Model-Based System Management for Multi-Tiered Servers

by

John Reumann

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
2003

Doctoral Committee:

Professor Kang G. Shin, Chair
Associate Professor Peter M. Chen
Professor Atul Prakash
Professor Robert L. Smith

To the glory of God

ACKNOWLEDGEMENTS

Thanks to Prof. Kang Shin's years of advice, support, and encouragement in the process towards completing the research for this thesis, this project is finally coming to completion.

I also want to thank my wife, Liz, for enduring side-by-side the twists and turns of the thesis process.

I thankfully acknowledge the work of my thesis committee: Peter Chen, Atul Prakash, and Robert Smith.

This thesis, of course, did not originate inside a vacuum. There were important people along the way, whose ideas, corrections, questions, insights, and support helped advance this process. I would like to thank all of Prof. Shin's Real-Time Computing Lab students for their interaction with me and my research. In particular, I would like to mention: Hani Jamjoom for good collaborative work and many discussions about research, Babu Pillai for collaboration and interesting debates and his willingness to help with just about anything, and Tarek Abdelzaher for debates on adaptive QoS. I would like to thank Ashish Mehra for the spirited discussions and collaboration, which led to an initial version of the Virtual Services concept. I would also like to thank Dilip Kandlur who served as my mentor at IBM T.J. Watson Research during my work there.

I would like to thank IBM Corporation and the National Science Foundation for their financial support for this work.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

CHAPTER I

Introduction

1.1 Motivation

The emphasis of software modularity in modern software engineering methodology has led to the development of multi-tiered service architectures as shown in Figure 1.1. Unlike traditional monolithic services, each service component implements only a small part of functionality, while relying on other (often standardized) service components to accomplish the functionality that is peripheral to the intended service functionality. For example, an HTTP server from vendor *A* may be integrated with a proprietary application via a CORBA interface, which in turn connects to an arbitrary back-end database using ODBC. Each service component may execute on its own specially-tuned server. Unfortunately, dependencies between such loosely-coupled services create system management problems that are not anticipated in the design of current OSs.

The multi-tiered scenario is becoming increasingly important as multi-tiered architectures find broader acceptance due to new technologies, such as CORBA [102], DCOM [61], J2EE [139], IBM's WebSphere [70], BEA's WebLogic [17, 18], and Microsoft's .NET initiative [145].

Many difficulties in multi-tiered server management arise whenever one attempts to configure different front-end applications to utilize shared back-end services (i.e., service consolidation) or attempts to enforce different policies for different users who use the same services. Such consolidation offers several tangible benefits. First, it reduces administrative overheads as the number of installed servers shrinks. Second, software redundancy is eliminated, thus reducing licensing costs and the waste of memory. Third, physical space and energy requirements can be significantly reduced.

The drive to ease software consolidation through OS support is not new. In fact, it has also inspired the innovation of the shared library abstraction in UNIX [9] and other OSs. Multi-tiered services can be viewed as the logical extension of the shared library concept for the networked server environment. For this reason, future server OSs should provide targeted support for service sharing analogous to the single-host concept of shared libraries.

Current OSs are ill-equipped to address the needs of multi-tiered applications, since they have evolved from single-server and workstation OSs. They provide only limited support for service distribution. In fact, the only support provided is basic network communication support. Nevertheless, multi-tiered services are built for such off-the-shelf OSs. This means that the ideal support sub-system will leave existing OS semantics intact,

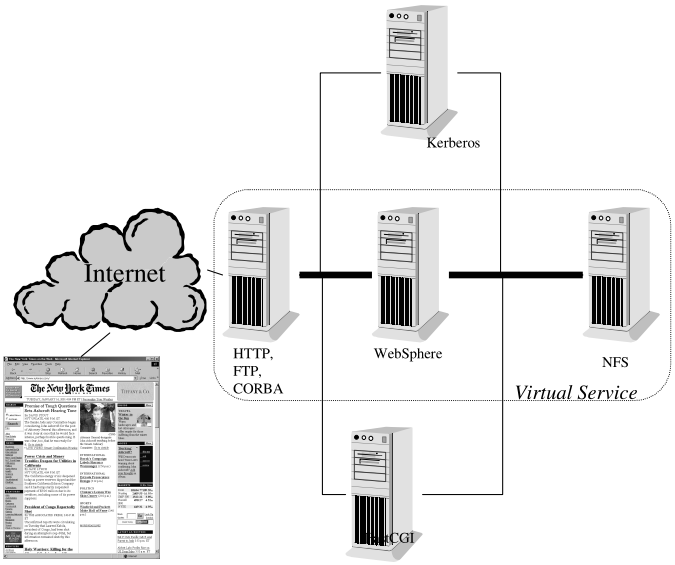


Figure 1.1: Multi-tiered server architecture

while adding mostly application and OS-transparent support for multi-tiered system management. One may wonder why the distributed OS concepts of the late 1980's and early 1990's [7, 99, 105, 122, 160] have not become the OSs of choice for today's multi-tiered systems. The first reason for this may be the semantical differences between single-host and distributed OSs, which complicate the porting of applications to distributed OSs. A second reason may be that many of the features of a distributed system, e.g., location transparency and single system view, may be counterproductive to the design of efficient server software and detrimental to the deployment of heterogenous computer hardware. Third, large, scalable, distributed servers [25, 74, 123] or applications can be built without specialized distributed OSs. However, these applications still suffer from manageability problems, which will be addressed in this thesis. In conclusion, migration from a single-host application to a distributed application atop a distributed OS is still a substantial task with limited benefits.

In multi-tiered systems, functionality is distributed across a number of hosts along service boundaries without requiring a distributed OS. Thus, application programmers can take advantage of their ability to distinguish between local and remote data access to optimize application performance. Moreover, it is possible to run different services on carefully-tuned server hardware. For example, a database server would most likely be hosted on a server with very large disk bandwidth and capacity. Thus any solution for multi-tiered system management support must be designed to cope with hardware and software diversity.

Perhaps, the most important reason why distributed OS platforms are not used to host multi-tiered services is that multi-tiered systems by-and-large require support that is orthogonal to the support that is provided by distributed OSs. Distributed OSs assume monolithic applications, i.e., all functionality needed to accomplish a certain task is implemented inside one potentially multi-threaded application. Multi-tiered services completely part with this basic assumption since their functionality is partially accomplished by independent, loosely-coupled sub-services. The challenge addressed by this thesis is to create

a *single service* abstraction (from a network of services) as opposed to the *single system* abstraction (from a network of machines) pursued by distributed OS designs.

The following scenario illustrates how and why multi-tiered system design differs from traditional distributed OS approaches. Suppose a service provider hosts outsourced applications for business clients using a shared hardware and software installation. A potentially large number of outsourced applications may require the same back-end services, e.g., DB2 databases, Web caches, search engines, and name services. These outsourcing centers can be very large; for example, IBM Global Services operates an outsourcing center in Raleigh, NC, containing over 10000 workstation-based servers and 250 mainframe computers. Therefore, it is highly desirable to consolidate shared back-end services. Unfortunately, this would cause many problems ranging from security to performance interference. Distributed OSs do not offer any answer as to how one could possibly consolidate such a service infrastructure. What is needed is a mechanism that maps back-end services' activities into different front-end services' contexts depending on who the back-end works for. To ensure that our solution remains largely invisible to the applications and the OSs, we adopt a model-based approach. This approach maps multi-tiered applications to generic service implementation patterns and applies generic performance controls that are known to be effective for applications of the recognized type. The patterns also aid in tracking the relaying of work from one tier to another.

1.2 Assumptions and Limitations

This thesis addresses the system management problem for multi-tiered servers under the assumption that multi-tiered activities are traceable without application modification. This means that there is *a traceable correspondence between the threads and processes that execute in the system and activities*, which are distributed and comprise all processing and communication executed on behalf of individual requests. Moreover, *messages that are passed between services must be recognizable*. Unfortunately, these assumptions are not always true without changing the applications.

First, the assumption of having a traceable relationship between activities and processes or threads may not hold. An example is a server such as the Flash [106] HTTP server, which bypasses the thread library and implements its own internal threading. The authors of Flash call this event-driven because each logical thread retains relatively little state, and threading is implicit, i.e., there is no explicit thread stack, and therefore, no use of OS threading. There is only one Flash process visible to the OS. However, this single process may be serving hundreds of connections simultaneously, each of which may represent different clients, and therefore, resource principals. In order to make the relationship between threads and activities observable at the system layer, it is necessary to instrument the application to export a unique connection identifier whenever it switches from serving one connection to another.

Second, the assumption that messages are recognizable may be violated. For example, this is the case when the client-to-server interface is implemented in a proprietary fashion using a random access medium (shared memory) communication channel, such as IPC shared memory. If clients submit requests to a server via a shared memory interface, it is very difficult to determine whose request a server is reading when it accesses the shared memory segment. In this case one would need to trace every memory access to trace the communication between clients and servers. However, such tracing would com-

pletely eliminate the benefits of using shared memory for communication. For all practical purposes, we must assume that communication over an unstructured, random access, communication channel, such as shared memory, is untraceable unless it proceeds through a shared library with a well-defined messaging interface.

Fortunately, the results of this thesis are not as limited as it may appear due to the above counterexamples. In fact, most service implementations satisfy our assumptions.

First, applications that implicitly encode threading into their code are very rare. In fact, in a survey of more than 30 service implementations, we found only a single widely-used service that does not use any of the following: a thread library, a virtual machine that provides threads, kernel threads, or kernel processes (the pre-2000 versions of MS-SQL). Later versions of the same service use Windows fibers (light-weight processes). There are many reasons against implicit threading, and the only reason for having them is a performance improvement over the use of generic threads. The arguments against implicit threading include:

- implicit threading is difficult to program,
- implicitly-threaded programs are difficult to understand and verify,
- a request that triggers an error in the program code will bring down the entire service,
- implicit threading is bad software engineering because it violates the separation of different programming aspects (functionality vs. threading),
- there is no need to reinvent threading,
- many services do so much application-level processing that the performance benefit of eliminating threading is negligible.

Second, while there are applications that utilize shared memory for inter-process communication, they typically do so through libraries that implement conventional communication abstractions over shared memory. For example, the Oracle database server uses shared memory for inter-process communication and for communication with local clients. This communication is encapsulated in a general-purpose linked library that implements the notion of full-duplex communication channels (virtual circuits) over shared memory. Applications never explicitly read from or write to shared memory. Instead, they enqueue messages into the shared library. Thus, messages are effectively traceable at the shared library interface, even though they are not traceable at the OS-IPC interface. One may argue that tracing shared library behavior is similar to customizing the OS to track specific applications. However, this is not the case because an unlimited number of custom applications may utilize a single shared library for communication with a *de facto* standard service. One would not even provide a shared library if there were no potential for reuse across multiple, independent applications. This means that we can possibly track hundreds of applications with only a small instrumentation of a certain shared library.

While there is the possibility that the assumptions made with respect to traceable applications and communications may not hold for some specific service implementation, we conclude that our assumptions hold in typical system scenarios. We also conclude that they are likely to hold in the future because system engineering and programming principles encourage service design to comply with our assumptions.

This thesis focuses on managing multi-tiered applications at the system or OS-layer by intercepting and interpreting its interaction with the OS. It is, however, necessary to interpret the term "system-level" or "OS-layer" broadly to not only include the OS itself but also threading libraries, communication libraries, and virtual machines; the term "runtime" may be substituted for "OS-layer." Despite the fact that this thesis specifically studies the case in which applications are built atop the OS without intermediary thread or communication libraries or virtual machines, one can directly apply this research to manage shared library and virtual machine-based service implementations.

1.3 Thesis Statement and Solution Overview

Current OSs are not able to effectively manage (police, monitor, and adapt) multi-tiered services because the correlation between OS abstractions and activities is not trivial and is disguised by intermediary tiers. This problem can be solved by exploiting the fact that services implement only a few different models of behavior. This thesis shows how a system can be policed and monitored by using models of application behavior in combination with OS modification for system object tagging, tag propagation, and tag-triggered interposition. Furthermore, this thesis shows that the changes do not unnecessarily slow down system operation and can be implemented without rewriting the service implementations.

This thesis specifically shows that:

1. System extensions that use models of application behavior provide configurable Quality-of-Service (QoS) control for concurrent multi-tiered services in the presence of shared back-end services in a manner that is transparent to the application.

2. By identifying the models of application behavior online it becomes possible to measure the performance and diagnose the bottlenecks of multi-tiered applications even in the presence of back-end services that are shared among multiple competing applications or client classes.

3. It is not necessary — not even for performance reasons — to build various specialized system management enhancements into each OS kernel in order to improve their operation in multi-tiered systems. Instead, OSs should provide a generalized layer of support for adding contextual information to existing system abstractions, configurable propagation of this contextual information alongside the flow of work, policing of interactions between applications and the OS, and the classification of system abstractions based on application behavior.

4. Even with advanced system monitoring and online policing capabilities that would facilitate online adaptation of QoS in multi-tiered systems, continuous QoS adaptation in a multi-tier system is not only of little benefit but can also produce suboptimal results when accounting for the cost of online adaptation in multi-tiered systems.

The basic solution can be summarized as follows. A request that enters the server farm is classified and associated with a system management context. As the request is picked up by a process, the process inherits the request's classification. If the worker process contacts a back-end service, its classification propagates with the subordinate request. System extensions use this classification to monitor and police the workload. Figure 1.2 corresponds to this rather simplified description. The individual problems and challenges addressed by this thesis are explained below in more detail.

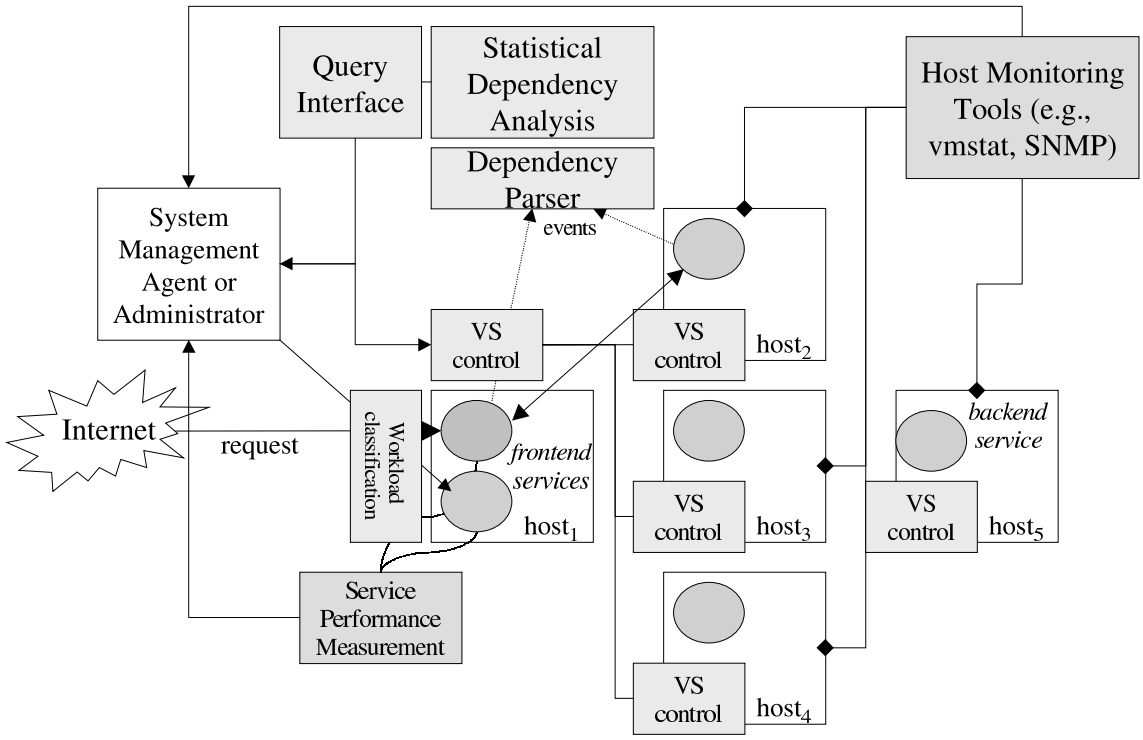


Figure 1.2: The multi-component approach to performance management in a multi-tiered service deployment as introduced in this thesis.

Context Preservation: A generic context abstraction that addresses the fundamental needs of system policing, security, resource partitioning, and monitoring in a multi-tiered system is introduced as a novel OS entity. The resulting solution is called *Stateful Distributed Interposition* (SDI). SDI also addresses many of the same configuration and state propagation problems that are addressed in configurable security framework research [49, 90]. In SDI, computations that propagate from one tier to another may be attributed with arbitrary contextual information, for example, security privileges, and resource limits. SDI can be configured to propagate context along with the work-flow through a multi-tiered service deployment without requiring application cooperation. Moreover, SDI permits the installation of context-dependent interpositions at all tiers. These interpositions may rewrite context or use its contents to control the processing of the intercepted activity. Thus, context can be used to police, monitor, or transform intercepted multi-tiered computations. An SDI-like OS mechanism is essential to building system support for multi-tiered services—including the solutions that are summarized in the following paragraphs.

SDI consists of an OS-level explicit context abstraction that is orthogonal to existing system abstractions, a network protocol for context propagation and remote access, and a μ-language to control the propagation of context. Moreover, the μ-language has features that permit its use as a control framework for multi-tiered activities.

Policing (Insulation of Co-Hosted Internet Services): Insulating co-hosted applications and different workloads that share some services in a consolidated server infrastructure, i.e., a service infrastructure with shared back-end services, is key to enhancing

the scalability of multi-tiered server farm designs. By sharing key hardware and software, one increases the utilization of resources and software licenses, thus implementing a more efficient approach towards server-centric computing.

Today's model of outsourcing, server co-location, lacks scalability. The co-location model completely insulates co-located applications by assigning each service for each outsourcing client to its own host. Every co-located server deployment operates as its own completely independent setup. The only resources shared among outsourced services are the uplink to the Internet, physical space, and power. Thus, every additional outsourcing client will incur costs that are almost as high as if the outsourcing customer was hosting his own services. The strict separation between co-hosted applications in current outsourcing environments is partly due to the absence of a resource reservation model for multi-tiered services. If one were to consolidate a system without this support in place, different co-located sites and applications would begin to interfere with each other.

This thesis proposes Virtual Services as a model for dynamic and deferred resource bindings in multi-tiered server farms, almost completely eliminating the interference between co-located services and workloads that share some services, if configured correctly. The idea is to delay binding any process or other OS resource abstraction to resource quotas until it is known for whom a service is performed. To on whose behalf a service is performed, the Virtual Service solution relies on administrator-specified workload tracking rules.

Monitoring (Interference Detection and Assessment): In order to determine resource allocations in a multi-tiered system, interference between co-hosted services and system bottlenecks must be considered. Picking the wrong resource control to tune a multi-tiered system will not yield good control over the performance of individual services and will not prevent interference between competing services and clients. This thesis proposes a problem-oriented monitoring and diagnostic subsystem, which detects interference and assesses its impact on the performance of different front-end services.

The proposed Performance Map (PMap) solution allows system administrators to diagnose a multi-tiered system without requiring them to understand all service interdependencies *a priori*. This solution is achieved by tagging requests that require monitoring and intercepting the resultant interaction of applications with the OS as they handle the tagged requests. The interception generates event streams on behalf of a distributed activity on all hosts that are involved in processing the request. These event streams are coalesced and interpreted according to a model-base of well-known service behaviors in order to construct a graph representation of service interdependencies. This representation can be used to manage, police, and analyze multi-tiered systems.

On-line Adaptation (Transparent Adaptation of Multi-tiered Services): A service hosting environment may experience temporary resource shortages due to an increase in the popularity of the services it provides. Several QoS adaptation schemes have been suggested to allow a single overloaded server to degrade gracefully under overload. However, none of the proposed schemes is applicable to multi-tiered service environments because they fail to account for the delays of resource allocation en-

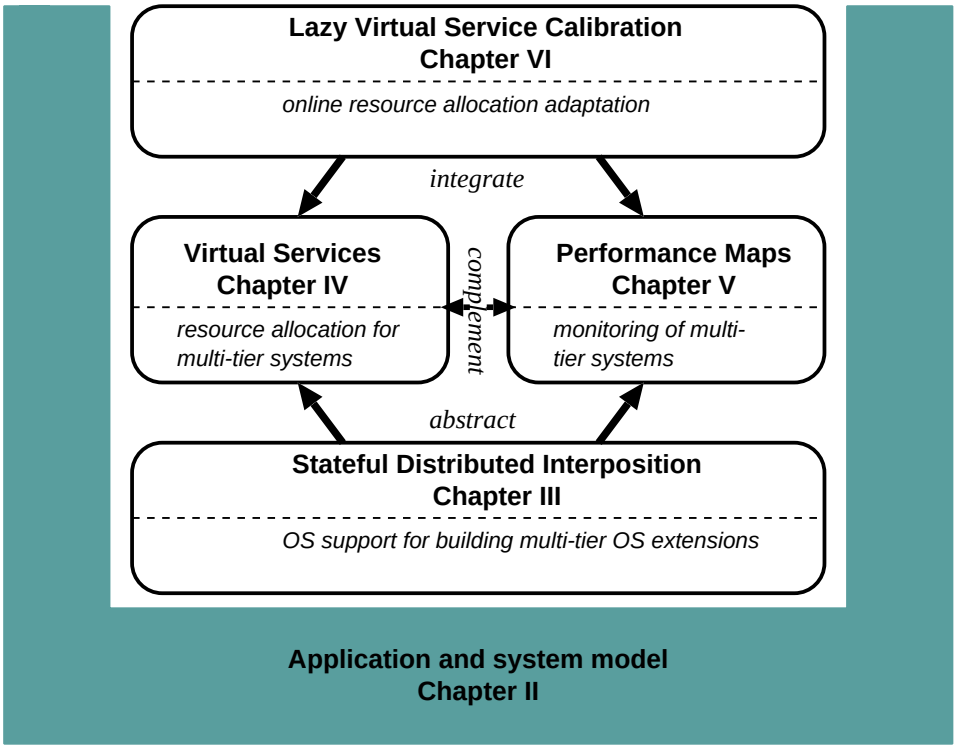


Figure 1.3: Structure of this thesis

forcement or the delay for obtaining a system state snapshot, both of which may be large.

This thesis shows that intuitively correct performance optimization approaches, such as dynamic programming based resource-packing, do not necessarily yield the best cost-optimizing behavior when accounting for resource allocation delays and the fact that resource schedulers at multiple tiers may make independent optimization choices. This thesis proposes slow calibration of resource allocations as an alternative resource allocation optimization approach with better performance characteristics. However, it also shows that the usefulness of resource allocation adaptation strongly depends on the delay of resource allocation enforcement and the delay of acquiring system snapshots. When delays are long and interarrival time distributions have large variances, online optimization generates no benefit over a static resource allocation.

1.4 Thesis Structure

Chapter II discusses the basic assumptions of this thesis and the fundamental concepts of multi-tiered systems and analyzes the design of typical server applications that are used in multi-tiered scenarios. This design study provides the foundational model of multi-tiered computations, which is then used to build application-transparent workload-tracking and policing features in the following chapters.

Chapter III presents Stateful Distributed Interposition (SDI). SDI is the general mech-

anism for the association of state with multi-tiered activities and is a rule-based facility that allows tracking this state across a multi-tiered system. The framework intercepts applications' interactions with OSs and applies context (or sets of tags) to system objects in a manner that is completely transparent to the applications. Rules for tracking these tags alongside the control-flow of multi-tiered applications can be specified as tracking rules. Furthermore, the infrastructure also supports policing multi-tiered activities based on the assigned tags, thus giving system administrators greatly improved control over multi-tiered applications.

Chapter IV introduces the Virtual Service abstraction, an application of SDI, which integrates the component services of a multi-tiered service into one virtual service for performance control purposes. The context or tags associated with system objects are resource-partition handles that are tracked as multi-tiered activities propagate from front-end to back-end servers.

Chapter V presents Performance Maps, another application of stateful tracking of multi-tiered activities. Performance Maps identify the interactions between the component services of a multi-tiered system. They are generated by OS interpositions and use forward- and backward-tracking of activities to map dependencies between services, workload, and resources. The purpose of Performance Maps is to aid in the configuration of Virtual Services, which presupposes a good understanding of the system's behavior. Furthermore, Performance Maps may be used for bottleneck identification and general system diagnostics in a multi-tiered deployment.

Chapter VI studies the design of an online resource allocation adaptation algorithm that uses Virtual Services for its resource control. The object of resource allocation as presented in this chapter is to optimize resource allocations with respect to an external objective function (i.e., utility or cost). Positive as well as negative results are presented.

Conclusions are drawn in Chapter VII. This chapter also summarizes avenues for future research. Figure 1.3 visualizes the relationships between the different technical parts of this thesis.

CHAPTER II

Multi-Tiered Service Design

2.1 Basic System Model and Terminology

This thesis addresses server-side problems of the following scenario. Independent clients utilize services on servers by connecting to them via the Internet. The server itself may host one or more distinct services, such as HTTP and FastCGI. Each hosted service should be configurable to its own per-service or per-user performance objectives. The hardware infrastructure of the server consists of multiple hosts, which are connected by a fast and highly-reliable Server Area Network (SAN).

Web-enabled front ends are integrated with proprietary applications, which, in turn, access back-end databases to accomplish their work. Such a system is called a **multi-tiered system** (Figure 2.1), with the independent, coupled applications representing the individual tiers. The tier number refers to how deeply nested an application is with respect to a composite service that is performed on behalf of a request. Service tiers are often located on special hosts that are optimized for the services that they host, but it is also possible that different tiers are hosted on the same server. In general, a database server would most likely be hosted on a machine with a fast persistent storage subsystem. In contrast, a payment authorization server would typically execute on a machine with a very fast communication subsystem to reduce the access latency to payment processing centers.

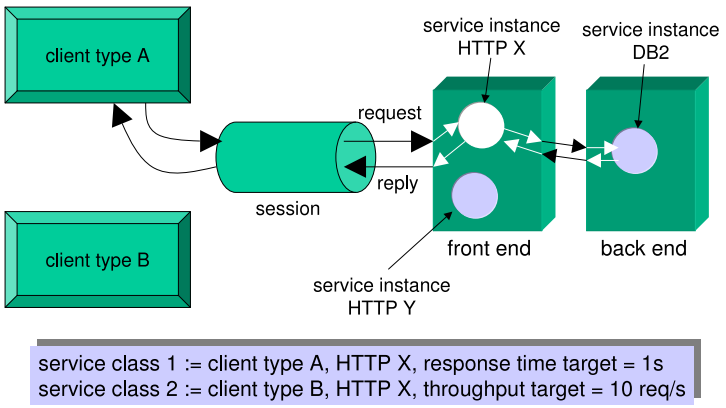


Figure 2.1: The terminology used to describe a multi-tiered system

A **service** is a collection of executable code with a well-defined request interface that is implemented using inter-process messaging, e.g., TCP. This means that service functionality is invoked by establishing a network communication channel with the service. The communication follows a specific protocol, such as, HTTP, SOAP or IIOP. The main advantage of service-based design is that the service behind the interface is an encapsulated black-box application. Encapsulation of reusable software components inside services is the translation of the shared library approach into the network-age. By placing different components on distinct hosts one can build much more scalable network services that also encourage software component reuse.

A **service instance** is an executing program that is ready to receive requests and process them. A service instance may consist of several processes and threads.

If the server provides multiple instances of the same service — for example, an HTTP server for company A and an HTTP server for company B — each of which may have to implement different performance characteristics, these distinct instances of the same service are referred to as distinct **service classes**. The term "service classes" also applies if specific client populations, e.g., clients from `umich.edu`, are treated as a separate class with class-specific performance objectives. In general, the term "service class" is a triple of performance guarantees, client population, and service instance (c.f., SLA [146]).

Services communicate with each other using a communication protocol, instead of the usual function call interface. An invocation over such a communication protocol is called a **request**. The service may or may not answer a request with a **reply** message depending on the service protocol, but usually they do. If a service A needs to issue several requests to some service B to obtain a result that it needs to satisfy one of its own requests, then all requests that A sends to B to satisfy one of its own requests are associated with one **session.** The boundaries between session and request are somewhat fuzzy since one can always view a session as a macro-request. The computation and communication that is executed on behalf of a request is called an **activity**.

Activities interact with the OS and depend on processes, messages, file descriptors, sockets, etc. for their execution. These abstractions are called **system objects**. What sets a system object apart from a system call parameter is that it does not live in the current call stack and that it usually survives the execution of the intercepted call. Typical system objects contain an OS-level context which is fixed and restricted to low-level attributes that are necessary to implement the system object.

A service that answers directly to requests from clients that are located on machines external to the server cluster is called a **front-end service** (oftentimes the HTTP service). If the front-end service contacts other services to complete a request, those helper services are called **back ends**. Back ends differ in how far removed they are from the front end. The client application is tier-0, the front-end tier-1, and all services that are directly called by the front-end are tier-2. Tier-3 applications are those that are directly contacted by tier-2 applications, and so on. Obviously, back-end applications may belong to several tiers in different request processing scenarios.

Quality-of-Service (QoS) is a broad term that means different things to different researchers. Unless stated otherwise, one may substitute it with average request turnaround time and request throughput. Note that the term QoS in this thesis always means server-side QoS. This QoS may not be the same as client-perceived QoS because the client will experience additional delays due to the Internet. However, the Internet is not under the control of the server infrastructure management, and therefore, this thesis defines and

measures QoS only in terms of front-end service performance.

The system is also assumed to process requests online (**online processing**). Online processing is different from offline processing in that only one protocol is used to submit a request to a server and to retrieve the result. Moreover, the request is not processed in any deferred mode, such as time-triggered, or by spooling the request to a file. Batch processing, in contrast, spools requests and writes output files that are retrieved through communication channels other than the request submission protocol (e.g., email).

For the most part this thesis does not assume that services will be modified to integrate with the mechanisms proposed in this thesis. It is also assumed that services are not maliciously sabotaging the proposed system-management mechanisms. Even though the proposed solutions will not fail entirely because of a malicious service, their effectiveness would be very limited in the presence of malicious services. This thesis also assumes that software is implemented according to best engineering principles, allowing for reasonable (documented, taught, and previously-implemented) design choices.

2.2 Multi-Tiered Server and Service Design Philosophy

The components from which modern multi-tiered systems are built are HTTP servers, proxies, distributed object frameworks, application servers, databases, and various components of standard or customized enterprise management software. While it is entirely possible to build component services of very limited functionality, it is a common design practice to embed substantial amounts of functionality into each component service. The reason for building powerful component services is that the functionality provided by the service must neutralize the disadvantage of using a slow, generic network interface to access each component service's functionality instead of a local method invocation inside a monolithic application.

A second development that has coincided and become synonymous with the emergence of multi-tiered component service architectures is the popularity of workstation or small PC-based server farms. Since network-oriented interfaces in multi-tiered servers allow the services to be scattered across multiple small, independent, and possibly heterogenous computer systems, clusters of small workstation and PC-based servers have started to replace mainframe-based server installations. Modern server farms can be scaled up to handle increased workloads by simply adding more inexpensive PCs. Therefore, this thesis assumes that a multi-tiered system is not only a system consisting of multiple, collaborating component services but also integrates many independent computers, which host those individual component services (a.k.a. hosts).

The remainder of this section will briefly define the main drivers for multi-tiered system implementation: component decoupling, system consolidation, and improved scalability. The main contribution of this thesis is to provide improved management support in consolidated systems by allowing activities that execute in a shared multi-tiered service network to be managed as if they were executing in a network of dedicated servers. The main goal of support for consolidation of service licenses, operating systems, and servers is achieved without negatively affecting the other drivers for the adoption of multi-tiered systems, namely their decoupled, loosely-integrated nature, and their scalability.

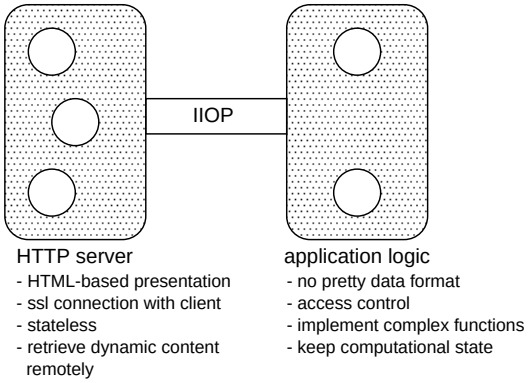


Figure 2.2: Decoupling the aspects of an application by integrating specialized component services.

2.2.1 Decoupling

The decoupling of major software components (Figure 2.2) achieved by encapsulating application functionality inside network services allows for a great degree of concurrency in the software development process. In developing a larger software system one may choose to purchase some component services, while completely rewriting others. This mix-and-match approach to building large software systems has greatly increased the efficiency of programmers. Today it is possible for a single programmer to build a deployment-strength e-Commerce server, such as an online store, in a few hours by simply integrating web-server, database, and application server components.

One of the problems of decoupling is that different component service implementations may interpret the semantics of the service interface differently, thus causing the failure of applications that depend on one specific interpretation of the service interface. Even though ensuring compatibility of different service implementations [120] is an interesting research problem, it is not the focus of this thesis. This thesis attempts to preserve the decoupling of components despite introducing coordinated performance management for multi-tiered systems.

2.2.2 Consolidation

Each component service may be part of multiple independent applications. A simple example would be a file server, which consolidates the storage and possibly backup functionality for all applications that depend on it (see Figure 2.3 for a different example).

Consolidation has many benefits. First, software license costs can be reduced. Second, configuration overheads are reduced if only one copy of the software needs to be configured. Third, software consolidation also leads to decreased hardware costs since some component service may be difficult to run as several instances on the same host, due to basic resource requirements for each service instance and port or address space contention.

Consolidation solves some of the economic problems of large software systems, but it introduces technical challenges. If component services are shared across multiple independent applications, it becomes more difficult to contain the faults inside each composite application. Ideally, component services would be built in a robust manner so that they

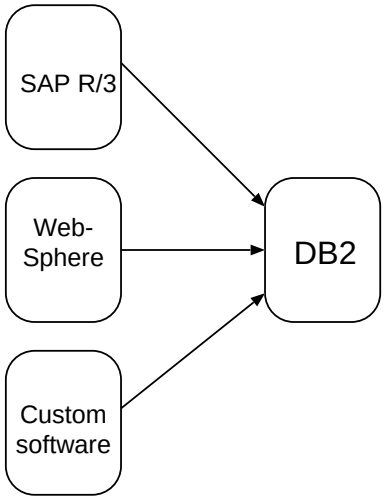


Figure 2.3: A back-end database can be shared among many different front-end applications, thus reducing software licensing cost and reducing maintenance overheads.

will not fail on any client input, thus avoiding the problem of fault containment altogether. Unfortunately, real systems usually contain many latent bugs. This problem is not explicitly addressed by the work presented in this thesis. However, by combining the work presented in Chapter III with Protected Shared Libraries [16], it may be possible to address the fault containment problem.

A second problem is potential name-space pollution. Services typically export some form of a namespace that allows front-end applications to refer to objects and methods provided by the component service. Unfortunately, competing applications may choose overlapping namespaces and manipulate each other's objects in the back end, thus leading to name and data conflicts. For example, a file server may be used by two applications that attempt to store their data in the same directory under the same file name. Fortunately, namespace problems are relatively easy to overcome by the common-sense approach of attaching application-specific prefixes to all names. With respect to file names most applications already manage their own application-specific directories. This programming mentality would simply have to be extended to database naming and to the naming of other objects that may reside in possibly shared services.

A third problem resulting from consolidation is that of performance interference, which is exhaustively addressed in Chapter IV. This problem is difficult to tackle because performance interference is a dynamic problem that depends on how a back-end service is utilized by different front-end services and users. Today's OSs do not provide configurable support for workload tracking and policing across multiple tiers, thus rendering fine-grained resource control and partitioning of multi-tiered systems virtually impossible.

2.2.3 Scalability

Well-designed multi-tiered systems scale well (Figure 2.4). First, the cost of scaling up is minimized due to the fact that multi-tiered systems are built from cheap, standard

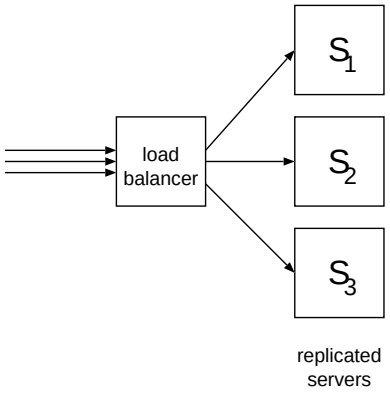


Figure 2.4: Connection balancing load-balancing devices allow spreading request load across multiple replicated servers. System throughput can be increased by adding another server.

components and require none or only little specialty hardware. Second, the system can be scaled-up component-by-component as opposed to upgrading to the next larger mainframe system. Thus, only the overloaded servers need to be either upgraded or clustered by adding more servers that provide the same functionality (Figure 2.4). Clustering is simplified because the interfaces between services are network-based and can be interpreted by moderately-intelligent load-balancers. Since load-balancing in a multi-tiered system is generally the same as balancing connections, it is possible to build simple network load balancers that spread load across replicated servers.

One of the drawbacks of increased scalability is that multi-tiered systems have more points of failure and are, therefore, more prone to experience downtime. Even though some researchers have discussed the possibility of building reliable clustered systems [11, 121], there are no good general solutions that provide low-cost fault-tolerance for real-world network servers. This thesis does not address the fault-tolerance aspects of clustered multi-tiered servers, even though the support of Chapter III may be used as a building block in addressing the fault-tolerance problem.

A second problem caused by the ease of scaling systems up to large sizes is that of understanding the system and its performance problem becomes virtually impossible once the number of applications and hosts grows beyond some humanly manageable amount. However, good system understanding is necessary to manage performance, diagnose performance anomalies, and to make upgrade and system configuration decisions. This thesis addresses the problem of understanding the dependencies between cooperating component services in Chapter V.

2.3 Service Implementation

It is difficult to improve system management support for multi-tiered systems without disrupting the overall design of a multi-tiered system. For example, if one introduced a resource management framework for multi-tiered systems, it would become necessary to tightly integrate the applications by adopting a shared resource management model. Moreover, applications would have to be integrated more tightly with the underlying OS to enforce a common resource reservation model, thus reducing the systems scalability.

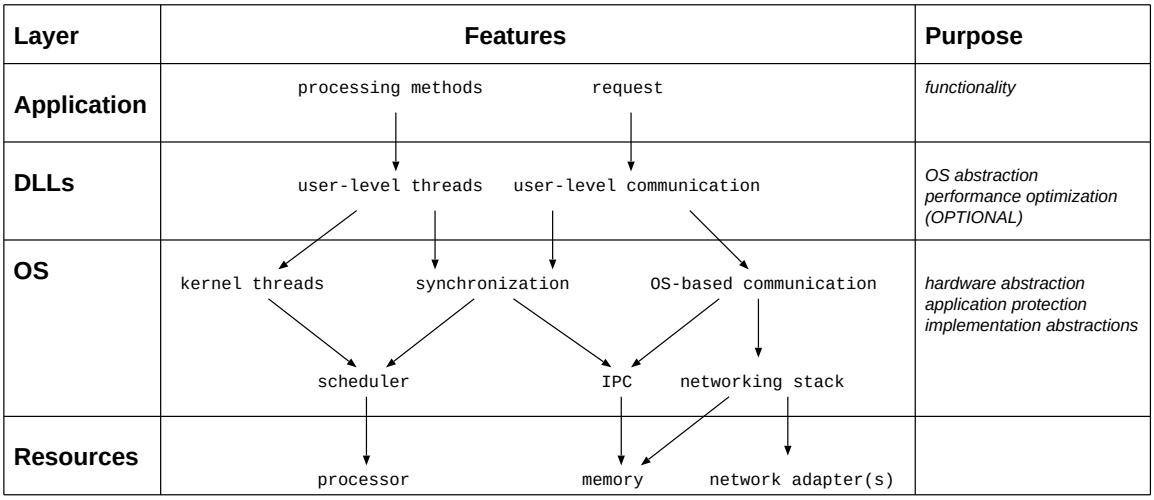


Figure 2.5: The processing of client-server workload maps to different abstractions and primitives at different system layers. For example, a request may be represented by an OS-level communication abstraction at the OS layer abstraction. By understanding how application-level workload is mapped to system-layer abstractions it becomes possible to take workload-management and monitoring actions from the lower levels of abstraction.

Today, the only feasible response to the system management problem in multi-tiered systems is to ensure that the deployment is not consolidated, i.e., each back-end service is used in only one application context. Unfortunately, it is not always possible to avoid all overlap between multiple multi-tiered applications. For example, a payroll application may need to access the same employee data as an employee benefits management application. This thesis addresses the problem of controlling the interference between competing applications.

To be able to control interference between competing applications without requiring re-design of the applications themselves, it is necessary to make some assumptions about the applications that allow tracing their request-related activities without requiring their explicit cooperation. This means that it must be possible to correlate OS-layer messages with client requests, and server-side processing with the request that triggered it. The remainder of this section describes the basic communication and processing models found in today's server applications and explains how it is possible to determine which activity (caused by a specific request) is being processed or propagated by observing the service implementation's behavior.

By dividing the service implementations into its two independent components, communication and processing, it becomes possible to model almost all widely-used service implementations. Figure 2.5 shows how the two key features of a multi-tiered service implementation, processing methods and request map onto system-provided abstractions. The objective of this thesis is to identify this mapping and to use it to affect the applications' interactions with the OS, thus externally controlling their request processing and resource consumption behavior.

Client/Server Protocol	DCOM	IIOP	HTTP 1.0	HTTP 1.0 (keepalive)	HTTP 1.1	DB2 (local comm.)	FTP
Communication model	request-per-message	pipelined (typically just connection recycling)	request-pre-connection	connection recycling	pipelined	Request-per-message	pipelined
Communication protocol	OSF RPC / UDP	TCP	TCP	TCP	TCP	FIFO queues (via DLL)	TCP (two connections)
Correlation between requests and replies	pattern matching	bulk requests, bulk replies by observing reversal from receive to send	OS (IOCompletionPorts)	JVM-based request queues	bulk requests and bulk replies by observing reversal from receive to send	pattern matching	Pattern and connection matching
observable	+	+	+	+	+	~	-

Figure 2.6: Mapping the different request submission models of widely-used client/server protocols to the communication models presented in this thesis.

The models of client-to-server communication and service processing behaviors introduced in this chapter are based on a comprehensive survey of application, network programming, and client/server literature [22, 41, 50, 57, 63, 119, 135–137]. The fact that computer programming textbooks heavily favor a few programming models for network services will make it likely for future generations of software developers to develop software that fits the basic models discussed in this thesis. A detailed survey of several applications and communication protocols is presented in Appendix A; Figures 2.6 and 2.8 list a few example communication protocols and service implementations (extracted from Appendix A) and map them onto the abstract processing and communication models addressed in this thesis.

2.3.1 Client/Server Communication Models

From an application programmer's perspective client/server computing often means using a service-specific API, which transparently connects to a server to provide some functionality. All communication between client and server is hidden beneath an API that handles the actual interaction between client and server. Each invocation of the API represents one or more client requests directed towards a particular service. Although the implementation of the client/server API could use arbitrary communication mechanisms between clients and servers to communicate individual requests, the communication between clients and servers usually follows one of only a few standard communication models (Figure 2.6).

Request-per-message: In this communication model the client sends a message labeled with a request ID to a specific server port. Once the server completes processing the request, it sends its reply back to the same port on the client machine from which it received the request carrying the ID of the request that it answers. Although one could easily implement variations of this scheme, e.g., the client identifies a different port to which the reply should be sent, this simple scheme is typically the one implemented. Client/server protocols that did not conform to this model (e.g., early versions of Windows Netmeeting)

have subsequently been altered to fit this model to allow communication to pass through firewalls, which also assume the simple request ID matching communication model.

Datagram-based request-per-message communication is not popular among client server application software implementors because it requires the service to reimplement congestion control, retransmission, and message ordering, which TCP provides at no cost. Except in cases when message ordering is not important, retransmission is optional (multimedia), or when applications are incompatible with TCP's congestion avoidance algorithm (also multimedia), datagram protocols for request-per-message client-to-server communication provide little benefit.

Request-per-connection: Despite the inefficiency of re-establishing a connection between client and server for every request, the request-per-connection model is frequently used. In this model the client establishes a connection with the server, submits its request, and waits for the reply to arrive on the backward-channel of the full-duplex client-to-server TCP connection. The server typically shuts down the connection immediately after sending its reply. Once the reply has arrived on the client side, the client closes the connection as well. The advantage of this model is that it is easily understood, easy to implement, and allow the server to terminate the connection. It gives this model an advantage in scenarios in which clients typically submit only a single request to the server and remain inactive for prolonged periods from the server's perspective. This processing model is very inefficient when clients tend to submit several requests in short succession because of the overheads of TCP connection establishment.

A salient feature of this communication model is that it works extremely well with load-balancers that balance the number of incoming requests among back-end servers.

Connection recycling: Instead of tearing a connection down after one request-reply sequence, one can keep the connection between client and server open and reuse it to submit another request after the client has retrieved its previous reply. This model of communication eliminates connection establishment overheads for all requests after the first. Since the server cannot be sure when the client is done, the connection is typically terminated by the client. This raises the problem of possible resource exhaustion on the server side due to misbehaving or failed clients. To counter this problem, the server may time out idle, open connections.

Request pipelining: While connection recycling reduces overheads when a client submits multiple requests to the same server, it does not reduce latency if the client is ready to send multiple independent requests at the same time. To send a second request the client would have to wait for a response to its previous request or establish a new connection with the server, thus increasing TCP connection establishment overheads and exhausting socket resources. When a client knows in advance that it is going to submit several independent requests to the server, it would ideally send those requests to the server simultaneously without waiting for intermediate replies. Once all outstanding requests are submitted to the server, the client starts waiting for replies. This model is the most efficient mode of client-to-server communication and implemented in protocols that are streamlined for performance. This mode of operation allows the server to work on several requests from the same client in parallel. A disadvantage of this model is that IP-based load-balancing devices cannot spread individual requests submitted by one client across several servers. Furthermore, pipelining requires a delimited request format.

Pipelining with connection recycling: The model of request pipelining can easily be extended by allowing the client-to-server connection to persist, thus combining the advan-

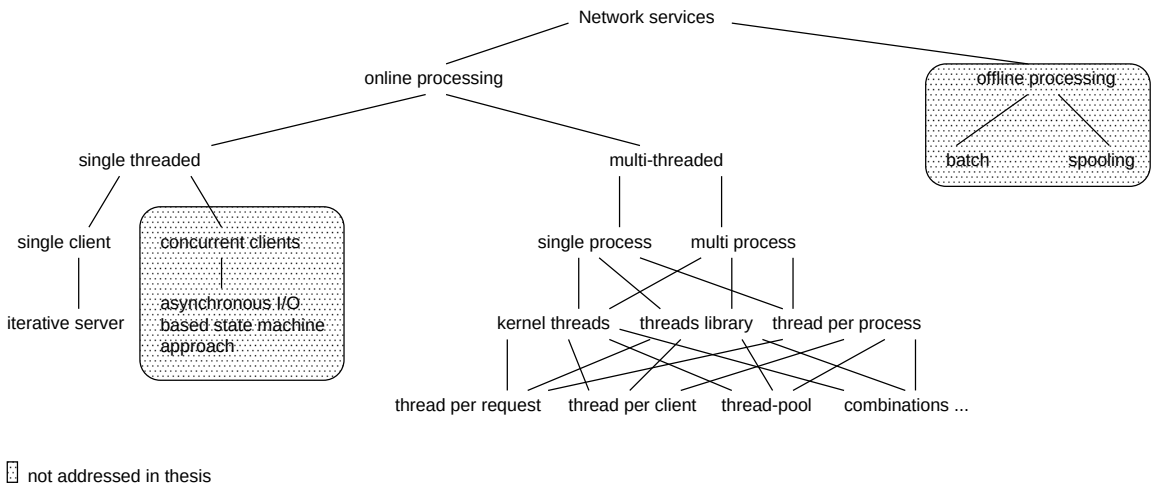


Figure 2.7: The different threading models for multi-tiered services.

tages and disadvantages of the basic models.

2.3.2 Service Processing Models

Requests are received and processed by a service, which may cause the service to submit subordinate requests to higher-numbered tiers. To correlate original client requests with subordinate requests that are submitted by the server-side component services, it is necessary to model not only how individual requests are submitted and received, but also how they are handled by the service implementations so that an OS-level observer is able to trace a multi-tiered activity.

The models of server-side processing considered in this thesis are derived from instructional literature [41, 119, 136, 137, 144], technical manuals for various services [29, 70, 71, 104, 143], and source codes. The processing models of actual multi-tiered service deployments are compositions of the basic models introduced here. In general, each tier implements its own processing model independent of the other tiers. Moreover, some modern services integrate different processing models within one tier.

The remainder of this chapter will not strongly differentiate between the different thread abstractions (LWP, processes, threads, kernel-threads, user-space threads, . . .) found in today's network services (see also Figure 2.7). It is assumed that threads can be uniquely identified regardless of the thread model chosen by the application. This, of course, can only be done if the thread model is implemented separately from the application logic, which is the case for all threading models except for the state machine server design approach.

Single worker: A process accepts an incoming request, handles and finishes it. Upon finishing the request, the single worker calls `accept` or `select` again. This is by far the simplest processing model, which is also referred to as an iterative server in the literature [136].

Forked worker thread: A master thread accepts an incoming request and forks a worker to handle it (e.g., **inetd**). Upon finishing request processing the worker exits. This process-

Application	inted	Apache	MS IIS	Weblogic	Servlets	DB2	MS-SQL	SAP R/3	Pre-2000 MS-SQL
Processing model	forked worker	thread pool	thread pool	Thread pool/staged	Nested worker	staged thread pools	thread pool and forked worker	staged thread pools and foked worker thread	thread pool of state machines
Threading provided by	OS	OS	OS	JVM	JVM	OS	OS	OS and LWP library	OS and application-level
Internal communication provided by	n/a	OS (IPC, semaphores)	OS (IOCompletion Ports)	JVM-based request queues	JVM	OS (FIFOs)	OS (IOCompletion Ports)	OS	OS (IOCompletion Ports)
Request processing traceable?	+	+	+	~ (requires tracing of access to request object)	+	+	+	+	-

Figure 2.8: The different internal processing architectures of some example service implementations.

ing model requires minimal configuration since the system will always generate as many threads as are needed to handle the incoming workload. Moreover, by implementing a process-per-worker model, one can mitigate the impact of memory leaks and other faults in the service's implementation. Unfortunately, this model does not take into account the fact that total throughput tends to decrease when a system has excess concurrency. Reasons include scheduling overheads, cache pollution [157], memory exhaustion, and thrashing. Therefore, the forked worker thread model can produce severe performance problems, especially on memory-constrained servers. Despite its weaknesses, this processing model remains popular for simple, low request volume services, and for services in which the start of a worker constitutes only a small fraction of total request processing time.

Thread pool: A master/dispatcher process accepts an incoming request and dispatches another existing process to service it. The main advantage of this processing model over the forked worker thread model is that it amortizes the cost of setting up a worker thread over an infinite horizon. Implementations that follow the thread pool model can be much more efficient than the forked worker model, especially when the heavy-weight `fork` system call takes relatively long compared to the remainder of a request's processing time.

The synchronization between a thread pool dispatcher and its workers can be achieved in many ways. One may use semaphores, FIFO pipes with many concurrent readers, file locks, and application-level request queues. Busy waiting by worker threads (a theoretical option) is a very inefficient solution that is not used.

A second implementation pattern that is frequently used in thread pool implementations is that of a request queue. Instead of passing the connection from a dispatcher to a worker that resides in a thread pool, the dispatcher may utilize a special request queue (e.g., OS-level message queues, and FIFOs) to release a thread from a thread pool. A request queue between the dispatcher and thread pool allows the service to temporarily accept many more requests than it has workers to service them.

Helper threads: A main thread creates helper threads to handle different aspects of a request, such as database queries and name service access. Helper threads are typically created for one of two reasons: increase concurrency within a service or functional encapsulation. An example of encapsulation would be a helper process that is created to execute a separate binary program, which is part of providing a service. This pattern iso-

lates the binary program from the processing context of the service logic. Helper threads are often created to increase concurrency when a service accesses the relatively slow I/O subsystem while executing other operations in parallel. Helpers may either be generated on-demand (forked worker-style) or dispatched from a pool of workers (thread pool). The difference to the thread pool and forked worker models is that both the dispatcher or parent and helper stay active on behalf of the same activity. Moreover, parent or dispatcher will resynchronize with the helper thread.

Staged worker: Work is passed between processes that implement different parts of request processing. The last stage finishes the request. This model is essentially the recursive extension of the thread pool model. Recursive dispatching is not very common but can be found in some commercial service implementations (e.g., ORBIX 2000 for multi-threaded objects).

Nested worker: Processing also passes through several stages as in the previous model. However, upon completing each stage, the previous stage resumes processing. This processing model is the recursive extension of the helper thread model. Such a nested processing model is the inevitable consequence of sequential service implementations. Passing control from one thread to another or from one service to another is analogous to a synchronous method invocation — the caller suspends until the callee returns the result.

Peer relationships: Peer relationships are the essential building block of multi-tiered systems. A front-end process (acting as a client) connects to a back-end process of a different service in order to accomplish part of the service required to handle its currently active request. The protocol between the two peer processes follows one of the previously-identified communication models. The processing model is best compared to either staged or nested worker.

If the front-end service's process suspends until it receives the result from the back-end service, to which it submitted a subordinate request, then the peer relationship implements a nested worker model. In contrast, the front-end process may move on to service another request, in which case the peer relationship is best described as an extended staged worker model. Both types of peer relationships can be found in multi-tiered systems. However, the nested worker relationship is the more common case.

2.3.3 Exceptions

Most service implementations can be described by the models presented thus far. So, it is possible to observe the propagation of activities that pass through these services without having to modify the service implementation by simply monitoring or tracing applications' usage of OS interfaces. Nonetheless, there are examples of applications and request submission protocols that do not directly fit the proposed models. In order to trace activities that pass through such services or protocols, it is necessary to either implement application- or protocol-aware observers, or even to modify the applications directly so that we can infer the mapping of messages and processes to activities.

An example of a communication protocol that cannot be tackled without protocol-awareness is FTP [111]. During an FTP exchange between client and server, the client opens a control connection to the server. Result codes for the clients' requests are sent back to the client via the same control connection. However, the majority of the the communication between client and server is conducted over a secondary data connection that the server initiates. Therefore, only the control connection of FTP fits the above models.

To trace all packets that are part of an FTP data transfer (i.e., the activity triggered by a GET), it is necessary to track the application-level protocol.

The reason for FTP's peculiar design is that the FTP application needs to remain responsive to control commands despite a potentially very full TCP socket data queue. This design has been reapplied to a number of multi-media transport protocols (RealPlayer and Windows Media). It should be noted that such bulk transfer protocols are not the targeted by this thesis because they are typically not interfacing to multi-tiered services. Typically, the transfer application is a monolithic application that streams or transfers bulk data directly from local data storage to the clients.

An example of a service processing architecture for which it is impossible to trace activities that are passing through it is the pre-2000 version of MS-SQL server [19, 45]. The pre-2000 versions of MS-SQL do not utilize Microsoft's light-weight thread implementation (called fibers) but multiplex processes between concurrent client connections once the process limit for the SQL server is reached. This multiplexing behavior occurs only if the maximal number of simultaneous connections allowed for the SQL server exceeds the number of allowed processes. To trace activities that pass through the SQL server it is necessary to instrument it, so that each process announces which request it is currently processing. The pre-2000 versions of MS-SQL server are a hybrid of a thread-per-client model and a state-machine-style server architecture, which is also not traceable by an outside observer. More recent implementations of the MS-SQL server utilize fibers and map each client session to a fiber.

State-machine-style server architectures are typical of older server implementations, such as the earlier versions of MS-SQL server to avoid the threading limitations (maximal thread count limits) of older OSs. Academic service implementations (e.g., Flash [106]) also choose state-machine-style implementations to optimize performance for a specific test scenario. Newer service implementations typically rely on OS's much improved process, threading, and light-weight threading support to implement the easier to maintain processing models that were introduced in this section.

As was mentioned in Section 1.2 batch and offline processing models are not the focus of this thesis and are not covered by the above processing models. Most offline processing models commit execution requests to a file, which is later executed by a computation server. The computation server typically submits the result to the client in a deferred mode via an outside communication channel, e.g., an email message. In this model sometimes no OS abstraction is actively making progress on the submitted batch request; the request is dormant. It is impossible to identify when a dormant request is processed by a server thread because the server may read requests and rearrange their execution order internally. Reading a request does not necessarily imply that the server will process the request immediately in an offline processing model.

2.4 Discussion

Our in-depth study of different server application implementation patterns suggests that the flow of work through a multi-tier system can be modeled from a few basic patterns. The mechanisms used to implement the same service model may differ across different OS and middle-ware architectures because different runtimes provide different implementation mechanisms. But this only means that the basic models are expressed in different runtimes and not that the service implements any fundamentally different service model.

To identify a processing or communication model within a runtime system one must capture categories of implementation mechanisms that are used by the services to implement the different service and communication models and link them to the abstract service and request models. Regardless of the runtime system (this thesis focuses on the OS) one must capture creation, communication, synchronization, and transformation functionality of the runtime and network layer to reconstruct the path of an activity from observation traces that are collected at the runtime layer or to police an application as it executes against the runtime's API.

The interception of multi-tiered activities at the runtime layer must accomplish three distinct goals: classification of activities, capturing activity propagation and activity processing.

2.4.1 Classification of Activities

Activity classification is a process that begins with the receipt of a request at a front-end server. When a network packet from an external client is received by a front-end server, it is possible to classify the incoming packet as part of a new or existing activity or class of activities. For example, one could associate all incoming packets from host `10.100.10.100` with the class of high-priority activities. But classification does not necessarily stop at the front-end server.

As activities interact with other hosts or the runtime system of the hosts on which it is executing, it may be possible to reclassify the activity spawned by a front-end request. With respect to reclassification the important runtime functionality to intercept relates to the applications' attempts to modify their state inside the runtime system. For example, the fact that an application is invoking a particular binary image (via `exec`) may reveal what type of workload it is executing (e.g., browsing vs. purchasing customer). There are many other mechanisms, such as `setuid`, `setgid`, `getenv`, that are used by applications to police and properly execute workload. All of this application-OS interaction is useful for system management because it can also be used to transparently infer information about the active request. One could possibly attempt to intercept all interactions of the applications with runtime systems in an effort to classify them. However, for the classification of activities it is generally sufficient to intercept a few interactions between the network stack and external clients and between the OS and the service implementations.

Since we assume that it is possible to map activities to processes, sockets, and other system abstractions, one can change the classification of an activity by re-classifying a process that issued a certain system call. Thus, activity-classification reduces to system object classification.

2.4.2 Tracking Activity Propagation

To track activities as they propagate from front-end services to back-end services, it is important to intercept a different set of runtime functionality, namely, its communication subsystem. Messages sent between collaborating services fall into two major distinct categories: local and remote communication. Local communication encompasses special IPC-based communication libraries, e.g., message queue abstractions and standard OS-based IPC (message queues and IOCompletionPorts). Remote communication includes network messaging systems, primarily TCP/IP.

The communication between services is essential to the understanding of the flow of work between services. Without understanding how an activity passes from one host to another, it is not possible to police resource contention in back-end services nor is it possible to generate meaningful statistics with respect to the behavior of multi-tiered activities. To achieve activity tracking behavior it is necessary to add a basic message tagging infrastructure to all communication layers including both local IPC and remote communication.

2.4.3 Tracking Activity Processing

Without tracking processing progress towards the completion of an activity it is impossible to accurately correlate activities across several tiers. For example, a middle-tier service that forks a new process to handle each incoming activity (forked worker thread) would obfuscate the relationship between the front-end request and any work that the middle-tier service submits to back-end services in order to service the incoming request. A multi-tier-oriented system management approach that is transparent to the applications must track the generation of a worker thread and properly account for the work generated by the worker thread by charging it to the activity in whose context the worker was created.

To accurately track progress in processing an activity, it is essential to track changes in the process or thread system, e.g., task state changes, process creation, and process destruction. Furthermore, it is important to integrate any tags that were applied to received messages into tags for processes. For example, a process that `accept`s an incoming message of a certain activity type should probably be labeled with the same tag as the connection that it just accepted or with one that is directly derived from the tag of the incoming connection.

2.4.4 Conclusion

The high-level service and communication models are expressed via system implementation mechanisms. This expression must be exploited if one wants to police services based on a high-level understanding of its service model instead of using proprietary control mechanisms offered by the service implementations themselves. Without any correspondence between a service's use of the underlying runtime system (e.g., the OS) and the introduced service models, it would be impossible to accurately account for or police multi-tiered activities. There would simply be no correspondence between OS abstractions (processes, sockets, file descriptors) and activities. One cannot manage arbitrary service implementations and activities in an application-independent manner if their behaviors cannot be described in an abstract model that is traceable at the OS layer. The next chapter introduces a framework that allows for fine-grained and customizable tracking of activities that use the OS as their runtime.

CHAPTER III

Stateful Distributed Interposition

3.1 Introduction

Current OSs provide inadequate support for the management of multi-tiered systems. Each host of a multi-tiered server setup is configured and managed in isolation. There is little coordination across hosts and there are no effective means to enforce policies with respect to distributed activities that span across several tiers. Thus, important OS features that are used to manage and control single-host applications, such as resource quotas, user identities, tracing, and shared environment variables, are not available to multi-tiered applications.

The task of coordinating system management policies across the multiple hosts that constitute a multi-tiered system cannot be left up to the applications themselves. For example, an application that maps all of its incoming requests to one OS-level user ID can access all resources that are available to this user ID. This implies that a simple bug in the application may potentially corrupt the security of the entire system (if the user is `root`). It may be more appropriate to change the server process to user ID mapping based on the incoming sessions' source address. Unfortunately, this policy can only be enforced if the application supports such dynamic user ID mappings.

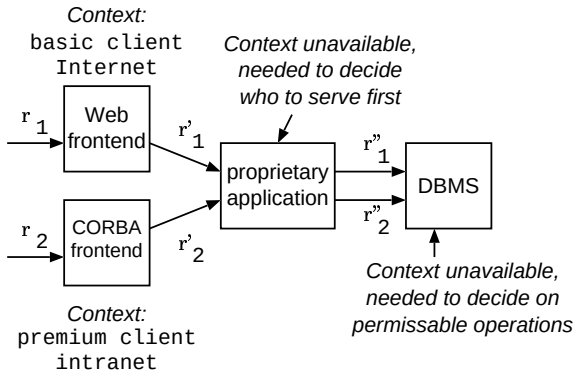


Figure 3.1: Context information is lost as requests r_1 and r_2 propagate across shared intermediaries. Neither preferred processing nor effective access control can be implemented at the system level.

This thesis proposes Stateful Distributed Interposition (SDI) as a mechanism to police applications' behaviors without their direct support. The idea is based on providing a distributed context service, which allows associating basic system objects (processes, messages, file descriptors) with auxiliary context. The second component is the ability to interpose policies on the system interfaces to control the interaction of applications with the system and other applications, and to track the propagation of context. For example, if the front-end host that received the request contacts a back-end host, the front-end process' context can be transparently transferred to the back-end process. This OS extension makes it possible to superimpose advanced stateful, distributed OS features, such as environment variables, scheduling classes, and security attributes without modifying the applications.

Propagating state across middle-tier applications that are oblivious to the fact that they are shared by multiple competing front-end services and users (i.e., service classes) is a problematic issue (as shown in Figure 3.1). For example, to achieve performance isolation between competing front ends in spite of shared backend services, it is necessary to propagate a resource context from the front-end to the backend machines [10, 118] and to enforce it in the backends' resource schedulers. *Virtual Services* [118] (Chapter IV), for example, associate each incoming request with a Virtual Service (VS) resource partition structure, which defines a number of resource limits. This VS affiliation is tracked from the sender of a message to its receiver, possibly across host boundaries, in a manner that is transparent to the applications. To propagate VS resource descriptors, the OS must become aware of this resource limit tracking abstraction at various places including the networking stack, which must tag communication messages to allow the receiver to identify the resource descriptor that should be charged for an incoming message. This chapter provides a framework that greatly simplifies such implementations.

Problems resulting from the loss of contextual information when tier boundaries are crossed range from trivial but annoying problems to serious security issues. We have all encountered the trivial, but nonetheless annoying problem that `telnet`ing from a login machine to a remote host to execute a program, the resulting X-Window is not automatically sent to the X-Server on the login machine. To achieve X-forwarding, the `DISPLAY` environment variable must be set appropriately on the remote machine. Of course, some login tools (e.g., ssh) fix this problem using connection forwarding. However, it would be better if the `DISPLAY` variable (or something equivalent) would automatically propagate among tiers, even if the login session was relayed across several intermediary login sessions.

Information loss across tier boundaries is far more serious when system security and system integrity are to be preserved in a multi-tier system. For a single-tier service, security problems are relatively easy to solve, because executions are triggered by a user who is logged in at the local node. Consequently, access to files, FIFO queues, executable programs, and hardware resources is controlled by processes' user and group IDs that are maintained by the local OS. If the user decides to utilize remote services (second tier) instead of executing local programs, then local security mechanisms fail. The remote server OS cannot identify the user who sent the network packet that it received, and hence, simply handles it as raw data. So, it is up to the services to reconstruct security context information and to enforce appropriate security policies. As a result, system administrators must configure security mechanisms in many distinct applications (e.g., Telnet, Ssh, Web, NFS, AFS, Samba, RPC, Corba, etc.), each in its own peculiar fashion. Solutions like Flask [130] and DTE [13] attempt to address this dilemma by proposing distributed

security attribute propagation to manage system integrity and security in a consistent and completely application-independent fashion.

Despite substantial differences among DTE, Flask, and VSs, we observe significant similarities with respect to administering or policing multi-tiered and distributed applications in a manner that is transparent to the applications themselves:

1. introduction of a new separate OS-level resource/security/integrity abstraction,
2. creation of associations between processes, messages, and the new abstractions,
3. propagation of associations across host boundaries (e.g., between shell and remote file server), and
4. interposition of security and resource constraint enforcement functionality on standard system interfaces.

This chapter lays the foundation for building stateful system management extensions that require coordinated policy application at all hosts of a multi-tiered system. The proposed *Stateful Distributed Interposition* (SDI) approach addresses the following requirements.

Keep State: Provide a customizable, distributed state abstraction allowing queries and basic operations on state variables. State variables are stored in a *context* object. It can be used to store security classification, monitoring descriptors, resource constraints, and the like.

Generate Context: Provide a mechanism that automates the generation of context, i.e., a classification facility.

Propagate Context: Automate the propagation of context between cooperating services and across system abstractions, e.g., messages, sockets, and processes. If necessary, state variables and context may need to be altered during propagation to match site-specific requirements.

Utilize Context: Use dynamic contextual information that is associated with system objects to trigger interpositions on standard system interfaces or policies that restrict or allow access to the intercepted interface. Stateful interposition allows system behavior to be influenced by context. This is an important advancement from prior interposition schemes, which operate only on fixed system state.

This chapter is organized as follows: We define our system model in Section 3.2 before outlining the overall approach and system-level architecture of SDI in Section 3.3. Section 3.4 argues for the high-level design and concepts of SDI. Important implementation aspects and practical design issues based on experience of an implementation of SDI for Linux are discussed in Section 3.5. Applications of the prototype system are presented in Section 3.6. Readers approaching SDI from an application-oriented angle may want to jump to Section 3.6 right after reading Sections 3.2 and 3.3 to obtain a feel for the use of SDI before returning to the theory of operation and design in Section 3.4. Section 3.7 evaluates the performance of our SDI prototype. Section 3.8 describes how SDI generalizes previous approaches to system design by interposition and how it generalizes previous domain-specific solutions that depend on distributed context. The chapter ends with concluding remarks in Section 3.9.

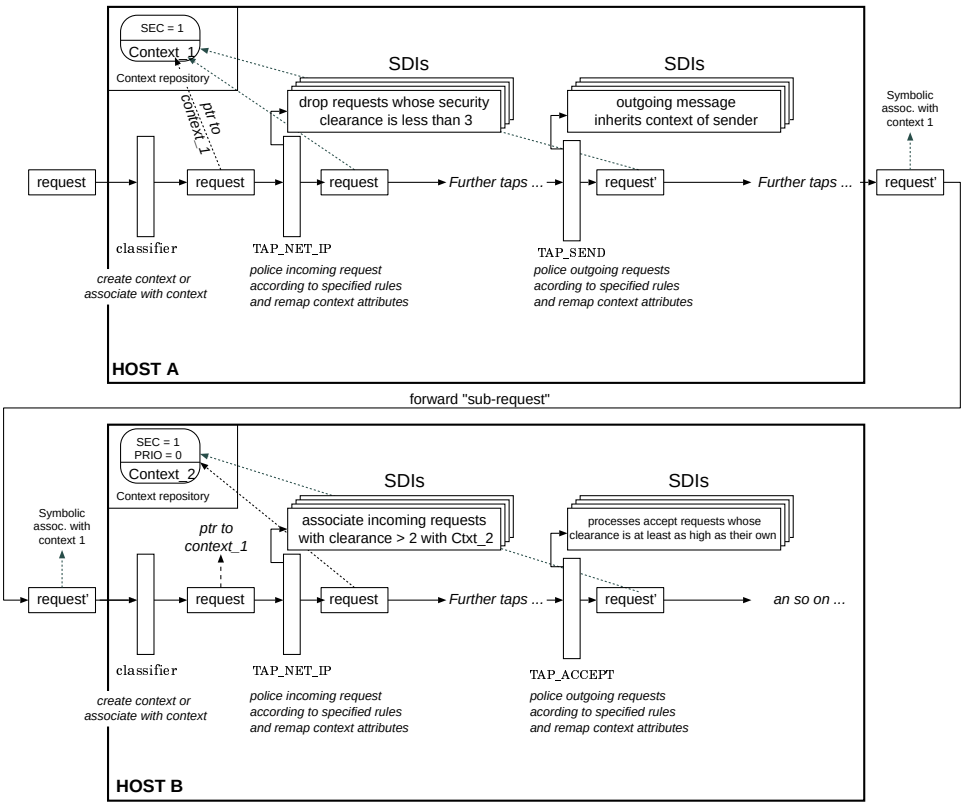


Figure 3.2: SDI provides mechanisms to associate additional state with incoming messages, and propagates it according to SDI rules as request processing progresses.

3.2 System Model Additions and Additional Terminology

In addition to the system model introduced in Chapter II, SDI assumes a typical layered OS design. This assumption allows the placement of interception points for multi-tier computations at the junctions between OS layers. These interception points are called *taps*. For example, a tap may be installed at the transition from network to IP layer processing or before and after specific system calls. Taps are used by SDI to police, monitor, and redirect processing to interpositions.

3.3 Architectural Overview

The SDI architecture provides a *context abstraction* that stores name-value pairs that are similar to POSIX environment variables. Unlike environment variables, which are embedded inside a process, context is provided as an independent system abstraction that can be configured to propagate with the workload across multiple tiers. This allows each context object to be associated with multiple system objects simultaneously — a necessity in distributed systems where multiple processes, sockets, and kernel threads may work on the same request or request class simultaneously. To allow context to be addressable from all hosts inside a server cluster, it is implemented as a network object.

Since context objects are separate from other system objects, it is necessary to pro-

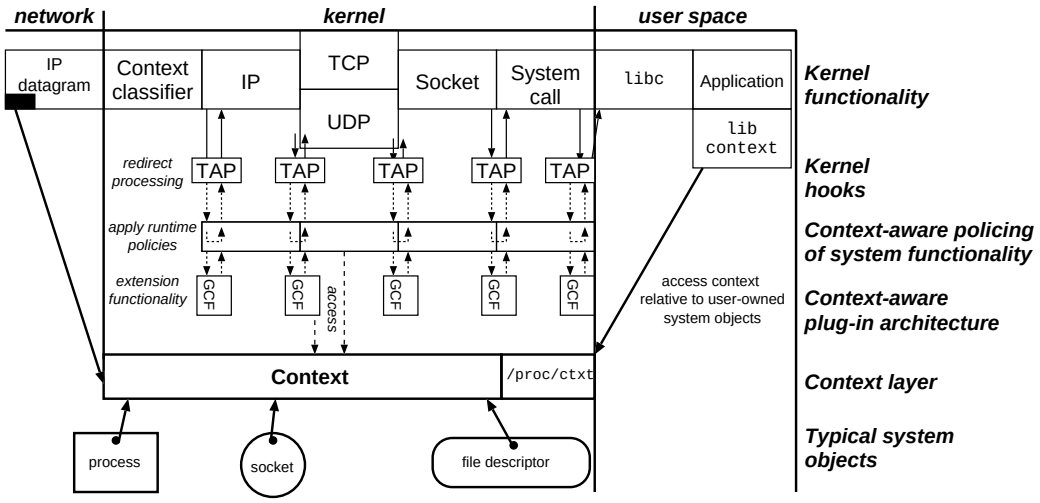


Figure 3.3: Architectural overview of SDI-Linux integration

vide a mechanism by which system objects can be bound to a particular (new or existing) context object. The initial context for "context-free" messages that are received from the outside network is assigned using *classifiers* (Figure 3.2). Context classifiers parse incoming communication messages to infer their appropriate context binding. This initial classification is only preliminary, i.e., it can be modified in later stages of request processing, i.e., when the activity interacts with the OS. Classifiers extract as much useful information from incoming messages as is possible and use it in making a preliminary context determination. Processes, sockets, and file descriptors may also be classified manually, by user-level scripts, or by the applications themselves.

The context of a packet or a process impacts its processing throughout the OS, at the system call interface, and, if applications use context attributes, at the application layer as well. This is similar to the effect that environment variables have on the behavior of an application. For example, the value of the `HOME` environment variable affects an application's interpretation of relative file names. Similarly, context attributes affect OS behaviors at so-called tap points, which are shown at layer transitions for the Linux OS in Figure 3.3. These layers may differ slightly across OSs. However, equivalent features can be identified in all OSs and wrapper layers may provide common interfaces [54].

Taps intercept the control flow of the OS, and as is shown in Figure 3.2, can apply context-dependent rules, or SDIs, to intercepted computations. SDIs are installed by the system administrator or the applications at the tap points. The name "SDI" is used for both the proposed framework as a whole and for the interposed rules since the framework is named for its configurable rules.

SDIs tackle several important aspects of context maintenance for multi-tiered applications, using a standardized format (Figure 3.4). First, they intercept the flow of a multi-tiered computation with conditional expressions that may include clauses referring to context. Second, they provide a mechanism to remap context attributes. SDIs can adjust context attributes with respect to the intercepted function and the context contents of the intercepted system objects. Thus, a previously-coarse context attribute, such as `REQ_TYPE = HTTP` can be refined once the application is observed to execute another program to

`REQ_TYPE = CGI`. It is also possible to bind a system object to an altogether different context object. This capability is used to propagate context. For example, if a process receives a message at the `TCP_RECEIVE` tap, one can (re)bind the receiving process to the received message's context, thus implementing context propagation.

The third feature of SDIs is to provide a means of policing intercepted multi-tier computations. Policing is a recurrent theme across security, performance management, and fault isolation mechanisms for multi-tiered systems. To this end, SDIs include an action directive, which may apply a standard policy, such as `DENY`, to an intercepted computation. In addition to a number of pre-defined standard actions, SDIs also permit the invocation of arbitrary system extension functions called Guarded Context-Dependent Functions (GCFs). For example, one could implement encryption and decryption GCFs that are called when messages whose respective contexts indicate `MSG_ENCRYPTED = 0` and `MSG_ENCRYPTED = 1` arrive. The fact that context is used by OS interpositions that are transparent to the applications, requires context to be implemented as an OS-level abstraction.

Applications access context through a simple library interface (Figure 3.3), which allows them to query and set the values of context attributes in a manner similar to the interaction between applications and their POSIX environment variables. However, the similarities are only limited because of POSIX environment variables are local. Unlike context they are valid only within a process context, and they do not implement any form of access control.

Applications can rely on OS mechanisms to propagate context alongside their communications with back-end hosts. They no longer need to implement their own context abstractions, which are incompatible across different distributed computation frameworks (e.g., CORBA vs. JDK). Distributed computation frameworks may take advantage of system layer context by implementing their internal context abstraction atop SDI. Thus, applications will benefit from fast OS-level context transfer mechanisms, context caching, and configurable context translation across tiers.

An additional benefit of the design of context as a stand-alone OS service as opposed to an application-level abstraction is that it creates potential cross-layer synergies. For example, an application protocol that processes multi-media data may not care if the packets that it sends are received 100% error-free, since the data is not error-free to begin with.

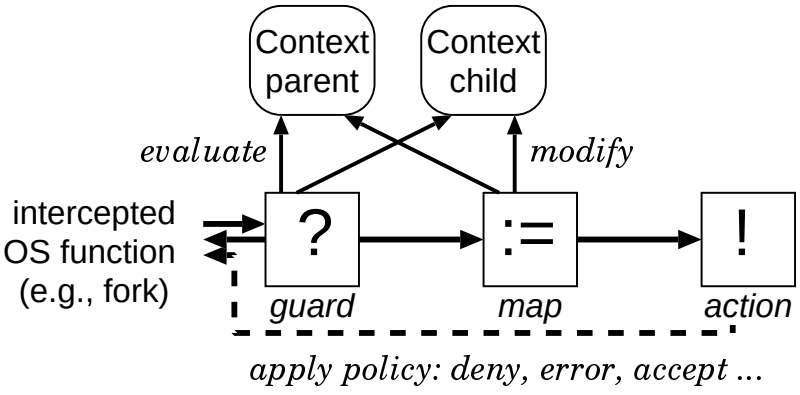


Figure 3.4: The structure of SDI rules.

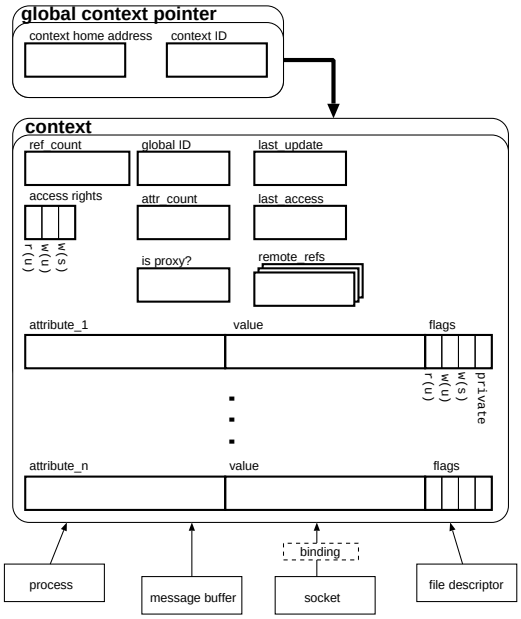


Figure 3.5: Context objects may be associated with multiple system objects (sockets, processes, etc.) possibly on different hosts via global context references.

To indicate this fact, the application could set the attribute `FastPath = 1` in its sending socket's context. At the same time, it would provide an SDI that instructs the `SEND` tap to copy the socket context to every message that passes through it. If the receiving host had an SDI at the `TCP_IN` tap to instruct the kernel to bypass error checking for messages whose context indicates `FastPath = 1`, the multimedia application would benefit from the realization of context-based efficiencies at the OS layer and experience less latency. One could not have achieved this objective easily if the `FastPath` attribute were implemented at the application level. The `FastPath` example can be expanded to implement Active Messages [149].

3.4 Stateful Distributed Interposition Concepts

3.4.1 What is Context?

Context is essentially an aggregate of attributes, which can be associated with system objects to add additional state. Context attribute names and values may be arbitrary bit strings, which can be created in, and deleted dynamically from, context objects. They are used by OS extensions and applications to save additional state. For example, a system security extension that filters network packets sent by unprivileged users must associate an additional security clearance with each process. To this end, it associates a context object with the process which specifies the appropriate security attribute.

Since the benefit of context depends on being able to take advantage of its contents, uniform attribute names are needed. For example, attributes such as user IDs and security clearance need well-defined names. The naming problem mirrors the attribute naming problem for POSIX environment variables and will most likely be resolved in the same

manner, i.e., by naming conventions. For example, every UNIX application interprets the value of the `HOME` environment variable as the current process' home directory. Similarly, a context attribute with name `HOME` should be interpreted consistently across applications. The naming problem is really a social problem and, therefore, outside the scope of this thesis. It could be resolved by using the same naming convention that SNMP uses to name its monitoring variables.

As front-end services interact with back-end services that execute within a different protection domain, their context must propagate (Figure 3.2) across protection domains. Context propagation crosses two important boundaries: system layer and host boundaries. A system layer boundary is encountered, for example, whenever a request is passed from the socket layer to the application. The attributes that may be applicable to the in-kernel processing activity with respect to the received message may or may not apply to the service that receives the request. Therefore, it is necessary to provide a rule framework that allows system administrators to specify in generic terms what should happen to context as an activity crosses layer and host boundaries.

3.4.2 Addressing Context

In order to be able to identify a specific context object in rules for context tracking and attribute value queries, it is important for applications and OS extensions to be able to correctly identify which context they are referring to. Since context acts as a state extension for existing system objects, it is primarily addressed relative to the system abstraction whose state it extends. For example, during IP processing one refers to the `DEADLINE` attribute of an incoming message as `[msg DEADLINE]`. Such relative references give interpositions and applications a more manageable local scope of context information. They need not process numerous kernel tables and descriptors to find and store relevant state. Eventually, one inevitably needs to address context objects by absolute, global context pointers (Figure 3.5):

- To resolve relative references to actual locations in memory (possibly locations at remote hosts) — required inside the context management subsystem itself.
- To group multiple system objects within one shared context — required for manual assignment of system objects to specific context objects.
- To utilize template contexts — these are abstract context objects that are not associated with any system object, and therefore, cannot be addressed using relative addressing.

Template context is important when individual per-request context objects only differ minutely (e.g., by a sequence number). The template from which an individual context object is created must be addressed using a global context pointer, since it is not associated with any particular system object — it is a template.

The global addressing mode requirements for context objects clearly distinguish SDI context from local POSIX environments, which can only be evaluated relative to the current process. In contrast, context can be associated with communication messages that travel across the network. Therefore, context must be implemented as a network-addressable or distributed object, to allow the receiving host to access the context that is associated with an incoming message.

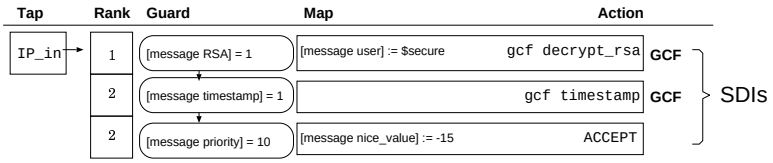


Figure 3.6: The structure of SDIs: a context-dependent guard triggers attribute value remapping, context rebinding, interpositions, and policies.

Distributed objects can be referred to directly [23, 102], via broadcast [31], or by name via a name service [100]. Name-based references provide the highest degree of flexibility in assigning and migrating (context) objects to arbitrary locations in a server farm. However, this flexibility is achieved by indirection, which does not scale to providing context for environments like HTTP servers, where context objects may be created at a rate of thousands of objects per second. In such scenarios, a name service would quickly become the system bottleneck. Obviously, broadcast-based context resolution causes similar scalability problems. Therefore, direct addressing, i.e., host-memory tuple addressing, is the only global reference method that scales to typical multi-tier service deployments.

3.4.3 Stateful Distributed Interposition

SDIs (Figure 3.6) are rules for the modification of contextual state based on previous contextual state and the intercepted interactions of computations with and within the OS. First, they provide a general mechanism to update context attributes and bindings based on the OS operations that multi-tiered computations invoke and their prior context classifications. Second, SDIs allow the invocation of interpositions depending on intercepted contexts' attributes. These features will greatly simplify the implementation of prior stateful distributed mechanisms like Active Messages [149], SPKI [49], Virtual Services [118], and future system enhancements for multi-tier systems.

The intended use of SDI for interposition-based policing, monitoring, and management of multi-tier services directly suggests the guarded clause structure of SDIs (see Figure 3.6) and leads to the SDI μ-language of Figure 3.7. First, as shown in Figure 3.7, SDI clauses are applicable to a specific system interface, i.e., a tap. Second, interpositions execute sequentially in rank order because each interposition may have side-effects that may interfere with, or build on other interpositions. Third, specific contexts and attributes may not be usable at all hosts, and therefore, must be mapped to meaningful values before they are used. Moreover, intercepting specific activities within a previously-inferred context may lead to its modification or replacement with a different, more applicable context. Such functionality is implemented in the *map* clause of an SDI. Finally, one may specify policing activities and user-defined interpositions.

The implied interposition programming model of SDI is similar to the programming models of Erlang [8] and Linda [56]. This similarity is not surprising, since SDI, Erlang, and Linda are all designed to tackle distributed programming problems in an extensible manner.

3.4.4 SDI Elements in Detail

Tap Points, part of every SDI, specify where in the kernel a rule is applicable. Tap

⟨*SDI*⟩ ::= ⟨*tap*⟩ ⟨*rank*⟩ ⟨*guard*⟩ ':' ⟨*map*⟩ ':' ⟨*action*⟩

⟨*guard*⟩ ::= ⟨*condlist*⟩
| ϵ

⟨*tap*⟩ ::= `IP-in`
| `IP-out`
| `IP-to-TCP`
| `IP-to-UDP`
| `...`
| `pproc_creat`
| `proc_creatt`
| `...`
| `ssend`
| `sendd`

⟨*rank*⟩ ::= ⟨*number*⟩

⟨*condlist*⟩ ::= ⟨*condition*⟩
| ⟨*condition*⟩ '∧' ⟨*condlist*⟩

⟨*condition*⟩ ::= ⟨*lloc*⟩ `'='` ⟨*rloc*⟩
| ⟨*lloc*⟩ `'>'` ⟨*rloc*⟩
| ⟨*lloc*⟩ `'<'` ⟨*rloc*⟩
| ⟨*lloc*⟩ '!=' ⟨*rloc*⟩

⟨*lloc*⟩ ::= `'['` ⟨*ctxt-holder*⟩ ⟨*attr*⟩ `']'`

⟨*rloc*⟩ ::= ⟨*lloc*⟩
| ⟨*number*⟩
| ⟨*string*⟩
| `'['` ⟨*ctxt-ptr*⟩ ⟨*attr*⟩ `']'`

⟨*ctxt-holder*⟩ ::= `process`
| `child`
| `parent`
| `message`
| `...`

⟨*attr*⟩ ::= ⟨*string*⟩

⟨*ctxt-ptr*⟩ ::= '(' ⟨*host-ip*⟩ ⟨*number*⟩ ')'

⟨*map*⟩ ::= ⟨*ctxt-assign*⟩
| ⟨*ctxt-assign*⟩ ',' ⟨*map*⟩
| ⟨*loc-assigns*⟩
| ϵ

⟨*ctxt-assign*⟩ ::= ⟨*ctxt-holder*⟩ ':=' ⟨*obj*⟩

⟨*obj*⟩ ::= `dup` ⟨*ctxt-holder*⟩
| ⟨*ctxt-holder*⟩
| `new`
| ⟨*ctxt-ptr*⟩
| `dup` ⟨*ctxt-ptr*⟩

⟨*loc-assigns*⟩ ::= ⟨*loc-assign*⟩
| ⟨*loc-assign*⟩ ',' ⟨*loc-assigs*⟩

⟨*loc-assign*⟩ ::= ⟨*lloc*⟩ ':=' ⟨*rloc*⟩
| ⟨*lloc*⟩ '+=' ⟨*number*⟩
| ⟨*lloc*⟩ '-=' ⟨*number*⟩

⟨*action*⟩ ::= `dc`
| `accept`
| `deny`
| `deny` ⟨*errno*⟩
| `shape` ⟨*cnt*⟩ ⟨*interval*⟩
| `shape` ⟨*cnt*⟩ ⟨*interval*⟩ ⟨*errno*⟩
| `smooth` ⟨*cnt*⟩ ⟨*interval*⟩
| `error`
| `error` ⟨*errno*⟩
| `gcf` ⟨*fct-ptr*⟩ '{' ⟨*string*⟩ '}'

⟨*errno*⟩ ::= ⟨*integer*⟩

⟨*cnt*⟩ ::= ⟨*number*⟩

⟨*interval*⟩ ::= ⟨*number*⟩

⟨*number*⟩ ::= ⟨*unsigned-long*⟩

Figure 3.7: The SDI grammar: duplicate first and last letters of a system call name specify interception taps before and after the execution of the default system action, respectively. The word ϵ is the empty word. A completely empty guard always evaluates true. *Context holders*, e.g., `socket`, always refer to the canonical object at the tap. For example, `socket` would refer to the sending socket in a SDI that is interposed on `send`.

points interpret the specified high-level actions, context attribute references, and conditional statements with respect to the system objects found at the tap point and the executed system behavior. They are part of an SDI-enabled system and need to be anchored in the OS.

The high degree of OS-dependence in each tap point raises the question of whether their implementation is too difficult, thus rendering SDI impractical for real world OSs. Fortunately, this is not the case because all tap-point implementations follow a generic implementation skeleton. In fact, the only tap point-specific functionality is to identify the intercepted system objects, which is a straightforward exercise, and to interpret action codes. The actual interpretation of SDIs takes place in the tap point implementations, but it is largely tap-point-independent. It is carried out by generic guard checkers and generic context mapping operators. Tap-point implementations are merely a "glue layer" adapting

a generic SDI interpreter to the specifics of an intercepted interface.

Context-dependent *guards* determine when to apply or skip an SDI, thus making interposition context-dependent. Guards are conjunctions of atomic conditions, which are evaluated at tap points relative to the contexts of the intercepted system objects. The SDI:

```
SSEND_TCP 1 [proc clearance] < 5 : : gcf check_send_perm.
```

is one of the first SDIs to be evaluated whenever an application attempts to send a message through a TCP socket. More specifically, it is executed right before the send functionality executes (tap SSEND_TCP). The example SDI restricts the custom interposition (check_send_perm) to execute only if the calling process' security clearance, a runtime-specified attribute, is below 5. This selective rule application mechanism distinguishes SDI from prior interposition approaches [21, 58, 78], which provide little control over the conditions under which a specific interposition should be invoked.

The second key element of the SDI grammar is attribute value *remapping* and the *rebinding* of context. As mentioned earlier, this is necessary because of the potentially heterogenous, multi-tiered computing infrastructure in which SDI-based system enhancements may be deployed. For example, the priority levels assigned on a front-end machine may have to be remapped to different values on the back-end machines [6]. Another typical example would be the remapping of user IDs from a web server environment to user IDs known to a back-end DBMS, as is done in application servers.

Value remappings permit assignment and the `+=`, `-=` operators. The reasons for including these operators in the grammar are that they are atomic and that assignment and counters are frequently used in system management tasks (e.g., in counting the number of packets sent, assigning user IDs, etc.). More generic arithmetic expressions which could have subsumed those operations were not used because each atom of a non-atomic arithmetic expression could be evaluated at different times due to the distributed nature of context. This means that some atoms' values could change during evaluation, thus generating phantom results that do not reflect the system state at any time. Furthermore, complex expressions could encourage users of SDI to specify expressions that look concise but are difficult to evaluate. Should some SDI-based solution require complex arithmetic expressions, they can be implemented within plug-in `gcf` actions.

The terminal component of an SDI is its *action*. Standard actions that are part of the SDI language are designed to simplify the policing of intercepted computations. The four parameter-free actions, `dc`, `accept`, `deny`, `error`, are easily understood. The `dc` (don't care) action simply states that the tap implementation should continue checking other registered guards. An `accept` policy indicates that the tap point implementation should terminate further guard checks and continue by executing the functionality upon which the tap is interposed. When a `deny` action is encountered, the tap point must interrupt the propagation of work immediately. For example, if an incoming packet triggers a `deny` action, the packet is simply dropped. Among other things, `deny` can be used to implement load-shedding under overload, to construct security policies, and to confine untrusted applications. Obviously, tap points have to be crafted carefully to interpret `deny` in a manner that is compatible with the intercepted OS behavior.

The `deny` action lacks the ability to log the action that was denied. This feature is provided by the `error` action. In addition to a plain `deny`, `error` also logs tap-specific information about the intercepted computation, including the intercepted context values

that were matched in the guard, the SDI itself, and potentially information about the intercepted system objects.

The parameterized versions of `deny` and `error` deal with the problem that a denied system call cannot simply die but must report to the caller of the failed call. Since applications may only be capable of interpreting certain error codes, the system administrator may explicitly specify an error that is understood by the applications. If the error code remains unspecified, the `deny` and `error` actions will return a general failure (e.g., `EINVAL` on UNIX).

The action codes defined so far only permit static (admit, deny) system control policies. A large class of performance-related system management approaches, such as load control and load sampling, require rate-based policies. To this end, we define the actions `shape` and `smooth`. First, the `shape` action oscillates between `dc` and `deny`. The `shape` action will return `dc` as long as it is matched at a rate below the specified upper bound. If the bound is exceeded, it behaves like `deny`, either failing with `EINVAL` or a system administrator-specified error code. Second, the `smooth` action differs slightly from `shape` in that it does not return error when its rate limit is exceeded but defers the current interaction until it is eligible to return `dc`. Each tap point may limit the number of deferred actions. Once this limit is reached, `smooth` behaves exactly like `shape`. Combining guards and the `accept`, `deny`, `shape`, and `smooth` actions makes sophisticated, class-based admission control schemes without much programming effort possible. For example, the SDI:

```
NET_TO_IP 1 [msg svc-cl] = 2, [msg type] = NEW_CONN_REQ : : \
            shape 2 10000.
```

would shape incoming connection request packets of service class 2 to a rate of 2 per 10 milliseconds.

Finally, the `gcf` action achieves unrestricted extensibility for the SDI framework. Whenever the generic action codes are not powerful enough to force multi-tier services to behave in a certain manner, generic interposition code may be interposed. For example, if the context of a sender carries a signature flag, its communication with other services should pass through a digital signature layer. Such a complex operation cannot be accomplished without special interposition code. However, the signing function may be generic and applicable at multiple tap points, e.g., network in, pipe write, and file write. The directives

```
HOST_A ->  IP_TO_NET 1 [msg sign] = yes  : [msg  sign] := x, \
                       [sock sign] := 1 : gcf sign { key_x }

HOST_B ->  NET_TO_IP 1 [msg  sign] = x : : gcf check_sig { key_x }
```

may be used to check signatures of packets that are exchanged between hosts. The sample GCFs `sign` and `check_sig` would have access to the same system objects that are available to the tap point implementations (`IP_TO_NET` and `NET_TO_IP`). Their return value may indicate that the message under consideration should be processed further or that it should be discarded immediately.

Since GCFs may require their own private data for each instantiation, e.g., encryption requires key management information, GCF-specific configuration data, specified between

the parentheses of a GCF action specifier, is passed to GCFs every time they are invoked from an SDI rule.

3.4.5 Initial Context Creation and Association

One difficulty within SDI is the creation of context for incoming requests that are not already associated with a context. As the grammar shows, the guards operate on contextual state that is associated with the message, not on the state contained inside of it. Since requests that are received from clients in the Internet are not associated with context, they cannot be matched by any guard — except the empty guard. This means that the classification of incoming requests would have to be implemented within GCFs that execute for every incoming packet. This is somewhat clumsy and inefficient. Classifiers which operate on intercepted network packet state and not contextual state solve this problem more elegantly.

Classifiers extract and interpret intercepted packet information in accordance with system administrator-specified context binding and creation directives. For example, a system administrator may direct that all requests coming from address 10.*.*.* must be associated with a duplicate of the template context representing intranet clients. The state created by a classifier is always associated with the intercepted packet. The created context object can, of course, be modified and replaced as the computation spawned by an incoming request progresses.

Classifiers are typically part of the lowest levels of the OS's networking stack, in order to ensure that all OS layers (including SDI) can rely on some preliminary context being already associated with an incoming message. A minimal classifier evaluates an incoming packet's source address, destination address, protocol, and destination port. Based on these it associates the incoming packet with an existing context or creates a new context object for it. More sophisticated classifiers may scan incoming messages for more information, for example, for the URL contained in an HTTP GET request. Since there is no conceptual limit as to what information classifiers may access to determine an incoming message's context, we provide only a schematic grammar for classifiers below. The details depend on specific classifier implementations (e.g., `IP_in`, `open`, `exec` classifiers).

⟨*CLASSIFIER*⟩ ::= ⟨*matches*⟩ ⟶ ⟨*map*⟩

⟨*matches*⟩ ::= ⟨*match*⟩, ⟨*matches*⟩
| ⟨*match*⟩

⟨*match*⟩ ::= ⟨*packet-property-name*⟩ ⟨*operator*⟩ ⟨*value*⟩

⟨*map*⟩ ::= `new`
| `dup` ⟨*context-ref*⟩
| ⟨*context-ref*⟩

⟨*operator*⟩ ::= `'<'`
| `==`
| `...`

⟨*value*⟩ ::= ⟨*string*⟩
| ⟨*integer*⟩
| ⟨*regexp*⟩

⟨*packet-property-name*⟩ ::= `source IP`
| `source port`
| `...`
| `TCP data`

The "packet-property" in the above grammar is used to capture protocol attributes that are specified inside a network packet, such as source address, presence of specific bit patterns, and the like. For example, the classifier
`SOURCE = 10.0.1.0/24` $\longrightarrow$ `CTXT_SVC_CLASS_1`
binds incoming packets from 10.0.1.* to a context that identifies service class 1. `Source` would be considered a packet property. The purpose of intercepting packet properties is to record the state that is expressed in network protocols, and therefore, only visible between client and server, inside a context that will be tracked as the work spawned by the request propagates across multiple tiers. Regular expression matching obviously only makes sense in combination with the "==" operator.

Oftentimes, one will force a default classification for an incoming request simply to remember some key request attributes or to execute it within an applicable default context. Upon receiving a user ID, a password, or other request-specific markers, an application process may modify this default context to better reflect a request's personality. For example, assuming the above example classification rule is specified for a server, one may also specify the following SDIs:

```
ACCEPTT 1 : proc := msg : dc.
CCONNECT 1 [proc svc-cl] = 1 : proc := MTIER_OP1 ; \
                               msg := MTIER_OP1 : dc.
```

The above SDIs instruct the server to bind the receiving process to the received message's context, which is derived by the classification specified earlier in this section. Furthermore, if the process connects to a back end, it will be labeled as a multi-tier process, thus implementing multi-step classification.

Other typical classification operations can be inserted at other locations in the kernel. For example, it is possible to associate specific process IDs, binaries, or files with a context object. These classification operations require simple additions to the kernel (explicit tagging of processes) or tables that map system call attributes to context objects for the calling process and the other intercepted system objects. For example, a classifier at `open` is configured to match a filename and possibly the open mode to a context reference for both the returned file descriptor and the calling process.

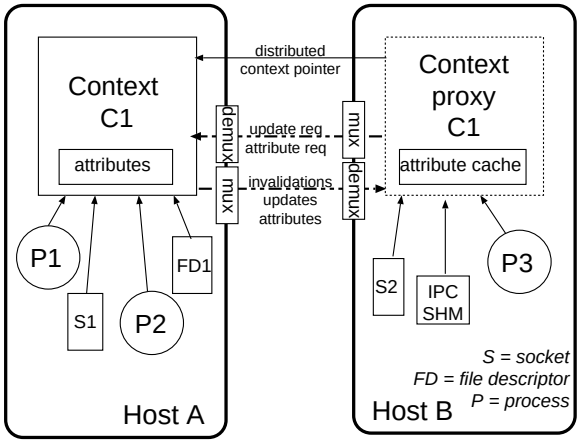


Figure 3.8: The relationship between primary context, its proxies, and references from system objects

3.4.6 Handling Distributed Context Efficiently

Since the context attributes that SDIs refer to may be located remotely, repeated access to the same context can incur high latency penalties. The standard remedy for this problem is caching, which is already addressed by a rich body of work in distributed and multi-processor shared memories [48, 88]. In line with these earlier solutions — especially [23] — distributed access to context attributes proceeds through a caching proxy object (see Figure 3.8).

The second problem is that it is difficult to determine when it is safe to discard a context object, which affects memory efficiency. Various garbage collection mechanisms [109] have been introduced to address this problem effectively. In our case, the garbage collection problem reduces to a two-level reference counting scheme. First, one must determine whether a context or a proxy is still needed locally. Second, for each primary context object, i.e., the originally-created context object, one must count how many proxies refer to it. Proxies are removed when there are no local references to them. Upon removal of a proxy, the corresponding primary context object is informed. This simple and frequently-used reference counting-based garbage collection scheme has one major problem: front-end servers typically access back ends repeatedly without keeping back ends completely busy. Thus, back-end servers would frequently discard and reinstate proxies.

To illustrate the caching problem caused by repeated back-end accesses, suppose that a certain percentage of HTTP requests require access to a back-end database. Whenever the database is idle, it will expunge all cached context object proxies that it may have, thus incurring a remote context access penalty for the next HTTP request that requires the database. To account for this access behavior, it is necessary to delay proxy removal by a configurable amount of time.

The third inefficiency is memory leakage due to host failures. In large multi-tier server networks, it is likely that a host that is still holding proxies to remote context fails or is brought down for maintenance. The problem of a back-end server becoming unresponsive without releasing its context references is addressed by using heartbeats, i.e., each host periodically announces its liveness to those hosts that hold primary context objects for its proxies. Communication messages between hosts act as implicit heartbeats. This requires the primary context object to record which remote proxies are referring to it (see Figure 3.5). Upon detecting a failed host, the context management subsystem reduces the reference count for each primary context accordingly. The reverse

problem, the failure of the host that hosts a primary context, is discussed as a failure mode in Section 3.5.6.

Finally, one must also allow for context to be exempted from garbage collection. Otherwise, it would be impossible to set up persistent, abstract template context objects.

3.4.7 Context Security

As in all OS mechanisms, security is an important concern. Context is a shared network object and may be used in critical system management tasks, such as user identification and scheduling operations. Up to this point, we have not discussed any mechanism that would prevent applications from creating or modifying critical context attributes that are used by the OS, e.g., a security-clearance attribute. It may also be necessary to prevent a host that masquerades as a primary context host from delivering bogus context attributes to back-end servers. The mechanisms introduced here focus on controlling context access on the local hosts, while leaving the integrity of the communiction links up to link or IP-layer security protocols.

To assure local access integrity, attribute access is controlled on a per-attribute basis since some attributes may contain important system information, while others may be informational, user-defined attributes. Two security principals are distinguished: the kernel, which is believed to be uncompromised, and potentially faulty applications. Attribute operations are controlled with respect to "read," "write," "add," and "delete." To prevent faulty applications from binding to an existing, potentially more privileged context, context binding is also controlled by binding permissions. Applications are prohibited from binding those context objects that are specified to be bound only by the kernel or super-user. This security scheme is analog to the UNIX filesystem. Each attribute should be considered the counterpart of a file in the filesystem. The context object itself is the counterpart of a directory.

Specifying access rights for each context and attribute is a tedious task, and sometimes applications and system administrators may forget to specify appropriate access permissions. Therefore, the default is to enforce the most restrictive access policy for each context and attribute: application-created context and attributes can be modified by their creator or the superuser, and kernel-created context attributes, including those created by classifiers and SDIs, can only be accessed by the kernel. An exception to this default is made when an attribute name is found in a system level attribute-name-to-permission-map, which specifies the default access permissions for a specific attribute name.

The above mechanism is secure as long as the root, superuser, or administrator accounts of the individual hosts have not been compromised. If the root account on any host has been compromised, then an intruder may tamper with context. The values that the compromised host supplies to its applications or other hosts are no longer trustable. However, the ability to tamper with context attributes gives intruders little additional power compared to a system without the proposed context abstraction as they can replace any service on the compromised host with hacked versions, and take over any IP address in a server cluster, thus bringing the cluster to a halt. It seems reasonable to expect that the hosts within one cluster are secure against such attacks and that the power of the SDI framework will give an intruder negligible additional power.

Another concern at the host level is the installation of SDI rules into the kernels. Since SDIs modify kernel behavior, we adopt the usual security policy for making kernel modifications. This means that super-user privileges are required for the installation of SDIs on any individual host. For more centralized control one can easily build a daemon process that carries `root`'s user ID and acts as a proxy for an accepted remote administrator. Control over who can install SDIs is very important because SDIs are essentially miniature kernel modules. Like faulty kernel modules,

faulty SDIs can cause a host to fail and are therefore kept out of the normal user's reach.

If an attacker gains access to a data center's network infrastructure, the attacker gains the ability to fabricate messages and to snoop on message exchanges. In particular, it becomes possible to tag messages with context references that illegitimately increase a message's privileges. Second, the attacker can disrupt the exchange of context information between hosts by fabricating false replies to client queries and by invoking operations on remote context objects.

The threat posed by illegal network access is potentially large. Instead of addressing this problem by our own security scheme, it can and should be addressed by using IPSec [79], which ensures network authenticity and privacy regardless of the message traffic. A system administrator who assumes that hosts may illegally connect to a multi-tiered system, should not only be concerned about protecting context but also about all other traffic between applications, which would be subject to tampering.

3.4.8 Context at the Application Level

Different context abstractions have been invented for distributed application frameworks, such as CORBA [102], J2EE [139], and WebSphere [70]. Each of these frameworks for distributed application development creates some form of context that can be passed as an optional parameter with an RPC. The problems of their context notions are that they do not integrate information relevant to other frameworks or OSs. Since they are application-oriented, they hide contextual state that could be useful in the OS. Moreover, the context abstractions of the different frameworks waste communication resources since context is transferred by value between hosts, regardless of whether it is used or not. System-level context is exported to the applications, to improve the efficiency of application-level context implementations and to promote the integration of application- and system-level context to realize synergies among kernel and applications.

Applications can refer to their processes' context in very much the same way as they refer to environment variables. If applications must access the context objects associated with file descriptors and IPC abstractions within their address space, they simply refer to the context via a specific system object key.

The default propagation of a process's context alongside its communication, i.e., from a process to a message, from the message to a network packet and back, generates a miniature version of a distributed thread, thus satisfying the requirements of many simple sequential applications, e.g., the propagation of user IDs. More complex applications that require remapping, e.g., if certain front-end users are mapped to different users on the back end, may instantiate SDI rules for this purpose. Instead of relying on SDI, applications could also manually remap context attributes before invoking other services.

Another reason for application-level context integration is that one cannot solve all context propagation problems without application cooperation. For example, if a particular service is implemented using an event-driven architecture or a user-level thread library, the OS cannot infer the right context-to-process-bindings automatically. The solution is to allow applications to specify explicitly which context object is currently applicable. To this end, the system allows processes to export a request ID, which is tracked by SDI like a process descriptor, i.e., it has its own request ID-to-context-binding. SDIs that refer to the context of a process will retrieve the context of a request ID if the application exports it. In order to avoid memory leaks, applications must add and delete request IDs explicitly. To guard against faulty applications, the kernel should only allow up to a maximal number of request IDs per process.

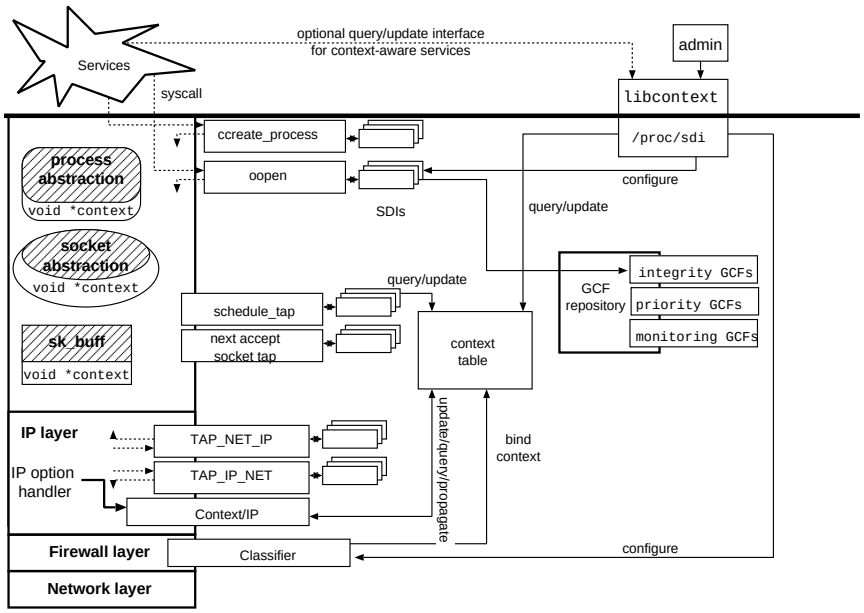


Figure 3.9: Detailed architecture of the Linux-based prototype

3.5 Implementation

We designed an SDI prototype for the Linux 2.2.14 kernel. A base module provides context objects and implements Context/IP, which is the communication protocol for remote context access (Figure 3.9). A second module provides classifiers and ensures that context associations propagate across hosts. Generic SDI parsing, including guard evaluation, attribute remapping, binding and SDI management functionality, is implemented in a third SDI kernel module. This SDI module is the basis for SDI administration and several tap point and GCF implementations. Tap points and GCFs are usually implemented as individual kernel modules, which are interposed at runtime using SDI.

When all of the prototyped modules are loaded, the kernel grows by a little more than 2 MB, most of which is allocated for the context index table. Each additional tap point, such as `TAP_NET_IP`, consumes approximately 2-5 KB to implement the required hooks and glue. GCFs which build on the tap point implementations require 2-3 KB. The size estimates for GCFs are only rough estimates, based on our experience with performance management and some security applications. Since there are no restrictions on what can be done in a tap point, GCFs may require arbitrary amounts of memory.

3.5.1 The Context Abstraction

Each context object contains a hash table of attribute-value pairs, a hash table of remote referrals, and a reference count for local references. It can be used as both primary and proxy. Proxy contexts mirror primary context objects and implement location-transparent context access for interpositions through its attribute access functions. Usually, the proxy mirrors the attributes contained in the primary context. To allow the system to work on small memories, proxy context objects may be evicted according to the LRU algorithm. Subsequent accesses to an evicted context will stall until its proxy is reloaded from the primary, while possibly evicting another context in the

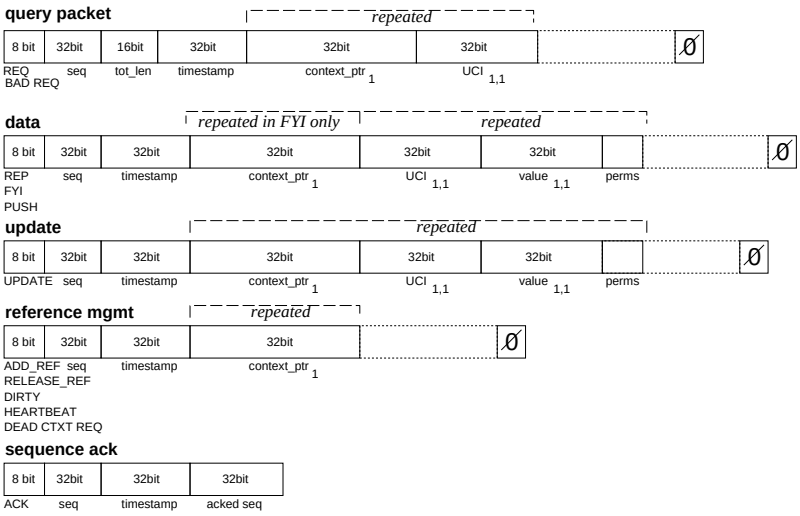


Figure 3.10: Context is maintained using pre-defined message formats. The message formats leave implementors enough freedom to deploy and experiment with caching, consistency guarantees, and various reference management strategies.

process. In practice, this feature may be unnecessary because today's servers have large memories, and the context objects are only minor consumers of host memory.

Operations on proxy context objects always propagate to the primary. On receipt of an update message, the primary simply invalidates all other proxy objects. The invalidated proxies will need to fetch the values from the primary on their next attribute access. Future implementations should also feature update propagation to improve context access latencies when several hosts are simultaneously processing each request. The invalidation-based approach performs better when context accesses are largely sequential.

We propose Context/IP for the execution of remote context operations and state transfer functionality. Context/IP's datagram formats are shown in Figure 3.10. The protocol is purposely designed in such a way that different proxy behaviors (e.g., update propagation vs. invalidation) are left runtime-configurable. This allows customization of the operation of SDI to the specifics of a particular installation. For the same reason, authentication, which may cause significant processing and communication overheads, is made an optional part of the message format.

Hosts usually answer attribute requests with reply packets that contain a complete context snapshot. There are two principal reasons for this design choice. First, context objects tend to contain only a few attributes, which can be transferred quickly. Second, if one attribute is accessed, it is likely that another attribute will be accessed soon. However, if a context does not fit into a single IP datagram or if the attribute request was a batch request for multiple attributes from different contexts, due to the protocol implementation's efforts to reduce communication overheads, the reply is sent via a FYI packet. FYI packets contain only the requested attributes (see Figure 3.10).

Disposal of context objects and sending of heartbeat messages are controlled by a periodic kernel thread which deletes context objects whose reference count has reached 0. If the context to be removed is a proxy context, it notifies the primary context object of the release of the proxy.

In addition to passive attribute manipulation and access functions, we implement *context-change triggers*. Context-change triggers actively initiate or wake up computations on context change. This is a useful implementation feature because it avoids having interpositions poll for context changes. For example, if one implements a resource quota mechanism using a context-

based counter, one must provide a mechanism to wake up computations when exhausted resource quotas are replenished. This is easily done using a context-change trigger on the resource quota attribute. Without the trigger, one would have to poll the resource quota frequently.

The attribute values provide only soft-state. Even though transactional semantics are relatively easy to implement [60], they would inevitably increase context access latency. Strict transactional semantics require distributed locks and multi-phase commit protocols, which require multiple network round trip times to complete. Any interposition utilizing consistent state would cause intercepted computations to slow down significantly. A soft-state approach gets around the latency problem without significantly limiting context usability. Soft-state is not problematic because most uses of context will only propagate attribute values along with distributed computations with few or no attribute updates during a context's lifetime. This assumption is backed by the usage patterns of environment variables and prior examples of distributed context in the literature.

The context abstraction must consider an important trade-off to allow many short-lived context objects and ones that are long-lived with numerous attributes to coexist. The ultimate goal is to provide index structures with minimal setup overheads *and* fast attribute lookup times. Fast lookup for potentially large context objects requires index structures. However, index structures typically require memory allocation and substantial initialization costs (see Section 3.7). In SDI, memory allocation overheads are reduced by using a LIFO queue of deallocated context objects. Instead of using the OS's memory management functions, disposed context objects are placed into this LIFO queue, which queues reusable context objects up to a certain administrator-specified memory limit. Most context object allocation requests can be satisfied from this queue without the need to run the kernel's slow generic memory allocator. LIFO allocation increases memory reference locality, thus boosting the hardware cache's efficiency.

The problem of possibly high initialization costs is addressed by *lazy initialization*, which amortizes context initialization costs over an extended time-frame. Lazy initialization works as follows. Instead of immediately initializing the hash indices for attributes and remote referrals in newly-created context objects, only doubly-linked lists of attributes and remote referrals are set up. The newly-created context object is marked semi-initialized and queued for later initialization by a deferred initialization kernel-thread. In the meantime, attribute access proceeds by traversing the linked attribute list. After 1s (our threshold for a long-lived context) the context is indexed by the deferred initialization thread and marked as completely initialized. Successive attribute accesses will proceed using the hash index. Thus, short-lived context objects incur only negligible setup overheads. Nevertheless, a long-lived context will be indexed eventually, thus eliminating the penalty of increased attribute access overheads for long-lived context. The optimizations for dynamic context creation increase tenfold the number of possible context creations per second compared to eager initialization.

The kernel needs to be modified in numerous places to accommodate additional state for system objects. All basic system abstractions, e.g., `sk_buff`s, processes, IPC message queues, sockets, and the like, are extended by a `void *` context pointer. Although it would have been possible to create an indirect, table-based association between system objects and their context, it is more efficient to use embedded context references. This modification does not make SDI any more invasive than it already is, since all system objects' destructors, which are part of the core kernel, must already be modified to decrease the reference counts of the context objects to which they refer. Fortunately, the required OS changes are small and readily implemented by experienced programmers.

3.5.2 Dynamic Context Creation and Propagation

The classification module manages the dynamic association of incoming IP packets with context references. The classifier hooks into Linux's firewalling layer and intercepts packets before they enter the incoming IP stack and, thus, before any interposition executes.

The classifier's mode of operation resembles a typical firewall [24]. The only difference between the implemented classifier and an IP firewall is that the classifier associates a context object with the intercepted `sk_buff` instead of policing it.

Classification rules match the protocol ID, source address, source port, destination address, and destination port of an incoming packet against user-specified classification rules. For each match, it is possible to specify whether the intercepted `sk_buff` should be associated with an existing context (via a context pointer) or a new context should be created for it.

The following command installs an example classification rule, which causes a matching incoming packet from 10.0.0.0/255.255.255.0 destined for TCP port 80 to be bound to a duplicate of context 2 on host 10.0.0.2.

```
sdi-classifier -p TCP --syn --dp 80 --sa 10.0.0.0 --sam 255.255.255.0 \
               --home 10.0.0.2 --id 2 --dup
```

The IP-based classification should not be considered final, since each classified message object may have its context subsequently altered by numerous SDIs. For example, assume a special HTTP interceptor had been implemented, which can be interposed at the socket or TCP receive message taps. This interceptor is configured with a string to be matched in the incoming request and a resulting classification. For example, the SDI

```
TTCP_RECVMSG [MSG SVCTYPE] = HTTP-INTRANET : :
         gcf HTTP_INTCPT { "/research/", 10.0.0.2, HTTP-RESEARCH}
```

could force the context binding of a message that was previously bound to a context of service type `HTTP-INTRANET` by the classifier, to be refined to context `HTTP-RESEARCH` if research data is being accessed. Furthermore, context-aware user applications may inquire of context bindings and remap a request's context or attributes within a request's context upon verifying an application-level password, secret, or the like.

Context propagation proceeds as follows. Whenever an `sk_buff` with a context association arrives at the outgoing IP layer, the context association is written into a new ContextRef IP option (see Figure 3.11) in the `sk_buff`'s data area. Thus, the receiving host can reconstruct the association between the incoming packet and its context.

Whenever a receiver obtains a context reference with an object that it cannot resolve locally, context is retrieved remotely via Context/IP. Since remote context access is a relatively slow operation (100 μs for Fast-Ethernet), an incoming packet can be deferred for some time before being processed. To ensure fast processing of incoming packets, a packet's context is pushed ahead of it, unless the sender knows that the receiver has already established a proxy for the message's context (see Figure 3.11). Thus, incoming datagrams can be processed without interruptions.

One may wonder why this two-pronged approach of an additional IP option and the new Context/IP protocol is chosen over an additional wrapper layer between IP and UDP or TCP. The disadvantage of introducing a wrapper layer is that smart network infrastructure, such as load-balancing switches and firewalls, could no longer be used. These devices peek directly into the packet content beyond the IP header to make decisions about packet forwarding and policing. Since smart network devices are basic building blocks of high-performance multi-tier server installations — with or without the endorsement of end-to-end argument advocates — their smooth

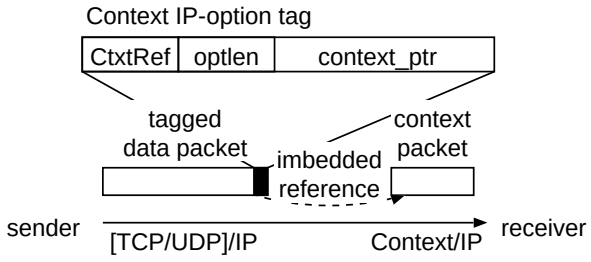


Figure 3.11: Senders may push context ahead of data packets to initialize a proxy before packet receipt.

operation should not be disturbed by the propagation of context alongside messages. The proposed combination of Context/IP and the ContextRef IP option is friendly to [TCP,UDP]/IP and layer-3+ load-balancing switches. The only potential problem with load-balancing switches and Context/IP is that the push mechanism will not work correctly unless the load-balancer's firmware is updated to send the Context/IP packet preceding a message with a context reference to the same host as the context-tagged packet. Without such an update, the back end may sometimes need to retrieve the context from a front-end server before servicing an incoming request, thus incurring a 100 - 200 μs context access penalty (Fast-Ethernet).

The second advantage of our two-pronged approach to context propagation is that hosts and routers that are unable to participate in the Context/IP protocol are not disturbed. According to the IP specification [110], routers and hosts simply ignore unknown IP options.

Many researchers categorically reject the use of IP options for any purpose because the presence of IP options causes packets to be forwarded over the slow path in today's standard routers. While this argument is valid for today's routers, it should not prevent us from advancing network infrastructure for server farm environments since it is possible to update router OSs for server farm deployments to process (or ignore) the ContextRef IP option on the fast path. Moreover, server area networks are often switched, not routed, networks. Therefore, the presence of an IP option will have only negligible impact on packet forwarding times.

3.5.3 Tapping the OS's Control Flow

Before evaluating a guard for a registered SDI, the tap point first attempts to resolve the context references of the intercepted objects (see Figure 3.12). If this fails, the tap records its current state in a continuation structure [47], requests the non-local attributes, and defers its execution until the attributes arrive from the primary context or the operation fails. In the case of `TAP_NET_IP` (see Figure 3.12), the continuation stores the intercepted `sk_buff`, the SDI under consideration, and the index of the guard condition or substitution at which execution stalled. The tap point returns control to the standard network interrupt handler, indicating that it will process the intercepted packet later. Evaluation of the SDI continues when the requested attributes or contexts arrive.

The structure of system call interpositions is much simpler than that of the depicted interrupt interposition in `TAP_NET_IP`, since they execute within a full-fledged thread abstraction. Therefore, deferral can be implemented by putting the current thread to sleep until all necessary attributes for guard evaluation and attribute value remapping are available.

Typical tap points are placed at layer transitions in the control flow of multi-tier services. Tap points must intercept all communication activity (`send, recv, read, write`) and the

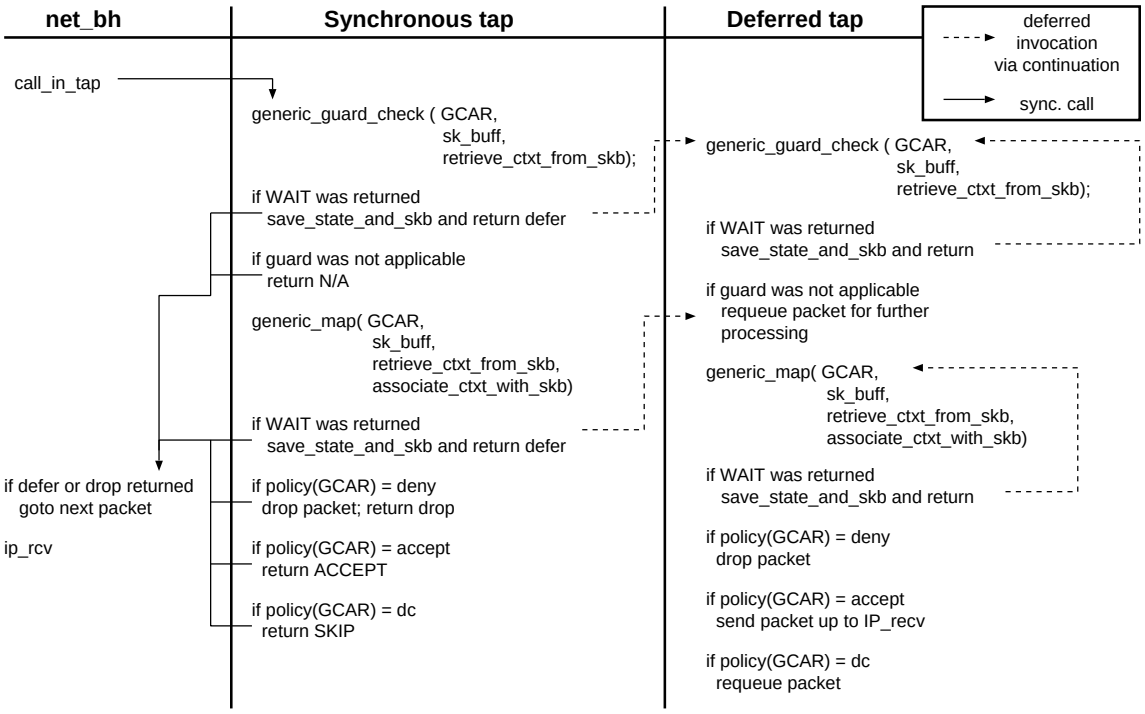


Figure 3.12: A sketch of the prototype's tap at the in-bound IP interface. The tap links into the Linux firewall `call_in` chain. Most other taps are of a similar structure.

creation of new system objects (`fork, socket, open`). Additional tap points are optional but can be very helpful in system management tasks.

Figure 3.13 shows an architectural overview of typical tap point implementations. The tap point module registers itself with interception hooks inside the kernel, which call the tap point implementation every time the hook is reached in the OS's control flow. The tap point implementation then uses generic functions to check the registered SDIs ($\mathrm{SDI}_1 - \mathrm{SDI}_\lambda$). For each SDI, it first evaluates its guard conditions in linear order. This check is done by generic functions. Then the tap point implementation applies the substitutions or assignments of the mapping clause in sequential order. Finally, it applies the action. The action interpretations are actually part of the tap point implementation, as is shown in the tap point data structure of Figure 3.13. If an applicable SDI specifies a GCF, the tap point implementation calls the GCF and interprets its return value as an action specifier.

GCFs are programmed replacements for standard actions (see grammar in Figure 3.7). If a tap point encounters a GCF directive, it transfers control to the GCF by calling it directly. The return code of a GCF may specify an action, which is then interpreted by the calling tap point, or it may indicate that the GCF has taken charge of the intercepted computation. GCFs are invoked with the same arguments that are supplied to the tap point for which they are registered and to an additional tap point ID. Hence, it is important to specify at which tap points each GCF can be registered. This is expressed in a GCF descriptor structure, which is read when a GCF module is inserted into the kernel (Figure 3.13 bottom right).

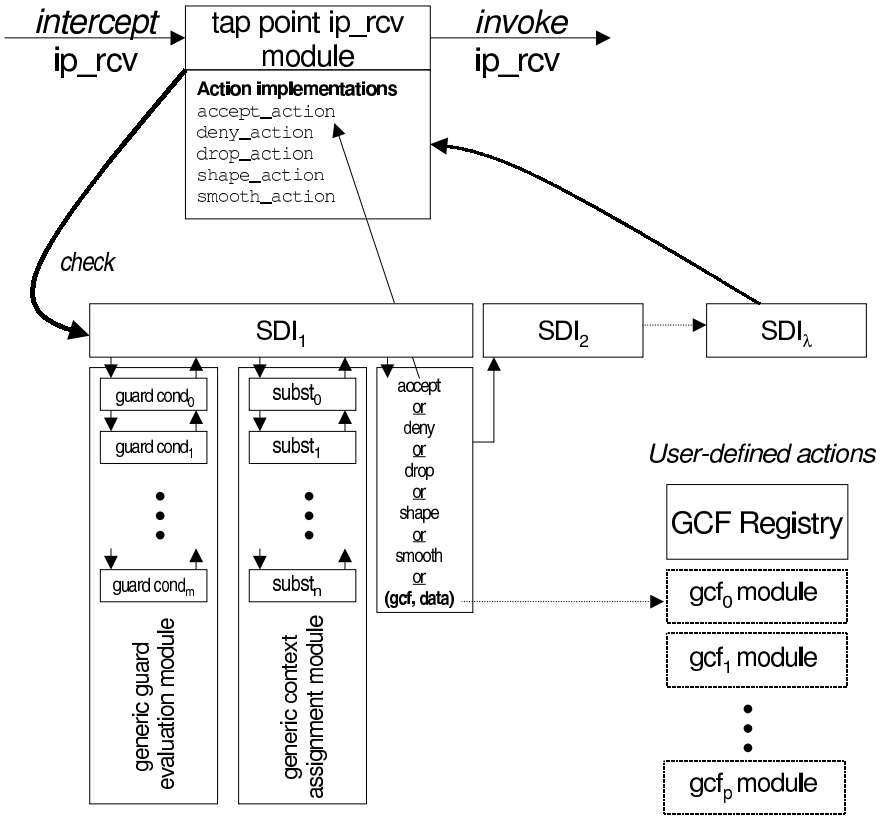


Figure 3.13: Modular structure of taps, SDIs, and GCFs in the current prototype.

3.5.4 The Command Line Interface

The command line interface provides commands to create (`ctxt_create`), manipulate (`ctxt_attr_add, ctxt_attr_set, ctxt_attr_del`), and delete persistent context objects (`ctxt_remove`). The association of context with an incoming message is controlled by the `sdi-classifier` command, which binds an incoming packet to a context object based on its [TCP,UDP]/IP properties. Context bindings are further manipulated by SDIs. These SDIs are specified according to the grammar of Figure 3.7 and passed to a parser (`sdi-config`), which translates the expressions into data structures that can be checked efficiently by the TAP point implementations.

As the SDI grammar of Figure 3.7 shows, GCFs may accept arbitrary additional arguments, which `sdi-config` cannot interpret on its own. To check the arguments passed to a GCF, `sdi-config` supports GCF-specific DLL plugins that parse the argument list passed to a GCF and translate it into a GCF-specific data structure. This plug-in for `sdi-config` and the GCF implementation itself are to be provided as a unit.

To ensure that context propagates properly across system layers, applications, and network connections, the system administrator must specify appropriate SDIs. SDIs for specific tap points can only be submitted after the specific tap point kernel module is loaded using Linux's `modprobe`.

In order to feed back information from the installed SDIs to the administrator, the `error` action is redirected to the user-level via the proc file system. A generic fault-handling daemon reads error messages from `/proc/sdi` and invokes error-handling functions that catch specific

error codes. The error handler is read from a DLL, which is specified in a special error code to DLL mapping file, `/etc/sdi-error.conf`. The DLL's error handler receives the entire error data structure and may take arbitrary actions. For example, in one case we designed an error handler that responds to back ends' receiving packets from untested front-end services by installing a replica of the back-end service and redirecting to it future requests from untested front-end services (Section 3.6.2).

3.5.5 Application Integration

Applications typically communicate indirectly with the SDI framework using a function call API, `libcontext`, which interfaces to the native `/proc/sdi` interface. This interface allows user-space applications to inquire of their own context bindings, create context, read attribute values, and adjust attribute values of user-space writable attributes.

User-level threads and event-handling libraries can take advantage of SDI by explicitly declaring their current internal thread or request ID in a registered memory location (`ctxt_register_thread_id_location`). Request IDs are added and removed using the `ctxt_add_thread_id` and `ctxt_remove_thread_id` functions, respectively. This simple feature reveals enough of the application's internal structure for SDI to police the application and to automate the forwarding of its context between tiers.

3.5.6 Failure Modes

The main cause of faults is the absence of context or attributes that are expected by SDIs. If a guard attempts to match a non-existing attribute, its value is assumed to be `NULL`. If a context and attribute remapping directive attempts to assign a value from a non-existent attribute to another context attribute, only that substitution clause is skipped; subsequent clauses are unaffected. Finally, GCFs may request non-existent contexts or non-existent context attributes, in which case SDI's attribute retrieval function would indicate a fault. It is up to the GCF implementations to handle such errors.

Internally, SDI may suffer from sporadic packet loss even though this is rare because SANs are highly reliable. Our switched Fast-Ethernet testbed, which features commodity Intel and SMC networking hardware, is found to experience error rates of less than 1 per 30 million. To mask occasional packet loss, SDI retransmits requests for which an answer has not been received within a specified timeout (default 10ms). Update operations are acknowledged periodically. Acknowledgments are cumulative for all update operations that were initiated by a specific host.

Because of the low error rates of the underlying network infrastructure, SDI acts optimistically with respect to transmission failures, i.e., the control flow of SDIs or applications does not wait for the acknowledgment of a context attribute change, which is possible due to soft-state. This choice keeps latencies for update operations low.

To guard against machine failures, heartbeats are to be exchanged at a minimal rate, r ($r = 1\text{Hz}$). If some context's home machine detects a silence of duration $3/r$, it expunges all remote referral entries from the failed remote host, assuming it has died.

In the event of a failure of a machine that hosts primary context, the proxies must recover. The failure is detected when access to its primary context objects time out (3 unanswered request retransmissions). On timeout, a host is considered dead, and the requesting host invalidates all references to any context object that resides on the failed host. Processes, sockets, etc., that are bound to a context of the failed host are unbound to refer to the `NULL` context.

3.6 Three Examples of Using SDI

3.6.1 Simple Context Propagation via Local IPC

In this example scenario, we show how context can be transferred within one application or two applications that communicate via local IPC message queues. IPC mechanisms are often used to bypass TCP/IP when two communicating tiers are co-located. For example, message queues are used in DB2 to connect the listener to the back-end server process.

As an example of SDI over machine-based IPC we demonstrate how the IPC message queue, mechanism is opened up to SDI.

The two system calls used to send and receive messages are `msgsnd` and `msgrcv`, respectively. The purpose of enabling context-transfer and SDI at this layer is that it allows to track a processing context as control is passed from one process to another. Furthermore, by allowing SDI policies to be submitted for taps in the communication flow between application that are linked via IPC, one can easily add security, scheduling, and redirection mechanisms. For example, a process could be denied access to an IPC message queue if it is executing on behalf of a remote client who is connected via the Internet while being allowed to access the same message queue as long as it is working on behalf of a local client. Such policies cannot be configured using stateless system configuration mechanisms that are currently at the system administrator's disposal.

In enabling SDI on the IPC message queue, it is necessary to deal with four entities: the message, the message queue, the sender, and the receiver. IPC message queues are an asynchronous relaying mechanism, so that one only needs to deal with three entities in the taps for `msgsnd` and `msgrcv`.

A default SDI that one would like to declare is:

```
iipc_msgsnd 1 [sender priority] = high : msg := sender : dc.
```

This SDI copies the sender's context to the message if the sender's priority is high. As the duplicate initial letter shows, this command is executed before the message is actually enqueued in the message queue.

When this context-tagged message is received, and one would like to copy the message's context to the receiver, then one would install the following SDI.

```
ipc_msgrcvv 1 : receiver := msg : dc.
```

In fact, this SDI always copies the received message's context to the receiver, thus forcing the receiver to process using the same context attributes as the sender. Note that any previous context affiliation of the `receiver` process is discarded/overwritten once it is bound to the new context. This SDI executes after the message has been removed from the message queue. A step-by-step illustration of the tap points behavior is shown in Figure 3.14.

Implementation Steps

The implementation steps required for SDI to enable the IPC message queue abstraction are most likely a superset of those required for SDI-enabling of almost all OS abstractions.

First, it is necessary to add a context reference pointer data field to the message and message queue data structures to allow recording of context references for each system object. It would have also been possible to transparently keep this reference through an intermediary message-to-context mapping table, but it would have also incurred greater overheads.

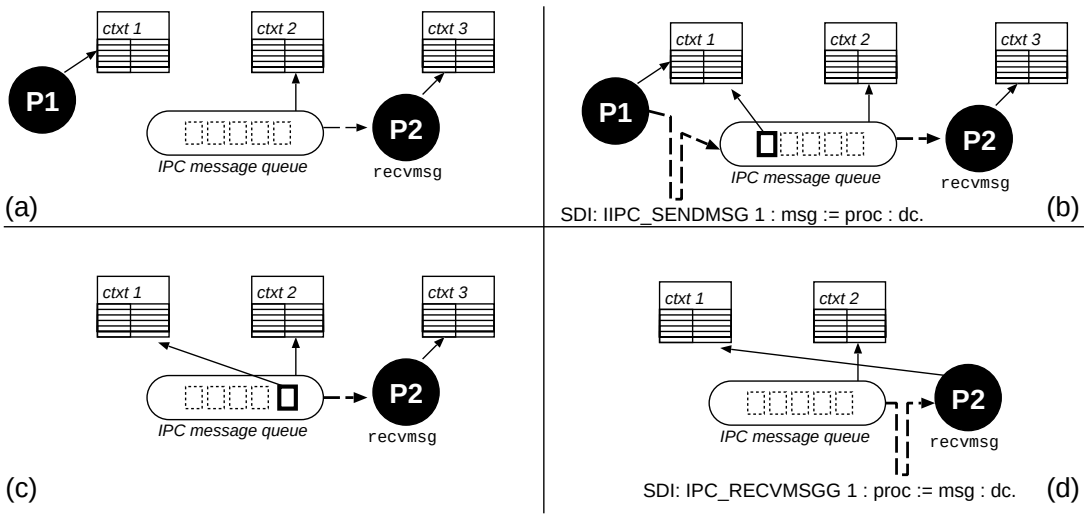


Figure 3.14: This figure shows the process of context propagation through the SDI-enabled IPC message queue mechanism: (a) the sender is associated with context object 1 prior to sending the message, the queue with context 2, and the receiver with context 3; (b) the created message inherits the sender's context binding through the specified SDI rule; (c) the message queue's context binding remains the same; (d) when the message is ready for delivery to the waiting process (P2), the `recvmsg` SDI rule instructs the framework to change the receiving process' context binding to context 1 instead of its prior binding context 2.

The second step is to define the interception points `TAP_IIPC_MSGSND`, `TAP_IPC_MSGSNDD`, `TAP_IIPC_MSGRCV`, and `TAP_IPC_MSGRCVV`. These definitions,
among accounting-related issues, include the implementation of a tap-point handler function that is inserted as a kernel module if interposition on the IPC interface is needed. This interposed tap-point handler takes as its argument a reference to both the intercepted message and message queue. The process context is implicitly given. The tap-point handler dereferences the context pointers that are recorded for the calling process, the message, and the message queue and passes those to the generic SDI matching module, which evaluates if there is registered SDI that applies to the observed call given its context attributes.

Once the SDI check returns a list of SDIs that are applicable to the current interception, the handler passes through the SDIs, applying their attribute mapping and context binding rules and interpreting the action code.

The only steps that are truly specific for SDI-enabling of the IPC message queue interface are the creation of functions that bind a message or message queue to a context object, and the interpretation of SDI's error codes (Figure 3.15), which is a simple operation. For example, the SDI action code `DENY` is translated by erasing the message and pretending it was delivered in the `iipc_msgsnd` interposition. In contrast, in the the interposition of `ipc_msgrcvv` the `DENY` action causes the offending message to be discarded and the search for a deliverable message to be continued.

This skeleton consisting of a glue layer that ties the tap point to generic context processing functions, allows the previously-defined tracking of context from sender to receiver. To implement more elaborate rewriting or redirection of IPC messages, e.g., to a remote message queue on a

```
static int real_msgsnd (int msqid, struct msgbuf *msgp,
                        size_t msgsz, int msgflg)
{
     ...

#if defined(CONFIG_CONTEXT)
        msgh->context = NULL;
#endif
        err = -EFAULT;
        if (copy_from_user(msgh->msg_spot, msgp->mtext, msgsz)) {
                goto uncharge;
        }

       err = -EIDRM;
        if (msgque[id] == IPC_UNUSED || msgque[id] == IPC_NOID
                || msq->msg_perm.seq != (unsigned int) msqid / MSGMNI) {
                goto uncharge;
        }
#if defined (CONFIG_CONTEXT)
        {
            int sdi_res;
            if (iipc_sendmsg_intercept)
                if ((sdi_res = iipc_sendmsg_intercept(msgh, msq)))
                {
                    if (sdi_res == -1)
                    /* this message is handled by a gcf */
                    {
                        err = 0;
                        goto message_handoff;
                    }
                    /* custom error ? */
                    else if (sdi_res < 0)
                    {
                        /* errors are downshifted */
                        sdi_res++;
                        err = sdi_res;
                        goto uncharge;
                    }
                }
        }
#endif

        msgh->msg_next = NULL;
        msgh->msg_ts = msgsz;
        msgh->msg_type = mtype;
        msgh->msg_stime = CURRENT_TIME;
...
}
```

Figure 3.15: This figure shows most of the work that is required inside the kernel for the invocation of the SDI framework from IPC `msgsnd`.

different host, one would insert a `redirect` GCF. An example of a redirection GCF is given in Section 3.6.2.

3.6.2 Protecting Back-End Service Integrity using Alternative Services

One of the concerns when co-hosting services in a shared services infrastructure is that new services may corrupt existing ones or as in the case of LVSC (Chapter VI) one may want to install alternative services that implement comparable functionality at different QoS levels. Therefore, server farm administrators often install new services on dedicated hardware running their own instances of standard services. Once a dedicated setup has been chosen, administrators typically do not consolidate the setup because consolidation requires reconfiguration.

To simplify the transition from experimental configurations to consolidated deployments, we provide a simplified example application that allows configuring the experimental setup as if it were configured for final deployment. Our extension replicates back-end services automatically if a request from an untested front-end service is received. Subsequent requests by an untested front-end service are redirected to the replica (see Figure 3.16). Once the new service has proven reliable, the system administrator simply deletes SDI classification rules, and requests submitted

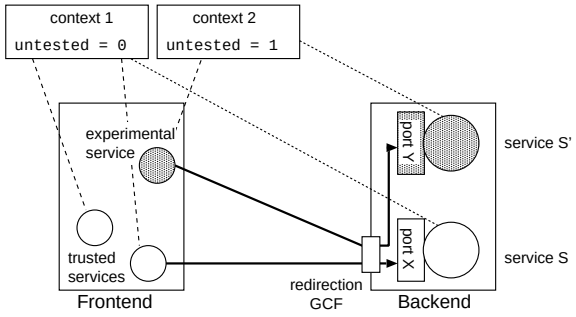


Figure 3.16: Back-end request redirection based on the requestor's untested attribute

by the recently-tested service are processed by the shared back-end setup. Replicas that are no longer needed can be deleted. This example is a variant of Flask's [130] interference avoidance mechanism. Flask enables the administrator to restrict services from accessing certain system calls and from interacting with other services.

Whenever a new service is installed in the system, the administrator binds its processes to a context with `untested = 1`. This is accomplished by a simple command line script that tells the SDI kernel module to bind each of the untested service's processes to a specific context (`sdi-classify-proc <pid> --home <ip> --id <id>`). Back-end services are configured in the same manner with the only difference being that their context's `untested` attribute is set to zero. Context propagation is configured as it was in the previous example.

We use the error directive to detect when an untested front end tries to access a specific back-end service. For each back-end service we configure the following rule:

```
ACCEPTT 1 [msg untested] = 1, [proc untested] = 0 : \
                            : error {ECONNABORTED}.
```

The client and server will have to recover from an aborted connection.

The error directive is interpreted by the accept tap, which passes a message containing the applicable SDI and information about the incoming connection (i.e., destination port and source address) to an error daemon via `/proc/sdi`. This daemon checks for each violation, if it has a rule that allows it to replicate the service listening on the destination port. If it finds an installation script, it creates a service replica that listens on a different port and automatically specifies a new SDI of the form

```
NET-IP-IN 1 [msg untested] = 1 :
            : gcf REDIRECT {$ORIGINAL_PORT $REPLICA_PORT}.
```

to redirect accesses by untested front-end services to replica back-end services for experimentation. The `$ORIGINAL_PORT` and `$REPLICA_PORT` are determined at replication time.

The redirection GCF is implemented in a 91-line "C" kernel module. The daemon plug-in responsible for creating redirections consists of an 83-line C program, and an 8-line Perl script consults a replica setup file (a simple ASCII text file) to automate the replication process.

3.6.3 Prioritized Request Handling

The purpose of this example is to introduce the problems encountered when trying to assure prioritized workload processing in a multi-tiered system, in which front-end servers act as request

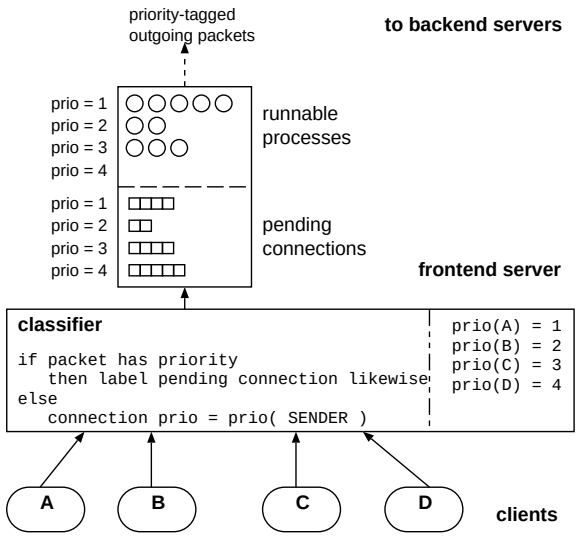


Figure 3.17: VS-SDI consists of a scheduler and `accept` extension. The classifier labels incoming requests with system administrator specified priority attribute mappings, which are enforced by the scheduler and the `accept` of pending connections.

anonymizers. This example is a preliminary version of the full-fledged Virtual Service multi-tier resource partitioning approach that is introduced in Chapter IV. This is why the example SDI application of this chapter is called VS-SDI.

The example setup consists of a 3-tier service implementation. The front end implements a Web interface (Apache), while the middle-tier application server is simulated by an FCGI application, which occasionally contacts a back-end database (Postgres) to process update and select operations. About half of all requests require access to the middle-tier service, and 10% of those requests requiring access to the middle-tier service also require the database. The Web server executes on its own front-end host, while the FCGI middle-tier and the back-end database share one back-end host. Our objective is to prioritize request handling throughout the server farm, i.e., requests from high-priority clients should be expedited at all tiers of the system.

Prioritized processing is impossible to achieve in the back-end servers, if their incoming requests are not tagged with a priority attribute. As far as the back end is concerned, all requests originate from the front-end service. Thus, high-priority clients can find themselves waiting either in the accept, network, or scheduler queue of a heavily-loaded back-end service or server because of low-priority requests.

The easiest part of the problem is to represent high- and low-priority clients by two context objects. Using the command-line interface as *root*, we create and initialize the two context objects as follows:

```
ctxt_create -p
context home=10.0.0.100 id=1 created
ctxt_create -p
context home=10.0.0.100 id=2 created
```

The commands

`ctxt_set_attr 1 PRIO := 1` and

`ctxt_set_attr 2 PRIO := 2`

```
/* CONTEXT-AWARE PRIORITY SCHEDULING FOR UNI-PROCESSORS
 * Note, declarations are stripped.
 */

static int
prio_goodness_handler(struct task_struct * prev,
                      struct task_struct * p,
                      int this_cpu)
{
  struct uci_value_pair *uvp_prio;

  /* Does this process  have any context at all? */
  if (!p->context)
  {
        if (prev_goodness_handler)
                return prev_goodness_handler(prev, p, this_cpu);
        else
                return -1000;
  }

  uvp_prio = get_uvp_local (p->context, UCI_PRIO);
  if (!uvp_prio || !uvp_prio->value)
  {
        if (original_goodness_handler)
                return original_goodness_handler(prev, p, this_cpu);
        else
                return -1000; /* this should not happen */
  }

  return (int)(uvp_prio->value) + 10000;
}
```

Figure 3.18: Code snippet of a context-aware scheduler interposition

set up the first context to represent low-priority work and the second to represent high-priority work. `PRIO` is an integer representing the priority attribute as the current prototype only supports name and value bit strings of length 32.

The next important step is to bind incoming workload to the contexts. For simplicity, we prescribe a simple binding based on the incoming IP address. To bind incoming workload from clients `10.0.1.*` to the low-priority context and the workload submitted by clients `10.0.2.*` to the high-priority context, we invoke the following commands:

```
sdi-classifier --sa 10.0.1.0 --sam 255.255.255.0 --home 10.0.0.100 \
               --name 1
```

and

```
sdi-classifier --sa 10.0.2.0 --sam 255.255.255.0 --home 10.0.0.100 \
               --name 2
```

Two interpositions are implemented to take advantage of the priority attribute: an interposition for the scheduler function and a selection function that chooses the next pending socket to accept. Their implementation is generally not the system administrator's responsibility. They should be created by experienced system programmers who have a good understanding of the kernel and the impact that their interposition may have. Warnings aside, as the code snippets in Figures 3.18 and 3.19 show, creating multi-tier-aware interpositions is in many cases straightforward.

After installing the interpositions and the `ACCEPTT` and `CCONNECT` tap in the kernel using `modprobe`, we set up the taps to propoagate context at all servers.

```
echo "ACCEPTT  1 : PROC := MSG  : dc." | sdi-config -a
echo "CCONNECT 1 : MSG  := PROC : dc." | sdi-config -a
```

```
/*
 * PRIORITIZED ACCEPT
 * Declarations and module maintenance have been removed.
 */

/* compare Linux source (net/ipv4/tcp.c for the original) */

struct open_request *
prioritized_tcp_find_established (struct open_request *req,
                                  struct open_request *prev,
                                  struct open_request **prevp)
{
  struct open_request *best_so_far = NULL;
  struct open_request *best_so_far_prev = prev;
  int best_prio_so_far = 0;

  while (req)
  {
      if (req->sk &&
          ((1 << req->sk->state) & ~(TCPF_SYN_SENT | TCPF_SYN_RECV)))
      {
          struct uci_value_pair *uvp_ptr_prio;
          uvp_ptr_prio = NULL;

          if (req->context)
          {
              uvp_ptr_prio = get_uvp_local (req->context, UCI_PRIO);
          }

          if (((uvp_ptr_prio) && (uvp_ptr->value > best_prio_so_far))
              || (!best_so_far))
          {
              best_so_far = req;
              best_so_far_prev = prev;
              best_prio_so_far = (uvp_ptr_prio)?uvp_ptr->value:0;
          }
      }
      prev = req;
      req = req->dl_next;
  }

  if (best_so_far)
  {
      *prevp = best_so_far_prev;
      return best_so_far;
  }
  *prevp = prev;
  return req;
}
```

Figure 3.19: Code snippet of our context-aware interposition for the selection of pending connections

After setting up the system, we verify that this configuration implements QoS differentiation for high-priority requests. To this end, we run two instances of the SpecWeb99 benchmark against the same Web server. As Figure 3.20 shows, high-priority clients suffer little from an increase in the workload of low-priority clients until the system capacity limit is reached. This figure clearly shows that without prioritization high-priority clients are affected by the low-priority clients even before the system's capacity is reached. The reason for the rapid response time increase beyond the system's capacity is that queues that build up at the database server propagate to the front-end server by blocking processes. This effectively reduces front-end processing capacity, thus causing the observed increase in response times and a 30% drop in total throughput. This and other problems related to implementing resource control for multi-tiered servers are discussed in Chapter IV.

It is important to note that context caching is vital to maintaining good service throughput with SDI. Without context caching, each network-level message would have to be accompanied by its context, which would tax the communication subsystem, thus increasing delays and reducing total throughput.

Increasing Classification Complexity

Instead of differentiating between high- and low-priority clients at the Web server, one may decide

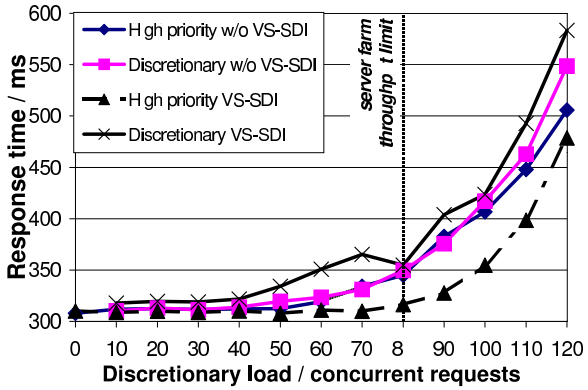


Figure 3.20: The performance of a multi-tier server farm serving high- and low-priority clients with and without the SDI-based priority mechanism

to treat all clients equally except for those high-priority clients whose requests require back-end database transactions. Those requests that require database access receive a priority boost. This could be used to reduce the difference in response times for requests that require database access and those that do not.

This means that the binding between high-priority clients and their priority class must be deferred until they actually trigger a database transaction. However, this cannot be controlled by the HTTP server since the middle-tier FCGI server, i.e., the application server, decides whether to contact the back-end database. Unfortunately, the middle-tier cannot correlate its requests with the original HTTP requests that are submitted by the network clients. SDI solves this problem.

This example configuration requires the creation of 3 context objects (`$x,$y,$z`) which are configured as follows:

```
ctxt_set_attr $x PSEUDO-PRIO := 1
ctxt_set_attr $y PSEUDO-PRIO := 2
ctxt_set_attr $z PRIO := 2
```

The classifiers, the accept differentiation, and the scheduler interposition of the previous example are installed. Note, however, that the context objects `$x` and `$y` only carry priority marker variables that do not affect scheduling.

All hosts are configured with the following SDI:

```
ACCEPTT 1 : PROC := MSG : dc.
```

In order to allow modified back-end request bindings to propagate back from the back-end server to the front end, we configure the SDI

```
TTCP_SEND 1 [PROC PRIO] =  2 : MSG := PROC : dc.
```

at the back-end server. The front end is configured using

```
TCP_RECEIVEE 1 [MSG PRIO] = 2 : PROC := MSG : dc.
```

Finally, one must configure the back end to boost the priority of incoming requests belonging to `PSEUDO-PRIO` 2 as follows:

```
NET-IP-IN 1 [MSG PSEUDO-PRIO] = 2 : MSG := (10.0.0.100 z) : dc.
```

The above configuration maps messages from `PSEUDO-PRIO` 2 to the persistent context object residing on host 10.0.0.100 with ID `z`, which represents high-priority clients.

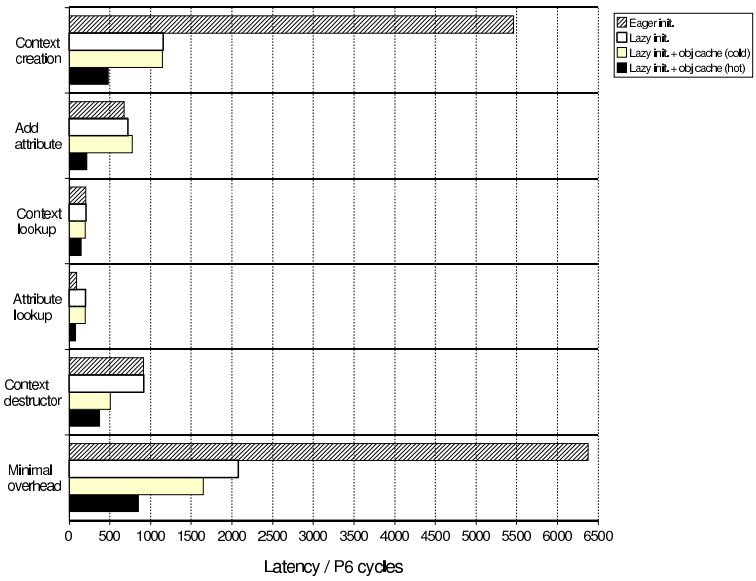


Figure 3.21: Base overhead for context operations and comparison of the performance optimizations for fast context creation described in Section 3.5

3.7 Evaluation

The performance of the SDI prototype for the Linux OS indicates that SDI can be implemented efficiently. We present both micro- and macro-benchmark measurements of our prototype and discuss their implications. The measurements lend further support to crucial implementation choices that are likely to become important in future instantiations of SDI.

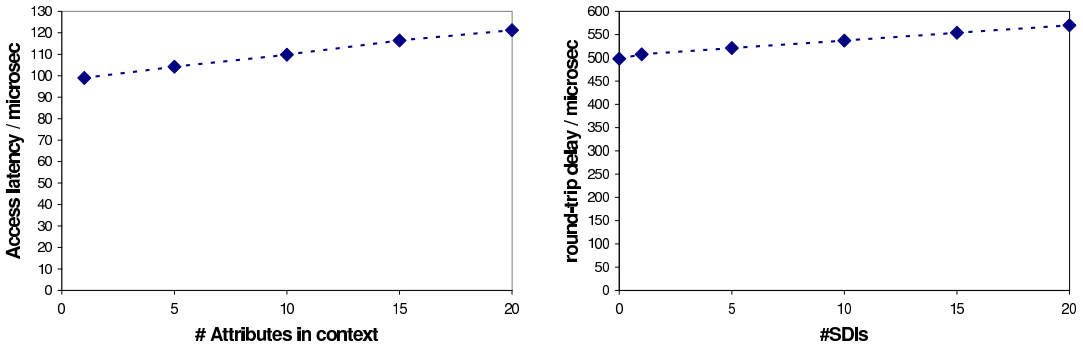
3.7.1 Micro-benchmarks

The first set of measurements is taken on a single 450MHz Intel Pentium II computer. The results, shown in Figure 3.21, demonstrate the relevance of the performance optimizations for dynamic context creation. Using both lazy initialization and a LIFO queue filled with disposed context objects, the minimal cost of context creation and destruction is reduced to 850 Pentium II cycles from over 6000 cycles for a straightforward implementation using the kernel's memory management and eager initialization.

As expected, lazy initialization increases the cost of attribute lookup. However, the penalty is in the low hundred cycles, whereas the number of cycles saved by not completely initializing the context object is in the range of 5000 cycles. Hence, it will take a large number of context accesses to offset the benefits of lazy initialization. Since the contexts of long-lived requests are eventually indexed, the performance penalty for attribute accesses lasts only for 1s.

One may have noticed in Figure 3.21 a seemingly odd performance impact of using the alternative context object (de)allocation queue; attribute lookups are accelerated. The reason for this anomaly is that context's memory locations are more likely to be cached if they are taken from the most-recently disposed context object. Hence, the execution time of context operations will suffer less from L1 and L2 cache misses.

We also measured SDI's network base performance in a small cluster of seven Intel Pentium-III 550 servers connected by a 100 Mbps Fast-Ethernet, SMC Tiger switch. Since server clusters



(a) Latencies for remote attribute access with different context sizes

(b) End-to-end delay for a UDP-echo application, depending on the number of interposed SDIs

Figure 3.22: Micro-benchmark results

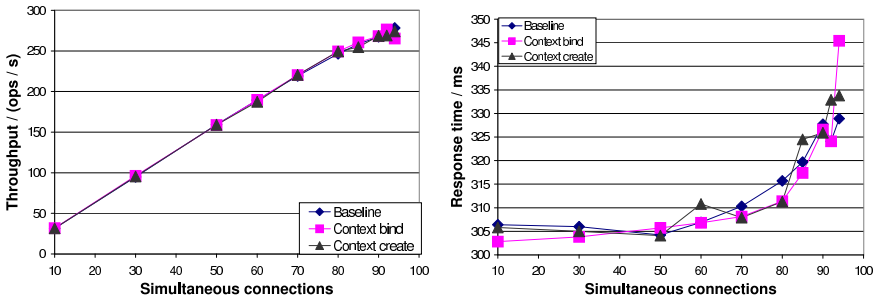
are migrating to Gigabit-Ethernet and even faster servers, the reported latencies are likely to be larger than what could be achieved on the best available hardware platforms. However, the performance numbers indicate that Context/IP and the context service, even in its current prototype implementation, will add only negligible additional latencies to multi-tier services.

Figure III.22(a) shows the delay that a context-dependent module, such as a GCF may experience in accessing remote context attributes. As expected, delay grows with the number of attributes per context, but for most context applications it will remain within a range between 100 μs (0 attributes) and 200 μs (100 attributes). This remote access cost will be incurred only if a local context proxy either has out-of-date information or has not yet been set up. Subsequent attribute accesses that can be served from the context proxy take only 130 Pentium II cycles.

Since attribute access will contribute little to application latency, we investigate the performance of SDI rule evaluation at the tap points. To assess the worst possible latency effects of SDIs, a UDP-based server was set up to do nothing but bounce any incoming datagram back to its sender. A single client was set up to send requests of 1 KB size to the server and time how long it takes for the packet to return. Both client and server machines are SDI-enabled.

The measured end-to-end delay is linearly increasing in number of context-dependent guards that are interposed (see Figure III.22(b)). Each additional SDI adds approximately 3 μs of latency. These delays are too small to cause a noticeable increase in the response times of complex cluster services. End-to-end service delays in the Internet are typically above 50 milliseconds. Nevertheless, to support thousands of simultaneously-installed SDIs, future versions of SDI should implement guard checks in decision trees instead of the linear lists of guards that are registered at tap points in the current prototype.

The performance impact of guard interposition at the system call layer varies, depending on the complexity of the system call in relation to SDI evaluation complexity. This has been noted in earlier interposition-based research projects [58, 118]. System calls' performance can deteriorate as much as 40% for simple system calls like `open` and as little as 2% for a complex call like `fork`. Fortunately, low-overhead system calls, for which the impact of interposition is the worst, contribute only little to most applications' total processing time [154]. Services spend most of their time executing application code and heavy-weight `send`, `read`, `recv`, and `write` system calls.



(a) (b)

Figure 3.23: Througput comparison between a system without SDI (baseline), binding incoming requests to existing context and to newly-created context

3.7.2 Macro-benchmarks

To obtain a realistic assessment of SDI's impact on true server performance, SDI's effect on a Web server executing the SPECWeb99 [132] benchmark is measured. SPECWeb99 generates a mix of dynamic and static request loads — an approximation of the request load found on a realistic server. The response times and throughput numbers shown in Figure III.23(b) and Figure III.23(a), respectively, are measured on a 450 MHz Pentium II-based server with 448 MB RAM. We use Apache 1.3 as a web server. The SPECWeb99-supplied Perl scripts are used to handle all dynamic workload except advertisement service, which is done in a FastCGI server.

The macro-benchmarks compare the performance of Apache on a server without any SDI support against the performance of Apache on an SDI-enabled system. The overhead of SDI is assessed for two scenarios. First, in a low-overhead scenario, two classifiers are set up to bind incoming requests to one of two persistent context objects. The command lines,

```
sdi-classifier -p TCP -y --sa 10.0.1.0 --sam 255.255.255.0 \
               --home 10.0.0.1 --name 1
sdi-classifier -p TCP -y --sa 10.0.2.0 --sam 255.255.255.0 \
               --home 10.0.0.1 --name 2
```

implement this binding directive.

In a second scenario, we attempt to approximate the maximal overhead caused by the association of workload with context by creating a new context object for each incoming request. Such a configuration is typical of an environment in which each request is managed with respect to its own performance, security, and monitoring goals. The command lines

```
sdi-classifier -p TCP -y --sa 10.0.1.0 --sam 255.255.255.0 \
               --home 10.0.0.1 --name 1 --dup
sdi-classifier -p TCP -y --sa 10.0.2.0 --sam 255.255.255.0 \
               --home 10.0.0.1 --name 2 --dup
```

configure SDI for the target scenario. The newly-created context automatically propagates up to the accepting socket and later to the processes that read from the so-classified socket. The measurements show that neither response time (see Figure III.23(b)) nor throughput (see Figure III.23(a)) of an HTTP server is affected by the presence of SDI. The lines labelled "context bind" and "context create" in Figures III.23(a) and III.23(b) represent the binding of incoming requests to existing

context and the creation of a new context object for each incoming request, respectively. The differences between the different configurations are so minimal that they are almost indistinguishable from normal measurement noise.

3.8 Related Work

The proposed SDI mechanism is the first to integrate extensible, distributed system state with interposition into a highly extensible, distributed system management and extensibility framework. While context and its management in distributed systems have appeared in numerous applications (e.g., security, resource management, and monitoring), each domain-specific solution only manages a few concrete contextual attributes, e.g., security classes. Context has not yet been proposed as a separate generic service for the design of system support for distributed systems.

The LINDA tuple-based computation and communication model [56] shares some similarities with SDI and other interposition schemes. LINDA proposes a computation model in which persistent processes post data tuples into a distributed tuple-space. Computation progresses as executions are triggered by conditional receives of these posted tuples. Additional contextual state can simply be integrated into distributed computations by extending the tuples. Unfortunately, LINDA's distributed state abstraction, tuples, are transient, so that propagation of additional attributes still requires programmer intervention. In particular, computation rules must preserve the unused state of their input tuples by adding it to their output tuples. SDI, like LINDA, achieves extensible state and state-directed processing. In addition, SDI manages per-computation state, while preserving the traditional invocation-based programming model, processes, existing service APIs, and the communication abstractions found in today's OSs.

3.8.1 Application Frameworks

CORBA [102], J2EE [139], and WebSphere [70] are environments for the design of multi-tiered applications. Each of these application environments provides its own notion of context, primarily for the implementation of access control mechanisms. CORBA uses context primarily in the implementation of CORBA Security. J2EE and WebSphere use context to map the application server's user IDs to back end user IDs before accessing a back end database.

In CORBA context is implemented as an optional parameter for every remote method invocation. To have any effect, the CORBA context abstraction must be unpacked by the server object. Without active intervention by the server application, context does not propagate across the tiers of multi-tiered computations. Since the applications are responsible for configuring and maintaining their context attributes, one cannot rely on their availability at the system layer and across application frameworks. SDI solves this problem. Moreover, SDI also addresses numerous inefficiencies of context abstractions in application frameworks, which result from the fact that context was typically introduced as an afterthought to fix certain problems (e.g., security). In contrast, SDI proposes context as a primary system abstraction.

3.8.2 Interposition

Since its proposal as a generic system extension mechanism by Jones [78], interposition has gained significant support. The SPIN OS [21, 107] effectively promotes interposition as the standard way in which system functionality is to be achieved. SLIC [58] pursues similar goals for commodity OSs. The basic objective of interposition can be summarized as calls to existing sys-

tem and service interfaces that are intercepted and redirected to interposed wrapper layers that improve or augment the intercepted interface's semantics without affecting the invocation syntax.

The above approaches adopt an event-based dispatcher scheme [107] in place of the traditional function call interfaces for OS layer interactions. SPIN, for example, maps all interactions between system layers to events which can be intercepted by interpositions. The default interpositions are the standard OS handlers. The event language is fixed at the time of OS design. Interpositions cannot create their own additional state and events. SDI addresses this shortcoming.

The lack of state integration in previous single-host interposition approaches is not an oversight. On a single host, necessary state information can simply be preserved in process or in socket descriptors or even reproduced on-demand. Hence, state is typically implicit in the variables of the OS layers or interpositions. This is why previous approaches work well despite their disregard for state. A solution for interposition in multi-tier systems cannot assume that state is always local; it must be stateful.

3.8.3 Domain-Specific Context Solutions

Active Messages [149] are a domain-specific incarnation of SDI. Active messages propagate pointers to the receivers' packet processing functions with every message that is exchanged in a distributed system, thus possibly short-circuiting unnecessary checks. Besides the shared packet reception pointers, there is no shared or extensible state that is transferred between tiers.

A rich body of work in network security [1, 13, 49, 93, 98] includes some of the basic features of SDI with respect to security attribute propagation, attribute remapping, and policing. However, these works fail to abstract from concrete state propagation problems (e.g., user ID propagation) to a more abstract concept of attribute propagation, and from the problem of security policy enforcement to generic policing of distributed computations. SDI makes these abstractions. Readers familiar with research in system security will quickly realize the synergies between SDI and the implementations of network security mechanisms.

For example, the Domain and Type Enforcement (DTE) architecture [13] stresses the need for flexible security attribute propagation along the path of inter-application communication. To address this need, the noted approach provides a rich policy framework supporting security attribute inheritance, remapping at tier-boundaries, and propagation in IP datagrams. We believe that much of DTE's functionality is not necessarily security-specific but should be captured by a generic service like SDI instead. Flask's policy-controlled integrity [130] mechanism — featuring sender-based packet redirection — is also a highly specialized instance of state maintenance, state propagation and label-based interposition on system interfaces. Similar functionality can be implemented using SDI almost effortlessly (Section 3.6).

The need for propagation of state information has also been noted in recent work on resource management. The Scout OS [129] and Lazy Receiver Processing [14, 15] emphasize the importance of processing incoming workload in the right resource context. To this end, they provide proprietary resource-binding mechanisms. Scout provides a compile-time processing path abstraction, which automatically propagates resource reservations across traditional OS abstractions. Different subsystems are arranged into a call chain, called *path*. Each path represents a flow of requests that are to be administered as one administrative entity. Each path can be controlled by its own resource controls. Unfortunately, it is necessary to recompile the kernel for almost every application. LRP is more flexible in that it binds incoming requests to Resource Containers [15] on the basis of dynamically-installed connection matching rules (source address and port, destination address and port). Both approaches fail to provide proper resource isolation when competing processes relay work to shared, remote processes. Virtual Services [118] (Chapter IV) solve this

problem by propagating explicit resource reservation handles along with all inter-application message exchanges. Cluster reserves [10] provide similar functionality at the application-layer by using Resource Containers in combination with application modification and a resource management daemon application.

3.9 Summary and Conclusions

We have introduced SDI as a useful, low-overhead improvement of OSs for multi-tier services. SDI associates state with multi-tier computations and facilitate state propagation as computations spread to multiple machines and sub-services without mandating application modification. SDI achieves the same extensibility and customizability for multi-tier, component-based systems that has been achieved by interposition for single-host OSs. Thus, component services and OSs can be fixed up to perform well in server farms, under constraints that were not anticipated at the time of their design.

Since contextual information significantly simplifies coordination across software layers, auxiliary system management and application support mechanisms that integrate across several software layers can be built more easily. Context-based access control, for example, can be enforced at the network layer while application-layer information (e.g., a user password) may still be taken into account. Other mechanisms that will benefit from SDI include fortification of previously unsafe service protocols, server-site monitoring mechanisms, integrity assurance, context-aware load-balancing, distributed resource management, and creation and propagation of transaction contexts in nested server activities [65].

The current prototype demonstrates that a distributed context propagation and interposition framework can be built in a manner that is independent of the applications without excluding them from using and improving context semantics. In already-built example applications, SDI is shown to significantly reduce implementation complexities. We believe that SDI can generally simplify the design of system software enhancements for multi-tiered systems.

Despite the prototype's promising performance, there is still ample room for future research. Some obvious extensions of SDI, such as hierarchically-nested context, have to be evaluated with respect to their additional expressive power, performance benefits, and their overheads.

A practical improvement target is the performance of SDI's guard matching, which is sequential in the current prototype. Therefore, runtime overheads for each system interface tap are linear in the number of registered SDIs. Instead of matching each guard clause of SDIs in a static, linear order, a minimized finite state machine checker representation should be generated automatically. While this would not change linear time worst-case complexity, the average case could experience a significant speed-up, since common guard conditions could be eliminated across SDIs. Guard checks should also be reordered automatically so as to minimize the average number of comparison operations executed, i.e., check the most selective guard conditions first. This would allow a greater number of simultaneously installed SDIs.

Finally, it is important to note that the proposed SDI is not intended to be a final standard for distributed context. It should rather be viewed as the beginning of a standardization process that will replace other existing Internet standards that provide narrower, application-specific, and less customizable abstractions of context (e.g., `identd` and CORBA). In order to provide a new, generic context service for IP-based multi-tier systems, it is necessary to address the main points raised in this chapter.

CHAPTER IV

Virtual Services

4.1 Introduction

Virtual Services (VSs) primarily address the control-side problems of managing the performance of shared services in a multi-tier deployment. In particular, VSs attempt to replicate the concept of resource allocation that exists for individual processes or monolithic single host services and apply it to a composite service, in which different components may be shared with other composite services. In essence, VSs extend the simplified SDI-based activity prioritization example of Section 3.6.3.

VS is a control framework that allows system administrators with an understanding of the hosted services and their interactions with the OS and each other to configure effective control policies. The principal mechanisms that are used to realize VSs are: *classification*, *interposition*, *policing*, and *state propagation*.

The problem of policing a multi-service system has drawn enough interest from Application Service Providers and other service hosting companies, so that researchers and commercial companies have started addressing the problems of sharing one server installation across multiple clients. Controlled resource sharing is typically achieved by the virtualization of resources in off-the-shelf OSs (e.g., *virtual web hosting* and *virtual servers*) [15, 27, 32, 51, 52, 84, 142]. The essence of these concepts is that one physical server is split into several virtual hosts (VHs). Ideally, neither the client nor the server application is aware of the fact that it is executing on a VH and not on a real host. Initial implementations of this idea were content-based, application-level VHs. Here, a service would serve different contents, depending on the IP address that was used to contact it, e.g., Apache's `VirtualHost` directive [84]. VMware [147] perfects this approach by creating a number of virtual hosts inside a host OS. Unfortunately, the number of co-hosted VHs is limited and the performance interference that occurs between co-located VHs is not considered. To solve the second problem, resource bindings for VHs were introduced [15, 27, 51, 141, 142]. With resource bindings, demand surges on one VH will no longer impact the performance of other co-hosted VHs. A service that is executed on one VH behaves as if it were executed on its own physical server. This still does not address the performance interference between services that may result from a single back-end service between different applications and customers (Figure 4.1).

When back-end services are shared among different clients in a multi-tiered system, VH-based insulation approaches fail (see Figure 4.1). Shared services like DNS, proxy cache services, time services, payment processing servers, distributed file systems, and shared databases, are worth consolidating. However, when services are shared between multiple front-end VHs, an obvious question arises as to which VH should host the shared back-end services. Since these services

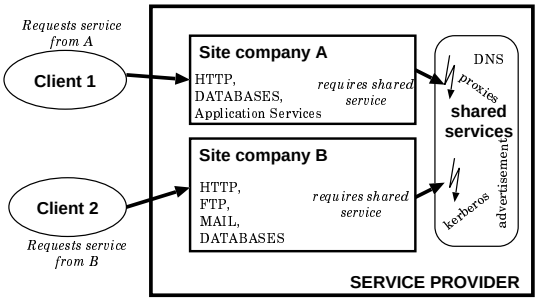


Figure 4.1: Service-sharing destroys insulation

work on behalf of multiple front-end services, their resource bindings should be dynamic to reflect who requested a particular service (i.e., the *works-for* relation). The problem of determining the works-for relation on-the-fly could be avoided entirely by replicating shared services on each VH. However, in this case the consistency of individual shared services becomes a major concern if they must share data, as well as software licensing, the resource inefficiency of hosting two identical services to maximize performance insulation, and the overall inefficiency of hardware virtualization.

To eliminate the performance interference caused by shared services, the *Virtual Service* (VS) concept is introduced. VSs link the tracking of activities to a resource reservation enforcement framework. The state tracked alongside multi-tiered activities is a resource partition identifier that is translated to a resource partition handle at every host. The VS framework dynamically binds OS abstractions (e.g., processes) to resource partitions in a manner that is transparent to applications. The goal of dynamically binding OS abstractions to resource partitions, is to improve control over activities that propagate throughout a multi-tiered infrastructure.

Each application's interaction with the OS is assumed to be an expression of a certain processing model (Chapter II), which, in turn, can be used to infer on whose behalf a specific OS abstraction (e.g., a process) is being used. In this chapter, it is assumed that the processing model of different applications is known in advance, to allow system administrators to set up appropriate activity-tracking, and message-tagging rules. For example, administrators may specify rules like: "If process P_1 `accept`s a service request from VS_x, the resulting P_1 activity should be charged to VS_x." If this behavior is not known in advance, the Performance Mpas introduced in Chapter V uncover the mechanisms that are used to relay work between service instances.

Each message and process is tagged with a VS identifier (VSID) to indicate the resource partition to which it should be charged. The VSID does not represent each individual activity but a class or type of activity (e.g., high-priority work). This is a reasonable approach because it is difficult to administer resource constraints for individual activities. Once some activity that uses the OS is classified as belonging to some VS, this VS association is maintained, regardless of the process context in which the activity continues. This means that the resource bindings for back-end services are delayed until it is known who they work for. This automatic and delayed resource binding enables insulation between services, in spite of shared back-end services. This capability also distinguishes VSs from other system partitioning approaches.

In the VS architecture, the dynamic binding of activities to VSs is inferred by intercepting system calls within the OS using *classification gates* and analyzing the information that is passed to the function call (see Figure 4.2). Gates intercept and track work that propagates from one service to another and are configured by the system administrator via simple rules. They automate

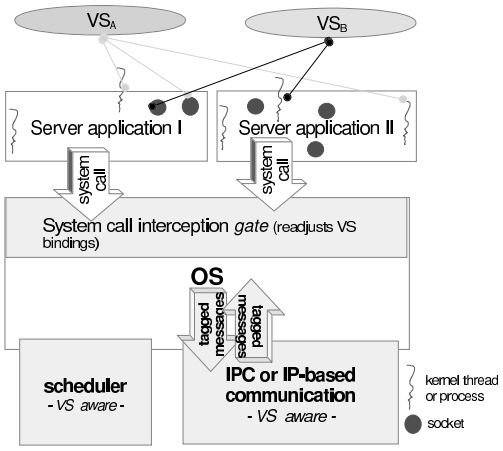


Figure 4.2: Virtual Service architecture

the binding of resources (e.g., newly-created processes and sockets) to VSs' resource partitions, and keep track of any work done on behalf of a VS.

Obvious problems occur when applications utilize special threading and communication middleware that multiplexes OS abstractions among multiple activities. This chapter does not explicitly address how to modify such middleware or runtime to allow VS-based performance control. However, the extension of the OS-level tracking mechanisms to arbitrary communication and threading middleware is straightforward. The OS modifications of this chapter should be viewed as an instance of providing performance control by modifying a general-purpose runtime system.

Other approaches that permit changing the resource bindings of processes and other system abstractions [15, 27] do not consider the problem of shared back-end services that do not readjust resource bindings on their own. As Figure 4.2 shows, the transparent VS architecture brings resource management to such applications.

Summarized below are the key features of the VS architecture:

- Dynamic resource bindings for shared services based on high-level model of works-for relation
- Separate application logic (what needs to be done) from resource management (how to charge the work that needs to be done)
- Applications may use several OS mechanisms to relay work to each other
- Minimal interference between competing VSs
- Modular implementation permits trade-off between the quality of insulation and the overhead incurred by the VS abstraction

Section 4.2 summarizes performance-management-related work. Section 4.3 introduces some additional design constraints and terminology. Section 4.4 details the state that is maintained for each VS. Section 4.5 describes the classification of activities in a VS-enabled system. Section 4.6 describes the dynamic tracing rules that are to be instantiated to track activities as they utilize resources at multiple hosts. Section 4.7 details the resource allocation enforcement based on the dynamic mapping of system object to VS resource partitions. Section 4.8 describes a Linux-based

	Dynamic System Domains	Eclipse BSD	HS	RC	Resource Manager	Scout	VS	WLM
OS	Solaris	Eclipse	Solaris	Digital Unix	Solaris	Scout	Linux	OS/390
Focus	VH on multi-processor	VH	Mulimedia end-host	abstract VH	per-user VH	Multimedia end-host	per-service resources	workload mgmt
Single server	Y	Y	Y	Y	Y	Y	Y	Y
Server farm	Multi-processor	N	N	N (pot. Y)	Multi-processor	N	Y	Y
CPU control	Y #CPU's/VH	Y	Y	Y	Y	program-mable	Y	priorities
Net control	Y (coarse)	Y	N	indirect	N	program-mable	Y	N
Disk control	Y (coarse)	Y	N	N	N	program-mable	N	Y
Memory control	Y (coarse)	N	N	N	Y	program-mable	Not yet	Y
Inference of resource principal	VH	N	N	some TCP-based	Unix user/group	N	rule-based	app-based
Recognize work delegation	N	N	N	N	N	hardwired into OS	Y intercept syscall	N
Application changes	Y	N	Y	Y	Y	OS part of application	N	Y

Table 4.1: Resource- and service-oriented server management solutions

prototype implementation. Section 4.9 presents experimental results and quantitatively shows that the VS abstraction solves the problems faced by application and Internet service providers (SPs) that co-host services. Section 4.10 describes limitations of the proposed approaches in a heavily-loaded, multi-tier network deployment and proposes an important front-end traffic control extension for VSs to enforce resource partitions even during prolonged demand spikes. Section 4.11 demonstrates the efficacy of the proposed load-shedding approach. Section 4.12 summarizes our findings, comments on possible future extensions, and states the relevance of VS with respect to this thesis.

4.2 Related Resource Control Approaches

There are two approaches to server management: *resource-oriented* and *service-oriented*. Resource-oriented approaches like Resource Containers [15], Eclipse [26, 27], Capacity Reserves [94], and Hierarchical Scheduler [59] provide necessary low-level support for the partitioning of resources. Furthermore, they support relatively static bindings of resource consumers to these partitions. VS and Workload Manager for MVS [6] are service-oriented. They charge services and client classes for their resource usage instead of creating static resource partition bindings for entities like processes, users, or sockets. Table 4.1 characterizes the properties of related approaches. This figure also highlights the novel features of VS.

Resource Containers (RCs) separate traditional OS abstractions from the OS's resource control functionality. Each application must identify the context of an RC to which its current resource consumption should be charged. The RC may contain basic CPU and network shares and various count limits on the number of resource consumers that can be bound to it. To control application performance, processes must explicitly bind to an RC. Subsequent activities are charged to the associated RC, and resource limits specified therein are enforced. Unlike VS, the RC abstraction does not automate the binding of resource consumers to RC's. Thus, the authors [15] have provided a novel abstraction without addressing the problem of binding processes and sockets to this resource abstraction in a dynamic manner that avoids extensive reimplementation of service applications and modifications of interfaces between peer applications.

The Solaris Resource Manager [141] is based on a resource reservation concept (called `lnodes`) which is equivalent to RC. In addition to the resource-reservation abstraction, `lnodes` are tagged with Unix user-group affiliations so that the resource context can be inferred from the user-group setting of current application activity. This mechanism reduces the need for manual resource bindings. The idea is to give each user-ID its own machine. Unfortunately, this concept fails if shared services do not change their user-ID when they work on behalf of different users. For example, the system's DNS server does not change its user-ID to that of the process requesting address resolution. Furthermore, this approach fails to take proactive steps to avoid resource depletion when critical sections are invoked — that is the Priority Inversion Problem [82].

In the context of Eclipse's hierarchical reservation domains [27], Bruno *et al.* discuss in [26] how Eclipse tackles the problem of sharing specific OS entities such as sockets among concurrent applications. Interference can be reduced by tagging each request that utilizes a shared resource with the appropriate reservation domain, thus delaying the resource binding of the shared OS abstraction. Request tagging is also used by VS. Unlike VS, Eclipse does not infer the tag for a request in the absence of application support and does not exploit these for the scheduling of an application that picks up a tagged request. Precursors of this work are the Hierarchical Scheduler (HS) [59] with configurable CPU scheduling policies and the Nemesis OS [64]. Nemesis provides comprehensive inter-application isolation for memory and file system. Both HS and Nemesis require applications to explicitly manage their own resource bindings. Like RCs, these approaches only create a resource partition abstraction.

Workload Manager's (WLM's) [6] notion of a *service class* is similar to the notion of a VS. Since WLM manages requests separately according to their *service class*, service sharing does not necessarily cause interference. Nevertheless, classifying requests into service classes is the hard part. For this purpose, IBM modified OS/390's services to classify all requests into service classes. This approach does not work for the multi-tiered scenario that was introduced in Chapter II since it is not desirable to modify hosted applications. Therefore, VSs must provide a transparent work classification mechanism.

Scout [129] takes an approach that differs from all previously-discussed approaches since it is primarily designed to be used in embedded multimedia server designs. Scout's path abstraction tracks the flow of work across different OS layers. Resources are reserved on a per-path basis. Since path abstractions are compiled into the kernel, resource consumption scenarios cannot change dynamically. For every new resource consumption scenario (i.e., new applications) the Scout kernel must be recompiled. In contrast, VSs can be configured dynamically to handle new scenarios of resource consumption and service interaction.

Sun's Dynamic Enterprise 1000 [142], Solaris Resource Manager [141], Ensim's recent VH product ServerXchange [51], and VMWare [147] are noteworthy commercial VH implementations resembling Eclipse. Other popular commercial solutions, such as Cisco's LocalDirector [32], Hy-

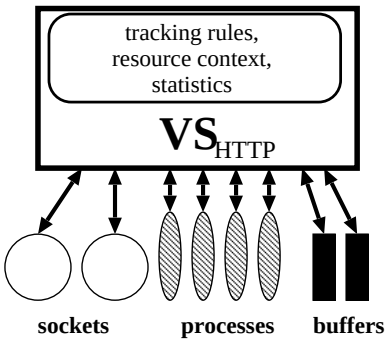


Figure 4.3: A VS and its members

draWeb [69], and F5's BigIP [52] are geared towards increasing the capacity available to ASPs through load-sharing in server clusters. These solutions also provide some coarse insulation between co-hosted services by shaping request flows. These mechanisms fail to fully utilize server resources because they enforce only static traffic-shaping rules. Hence, insulation fails when the workload created by individual requests differs significantly among co-hosted sites. Moreover, the insulation provided by such approaches is coarse at best because the amount of work required to service a request is not always obvious from the request itself. This is especially true when requests may access back-end services.

4.3 Additional Design Constraints and Assumptions

The VS approach tags OS entities, such as processes, sockets, IPC shared memory segments, etc., with a VS identifier. In an SDI-enabled system the VSID would be implemented as a straightforward context attribute. However, in systems that do not support SDI but still wish to support VSs, it is necessary to tag actual system objects, which requires a simple kernel extension.

4.4 The Virtual Service Abstraction

The VS abstraction is the state that we would like to associate with system objects to ensure that their processing, communication, and possibly I/O requirements are mapped to the appropriate resource partition.

The VS abstraction treats multi-tier services as if they were a single application executing on its own dedicated server. To create this illusion, a VS is associated with a basic resource context (Figure 4.3). The resource context summarizes the resource limits and statistics for activities that execute on behalf of the VS.

Each VS is uniquely identified by its descriptor (Figure 4.4). To allow the system administrator to manage VSs, each VS has an integer virtual service identifier (VSID). The VSID is guaranteed to be unique on each machine. For VSs that use resources on multiple machines, it is up to user-space software to guarantee the uniqueness of identifiers. When setting up the distributed service, external administration software can force a specific VSID onto the newly-created VS. Such global VSIDs are taken from their own number range and will never conflict with local VSIDs.

Like RCs [15] VSs propose hierarchically-nested resource contexts. Hierarchy is necessary because SP's clients should be able to decide themselves whether they want all of their services to share resources or they want to insulate them from each other.

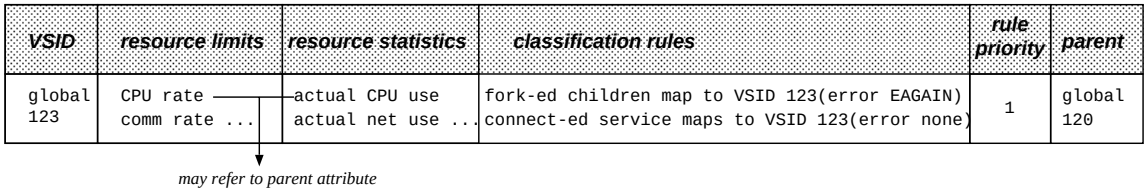


Figure 4.4: The VS descriptor

The parent field of the VS structure points to the parent VS. Oftentimes parent VSs will be used to implement abstract VSs, i.e., placeholder services to which all services of an SP's business client belong. The highest-level VS is the `root_vs` with VSID 0, which accounts for all unclassified work.

Hierarchy is again reflected in VS attributes such as resource usage statistics and resource limits. By default, newly-created VSs share the attributes, i.e., resource control settings and statistics (CPU time used, packets sent, etc.) of their parents. This means that the fields in the child refer to its parent's pendants (Figure 4.4). To manage the child service directly, attributes of interest need to be detached from the parent. For example, to control the number of processes for a backend service, one detaches the backend service's process count limit via the `vsctl(DETACH_PROCESS_LIMIT, VSID)` call.

To instantiate a cluster-wide VS, the administration software must create VS descriptors with one global VSID on all cluster nodes. On each of those nodes, local resources may be reserved using the VS descriptor's resource context. Before reserving VS resources, the administration software will monitor the VS's resource consumption via the statistical VS attributes or Performance Maps (Chapter V). Once enough statistics are available, resource reservations will be calculated to stabilize VS performance.

Most of the VS state discussed so far could potentially be realized using RC's [15] or Reservation Domains [27]. However, they do not provide configurable classification rules. Classification rules indicate how VS-membership is to be updated when certain system calls are invoked by specific VSs. As was explained in Chapter II each computation implements a service and communication model and its interaction with the OS and middleware is an indicator of the implemented model. For instance, if a process member of the VS in Figure 4.4 calls `fork`, the OS knows exactly that this is a way of relaying work and that the created process should inherit the parent's VS affiliation. The implemented service model directly impacts the way in which the members of a VS are to be tracked.

4.5 Determining Virtual Service Membership: Classification

There are two ways of assigning members to a VS: either they are announced or the OS infers who they are. For VSs membership is mostly tracked by the OS without requiring continuous application or administrator intervention (rule-based classification). Nevertheless, especially at service startup time it can be efficient to create some associations between VSs and other OS entities explicitly (manual classification). For example, if one knows that one specific kind of service request (identified by its own VS) always enters the system through one specific process or socket, a manual classification of these processes or sockets as members of a specific VS should be used. This avoids having the OS infer VS bindings repeatedly, thus reducing overall overheads.

Incoming, unclassified packets are classified using packet matching rules that resemble firewall directives. The only difference between a firewall classifier and packet classification for VSs is that the policy part of the firewall rule is replaced with a VSID mapping. For example, the rule `source=10.10.0.*` $\rightarrow$ VSID$_i$ would map all packets comming from source 10.10.0.* to VSID$_i$ (see Section 3.4).

As a VS begins to respond to requests, new sockets, processes, and IPC resources may be created. Each of them must be associated with a VS because they incur system load and are used to relay work. Usually these new members cannot be added explicitly since the administrator does not even know of their existence and the application does not cooperate with the VS abstraction. Therefore, membership for these new entities is implicitly determined by the classification rules.

Not only do new entities need to be associated with a VS, but VS-membership may also change over time. For instance, if some process is observed to be operating on a particular data set that is characteristic of some separately-managed VS, the process is added to that VS and removed from its current VS.

Classification, i.e., associating an OS entity with a VS, takes place when the OS can infer something about the application, i.e., at system call time at tap points. One can limit VS-membership inference to those times because we assumed in Chapter II and Section 4.3 that VSs interact with each other over a limited set of OS mechanisms. This means that the *works-for* relation cannot change unless a system call is invoked. Therefore, there is no need to update VS-membership at any other time.

The **classification rules** that the OS examines at system call interception consist of a conditional clause, which defines when the classification rule is applicable, and a classification directive. This is formalized as:

$$(\texttt{syscall}, S_1, \ldots, S_m, P_1, \ldots, P_n) \rightarrow (S'_1, \ldots, S'_m)$$

where S_i represents the VS of the i-th affected entity. For example, the only affected entity in the `exec` call would be the calling process. The calling process's VS is always identified by S_1. P_j represents the j-th intercepted property, for example, the program name passed to `exec` or the incoming IP address of an `accept`ed connection. Properties are not necessarily OS entities. A classification rule also specifies S'_i: the resulting VS classification of the i-th affected entity. This classification is applied only if the conditional (left-hand side of the rule) matches the intercepted system call. S_i and P_j may be wildcards. The prototype implementation requires S_1 to be specific. The system call is always specific, since the dimensionality of the condition tuple depends on it.

The key system functionalities that need to be intercepted to correctly classify system objects are the network communication — to classify incoming packets — and state *transformation* system calls that reveal detailed information about the workload that is currently executing.

Whenever the kernel intercepts a characteristic argument to a system call, it is possible to classify the caller and other affected entities more accurately. For instance, the program name in the `exec`-call allows a more accurate VS classification of the active process if the program is typical of a specific VS. Other frequently-used system calls that affect VS classifications are `setgid`, `setpgrp`, and `setuid`.

Conflicting rules: Rule matching can lead to ambiguity. Multiple condition tuples may match the current system call interception. To solve this problem, VSs are ranked. The rule that matches the highest-priority VS explicitly is used to determine the resulting classification. Should there be a tie between several rules, the most specific rule is applied. If this does not resolve ambiguity, the result is unspecified.

Category	System call	Proposed classification mechanism	Proposed VS error	Used by
creation	fork	Classify new process based on forker's VS	EAGAIN, block	Multi-threaded services
	open	Tag created file descriptors with a service affiliation based on creator's VS	block	User services, FCGI
	socket			All network services
	pipe			User services, FCGI
	shmctl	Tag new shared memory segment based on creator's VS	EINVAL, block	
communication	accept	Classify caller based on its VS, the VS of the incoming connection and incoming source address	EWOULDBLOCK, block	User services, FCGI
	connect	Classify caller and connecting socket based on the destination IP + port	EINPROGRESS, block	Frontend services, HTTPD with distributed FCGI
	send	Classify caller and sending socket based on destination IP + port	EWOULDBLOCK, block	Frontend
	sendmsg			
	sendto			
	recvmsg	Classify caller and receiving socket based on incoming packet's VS and IP + port		NFS, DNS, RPC, Multimedia
	recvfrom			
	recv			
	shmat	Classify caller based on shared memory's VS	EACCESS, block	Apache (thread synch)
	msgsnd	Tag message based on caller's VS	block	Proprietary services
	msgrecv	Classify caller based on the incoming message's VS	ENOMSG, block	
synchronization	semop	Classify caller based on the semaphore's VS	block	Proprietary services
transformation	exec	Classify caller based on the executed program's name	block	HTTP-CGI, Inetd, rexec, rsh
	listen	Classify caller based on its and socket's VS		TCP + Unix Domain socket-based services (Fast CGI)
	bind	Classify caller and socket based on the IP + port pair		Standard services
communication, synchronization, transformation	write	Tag the message based on caller's VS or inherit file descriptor VS	EAGAIN, block	All services
	read	Classify caller based on read message's VS or inherit file descriptor VS		

Table 4.2: System calls that affect VS-membership

4.6 Tracking Virtual Service Membership

Once the correct VSID for a process or socket has been determined, it becomes necessary to track the VSID along with the propagation of the activity that is associated with that VSID. The system calls and functions that are relevant with respect to the tracking of activity propagation fall into three major categories: creation of system objects, communication, and synchronization (see Table 4.2).

Creation: If an entity, A, creates another OS entity, B, B's future VS affiliation depends solely on A's VS affiliation. Examples are the creation of sockets, IPC shared memory segments, message queues, pipes, and the like. The canonical default rule is for the created entity to inherit its creator's VS affiliation by copying the VSID from the creator's context into the created object's context.

Communication: Communication is used to relay work within and beyond machine boundaries. Therefore, intercepting intra-VS communication is essential to VS maintenance in server farms. If it is possible to determine the VS affiliation of each request that is picked up by a service, the resulting activity can be charged to the correct VS. This does not depend on whether the request originated locally or remotely.

Communication affects at least three entities: *sender*, *receiver*, and the *message* itself. To make inter-process communication more efficient, most OSs implement asynchronous communication as opposed to the *rendezvous* concept. This adds sockets, pipes, and the like to the set of affected entities, each of which may be reclassified upon system call interception.

Due to the temporal separation between sending and receiving of a message, reclassification

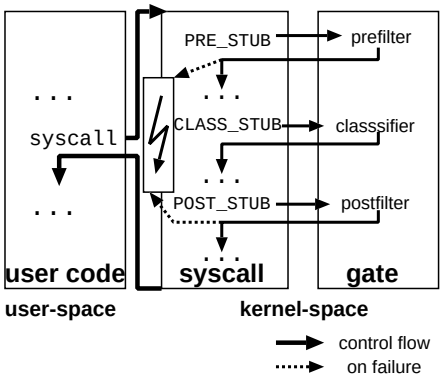


Figure 4.5: Gated system calls

of the affected entities is difficult. Therefore, communication-based VS tracking is done in two stages. First, when a message leaves the sender it is tagged with a VS affiliation, much like what is done in the case of creation-type calls. This can be skipped if the communication is a one-to-one connection. In this case, the connection itself is labeled at setup time with a VS affiliation that implicitly applies to each message that passes through it. The second stage is message consumption. At this time, the receiver's VS affiliation may change based on its previous VS affiliation and the received message's VS affiliation.

Synchronization: The set of affected entities in synchronization includes the executing processes and all processes in the wait queue for the synchronization primitive. Activities that are performed under the protection of a synchronization primitive may be associated with its VS.

Synchronization can also be used to infer collaboration among a set of processes. Previously-unclassified processes may inherit the VS affiliation of the synchronization primitive. This is an effective tool since many multi-threaded server applications expose their process sets when they synchronize for thread control purposes.

The process(es) that execute under the protection of the synchronization primitive must not be allowed to stall processes in the wait queue that have unused resource allowances because otherwise, priority inversion [82] will result. This is also a problem when a single-threaded backend service is shared among several VSs. This will be discussed further in Section 4.8 (`accept`).

4.7 Enforcing Per-Virtual Service Resource Quotas

4.7.1 System Call Gates

Whenever a VS receives a new member either because of classification or as the result of activity propagation, its resource limits could potentially be violated. This means that classification and resource limit enforcement are inseparable. To this end, this thesis introduces control *gates*, a combination of system call filtering and VS classification. Each system call that is used to track VS-membership is controlled by a gate.

If the gate's filtering code indicates a resource limit violation as a result of the new classification, the system call will either fail with an administrator specified `errno` code, block, or execute in best-effort mode. Otherwise, VS-membership is updated as specified in the classification rules. Figure 4.5 depicts the basic anatomy of a gate:

1. The *prefilter* checks if the caller is (a) classified and (b) if its VS affiliation permits the execution of the gated call.

2. The *classifier* applies a matching classification rule. To execute the classifier for creation-type calls it executes after the new resource has been created.

3. Finally, the *postfilter* checks whether the resulting classification violates any VS resource limits. The resource limits that are considered are: count limits on the number of processes and sockets. Other resource limits, such as CPU and network bandwidth are enforced silently by the packet and CPU schedulers and need be checked by the gate mechanism. If a resource limit is violated, the system call fails or retries as is described in the next section.

4.7.2 Failing System Calls

Gates may detect resource limit violations. For example, during the execution of `fork` it may become apparent that successful call completion would result in a violation the VS's process count limit. The appropriate remedy is application-dependent. One may decide to:

1. Wait until VS resources become available.

2. Return an error to the caller indicating resource exhaustion.

3. Not apply the classification that led to the resource limit violation and silently reclassify the caller as best-effort. If the best-effort VS has exhausted its resource share, there is no other option but to fail the system call.

In the first case, the OS will add the caller to a FIFO wait queue for the requested resource. For example, in the case of `fork` this means that the forker will sleep until the VS's process count drops below its process count limit. The resulting delay may not be acceptable to the calling application.

Applications that cannot be delayed, should receive an error upon resource exhaustion. Unfortunately, existing applications may not be able to handle arbitrary errors. Therefore, it is up to the administrator to configure the error that will be raised if a VS-level resource limit violation is observed at a particular gate. In this way, only errors that the application is able to handle will be raised. For example, the administrator may choose to raise the `EAGAIN` error for some VS that exceeds its process count limit upon `fork`. This behavior is specified at the time of gate configuration [e.g., `vsctl(SET_FORK_POLICY, VSID`$_x$`, ..., EAGAIN, ...)`]. Most server applications are capable of handling errors that result from resource exhaustion gracefully. They simply record the error in a server log-file to support system tuning. If neither blocking nor returning an error is acceptable to the hosted service, the execution should continue in best-effort mode (VSID 0).

4.8 Implementation

The VS abstraction is implemented in loadable modules for the 2.0.36 version of the Linux kernel. Figure 4.6 shows dependencies among the VS modules. To implement the gates, only a few lines of call-back code need to be added to the intercepted system calls to trigger VS classification. The VS structure itself (Figure 4.8) contains the previously-described membership information, statistics, and resource limits. The VS structure, VS hierarchy management and most of the gates should be relatively easy to port to other OSs, since they only minimally depend on Linux internals.

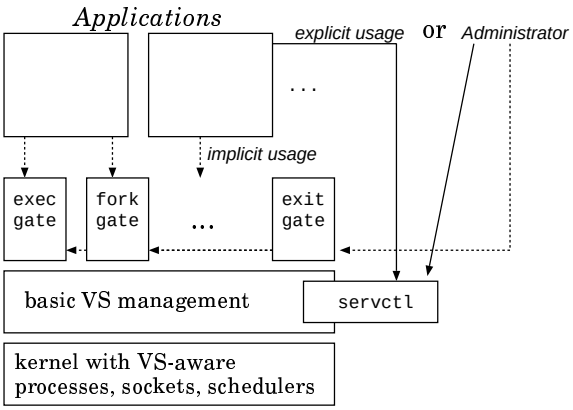


Figure 4.6: VS module dependencies

The placement of the call-back code in the original system calls is Linux-specific and needs to be revisited for every target OS/middle-ware-system despite the fact that the required modifications would be very similar.

VS-level *fair-shares* [77, 94] for CPU and network are implemented to provide strict VS-level resource guarantees. VSs that are neither directly nor indirectly (via a parent VS) associated with a share are scheduled on a best-effort basis. Best-effort VSs use all unreserved resource slots. Any excess capacity is shared between VSs that own resource shares in a round-robin fashion (see also firm Capacity Reserves [101]). The implementation of VS resource shares is not portable across platforms. Nevertheless, numerous implementations of capacity reserves and fair-shares exist. Therefore, requiring VS-level fair-shares does not limit the applicability of the VS approach.

VS statistics are cumulative aggregates of the VS's members' statistics. The attributes include a wide range of statistics that Linux keeps for processes and sockets, such as page faults and virtual time elapsed.

To set up the VS hierarchy and adjust CPU limits, VS membership, policies, attribute inheritance, resource limits, and query VS attributes, the OS offers a new system call (`vsctl`). It takes a command, the size of the argument, and an argument structure as parameters.

Gates are implemented as loadable modules. Currently supported are `fork`, `exit`, `exec`, `open`, `accept`, and `socket` gates. Upon insertion of a gate module, the call-back stubs that are placed in their corresponding system calls are activated so that the gate's *prefilter*, *postfilter*, and *classifier* are executed each time the control-flow of a server application passes through the intercepted system calls. Each gate also registers its own `vsctl`-handler to enable gate configuration.

The advantage of a modular gate design is that one only needs to add those gates to the kernel that are absolutely necessary to classify VS membership and insulate services. This is very important because the insertion of each gate into a running kernel increases system overhead slightly (see Section 4.9 for more detail). The remainder of this section describes the implemented gates.

Fork: Upon interception of `fork` the created process is classified as a new member of some VS. To determine the resulting VS affiliation, one must check the `fork_policy` object of the creator's VS. The `map_to` attribute of the `fork_policy` specifies the affiliation of the created process. If the VS specified by `map_to` has reached its process count limit (set via the `vsctl` call), the failure behavior that was configured for that VS is invoked (Section 4.7.2). Figure 4.7 shows a high-level control-flow graph for this gated system call.

Exit: If a process exits — including ungraceful `SIGSEGV` and other uncaught signal exits — it

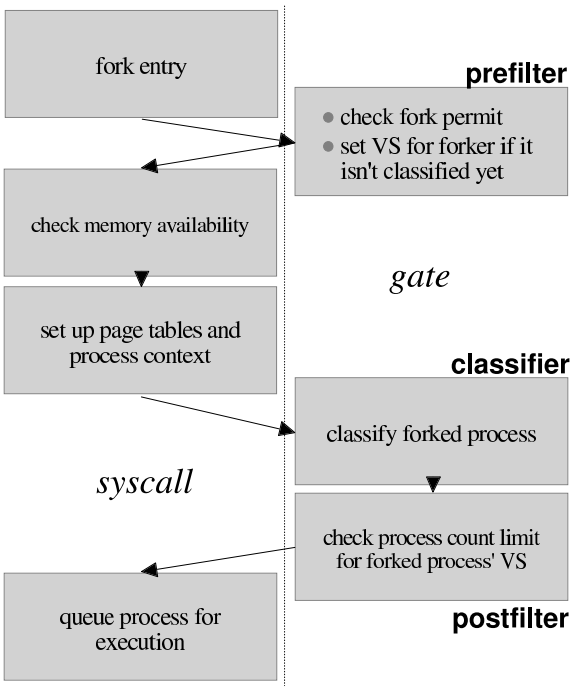


Figure 4.7: Control-flow of the fork gate

must be removed from the VS with which it is associated. This gate is not configurable.

Exec: Upon calling one of the `exec`-family system calls, the caller can be reclassified based on the name of the program that was invoked. The gate code checks the name of the program against a hashed mapping table, i.e., the `exec_policy` field in Figure 4.8.

Open: The `open` gate acts like the `exec` gate. The only difference is that the file descriptor may be tagged with a VS affiliation at the same time. Moreover, the `open` gate uses a prefix-tree to match the file names. Thus, whole directories — identified by a shared prefix — can map to one VS. This is important because large numbers of data files residing in one directory subtree may yield identical VS classifications.

Socket: The socket gate resembles the `fork` gate. The `socket_policy` of a VS specifies the future VS affiliation of the created socket. Once messages are relayed via such a classified socket, they are tagged with the VS affiliation of the socket in their IP Type-of-Service field (TOS), thus allowing VS information to propagate through the network. Since the TOS field may be used by DiffServ to provide differential QoS in a WAN, this field can only be used inside server clusters. If the TOS field cannot be used or one needs more than 256 VSIDs (the TOS is eight bits wide), one may introduce a new IP-option [110] to hold the VSID. In fact, our most recent adaptation of the code, overcomes the limitations of this first prototype, by utilizing a 32-bit-wide VSID IP option to mark network packets. If the prototype implementation was directly layered atop SDI then a 32-bit wide VSID would have been the default.

Close: Closed file descriptors' and sockets' VS affiliation must be removed.

Accept: The `accept` gate is quite complex. It first determines the highest-priority VS among the caller, listening socket, and incoming connection (see Figure 4.4). Then the winning VS structure is checked for a VS mapping based on the incoming IP address and the VS affiliation of the listen-socket, process, and incoming connection. The VS affiliation of the incoming connection can only

```
struct service_struct {

            int sid;
            struct service_struct *parent;
            char name[MAX_SERVICE_NAME_LEN];
            int precedence;

            // int_or_ptr is either a value or a
            // pointer to the parent's int_or_ptr
            int_or_ptr process_count;
            int_or_ptr socket_count;
            int_or_ptr byte_count;
            int_or_ptr vtime;
            int_or_ptr majflt;
            int_or_ptr minflt;

            member_struct *processes;
            member_struct *sockets;
            member_struct *services;
            fork_policy_struct fork_policy;
            exec_policy_struct exec_policy; ... more ...
            cpu_policy_struct cpu_policy;
            comm_policy_struct comm_policy;
};
```

Figure 4.8: The VS struct

be determined if it was initiated by another server with VS support and its VSID is from the global VSID range. The VSID is stored in the incoming SYN packet's IP TOS bits. For local `accept`s, the VS of the incoming connection is the connecting socket's VS affiliation. Both socket and receiver may be reclassified.

The difficulty with `accept` is that it should not block if the first pending connection on the listen queue leads to a violation of resource limits. There may be a connection that can be `accept`ed without violating any VS resource limits. Therefore, the implementation scans the listen queue for the incoming connection whose VS has utilized its resource reservation the least.

Concurrent Gate Versions: A powerful implementation feature is to allow multiple versions of a gate to be loaded at the same time. Hence, VS rules may specify which gate version they want to use when their process members invoke the corresponding gated system call. This way it is possible to eliminate unnecessary checks for specific VSs. For example, if `fork`ed-off processes should always inherit their parents' VS affiliation, it is unnecessary to check for a (`fork`, VSID_x) $\rightarrow$ VSID_x mapping as is required for general VS classification. One can implement one `fork` gate version that always applies the parent's VS affiliation to the `fork`ed child. Another example is the `accept` gate, which is quite complex in its general form (find a VSID mapping for incoming IP-header). In a server-farm setup it is likely that incoming service requests are already classified by the front ends and that the applications that process requests in the back ends only need to inherit these classifications. Since the frontends' classifications are propagated with every communication packet, the back ends' `accept` gates only need to apply the incoming connection's VSID to the accepting process. Such an optimized version of the accept gate is used in the experimental evaluation of VSs. In the experiments, incoming requests are classified as they are picked up by the HTTP server. Whenever the HTTP server relays work to a shared back-end Fast-CGI (FCGI) service, the back-end FCGI inherits the classification of the requesting HTTP server process.

Classifying Incoming Traffic: Traffic that enters a server from the outside is typically not classified, since client machines are not VS-aware. To allow charging network stack processing to the applicable resource quota or to allow sophisticated classification rules at the `accept` and `recv` gates that take a message's VS classification into account, it is necessary to provide a simple tagging facility. To this end, the kernel provides a simple firewall extension that is configured in the manner of `ipchains`. Each rule consists of an IP + TCP|UDP packet property matching clause and a mapping to a VSID. Whenever an incoming packet that matches a specific rule is found, its

`sk_buff` data structure is marked as a member of the identified VS.

If multiple rules match a packet, then the one that was installed first will be applied to the incoming packet. An example rule could be specified as follows:

```
vsfilter -p TCP -y --sa 10.0.0.0 --sam 255.255.255.0 --da 10.1.1.1 \
         --dam 255.255.255.255 --vsid 10
```

This rule specifies that TCP SYN packets from subnet 10.0.0.* destined for server 10.1.1.1 should be mapped to VSID 10. These markings do not interfere with classical firewall operation because `ipchains` allows a chain of firewalls to be registered. The classification module simply returns `FW_SKIP` to indicate that it did not decide to drop the incoming packet but that other firewalling modules should continue checking the packet.

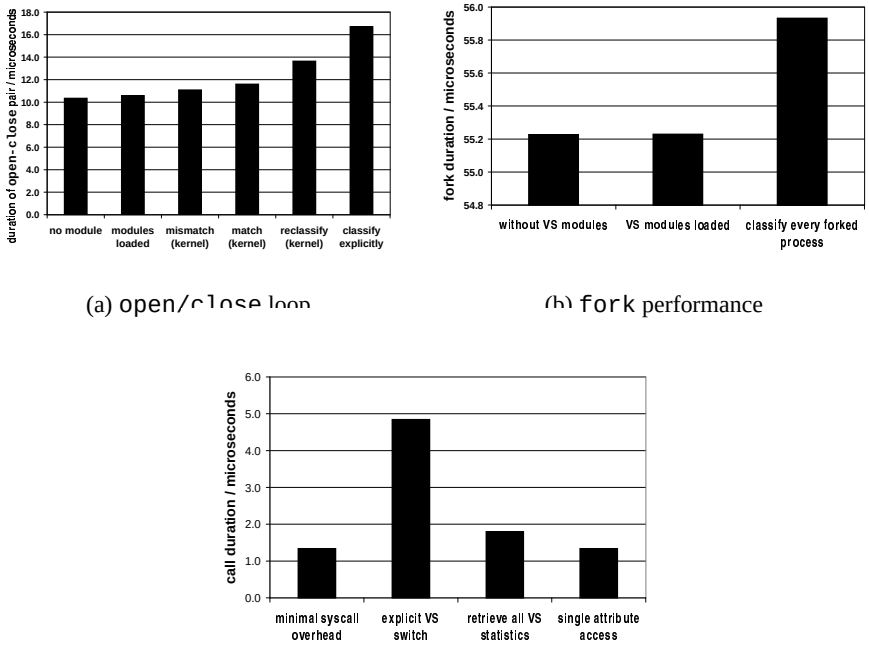
4.9 Evaluation

The performance of the VS architecture is measured on a small Web server running on a Dell 450 MHz Intel Pentium II PC with 448 MB RAM and one UDMA HDD. The clients, three 300 MHz Pentium II machines with 128 MB RAM each, are connected through 100Mbps Ethernet. The measurements are with respect to Apache 1.3.6 (HTTP 1.1) on the Linux 2.0.36 kernel. The workload is generated by the commercial SpecWeb99 benchmark [132]. SpecWeb99 attempts to model a realistic workload including 30% dynamic requests. The size of the file sets grows linearly with the number of simultaneous connections offered to the Web site. Therefore, it generally does not completely fit into the server's file cache. Dynamic requests and the use of Apache explain the low HTTP throughput of the server (ca. 220 ops/s). Since the VS abstraction is an application-transparent mechanism, neither applications nor libraries had to be modified for these experiments. The management of the VS hierarchy and gate configuration is done from the command line using utilities that feed their arguments into the appropriate `vsctl` call.

4.9.1 Baseline Performance

Basic performance measurements show that the dynamic VS classification layer degrades overall system performance only minimally. If one intercepts a complex system call like `fork`, the overhead of classifying the new process is small — only 1.3% — (see Figure 4.9(b), classify). Nevertheless, the raw performance of intercepted system calls can decrease significantly if the intercepted call is very simple like `open`. A 30% cost increase for the `open`/`close` pair can be observed if the VS affiliation changes with every execution of the loop (reclassify in Figure 4.9(a)). Just finding a classification rule (match) or not finding one (mismatch) without reclassification is much cheaper. The high relative overhead for simple calls results from the almost constant classification and VS binding overheads. An important point shown in Figure 4.9(a) is that binding processes to VSs from user-space (explicit classification) performs much worse than kernel-based classification because of the system call overhead. Explicit classification requires the executing process to classify itself and the resources that it uses and creates by calling the `vsctl` system call.

The measurements compare the performance of a sample program using (implicit) classification rules against a modified version of the program that explicitly updates its own VS bindings. The performance numbers strongly support the use of kernel-based (implicit) classification. For the sake of completeness, Figure 4.9(c) summarizes the cost of querying VS attributes and administering the VS hierarchy.



(a) `open/close` loop

(b) `fork` performance

(c) VS management performance

Figure 4.9: Performance of intercepted and new system calls

To estimate the overall performance impact of the kernel modifications including scheduler changes, resource limit enforcement, and the cost of system call interception, Figure 4.10 shows how the performance of the Apache HTTP server [84] is affected by the OS changes. According to these measurements, the VS abstraction affects the system's HTTP performance only up to 2.5%, depending on the number of simultaneous client connections. The bi-modal shape of the performance loss graph in Figure 4.10 can be explained as follows. Apache keeps some spare processes alive to serve incoming connections faster. Once the number of simultaneous connections offered to Apache increases to an extent that there are not always enough of these spares, Apache begins to fork more connection-handling processes on-demand, which explains the increase in overhead up to 80 simultaneous connections. Beyond this point, the file system cache hit ratio goes down so that the low performance of the file system begins to dominate overall system performance, resulting in request queue overruns that decrease the relative impact of the previously outlined OS changes. The problem of request or incoming packet queue overruns is addressed separately in Section 4.10.

4.9.2 Implementing VHs using VSs

Another series of experiments on Apache shows that the VS abstraction may be used to insulate VHs. Unlike other applications, Apache itself provides some basic resource controls (process count limits) to insulate VHs. Common insulation techniques for collocated services are compared to the capabilities of VSs. The goal is to divide the previously-measured server into two VHs of equal capacity in both cases.

In the experiments, two copies of Apache are executed on the same host, each listening on

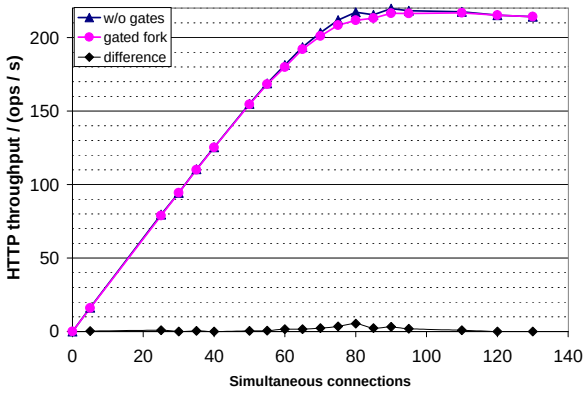
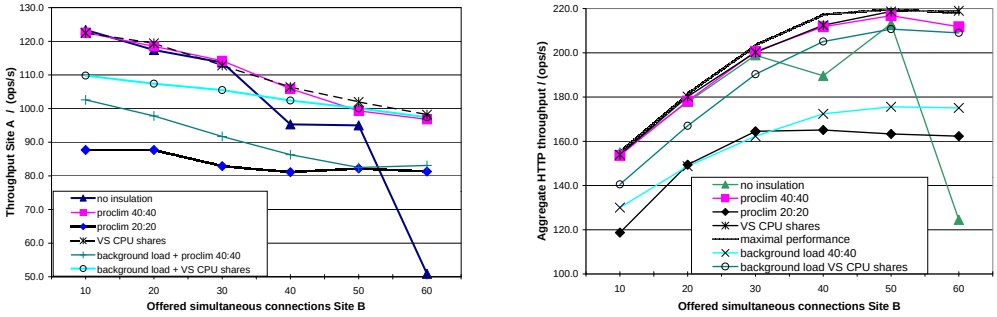


Figure 4.10: VS effect on HTTP throughput



(a) Site A performance in a two-site setup

(b) Aggregated performance in a two-site setup

Figure 4.11: Performance loss when hosting two sites of equal capacity on one server

its own IP address using IP aliasing for the Ethernet interface. Running two copies of Apache, each instance can be controlled by adjusting the `MaxClients` directive, which limits the number of concurrent sessions for each site. This is an effective means of performance insulation if the average work per HTTP operation is known for each site. Since both sites have their own copy of similar content, setting the `MaxClient` directive for the Apache servers to the same value yields acceptable insulation.

To test VS-based insulation, the Apache servers are launched as if they were executing on their own physical hosts (using very large process limits). Two VSs, `www1` and `www2` are instantiated with the following `fork` classification rules.

$$(\texttt{fork}, \texttt{www[1|2]}) \rightarrow (\texttt{www[1|2]})$$

Each site's initial `httpd` process is explicitly added to its corresponding VS via a simple command line utility:

```
$> vsaddprocess <VSID> <PID>
```

Each site is given a 50% CPU share.

In the measurements that are discussed here, Site A is offered a constant load of 40 simultaneous connections while Site B is offered between 10 and 60 simultaneous connections. These parameters are chosen to cause eventual server saturation at about 80 simultaneous connections.

Without insulation between the sites, A's performance degrades significantly once the server is offered a total of 70 simultaneous connections (A=40, B=30) [see Figure 4.11(a)]. From this point on, B begins to "steal" resources from A, thus contaminating the file cache to A's disadvantage. The lack of insulation can be fixed in Apache itself by restricting the maximal number of concurrent processes. This comes at the expense of some loss of aggregated performance under peak load [Figure 4.11(b)]. This loss is due to the fact that incoming requests must be rejected when the process limit is reached. This queuing phenomenon — for M/M/m/c systems described by the Erlang-loss formulas [153] — is especially evident when looking at the smaller process count limit (20:20). VS CPU shares eliminate this problem.

Apache's process limits also fail when background activities compete for CPU time, e.g., monitoring. To simulate the effects of background load, ten background load generators are invoked. As expected, aggregated performance and A's performance drop significantly if Apache's process limits are used for site insulation. In contrast, the VS abstraction keeps A's performance stable since only non-dedicated resource slots (beyond A's and B's resource limits) are used to process background load. Therefore, VS-based resource insulation outperforms conventional VH "insulation tricks."

One may argue that a modified, CPU-share-aware Apache could achieve the same quality of insulation. However, VSs obviate the need for modifying applications to get a better handle on performance management.

Since this experiment did not involve access to any shared services and work is relayed only from a parent process to its child, Eclipse or RC's could probably be tuned to perform just as well as VSs. Beyond establishing the competitiveness of the VS approach, the next set of experiments focuses on its novel contribution.

4.9.3 Insulation Despite Shared Backends

Instead of letting the sites A and B execute CGI scripts to serve the advertisement banners (part of the benchmark), a shared, single-threaded Fast-CGI server (FCGI) is used. Queuing theory suggests that the impact of this shared FCGI will be the worst when (a) it exhibits highly variant execution times and (b) a high percentage of requests are forwarded to it. Therefore, the FCGI server is modified to execute a busy wait cycle randomly chosen between 0 and 10 ms (uniformly distributed), before serving incoming requests for advertisement. Furthermore, the percentage of advertisement banners requests is raised from 13% to 30% on each site. Other dynamic requests are eliminated from the benchmark's workload. The Apache sites (A and B) used a TCP connection to retrieve advertisement from the shared FCGI service. The load offered to Site A is kept at 30 simultaneous connections while the load offered to Site B increased from 10 to 60 simultaneous connections — with the changed dynamic mix, the server saturates at a total of 60 simultaneous connections of offered load.

As in the experiments of the last section, two VSs (`www1` and `www2`) are created and each is assigned half of the server's capacity. The first experiment (Apache insulation only) executes the FCGI outside the VS context of either site so that it could utilize all unused server capacity.

In the second round (dynamic FCGI-to-VS binding) the additional `accept` and `socket` gates are loaded into the kernel to police access to the FCGI, which receives its requests via TCP. The following classification rules are instantiated:

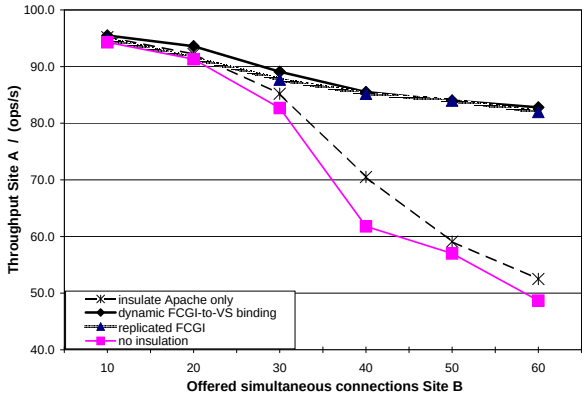


Figure 4.12: A's performance while increasing load on B

$$(\texttt{accept}, \texttt{www[1|2]}, \text{req} = \texttt{www[2|1]}) \rightarrow (\texttt{www[2|1]})$$

The `accept` rules cause the FCGI to change its resource binding if it is executing in the VS context of `www1` (`www2`) and receives a request from `www2` (`www1`) to `www2` (`www1`). Moreover, `accept` reorders requests in the order of their VS's remaining resource share as explained in Section 4.8. The default `socket` rule associates a new socket with its creator's VS. This ensures that requests sent to the shared FCGI will have appropriate TOS markings. To establish a baseline for optimal insulation, the FCGI script is replicated in each VS context (replicated FCGI). This cannot be done in real setups because some applications are not designed to be replicated, and others are too heavy-weighted to replicate.

As Figure 4.12 shows, sharing the FCGI without the `accept` gate breaks the insulation between sites A and B (Apache insulation only); the performance of Site A decreases rapidly as the load on Site B increases. This effect can be traced back to the contention for the shared FCGI. With the `accept` gate (dynamic FCGI-to-VS binding), the performance of Site A drops much slower, nearly at the pace for replicated FCGI. The benefit of using the `accept` gate is a performance improvement for the well-behaved site ("well-behaved" means that its clients do not overload the site) of approximately 60% under maximal load. Further experiments show that the `accept` gate for dynamic VS bindings performs almost as well as if the shared service were replicated for each VS (replicated FCGI). The ill-behaved Site B suffers from overloading its CPU share. This results in a 10% loss of aggregated performance compared to the ideal case of a replicated FCGI under peak load. The reason for this is that the ill-behaved site uses its resources mainly on serving static HTTP requests. Only when the number of queued-up FCGI requests is large will its FCGI requests be processed. During those times Site B operates mostly sequentially and some requests are lost because all server processes are in waiting state.

Without changing Apache this problem could not have been solved using RC's or any other approach presented in Section 4.2, because the resource binding for the FCGI must be dynamic, assuming it cannot be replicated.

4.10 Limitation of Virtual Service-Based Resource Control: Persistent Overload

So far, VSs seemed to be a straightforward application of the SDI principle to resource management. Activities are classified and mapped to VSIDs. Individual processes and other sytem

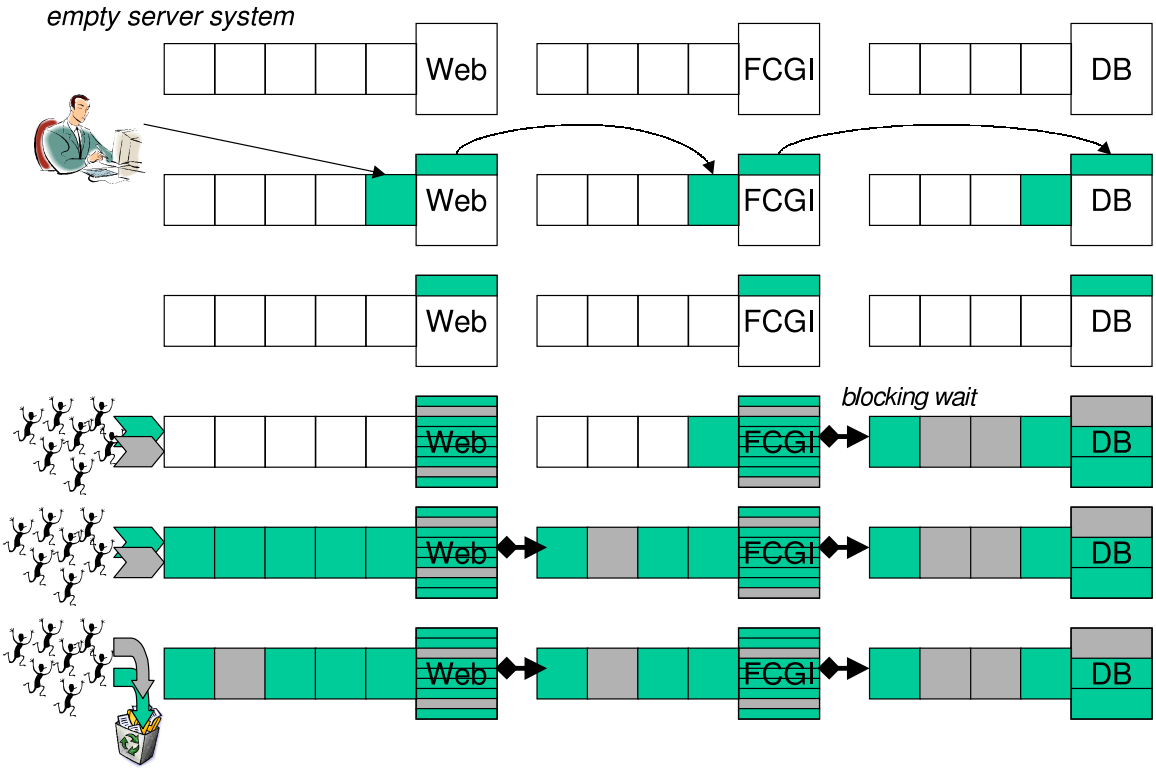


Figure 4.13: When requests exhaust one of the server's buffer limits, new arrivals will eventually be dropped indiscriminately. This leads to failure of resource-share enforcement.

objects are labelled to reflect the VSID of the activity to which they belong. Resource quotas are tracked and enforced by the OS in VS gates, i.e., stateful interpositions for resource control. Unfortunately, it is not enough to simply track and charge resource usage to the right resource budget at each local host. To achieve the division of resources specified in per-VS resource quotas, cluster-wide feedback and traffic control may become necessary.

4.10.1 The Problem

VSs are an effective means for insulating different customer types, front-end services, and otherwise identifiable workload from each other despite shared back-end services — unless overload conditions persist. The previous evaluation did not stress the system to the point where the server becomes persistently overloaded, i.e., it begins to drop incoming traffic.

Network servers are not usually persistently overloaded because system administrators attempt to carefully project capacity needs and size their systems' capacity to match workload requirements. Temporary overload situations that are due to the bursty, random nature of request arrivals at network servers can be absorbed by the server's internal buffers; VS-based resource insulation works well in this scenario. However, when the system experiences a prolonged overload condition (more than several seconds) servers will not be able to buffer incoming requests. For example, the terrorist acts of September 11, 2001 caused an extreme demand spike for web sites with news content. The best-known Internet news sites were unavailable most of that day due to excessive demand. It is this kind of a scenario that causes VS-based resource insulation to fail.

To better understand the problem that persistent overload causes for VSs one should view the

multi-tiered system as a queuing network with queue limits and blocking (Figure 4.13). Such a system is operational as long as its utilization, ρ, is less than 1. If $\rho \sim 1$ then the dependent server's queues fill up until finally the front-end servers' queues overrun, thus discarding incoming workload indiscriminately. This conjecture is an obvious inference from Little's law [153].

There are two principal solution approaches for the problem of queue overruns. First, one could remove the queuing limit of the applications or the network stacks by inserting an intermediary buffer for network traffic before it reaches the network stack processing functions. Unfortunately, this approach only works if the overload does not last very long because eventually server memory will be completely used up by stored network packets. The second approach, which is the approach chosen in this thesis, is to shed traffic proactively, i.e., before queues overrun. This shaping mechanism must be coordinated with the resource quota allotments for VSs because traffic shaping affects which VSs will receive requests and which ones will not.

4.10.2 Adaptive Traffic Shaping

The rationale for shaping incoming traffic is that the majority of work in network servers is caused by requests received over the network. Although any computer system executes some background, or scheduled work, most of its processing happens in response to its input. Since the primary input device for a network server is its network interface, shaping network requests directly reduces server load, thus preventing the failure of VS-based server partitioning goals.

In dropping incoming traffic to combat overload one is faced with three difficult problems: *when* to restrict incoming traffic, *what* traffic types to restrict, and *by how much* the incoming traffic should be reduced? There are no simple answers to these problems because the answers heavily depend on the workload, the specific setup, and capacity constraints of the server deployment. Therefore, this thesis proposes an adaptive traffic shaping mechanism that takes its cues for traffic-shaping from the difference of effectively achieved resource allocations and the resource partitioning configured for different VSs.

Identifying resource share enforcement problems: Section 4.9 shows that the VS approach is capable of differentiating between competing customers, but even the well-behaved customers experienced some performance degradation as the server's overall request load increases. While part of this degradation is inevitable due to cache pollution and memory exhaustion effects, the root cause of the problem is that the server accepts more work for the co-hosted site B than it should. The challenge is to effectively detect and prevent this condition without human intervention.

A naive approach to recognizing a resource share enforcement problem would be to set a threshold for CPU utilization and declare servers overloaded when their CPU utilization exceeds the threshold. A trivial example may show that this approach is incorrect. Assume two sites A and B are co-hosted on the same server. Server B receives no request but site A receives more requests than the server can handle. Using a simple threshold approach, an alarm would be raised. However, the alarm is unnecessary if the only objective is to insulate site B from site A. Insulation is trivially achieved despite high server load because site B experiences no demand. Hence, there is no need to shed any traffic in this scenario. Furthermore, in some cases a server may reach its processing capacity without any resource being exhausted. This can happen when the server software utilizes timers or waits for external services, e.g., a web service. In such a case a threshold scheme would not work correctly either. The resource share problems identified by the proposed traffic-shaping add-on for VSs is defined as follows.

Definition 1 (Failure of VS Insulation:) *A failure of VS insulation mechanisms is said to occur when two VSs compete for resources and there are requests of VS X that are not served despite*

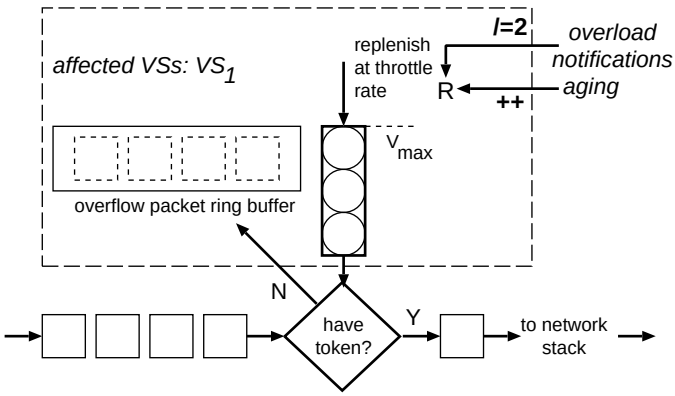


Figure 4.14: The request shaping rules are updated dynamically by overload notifications and aging. Excess traffic is buffered temporarily. This Figure only depicts one individual shaping-rule. A front-end server would usually be configured with many different rules, each with its own independent rate adaptation loop, target VSs, etc.

the fact that VS X still has not exhausted any of its resource quotas. Moreover, at the same time at least one other VS consumes more than its fair share of resources of some resource.

This definition can easily be reworded into a simple test procedure. The **necessary condition** for a failure of VS's resource insulation for VS X is the discarding of requests that would have been processed in the context of VS X. The **sufficient condition** is that VS X' has not reached any of its resource limits and that other competing VSs have consumed some resources at, or in excess of, their resource consumption limit.

Determining if any resource shares are exhausted is not a difficult problem because the VS abstraction provides VS-specific resource consumption statistics. The problem that has not been addressed yet is to identify which VSs are being starved, i.e., their requests are being dropped.

Ideally, one would keep a list that tracks those VSs whose requests are being shed and a competition matrix that identifies which VSs share processes, sockets, and other system abstractions. If both the list and the matrix were available one could quickly identify those VSs that are being starved.

The matrix of competing VSs is relatively easy to maintain. The VS extension only needs to register when it changes the VS-affiliation from one VS to another and make the appropriate entry in the (symmetric) VS competition matrix. In fact, it is even very feasible to maintain such a competition matrix, M_p, with respect to an individual TCP or UDP port p. Given such a matrix, M_p, for port p, the set of all competing VSs with respect to a specific port is obtained by summing up M_p, M_p^2, M_p^3.

To determine that requests are possibly being shed, it is necessary to identify at the TCP or UDP layers that packets destined for a specific port are being dropped. Unfortunately, it is not obvious to which VS the resulting work would eventually be charged.

The VS of the receiving service thread is a function of its previous VS affiliation, the VS affiliation of the incoming packet, and the VS affiliation of the communication abstraction (e.g., socket) through which a request packet is received by an application (see page 80). This mapping is determined by the VS classification rules that are installed at a specific server, and therefore, the set of possibly starved VSs for a particular shed packet can be determined if one knows, the VS of the discarded packet, its IP attributes, the receiving communication abstractions' past VS affilia-

tions, and the VS affiliations of the processes that have received packets using the communication abstraction under consideration. If the classification mehanism records for every communication port, which VSs have been receiving packets through it, then detection of interference is straightforward. For example, assume that HTTP traffic from subnet 10.0.1.0/255.255.255.0 is mapped to VSID 1 and all traffic from 10.0.2.0/255.255.255.0 to VSID 2. If the HTTP port has been observed to drop packets due to buffer exhaustion then one only needs to know which VSIDs are applied to the HTTP threads receiving requests. In this example both VSID 1 and VSID 2 could be reported as conflicting VSIDs for the HTTP server. An examination of their resource statistics will reveal which VS, if any, is starved and which one is over-utilized.

If starvation of a VS is detected, a signal is sent to one or more front-end hosts that are able to police traffic that enters the server farm. The remaining question is how to divide and police traffic.

Policing traffic: Since incoming traffic at the front-end hosts may not be associated with any particular VSs a second classification mechanism for early packet classification becomes necessary. Without such a mechanism, it would be impossible to discard packets that are destined for certain VSs before they are processed by the IP layer functions. The classification rules (referred to as "rules" see Figure 4.14) may be implemented in a manner that is completely orthogonal to VSs-based traffic classification. However, to avoid confusion one may opt to use the same classification rules as described on page 80 with an expanded state as shown in Figures 4.16 and 4.14.

Overload control rules, as introduced in this thesis, identify traffic classes using IP attributes, such as those used by firewalls, and associate each so-identified traffic class with a number of VSs that may receive requests via IP traffic that matches the rule. In general it is possible to utilize more additional information by parsing deeper into the packet, e.g., for certain URLs and cookies, but this is not implemented in the current prototype. By default the VS classifications for the `accept` and `recv` gates could be used as rules. However, administrators may want to use different (finer or coarser-grained) traffic classes for overload control.

When a front-end host that is configured with overload control rules receives notice that a specific VS is competing with and starving other VSs, the front end will enforce rate limits for traffic matching those rules that were labeled as matching traffic for the "over-consuming" VS.

The important question for a front-end host is by how much should incoming traffic be reduced once the front-end host is notified of a VS resource insulation failure in some back-end host. Since this question depends on the dynamics of request processing, the throttling mechanism is designed in an adaptive manner.

Priming the load-shedding mechanism: Assuming that a specific rule has never been rate-limited, the front-end server has no indication as to what the average traffic rate is for traffic matching the rule. There are two viable options to determine the initial rate limit. First, a front-end server could enforce a default rate limit for the rule (configurable for each installed rule). The second option is to learn the traffic rate by first installing the rule with a traffic counter to measure the incoming packet rate. Then the initial rate limit can be based on the observed traffic rate.

Throttling: Once the rules are primed with an initial rate limit, the front-end server may receive additional overload indications for a specific VS. The front-end server will then be forced to reduce the rate limits for those rules that may cause the overload. It is important to note that only those rules that actually registered a packet flow can possibly cause the back end's overload. Hence, only those rules' rate limits will be reduced.

The rate limit reduction problem is very similar to the congestion problem that is addressed

by TCP's congestion control algorithm [135], since both involve unknown server capacity (a.k.a. link bandwidth) and an unknown number of flows that contribute to congestion. Since the TCP congestion control algorithm (exponential decrease, linear increase) has been shown stable for network flow control [127], this thesis suggests its adaptation to alleviate back-end server congestion. The adapted version of the exponential decrease, linear increase model works as follows.

On receiving a notification regarding the fact that VS X starves other competing VSs, the front-end server cuts the rate limit for all traffic classes that contribute to load for X in half. Contributing to overload means that the traffic class registered at least one additional packet since it was last checked. To allow the new rate limits to take effect, the front-end server should ignore additional overload notification for a configurable time period. This time period should coincide with the average request processing duration for requests that are submitted via the policed traffic class. SNMP, Performance Maps (Chapter V) or other measurement mechanisms [91, 108] may be used to measure it. Implementation experience shows that reasonable overestimation (e.g., a default of 500ms) of this time period does not measurably impact performance of the shaping-based mechanism (performance differences are within measurement noise).

Every time a rate limit is reduced (by cutting it in half), the front-end server registers which back-end server reported the overload condition and records the new rate limit as a hysteresis value. If the overload situation completely subsides, e.g., due to gaps in the clients arrival process and reappears at some later time, then the hysteresis value is a good starting point for shaping incoming traffic. The reason why a hysteresis value is stored for each individual server is that different back-end servers' congestion require different throttling at the front ends. For example, an overloaded back-end server that serves long-lived requests may require more aggressive front end load-shaping than a congested back-end server that serves short-lived requests.

Aging: As long as there are no overload indications, the front-end server linearly increases the rate limit for all rules that are shaped to a rate limit (aging). If the rate limit significantly exceeds the offered rate over a configurable time period and there are no overload indications affecting the rate limit, then the rule is flagged as inactive and will no longer be enforced to reduce packet inspection overheads.

Hysteresis: The value recorded for hysteresis is used to quickly reduce traffic rates to a rate close to the lowest rate limit that was ever in effect due to overload notifications from a specific back-end host. To allow for long-term changes in the system's capacity and processing requirements, the recorded hysteresis values also age. The initial design of the overload-shedding facility for VSs did not allow for the suggested hysteresis mechanism. The result was that rate limits for different traffic classes oscillated wildly in the transition from heavy load to overload. Moreover, the traffic shaping mechanism did not effectively solve the starvation problem without hysteresis. The question is why TCP works relatively well without hysteresis and why one needs hysteresis in enforcing rate control for a server system. The main reason for having a hysteresis value is that the server's work is much longer-lived than packet transmissions, i.e., server bottlenecks do not disappear quickly while network backlogs clear up almost instantaneously. Therefore, incoming work must be reduced aggressively for a network server to allow pending work to finish processing — any additional traffic will have long-lasting overload effects and distort future overload indications. This is not so important in congested networks because bottlenecks clear up quickly.

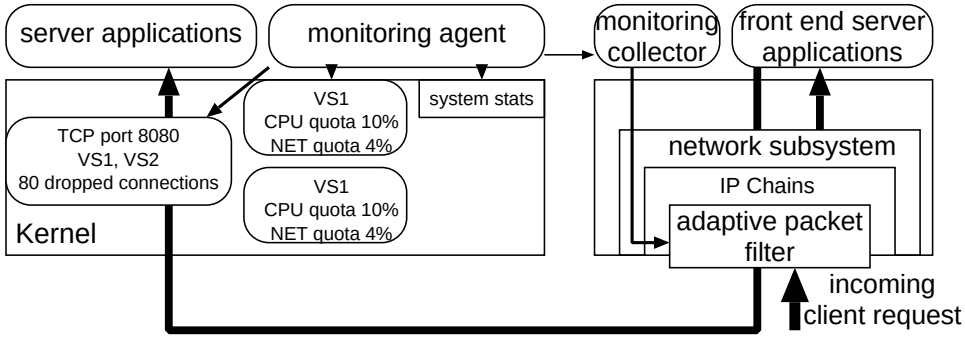


Figure 4.15: Integration of load shaping and VSs

While the TCP-derived algorithm effectively reduces the starvation problem, it is only one of many possible algorithms. The important contribution of this extension is that the failure of VS resource insulation under persistent overload must be addressed and that feedback-driven traffic shaping at front-end servers is a robust solution to the problem. This thesis does not focus on the convergence properties of different rate control algorithms since this work has already been done [116, 117].

4.10.3 Implementation

The implementation extends VSs for the Linux kernel. The VS traffic filtering module itself is implemented as a kernel module, which registers itself as a firewall with the kernel. So, it receives all incoming packets for inspection.

The implementation also includes an overload detection instrumentation of TCP and UDP, which reports its findings (lost packets for specific ports) via a file entry inside the `proc` file system. Whenever a received packet is to be discarded, the kernel simply logs the port for which the packet was destined. Moreover, the kernel also keeps track of which VSs have been seen sending requests to a particular socket, the socket's different VS affiliation, and the VS affiliation of the receiving processes. The VS sets and lists of exhausted packet buffers for individual ports are reset every time the statistical file entry is read.

The firewall module also exports a separate `proc` entry that allows it to receive overload indications of the form:

```
VSID OVERLOAD-SEVERITY RESOURCE REPORTING_HOST_ID [TCP|UDP] PORTID.
```

The reason why the kernel's overload identification extension is separate from the traffic-shaping firewalling plug-in is that the front end's firewall plugin should also be able to receive remote overload notifications from remote back-end hosts—not just from the local host. The prototype implements forwarding of overload signals in a user-space daemon process (Figure 4.15) that communicates back-end overload reports to front-end collectors. The collectors, in turn, report any received overload report directly to the traffic-shaping module, which applies the previously-described overload control mechanism.

The implementation also includes an active component that executes the aging of both hysteresis and rate limits and a second active component that dequeues back-logged packets. The need for the backlog was not discussed in the conceptual discussion of packet shaping because it is not needed to reduce overload. However, having a backlog improves the system's performance because rate adaptation by using only packet shaping can be very coarse. The reason for this is

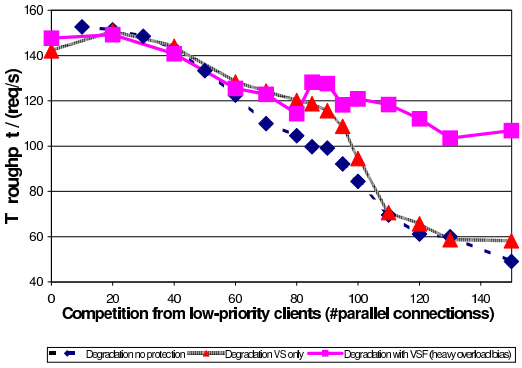
```
struct VSFilter
{
  /* global filter list */
  struct VSFilter *next;
  struct VSFilter *prev;

  /* list of draining filters */
  struct VSFilter *nextd;
  struct VSFilter *prevd;

  /* list of retired filters */
  struct VSFilter *nextr;
  struct VSFilter *prevr;

  /* list of filters in a VS-based filter set */
  struct VSFilter *nextfs;
  struct VSFilter *prevfs;

  /* inactive, active */
  VSFState         state;

  /* matching part */
  struct {

      int proto;

      /* source */
      u32 saddr;
      u32 saddr_mask;

      u32 sport;
      u32 sport_max;
      VSFCompare sport_op;

      /* destination */
      u32 daddr;
      u32 daddr_mask;

      u32 dport;
      u32 dport_max;
      VSFCompare dport_op;
      enum  {VSF_SYN_DC = 0,
             VSF_SYN_SET,
             VSF_SYN_CLEAR} syn;

      unsigned int saddr_accept_cnt;
      unsigned int saddr_reject_cnt;


      unsigned int daddr_accept_cnt;
      unsigned int daddr_reject_cnt;

  } match;


  /* mapping part */
  struct {
              u32 vs;
              int policy;
  } map;

   /* a null terminated of rates */
  int             *acceptance_rate_hysteresis;
  /* the matching null terminated list of host_ids */
  u32             *arh_host_ids;

  int             tokens;
  int             refresh_rate;

  /*
   * stats
   */

  int             total_accepts;
  int             total_rejects;

  /* remember old stats  ( scratch pad ) */
  int             last_accepts;
  int             last_rejects;
  struct timeval  last_refresh;

  int             max_backlog;
  int             backlog_cnt;

 /*
  * This is an index into a possibly graduated
  * back-off table (50% by default)
  */
  int             stepsize;


  int short_run_rate;
  int long_run_rate;

  /*
   * backlog of out-of-spec packets
   */

  struct sk_buff_head packet_backlog;

};
```

Figure 4.16: The filtering rule data structure

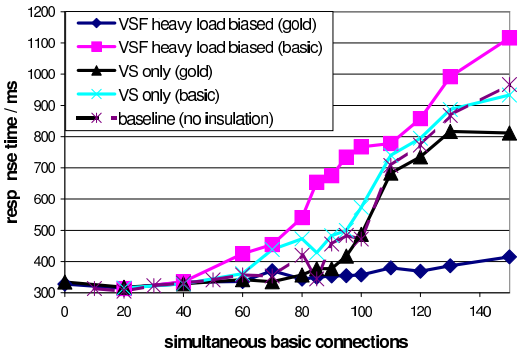
that TCP [135] is very sensitive to packet loss and reduces its own packet rate (too) aggressively every time a packet is shed for rate control purposes. The problem is even worse with SYN packets because their retransmission is very slow (several seconds). Fortunately, if the server deals with very many simultaneous connections, the effects of TCP's own adaptation cycle are diluted and do not affect average performance. However, since some services serve only few simultaneous connections, the backlog is instrumental in providing graceful degradation for those services. The backlog is most important for shaping UDP traffic because UDP does not implement automatic retransmission and applications that rely on UDP often suffer severe performance degradation in the presence of packet loss (e.g., NFS).

Each rule data structure (Figure 4.16 and Figure 4.14) contains a FIFO backlog queue to store packets that arrived "prematurely." Those packets are dequeued by the periodic backlog dequeuing thread. Rules with backlogs are stored in a list of "clogged" rules. Each individual rule may be referenced by arbitrary VSIDs, each of which may be the target of traffic that passes through the referenced rule.

The distribution of overload signals is currently implemented in user-space because it is much more convenient to use the user-level communication libraries than using sockets at the OS layer; there is no conceptual reason for this choice. The overload signal is generated on the servers by scanning for the occurrence of packet-shaping on a specific port and computing the set of VSs that may be affected by it. The monitoring daemon checks whether there are any VSs for which there is some request demand but that consume less than their allocated resource partition (uses the



(a) Degradation gold customer (short listen queue) while increasing competition from basic client



(b) Degradation of client response times

Figure 4.17: Performance degradation experienced under sustained, i.e., very heavy, overload

`vsstat` system call). It then identifies other competing VSs that utilize more than their allowed resource share on the same resource. If both conditions are met, an overload signal is generated.

The overload signal is delivered via TCP connection to a number of front-end overload signal collectors (Figure 4.15) since different front-end servers may be feeding into the same back-end server. The front-end overload collection mechanism is only a simple skeleton that communicates the overload indication directly to the firewalling extension inside the kernel via the previously-mentioned `proc` entry.

The kernel extension determines which rules are to be enforced in response to the overload report and adjusts their rate limits as is described in the general description of traffic shaping.

4.11 Evaluating VS under Heavy Load

The efficacy of the proposed load-shedding mechanism under persistent overload is evaluated for a testbed configuration consisting of a HTTP front-end server, FastCGI middle tier, and Postgres database back-end server. The Postgres server was configured as the system bottleneck. As in the previous example, the clients execute the SpecWeb99 benchmark against this server setup. Unlike the previous example, the clients are configured to request far more simultaneous connections than the server can handle. The simultaneous requests from one client host (gold) are held constant while the simultaneous connections requested by competing hosts (basic) are increased. The performance measurements show the performance degradation experienced by the gold clients that is due to its competition for service with basic clients. This evaluation refers to the filtering extension of VS (including filtering rules, overload detection, and throttling) as VSF (VS Filtering).

Figure 4.17 shows that VS-based resource insulation works well until the server becomes permanently overloaded (at about 85 additional basic connections) as is shown in Figure 4.17[b]. The reason why VS-based differentiation works well up to a threshold is that the system oscillates between being lightly-loaded and very short bursts of loaded intervals. The server with enabled VS processes requests in accordance with VS resource limits when it is heavily-loaded, and always manages to process incoming packets without exhausting the OS's queue limits. However, as the lightly-loaded time intervals shrink with increasing load, packet queues inside the OS grow until incoming packets must be dropped. At this point (85+ connections) all clients suffer equally, re-

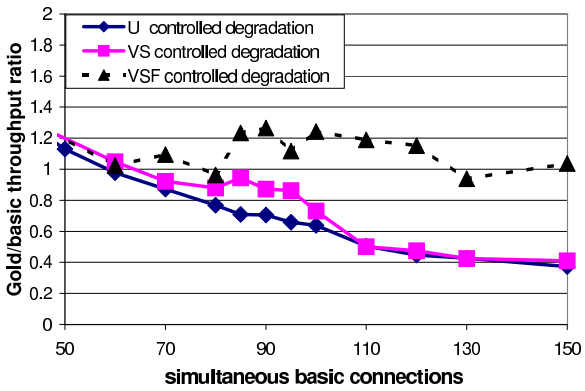


Figure 4.18: Achieved throughput ratios of gold versus basic customers

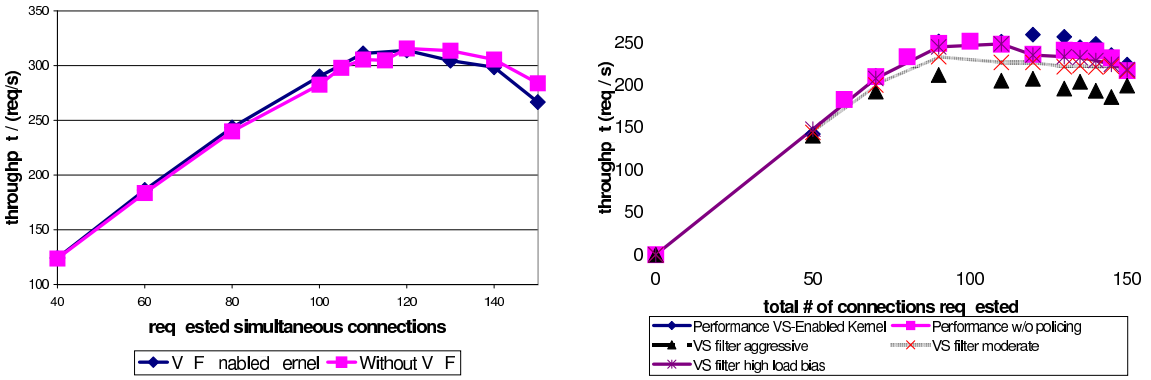
gardless of their VS affiliation. However, when the implemented traffic-shedding feedback loop is enabled (VSF heavy-load-biased[1]), the server continues to serve the gold clients much faster and with much less throughput degradation (Figure 4.17[a]) than without the mechanism. Some throughput degradation is inevitable due to the systems performance loss due to higher task switching and other overheads that are pointed out in the preceding evaluation of the basic VS mechanism (Section 4.9).

The experiments confirm that a comprehensive mechanism for server overload control must not exclusively rely on techniques for scheduling and dequeuing workload (VS) but it must also feature intelligent enqueuing mechanisms at the front-end servers to prevent buffer exhaustion (VSF). Without such a mechanism, VSs that have only small incoming request flows could be easily starved by competing VSs with much heavier traffic volumes. Voigt *et al.* [148] suggest that the buffer exhaustion problem should be addressed by giving each VS its own queue of pending connections. However, such a solution does not take into account the fact that the VS affiliation (or service class) may depend on the identity of the task that dequeues a request. Moreover, separate queues do not account for the fact that the overload may occur in the back-end tiers. Thus, front-end workers would actively process additional (basic) requests, which may tie up processes on a back-end service that should really be waiting for gold requests (see Figure 4.13). Voigt's solution, unfortunately, does not scale to multi-tiered server architectures. In contrast, VS in combination with VSF does.

Figure 4.18 shows that the resource partitioning (50% gold vs. 50% basic) is approximated. The relatively higher throughput for gold customers is explained by the fact that gold customers seldomly exhaust their resource budget, and therefore, their traffic is not policed most of the time. However, basic customers send more requests than allowed by their resource share, thus triggering traffic shaping. Traffic shaping naturally incurs some throughput loss. Fortunately, only those VSs whose request rate exceeds their resource quota suffer noticeable performance degradation.

Packet filtering is not free of cost. The overall performance loss due to filtering packets at the front end whenever an overload condition is encountered is small. Figure 4.19[b] shows that filtering reduces overall throughput by at most 10% for a high load biased filter. The more aggressive VSF settings reduce throughput by too much and are therefore considered unacceptable. The per-

[1] A heavy load bias means that the packet-shedding mechanism will not be activated unless overload surpasses a very high time threshold.



(b) Impact of filter module on aggregate throughput (filter active)

Figure 4.19: Total throughput degradation due to overload avoidance mechanism

formance loss is only incurred during a small load window, in which the system oscillates between overloaded and partially-loaded. Different load biases (time thresholds for acting on overload signals) affect throughput loss because more aggressive filtering results in more requests being shed at the front end. Since VSs are capable of controlling short overload bursts, it is best to configure the filters using a heavy-load-bias.

Some performance loss is due to the overhead of adding another filtering layer to packet processing as is shown in Figure 4.19[a]. Since traffic-shedding is not enforced (filter inactive), this graph shows the performance loss that is due to the interposition of additional filtering code at the network layer. The measurement is taken using the same SpecWeb99 setup, however, without access to the back-end database to increase total throughput and thereby amplifying the impact of VS filtering. The performance loss due to the filtering interposition at the network layer is less than 3%. While this number is not negligible, it is acceptable for a prototype implementation. Most of the additional overhead is due to one additional function call that is executed for every received network packet.

4.12 Summary and Conclusions

This chapter demonstrates that VSs are an effective, application-transparent resource management abstraction when back-end services and resources are shared in a multi-tier setup. Furthermore, the implementation shows that VSs can be integrated into an off-the-shelf OS without incurring the overheads of traditional virtual host-style virtualization (50-75% [147]). To manage VSs, a limited understanding of the managed applications, in line with the models of application behavior introduced in Chapter II, suffices. On the basis of a well-understood model of service interaction, the VS architecture can be configured to transparently and dynamically update the relationship between classical OS abstractions and the resource partitions that are associated with a VSID.

VSs are shown to be able to emulate the VH abstraction. Furthermore, it is shown that VSs provide sound insulation between competing services in spite of shared back-end services. SPs who multiplex hardware and software resources among their “business clients,” could benefit greatly

from the proposed solution. Given the great interest in application outsourcing and the trend towards multi-tiered applications, future versions of commercial resource management approaches will also need to consider the interference caused by shared back-end services. They may use VS tracking to minimize this interference, thus improving resource management for multi-tier services significantly.

The limitations of a pure VS-based solution with respect to persistent overload conditions are also examined and traffic shaping is proposed and shown to be an effective remedy for dealing with persistent overload conditions. While traffic shaping alone cannot create the illusion of a virtual server, it does allow VSs to differentiate different workloads effectively even when the offered load by far exceeds server capacity.

The proposed VS-based resource control architecture for multi-tiered systems with shared back-end services solve a new and important problem, which has not been addressed by any other application-independent resource management solution. The VS approach is unique in that it uses the applications interaction with the OS as an indicator for adjusting the binding between system abstractions and resource partitions. Previous provisioning approaches have always relied on the applications to explicitly manage their own resource quotas. This application-based approach becomes increasingly complex as applications from different vendors are integrated into multi-tiered applications. However, as shown in this chapter, there are fundamental mechanisms that can be implemented in an application-neutral layer, such as the OS, that allow administering the performance of server applications at the system layer.

The VS approach also validates the design of the SDI abstraction for multi-tiered environments. The resource partition abstraction (VSID) is essentially a context attribute in the SDI framework and all classification and activity tracking can be accomplished using SDI tracking rules. Furthermore, the VS-gates validate the integration of generic interposition functions into SDI, because gates need to interpret the VSID context attribute in a manner that cannot be achieved using generic SDI policies at many different system calls.

This chapter expands resource management to multi-tiered deployments and considers both temporary and persistent overload conditions. While the current VS prototype implementation considers resource insulation with respect to CPU, resident program memory, and network bandwidth, it does not address how to control various OS caches that benefit the applications, such as the file cache. For example, to improve insulation, each disk-bound VS should be equipped with its own file cache [30]. To accomplish this goal, the `inodes` in the file cache must be tagged with their VS affiliation. Furthermore, one must limit the total number of `inodes` in the file cache for each VS. If an `inode` is shared by two or more VSs it should retain the tag of the highest priority VS that is using it. Otherwise, priority inversion would result. Content servers with very large `inode` working sets might benefit from such insulation. Furthermore, the current implementation of VSs uses kernel-level threads as the primary processing abstraction. Demonstrating that it is possible to instrument a user-level thread library (e.g., pthread) in a manner that is similar to the instrumentation of the OS's process and scheduling subsystem is a straightforward, nonetheless, labor-intensive exercise that must be completed before the proposed VS-approach can be deployed.

CHAPTER V

Performance Maps

5.1 Introduction

It is very difficult to determine the service, interference, or communication pattern between services that caused a performance problem. The lack of understanding of a multi-tiered system's interdependencies also complicates the configuration of VSs. This calls for a monitoring mechanism that allows service providers (SPs) to correlate observed performance with the peer relationships among all services without requiring full *a priori* understanding of the services' implementations.

Since traditional network, OS, and application-level monitoring solutions [67, 131, 138] are only geared toward optimizing the performance of a host, a network, hardware resources, or one specific application, they are only of limited diagnostic and analytical use in multi-tiered environments, in which the boundaries between hosts and applications are blurred. Interactions and dependencies between coupled applications generally go undetected by conventional monitoring tools. This means that traditional tools are of little aid in configuring a framework like SDI or VSs in a multi-service environment.

A VS (Chapter IV) encompasses a service class, such as a hosted Web service — including required back-end services — for which it provides resource guarantees. Unfortunately, leveraging VSs requires knowledge about the workload-propagating behavior of, and dependencies, among services, which may be difficult to obtain if proprietary applications are part of the multi-tiered setup. Furthermore, one must carefully assess which service classes need to be managed and which ones are performing well, since micro-management of (well-performing) applications waste resources. Performance Maps (PMaps) are designed to infer the required dependency information from a running system through observation.

PMaps are designed to improve troubleshooting and performance management for service infrastructures that consist of hundreds of externally-developed multi-tier applications. The most important contribution of our PMap solution is its model-based approach to system monitoring, i.e., partial event traces are intercepted and checked against the service behavior models of Chapter II to infer application execution characteristics and assess key performance metrics. This chapter describes the PMap approach and validates it using a Linux prototype in a multi-tier scenario with respect to the following objectives:

- Generate a dependency map, such as Figure 5.1, automatically
- Should not require modification of applications
- Recognize peer relationships between services even if they involve several tiers

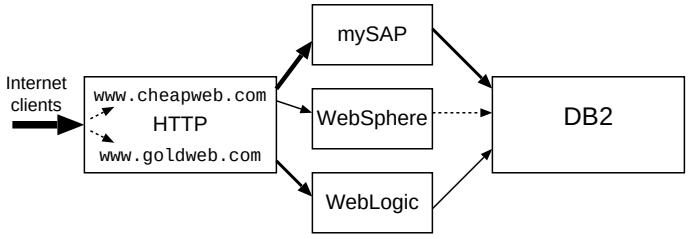


Figure 5.1: PMaps map dependencies within a multi-tier setup.

- Quantify the interaction between tiers with respect to service classes
- Provide troubleshooting support
- Capture the temporal structure of service interactions
- Keep the invasiveness of monitoring low.

The remainder of this chapter is organized as follows. An overview of the architecture is given in Section 5.2. Section 5.3 briefly states some additional assumptions. Section 5.4 describes the state that is attached and tracked alongside multi-tiered activities to build PMaps. Section 5.5 outlines the classification of activities that are subject to monitoring. Section 5.6 describes the tracing of activities. Section 5.7 describes the process of inferring the causal dependencies between tiers from activity traces. These interdependencies are quantified using temporal information about the interaction of services in Section 5.8. To reduce the amount of overhead caused by PMap inference back-tracing is introduced as a part of the PMap generation process in Section 5.9. Section 5.10 details a PMap prototype implementation for the Linux OS. Section 5.11 evaluates its performance and efficacy. Related approaches are compared to PMaps in Section 5.12. Concluding remarks are stated in Section 5.13.

5.2 Overview

Figure 5.2 outlines PMaps' approach towards online analysis of multi-tiered services. Kernel (and possibly DLL) instrumentation generates events, which are collected locally and periodically forwarded to a central analysis station. The central analysis station infers the causal and temporal order of events and interprets them with respect to the service models of Chapter II. Once the system has determined the applicable model, statistics are extracted and aggregated in a statistical graph representation similar to the one shown in Figure 5.1. The key hurdle in creating this representation is to infer causality among observed events because the inference of causality depends on the implemented service model and the completeness of the trace. At the same time the service model detection process depends on the observation of causally-ordered events.

5.3 Additional Terminology and Assumptions

Events are concise digests of what happens inside a computer system at a particular time (see Figure 5.3). They contain sufficient information to generate statistics of interest (e.g., processing time snapshots) but discard all information that is considered irrelevant (e.g., the status of subsystems that are unaffected by the event) to avoid overwhelming the system with monitoring data.

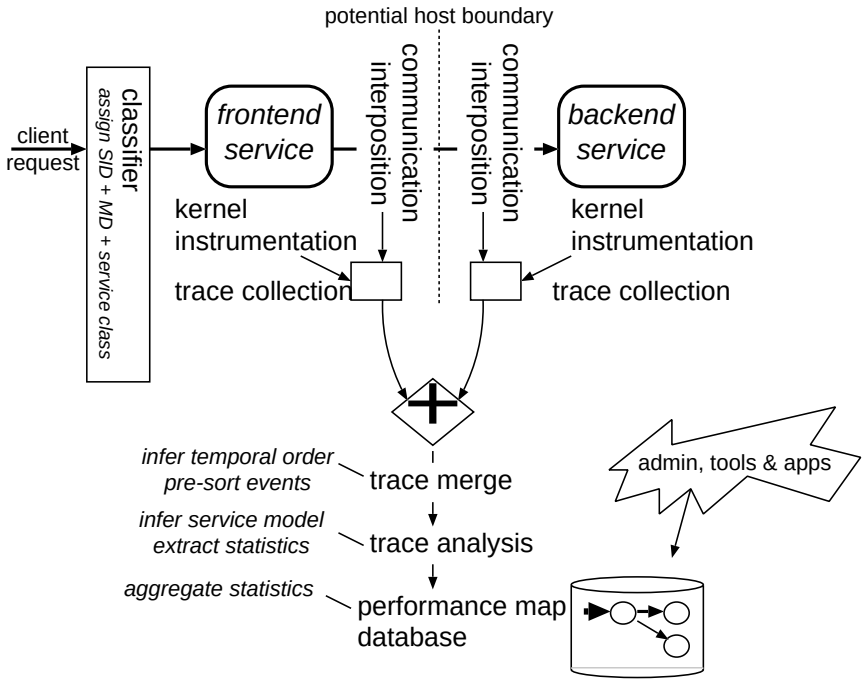


Figure 5.2: Basic PMap solution architecture

One may argue that the estimation of performance and interdependencies for applications on the basis of OS- and middle-ware-generated events is impossible in the general case because this would require an algorithm that can identify requests and corresponding replies for all applications. The impossibility argument against the existence of such an algorithm is called the *end-to-end argument* [124]. In spite of this argument we present a model-based approach to event-trace interpretation that capitalizes on the observation that service and application programmers implement their applications using only a few standard design patterns that can be identified by an observer external to the core application logic.

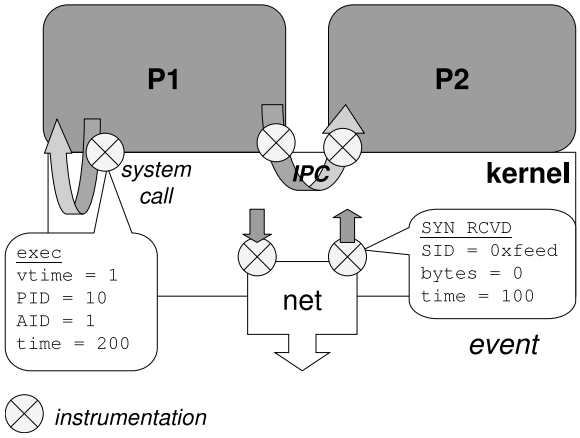


Figure 5.3: PMap's instrumentation points generate event records

5.4 Associating State with Activities

To track an activity across all tiers from the time it is received at a front-end server, it is necessary to identify each activity with a unique transaction identifier [65] and to log its interaction with all services. From the example of VSs it is obvious how one could construct such an identifier and track it as an activity propagates across the tiers of a multi-tiered system. This tracking mechanism generates a skeleton view of a tracked activity.

With respect to mapping the service interdependencies one also needs to track the services themselves and match system objects (processes, sockets, etc.) to the service to which they belong. Some or all of the processes, ports, and sockets that facilitate the activity may no longer exist as soon as the activity completes. Thus, careful tracking is necessary. This tracking feature is orthogonal to the tracking of an activity that starts at a front-end service. This information is tracked in the form of an application identifier that is recorded inside the system objects context and recorded whenever an event is generated for an activity.

Unlike in previous tracking examples, in which the amount of interposition on the path of a multi-tiered activity was fairly limited, monitoring the progress of an activity potentially requires a very large number of taps to record events. Since each tapping into the control-flow of a multi-tiered activity incurs with overheads (Section 3.7), it should be possible to control the amount of monitoring information that is generated for a specific activity. Thus, each activity is also associated with a so-called *monitoring descriptor* that specifies which events are to be generated along the path of this specific activity.

5.5 Classification of Activities

The initial classification of activities is similar to that of VSs. Traffic that is submitted to the multi-tiered system from remote clients is labeled based on packet-matching rules. The difference between the classification for PMap generation and for VSs is that an incoming activity is not mapped to a static label, i.e., a VSID, but to a dynamic sequence number and a monitoring descriptor pair.

In addition to the tagging of activities, we also apply and track an application identifier applied to individual processes. It is inherited by every system object that is created by a labeled process. Thus, a process forked from another process with application ID, X, will also be labeled with application ID, X. As stated earlier, the inclusion of an application identifier greatly simplifies the process of folding multiple distinct activity traces into a single set of service statistics.

5.6 Tracking Activities

The essential tracking mechanism is the same as in the VS example. However, the tracking rules are no longer flexible. In the example of VSs it was up to the system administrator to specify how the VSID would be inherited and propagated across different system calls and network communication. This flexibility does not apply to PMaps because PMaps do not enforce flexible policies and policy-based accounting for competing activities. Each trace generated must describe an activity as accurately as possible.

The tracking policies used for the generation of an activity trace are:

1. Each message sent by a labeled process inherits its senders label.
2. If two message chunks are combined into one message chunk, then the chunk obtains the label of one of the messages.

3. If a process creates another process, the child inherits its parents label.
4. If a process reads a message it inherits the message's label.
5. If a process accepts an incoming connection, it inherits the incoming connection's label.
6. Each socket is labeled in the same way as the last message that is written to it.
7. Each socket is labeled in the same way as the last message read from it.
8. Messages sent between two remote hosts that originate in the communication protocol and not within the application are labeled in the same way as the last application-initiated message that was sent through them.
9. Each file descriptor receives the same label as the process that last read from it.
10. Each file descriptor receives the same label as the process that last wrote to it.

The above label-tracking rules accurately construct a skeleton of potential dependencies between an activity, sockets, and processes. This skeleton, however, is over-determinate. The fact that a process is associated with a specific activity sequence number label does not necessarily mean that a certain system call that is logged under a specific activity ID was truly triggered by that activity ID. The operation could simply be overhead or could belong to the next activity, whose activity ID has not yet propagated to the process. Thus, there are subtle accounting inaccuracies that cannot be overcome with our simple approach to activity association.

Another problem with the above activity-label tracking system is that a multiplexed process or connection will incorrectly be associated with one activity. Moreover, the multiplexed service will produce discontinuous traces because a multiplexed process will continuously change its activity ID. This greatly complicates the analysis of generated traces. Note that unlike in the Virtual Services application, we must be prepared to intercept traces of activities passing through service implementations that may not fit into the processing models that are easily traceable. Knowing about a service that multiplexes its processes among competing activities (e.g., pre-200 versions of MS-SQL) allows the system administrator to avoid installing ineffective VS controls for those services and enforce resource constraints in other ways or at other locations.

This leads to a small modification of the activity ID concept. Since activities can be discontinuous at intermediary tiers due to the multiplexing behavior of one or more services or, for example, multiplexed sockets, we generate new activity IDs at every tier. Thus, it is possible to observe independent fragments of multi-tiered activities that are obfuscated by either connection or process multiplexing. This practical consideration only minimally affects the labeling operation described above, but it greatly enhances the robustness of trace analysis.

5.7 Inferring Service Dependencies from Activity Traces

5.7.1 Basic Approach

The model-based approach attempts to "understand" request submission to correlate front- and back-end computations. For this correlation, it is necessary to first generate a "complete," causal trace of all events (network and processing) generated by activities. If activities are traced by an activity ID, then all causally-related events are labeled with the same activity ID. The events of each specific local host are ordered consistently with that hosts local timestamp. The events between hosts, i.e., communication message are ordered by precedence, i.e., a message cannot be

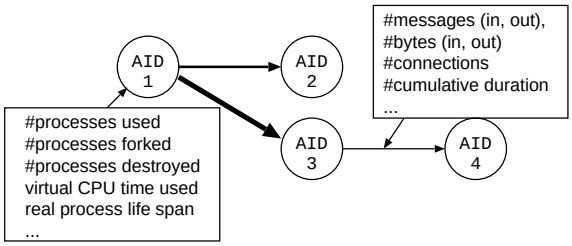


Figure 5.4: An example dependency map

consumed before it is received. This ordering can be achieved in two possible manners. First, it is possible to associate each message with a sequence number, or secondly, if one knows the clock drift between the local hosts, then it is possible to order all events with respect to a global clock.

The qualitative analysis of the dependencies between services needs no more additional information than the collected per-activity ID trace. By abstracting from individual process IDs and communication channel IDs it becomes possible to compile a dependency map as shown in Figure 5.4. As is shown in Figure 5.4, a dependency map contains much useful information about the performance of a multi-tiered service. However, by basing the correlation of events that are generated at multiple tiers solely on the propagation of the activity ID, we have produced a fragile monitoring tool that will possibly generate many phantom dependencies due to the fact that the activity ID is not simply purged across all hosts when an activity is over. Doing so would require large unnecessary overheads. In order to eliminate phantom dependencies, we reconstruct the ongoing activity in accordance with the observed model and insert missing events, such as the disassociation of a process with an activity based on the model-based interpretation of the trace. For example, when a process that is part of a thread pool begins work after an idle period and subsequently picks up a request belonging to a different activity ID, it should be disassociated from its prior activity context and the resumption of processing should be charged to the new incoming activity. This fix-up of event traces is the essence of model-based interpretation of traces.

5.7.2 Enhancing the Robustness of Tracing: Model-Based Interpretation

While model-based interpretation yields little benefit in a purely qualitative dependency analysis it is greatly beneficial to quantitative analysis. Only model-based interpretation of certain event traces makes it possible to charge processing to the correct activity. The ability to recognize a model also allows traces to remain identifiable even when certain events are missing from the trace. This ability allows reducing the amount of monitoring overhead caused by PMap generation.

The models considered by PMaps are the same as those introduced in Chapter II. The following list briefly summarizes their characteristics.

Single iterative worker: A process accepts an incoming request, handles and finishes it. Upon finishing the request, the single worker calls `accept` or `select` again.

Forked worker thread: A master process accepts an incoming request and forks a worker thread to handle it (e.g., **inetd**) and calls `accept` or `select` again.

Dispatcher: A master process `accepts` or `selects` an incoming request and dispatches another process to service it (**Apache, Servlets, Sybase, Weblogic**). The dispatched thread either `accepts` or `read`s from the incoming connection.

Helper threads: A main thread creates helper threads to handle different aspects of a request, such as database queries and client communication (**login services, CGI**). In contrast to the dispatcher model the main thread will not accept a new incoming session before the helper has finished. The work of both the main and helper threads is fully charged to the activity.

Staged worker: Work is passed between processes that implement different parts of request processing. The last stage finishes the request (**DVM** [128],**tcp_wrapper, Oracle**). The characteristic of this model is that the previous stage accepts new work before the dispatched worker completes its work. This is the recursive extension of the dispatcher model.

Nested worker: Processing also passes through several stages as in the previous model. However, upon completing each stage, the previous stage resumes processing (**DB2, WebSphere, FastCGI**). This is the recursive extension of the helper thread model.

Combinations: Services may utilize all or a subset of the above techniques in servicing incoming requests, either by composing the above methods or by implementing them as alternatives. This requires the monitoring tool to be able to identify behavioral components as they unfold in an event trace.

A more succinct summary of these models is given in Table 5.1. Appendix B details the automation of checking for the presence of each specific model.

It is important for PMaps to recognize the above implementation models because they lead to very different interpretations of resource consumption by a multi-tiered activity. For example, without an understanding of service models, a monitor cannot distinguish between the forked worker and the helper thread models, but accounting for the parent's processing depends on the implemented service model. If the a newly-created thread is a helper thread, its parents work should still be charged to the same activity. However, in the case of a forked worker, the parent should be disassociated from the activity as soon as the worker takes over. Thus, incorrect understanding of the service model leads to inaccurate accounting, which, in turn, hampers problem-tracking and leads to misconfiguration of VSs. How to set resource limits and tracking rules for a VS would not be clear.

Since we assume that tracking may potentially be incomplete and potentially obscured by some non-instrumented intermediary hosts, we propose to identify each session between two processes with a unique session ID. This labeling method is as powerful as the activity ID propagation model because by correlating a process, its incoming session identity, and the identity of an outgoing session allows regenerating the notion of a system-wide activity ID. Moreover, the ability to recreate activities beyond services that are not traceable, makes this approach more appealing for real system deployments.

The remainder of this section explains how events are correlated online.

To obtain a good approximation of causally-ordered event histories for activities, this chapter proposes a method that sorts events into (possibly multiple) *event chains*, each of which is built using inferred causality [126]. Similar approximations of causality have been used in building fault-tolerant systems [5, 115, 125][1]. The approximation of causal order based on the proposed method is still overdeterminant compared to true causality. Nevertheless, it is just a fraction of the total events that are generated by the system.

[1]By ensuring that messages are delivered in causal order to all hosts, including backup hosts, one can ensure that the backup's state would have been possible for the crashed primary.

Model	Observable features	Accounting
Single worker	- Worker accepts external client connection - Original worker closes the connection after processing request - Close leads to session end	Obvious
Forked worker	- Listener accepts external client connection - Listener forks worker process - Listener closes socket “before” worker (a) Worker writes result to socket (a) Worker closes socket (a) Worker close leads to session end (b) Worker reads from socket (b) Worker hands-off socket (dispatch/fork) - Worker exits	- Listener work between accept and its next accept or suspension counted toward activity - Worker completely counted toward activity - Recursive counting for multiple fork operations
Dispatched worker	- Listener accepts client connection - Listener waits for next external connection - Connection is used by worker (read or write) - Worker is not forked by listener (a) Connection is closed by worker (a) Worker’s close initiates session teardown (b) Worker hands-off socket (dispatch/fork) - Worker does not exit	- Listener part of activity until suspend or next accept - dispatched worker counted towards activity from resume or semop preceeding its first socket use - Worker start time can be approximated by first read (a) Worker counted until it tears down incoming session (b) worker counted until suspend before next use in a different activity or use in different activity
Forked helper	- Like forked worker but parent’s close terminates incoming session	see forked worker
Dispatched helper	- Like dispatched worker but parent’s close terminates incoming session	see dispatched worker
Staged Worker	- Recursive dispatcher model (submodel b)	see dispatched worker
Nested Worker	- Like helper model but parent waits for child	see forked worker
Peer relationship	- Listener or worker accepts an incoming session - Incoming session is initiated by tracked process	- Begin new model on listener side - back-end statistics counted towards activity

Table 5.1: Summary of different processing models for most common features. Real implementations must consider variants, e.g., dispatcher uses `select`, worker issues `accept`.

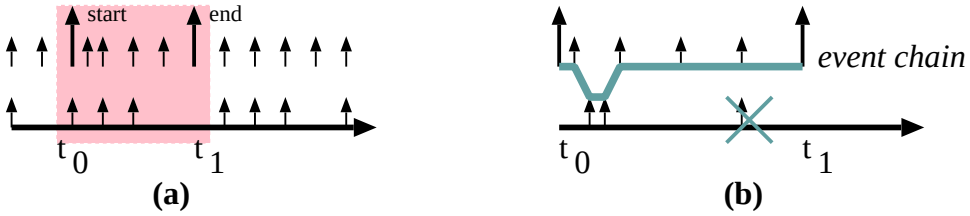


Figure 5.5: Distilling an event chain from interleaved event streams that are generated by the server farm's hosts: (a) activity boundary determination, (b) model-based refinement.

The first approximation of an activity's trace are those events that are delimited by the start and end times of the activity, computed as the difference between the start of the front-end session that triggers the activity and the closure of that session (Figure 5.5[a]).

To better approximate "true" causality, the temporally-ordered event histories are checked against the known processing models online. As certain behavior models become incompatible with an observed trace, the observer expunges unrelated and irrelevant events from each event chain, thus condensing an activity's trace to those events that are necessary to describe its progress. Furthermore, additional events may be inserted to indicate, for example, the switch of a process from one activity to the next.

In the refinement process of checking the incoming events against processing models we simplify the search for related events by introducing a condensed event record that contains only information that is relevant for determining the causal relationship between events, these event records are called *abridged events*.

Definition 2 (Abridged Events) *An abridged event is a distillation of an instrumentation-generated event of the following format: type* $\times$ *thread*$_1$ $\times$ *SID* $\times$ *thread*$_2$ $\times$ *timestamp. Values that are not applicable for a specific event type are set to* ϵ*. When comparing two abridged events,* ϵ *is automatically skipped. If one wants to match an undefined attribute of the abridged event, it must be matched explicitly using* `NULL`*. An abridged event may contain two thread IDs to capture direct communication between threads, e.g., signals. SID (Session ID) is used to identify communication channels. For any interaction between a process and a communication channel, the SID is always specified. SIDs are unique within the entire server cluster.*

The definition of *abridged events* is used to describe the event-to-event chain association algorithm. Assume $\mathcal{H}$ is a temporally-ordered history of events e_0 thru e_{n-1}. $\mathcal{E}$ is the set of all event chains $\mathcal{H}_0, \mathcal{H}_1, \ldots, \mathcal{H}_m$ (current approximations of concurrent activity traces). The operator $\mathcal{H}_i \sim \mathcal{H}_j$ evaluates true ("matches") if the abridged event patterns of $\mathcal{H}_i$ and $\mathcal{H}_j$ match. Tasks in the system (i.e., processes and threads) are uniquely identified by Task IDs (TIDs).

for $(i := 0; i < n; i++)$ **do**
  if $e_i \sim$ (*, `NULL`, $SID, $TID, *) **then**
    if $\neg\exists j \in \{0, 1, \ldots, i-1\} : e_j \sim$ (*,$SID, *, *) **then**
      create new event chain E
      $E := E \cdot e_i$
      $\mathcal{E} = \mathcal{E} \cup E$
      continue
    else if `is_session_termination`(e_i) **then**
      for all $E \in \{X \in \mathcal{E} : \exists x \in X \text{ s.t. } x \sim$ (*, `NULL`, $SID, *, *)$\}$ **do**

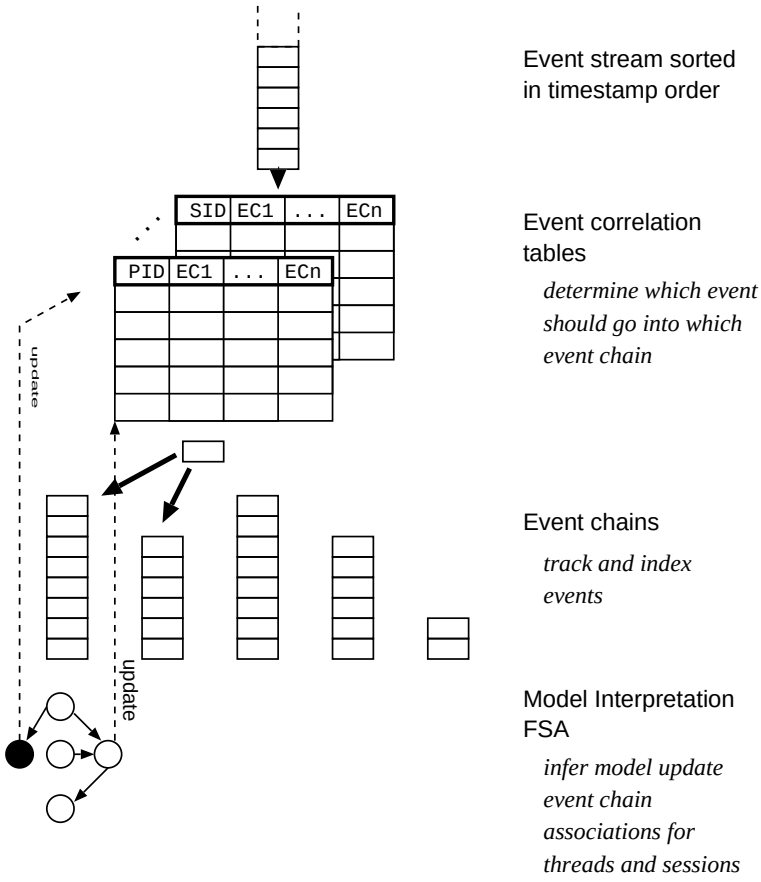


Figure 5.6: Model-based event chain inference in a nutshell.

$E := E \cdot e_i$
if `check_if_event_chain_complete`(E) **then**
`harvest`(E)
$\mathcal{E} := \mathcal{E} \cup E$
end if
end for
continue
end if{*End of special front-end session cases*}
end if
`strong_assoc_found` := 0
for all $E \in \{X \in \mathcal{E} : \exists x \in X$ s.t. $e_i \sim (x_0, x_1, x_2, *, *) \vee e_i \sim (x_0, x_1, *, x_3, *) \vee e_i \sim (x_0, *, x_2, x_3, *)\}$ **do**
if `fits_model`(E, e_i) **then**
$E := E \cdot e_i$
`update_models`(E)
`strong_assoc_found` := 1
end if
end for
if $\neg$ `strong_assoc_found` **then**

```
    {order of if-clauses matters}
    if is_send_type_evt(e_i) ∨ is_proc_evt(e_i) then
      if fits_model(E_byTID[$TID], e_i) then
        E_byTID[$TID] := E_byTID[$TID] ·e_i
        update_models(E_byTID[$TID])
      end if
    else if is_recv_type_evt(e_i) ∨ is_comm_evt(e_i) then
      if fits_model(E_bySID[$SID], e_i) then
        E_bySID[$SID] := E_bySID[$SID] ·e_i
        update_models(E_bySID[$SID])
      end if
    end if
  end if
end for
```

The algorithm's separation of the event stream is visualized in Figure 5.6. The algorithm's loop construct is not necessary when this algorithm is implemented as an online algorithm because the loop's body will execute implicitly each time a new event arrives. The functions `fits_model` and `update_models` tie the online model evaluation process into the causality inference step.

Without a good `fits_model` implementation the algorithm will generate event chains that are larger than needed. For example, if the service implements a staged worker model (e.g., Oracle's dispatcher), numerous activities by the low-numbered stages would be associated with every long-lived activity that utilizes the same (dispatcher) processes. To avoid unnecessary associations it is necessary to identify the processing model early. For example, staged worker behavior becomes obvious when a dispatcher thread accepts another incoming session that belongs to a different activity than the one that it is already associated with. However, it would be dangerous to commit to a processing model too early, especially when several processing models can explain the occurrence of an event. This is why the model identification procedure is conservative, i.e., processing models are only eliminated if they are incompatible with observed events.

The interpretation of event traces with respect to known processing models consists of two components. First, for every chain, PMaps identify those processing models that are compatible with the event chain (`update_models` in the above algorithm). Second, prior to the addition of a new event one asserts that there is at least one processing model, which explains the new event (`fits_model`). Figure 5.5[b] shows that the interpreting approach to event chain augmentation further reduces the size of event chains.

The procedure outlined in the proposed algorithm is essentially a pattern matching algorithm that operates on a stream of events and divides it into per-activity sub-streams (Fig. 5.6). Furthermore, specific patterns cause a rewriting of the event stream, either by removing an unnecessary event, e.g., adjacent communication messages that are sent without any intermediary action by the sending process, or by adding new events, e.g., to indicate the switch of a process from one activity to another.

Once an event chain is complete (i.e., the front-end session is closed), the event chain and its entries in the correlation tables are removed from the tracking process. The event chain is then post-processed by a statistical aggregation process. This process first generates the dependency map skeleton (Figure 5.4) and then derives statistics for the individual processing models. The process of service statistics compilation is accomplished by building a GANTT chart of the activity represented by a specific event chain (Figure 5.7).

To compile the event chain into a dependency map (see Figure 5.4), one abstracts from pro-

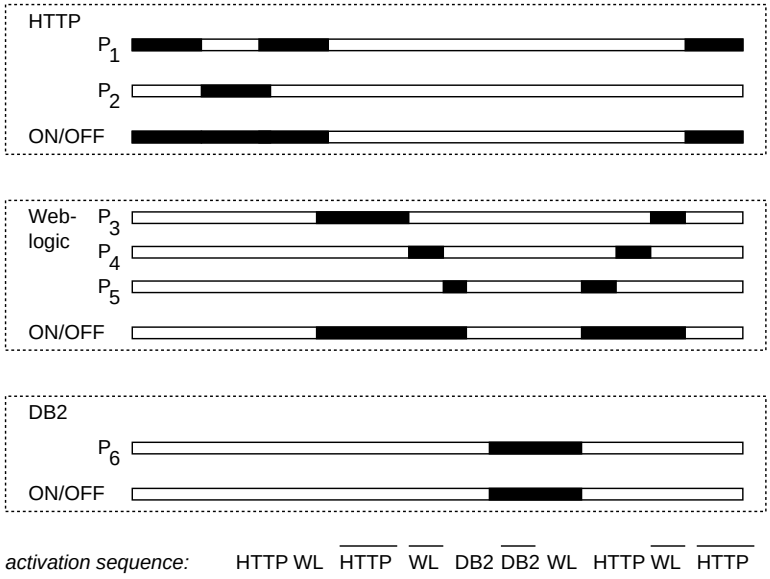


Figure 5.7: Constructing a GANTT chart of an activity for statistical analysis purposes (black bars represent ON)

cesses and threads, because they are only a low-level mechanism to implement a service. A GANTT is used to consolidate all thread processing (Figure 5.7). All processing of threads that belong to the same service (identical AID) is combined into a single per-service processing line.

The information extracted from a GANTT chart for each individual service is represented as a node in a dependency map. The nodes are labeled with statistics, such as the number of processes that are assumed to be working on behalf of a particular front-end session, their aggregated lifetime, and the CPU time spent on servicing the intercepted requests.

Communication between processes, as recorded by the event chains, is reduced to simple edges connecting the services to which the communicating processes belong. The edge data structures are annotated with the total number of bytes and packets exchanged, connection establishment delays (if applicable), and how many of the sends and receives on the connection were blocking and non-blocking. The annotation also includes the number of times the initiator of the communication channel switched from sending to receiving (reversals) as well as the total number of times the communication channel between the two communicating services was established during the analyzed activity. The representation of interdependencies along with its related statistical measures is defined as a *dependency map* (Figure 5.8, left).

Figure 5.8 shows that the dependency map is rooted in a service class node since per-service class statistics are of interest. The other nodes of the graph are the service tiers with a node's depth being equal to its tier number. The edges between nodes represent peer relationships between services.

It is unfortunately not possible to build all statistics that are of interest without some consideration for global time consistency.

5.8 Using Temporal Information to Build a Performance Map

Without accurate and consistent timestamps the dependency map representation cannot capture some important features of the interaction between processes:

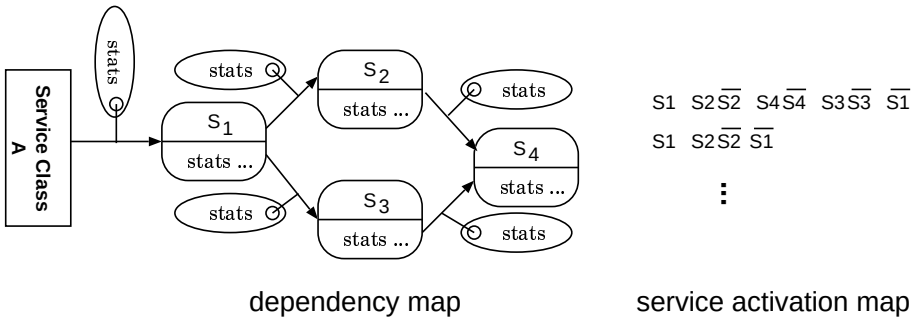


Figure 5.8: A PMap combines dependency and activation information

- Delays from message send to receive cannot be measured accurately.
- Overlapped processing across tiers cannot be identified properly. For example, a service that forks a helper thread to mask the latency of a slow back-end service access may only be minimally affected by the performance of that back-end service. Identifying such behavior is especially important if one wants to assess the expected benefit of improving back-end response times with respect to front-end client's response time.
- It is not clear how to translate the start and end times of an activity to local times at the back-end hosts that are utilized during the activity. This may cause processing action at the back end to be charged to an activity despite the fact that the activity is already over.
- The relation of the message transmission times to processing activity is not clear without accurate time-stamping. One cannot decide whether improving the delay of a network link will have a noticeable effect on the performance of a multi-tiered service.

For this reason it is best if the timestamps of the individual hosts are adjusted to a "correct" global clock. Thus, the GANTT chart of Figure 5.8 will include accurate information regarding the ordering of service activations, the relative processing progress in multiple parallel services, and the delays incurred by the network. The implementation also showed that being able to temporally restrict the search for events that are possibly causally-related greatly improves the performance of the dependency inference engine even in the event that the activity-ID-based tracking alone cannot capture an entire activity, possibly due to a hidden communication channel such as IPC shared memory.

In addition to timestamp accuracy, time is important to achieve a second objective. The dependency map representation is very concise if compared to the full event stream generated by an instrumented server (e.g., a PIII 550 generates up to 5 Mbps of events). In contrast, dependency maps typically only consume consume a few KBs of memory. This reduction obviously implies some information loss. In particular, what is not captured in a purely statistical dependency map is the temporal order in which a front-end service accesses its supporting back-end services. Edge statistics contain no information about timing. For example, looking at the left side of Figure 5.8 it is impossible to tell whether S_1 contacts S_2 or S_3 first or concurrently.

It can be helpful for a system administrator to view the temporal behavior of a service or even better all different access patterns to a back-end service. For example, a web server may access a back-end WebSphere server or a back-end Weblogic server but never both during the processing of one request. The dependency map does not convey this information. It will simply show that the front-end web server depends on both, the WebSphere and the Weblogic servers. To solve this

problem, one should record some timing information with each dependency map. In particular, it may be useful to retain some (shortened) event chains representing real activities. A GANTT chart of an execution (Figure 5.7) is used to infer service activation and deactivation events. Those events are recorded in temporal order, e.g., $S_1 S_2 \overline{S}_1 S_1 \overline{S}_2 \overline{S}_1$. Such an ON/OFF sequence is called an *activation sequence*.

Definition 3 (Structural equivalence) *Two activation sequences X_1 and X_2 are said to be* structurally-equivalent *if they are equal except for difference in their timestamps.*

Using the definition of structural equivalence one may choose to record only those activation sequences that are not structurally equivalent to any already retained ones. The set of all structurally different activation sequences is called the *activation map* for a particular service class (Figure 5.8, right). The combination of both dependency and activation map is called a *Performance Map* (PMap). A PMap includes both statistical dependency information and information about different execution patterns (all of Figure 5.8). Of course, this information is only a model of the real system. However, it provides valuable insight that is not obtainable by any existing approach.

Unfortunately, there may be too many structurally-different activation sequences for each service class since two activation sequences that only differ in their length would already be considered structurally different. Since the analysis station must retain all information in memory—logging to, and searching disk storage would be too slow—a mechanism may be needed to reduce the number of activation sequences to be collected.

One of the following two options may be appropriate to reduce the memory requirements for storing activation maps. First, one could simply enforce a limit on the number of activation sequences stored per activation map; this is by far the easiest implementation choice. The problem with it is that one may miss some rare, yet critical, interactions between front-end and back-end servers. To avoid such omissions, one may store a new activation sequences only if it utilizes at least one service S_y that is not mentioned in any of the already-stored activation sequences. This typically leads to very compact activation maps of 2-3 activation sequences for a 3-tiered service (one with and one without back-end utilization) because different interleavings of parallel activities no longer generate separate activation map entries. The drawback of this approach is that the so-reduced activation map is obviously an incomplete account of services' interactions, since it assumes that the first utilization of a particular service exhibits the typical access pattern. While this may be true in most cases, it may be inaccurate in others. To be reasonably certain that the activation map reflects the true back-end access pattern for a specific service class, the system administrator should reset the activation maps after reading it and regenerate it to be sure that the reported activation map does not change.

5.9 Focusing the Monitor's Operation Using Classification and Back-Tracing

The monitoring system described thus far generates PMaps representing different types of activities by interpreting the event streams generated by the instrumented kernels of several hosts. The proposed extraction of a PMap from a running multi-tier system obviously requires significant processing at the analysis station and imposes instrumentation and event-forwarding overheads at all tiers of a multi-tier activity. Therefore, it is prudent to generate PMaps only on-demand, i.e., when performance failures are observed. Ideally, PMaps would impose additional overheads only for those services, service classes, or hosts whose performance is already failing, or those that directly contribute to performance failure. This objective can be achieved by the following

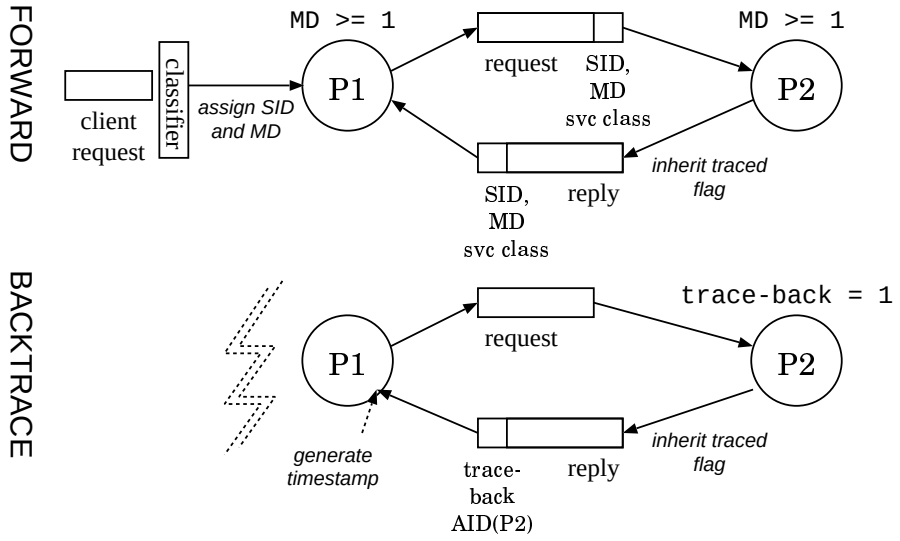


Figure 5.9: Forward-tracing determines statistical properties of service interaction. Backward-tracing is used to determine a back-end service's fan-in.

diagnostic procedure, which should be automated if PMap-like mechanisms were to be deployed commercially.

Using conventional monitoring techniques, such as SNMP, service probing, Tivoli, Windmill, or simply customer feedback, the system administrator identifies service classes whose performance is unacceptable. Classification rules are then installed at the front-end services for each of the failing service classes. For example, to identify a failing HTTP-based intranet application, the system administrator may specify a classification rule as:
`src = 10.0.0.1` $\wedge$ `dpt = 80` $\rightarrow$ `monitoring = yes, service class = intranet`.
This instructs the system to generate events that are necessary for the creation of a PMap of service class `intranet`. The current PMap prototype accepts service class specifications in a similar firewall-inspired format.

The implementation ensures that events are automatically generated for all processes, sockets, connections, network communication, and possibly I/O requests that are part of a monitored activity. Each monitored process or socket must keep generating events as long as it maintains any network connection that is labeled as being monitored. Furthermore, every network connection it originates while being monitored, must also be subject to monitoring. Given this description, one can disable monitoring for a process only if it has no open connections that are subject to monitoring.

The prototype, which partially implements the above-outlined monitoring behavior still generates a large number of events that will eventually be discarded. In particular, the prototype only resets the monitoring descriptor for a process when the process has closed all of its open connections. A better method to reduce the number of events generated inside the kernel may become necessary, should the instrumentation overheads grow too large. Currently, the monitored servers experience less than 1% overhead instrumentation and event collection.

Since the standard PMap generation process traces multi-tier services from their origin (i.e., from the receipt of a request at a front-end service), it is called *forward tracing*. Forward tracing can reveal slow back-end services, ping-pong communication patterns between tiers, and the like.

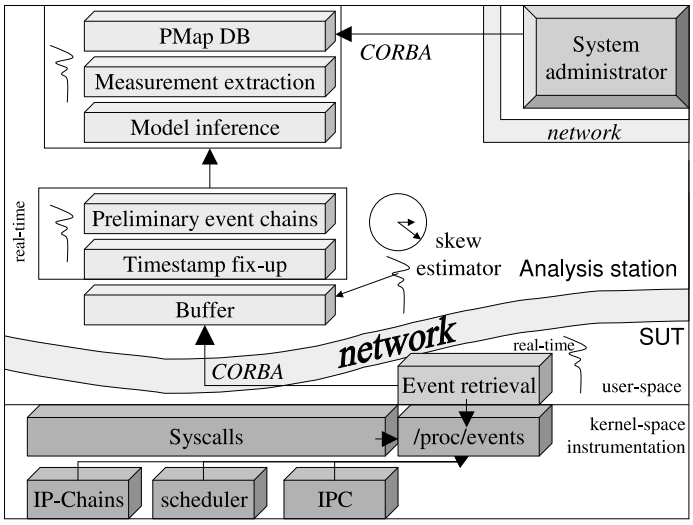


Figure 5.10: Overview of the prototype's implementation.

However, it cannot identify interference-related problems, i.e., performance failures that are caused by other service classes' competition for the same back-end service. Nevertheless, identifying interference must be part of a diagnostic support system for multi-tiered systems.

To enable system administrators to identify interference as the root cause of a service class' performance failure, a *trace-back* mechanism is introduced. A trace-back of service S or host H determines the set of services that depend on S or H, i.e., the back end's fan-in. To this end, PMap-compliant instrumentation must allow placing a *trace-back trigger* on services and hosts. A trace-back-labeled service will apply a trace-back label to each reply packet (i.e., a packet sent over a connection that the back-traced host or service did not initiate or any UDP message) by placing a marker in its IP header. The client-side instrumentation responds to the trace-back marker by generating a trace-back event, identifying itself and the back-end service that applied the trace-back label. Thus, it is possible to recursively determine the fan-in of a particular back-end service or host without exhaustively tracing all front-end services in the entire system. Once all front-end services that feed into a problematic back-end service or host are identified, one can forward-trace them individually to obtain the statistics needed to analyze and quantify interference. Backward tracing is to be deployed whenever system administrators are unable to identify any problems within the PMap of a service class.

5.10 Prototype Implementation

The PMap prototype tool consists of a user-level analysis component and kernel instrumentation and support functionality for Linux 2.2.14 (Figure 5.10). The user-level component is implemented in C++ using CORBA for communication between the central event-analysis station and the local event-collection daemons as well as a permanent TCP connection for event streaming.

The kernel-level instrumentation itself can be divided into service class *classification*, inside the networking stack, *tracing*, throughout the kernel, and *event generation*, inside the network, process, and IPC subsystems.

Classification: Whether the activities spawned by a request belonging to a specific service class are subject to tracing or not is determined by firewall-style rules of the form:

(src net, netmsk, dst net, netmsk, dst port) $\longrightarrow$ (svc class, MD), which are installed at the front-end servers. These rules are interpreted by a firewall plug-in.

The service class ID determines how the collected measurements are to be aggregated. In SQL databases one would use the service class as a `GROUP BY` attribute. The monitoring descriptor (MD), a bit vector specifying enabled monitoring events, is assigned at the same time as the service class. Instrumentation can access this classification state by accessing the appropriate extension fields in process descriptors, message buffers, and sockets, in which both MD and service class are recorded. For example, the monitoring descriptor `MD_PMAP_EVENTS` enables PMap's core events (the only such descriptor value supported by the current prototype).

To establish causality between message send and receive events, communication messages are tagged with a unique activity ID, a.k.a., session ID (SID) of the form: `HOST_ID : SESSION_NO` with configurable length of the host ID. A session that is identified by a SID is one instance of an end-to-end communication channel. A SID, however, is only assigned in two cases: either an incoming connection has been classified as being subject to monitoring or a monitored process initiates it.

Additional Kernel State: To be able to represent services by AIDs, a new service abstraction is created at the system level. Each AID is represented by a service structure to accommodate multiple trace-back triggers. A process is associated with a specific AID by setting the AID pointer inside its `task_struct` to the corresponding AID structure. Whenever the process forks child processes, the children inherit the parent's AID as they presumably belong to the same service. Windows already features a service/application abstraction, but the prototype's AID abstraction is more flexible. The AID of PMaps allows arbitrary processes and threads to be grouped together under one AID. For example, one may choose to aggregate a Web server and the FCGI server it utilizes under one AID.

In addition to the AID, the process socket and message structures are extended to store SID, MD, and trace-back flag where applicable.

Instrumentation: The instrumentation of the kernel requires only a few new lines of code for each instrumented function call. The code checks if the instrumentation at this specific interception point is enabled by evaluating the MD that is stored in the intercepted process, message buffer, or socket. If it is, the instrumentation will record a timestamp, a function ID, the service class of the intercepted system object, a SID for communication-related events, and an AID and PID for processing events plus some additional process or communication channel statistics for the purpose of troubleshooting (Table 5.2), which are noted in the node and edge annotations of a PMap.

On accepting a traced session, a process generates an accept event, recording the SID, the session's service class, its own PID, and the AID of the service to which it belongs. Furthermore, the process will copy the service class and MD of the incoming session into its process descriptor. From this time and on, a process will generate processing- and communication-related events, as specified in the MD, until it has no more open sessions, in which case the process's tracing bit is reset. As long as a process maintains open connections, all received MDs are *OR'ed* together, to ensure that all necessary events are generated. Table 5.2 lists the minimal event set required for the generation of PMaps. This event set is enabled by setting the PMap monitoring bit in the MD.

Propagation: SIDs, service class IDs, and MDs must propagate from client to server across host boundaries to allow tracing multi-tiered activities, adjust tracing preferences on a per-host basis, and decide which events should be generated. The propagated service class ID allows remapping the MD at individual tiers, i.e., the system administrator may direct different hosts to generate more or less events for a specific service class' activities. This flexibility allows monitoring different hosts at different granularity and may reduce the number of events to be generated and

Category	Event	Description	Recorded information	Linux calls mapped to this event
processing	CREATE	create a worker thread	parent & child PID	`fork, __clone`
	EXIT	exit a worker thread	PID, vtime	`exit`
	SES_CREATE	create a session	PID & SID	`connect, accept, sendto, sendmsg, recvmsg`
	SES_INDICATE	signal a session request	SID	network stack internal event at SYN_RECEIVED and at SYN_SENT
	SES_ACTIVATE	accept a session	SID & PID, vtime	`accept, recv(udp)`
	SES_DONE	close a session	SID & PID, vtime	`close`
	SUSPEND	processing suspended	SID & PID, vtime	`sleep, select, ...`
	RESUME	processing resumed	SID & PID, vtime	internal to scheduler
communication	SEND	send a network message	SID, PID, bytes	`send, sendmsg`
	RECV	receive a network message	SID, PID, bytes, block(?)	`read, recv, recvfrom, recvmsg`
	SIG	wake worker(s)	PID(sender), PID(receiver)	`signal`
	IPC_SEND	send an IPC message	PID(sender), FD, res_id	`write (to pipe), msgsend`
	IPC_RECV	receive an IPC message	PID(receiver), FD, res_id, blocking (?))	`read (from pipe), msgrecv`
synchronization	SYNC	synchronize	PID, SYNC_ID	`semop, semctl, ipc`

Table 5.2: List of events required for PMap generation. The fifth column casts Linux OS functionality into event categories. `Vtime` and `bytes` are place-holders for a variety of process and connection statistics, respectively. The level of detail recorded in individual events is an implementation choice.

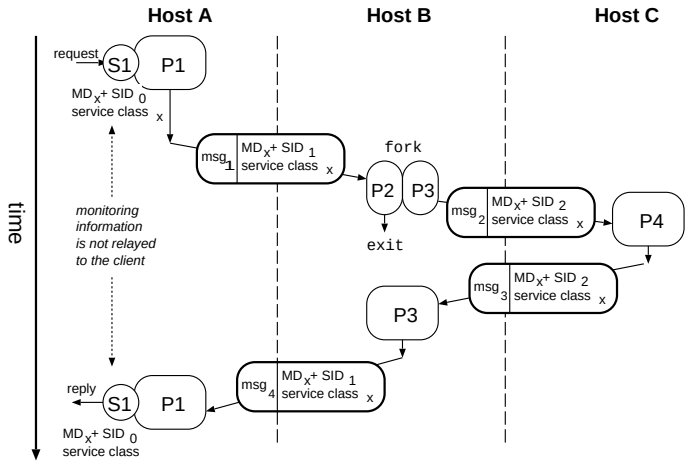


Figure 5.11: An example trace of an activity.

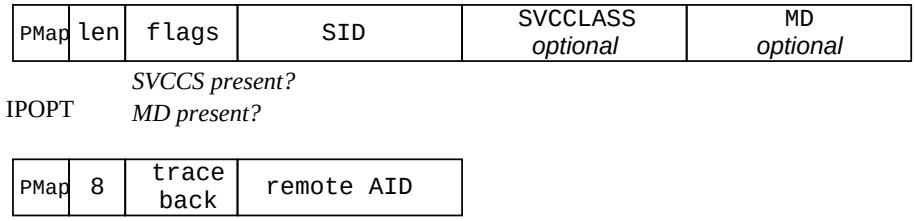


Figure 5.12: Format of the tracing IP options.

analyzed.

Both internal message abstractions and actual network packets are tagged with SIDs, service class ID, and MDs. To permit the deployment of PMap-unaware servers in a server farm, the propagation of SID, service class, and MD inside network packets must happen in a manner that is compatible with the underlying communication protocols.

Compatibility is achieved by exploiting extensibility in the lower-layer communication protocols; IP options of IPv4 and option headers of IPv6. IPv4 options may contain up to 20 bytes of information — enough space for 8 - 16 bytes of tracing information (see Figure 5.12). To utilize this space, it is necessary to modify the kernel's IP option processing functions. When an `sk_buff` with a valid SID, service class, and MD is about to be sent, this information is translated into a monitoring IP option. When an IP packet with the monitoring IP option set is received, the SID, service class, and MD are extracted from the IP options field in the IP frame and stored in the receiving `sk_buff`. Depending on the MD, certain kernel functions may begin logging events when handling the received buffer (e.g., SYN_RCVD or SYN_SENT). Figure 5.11 illustrates the entire process of event generation for, and tracking of, an example multi-tier activity.

Integration with existing monitoring tools: Integration of existing monitoring tools (e.g., LTT [155, 156]) and PMaps can be achieved using the MD. For example, LTT can be modified to generate an event whenever it intercepts an I/O system call from a process whose MD has the "monitor I/O flag" set. Since the MD also propagates with every communication message, it is also accessible to existing network monitoring tools, such as `tcpdump`, Pandora [108], and

Windmill [91], which may capture more specific information about packet timings. Those tools, however, need to be modified to interpret SIDs, MDs, and service class IDs as they are propagated in network packets.

Event Collection: Each server of a multi-tier system runs a real-time event-retrieval daemon, which extracts the events collected inside the kernel and forwards them to a central analysis station. The daemon is scheduled as a real-time process because its responsiveness must be guaranteed, especially under heavy system loads, which typically generate larger event volumes. The local event collector forwards an event block whenever a configurable number of events is available but at least after a configurable timeout (currently 1 s) if there are events that have not yet been forwarded.

Clock Synchronization: It is assumed that clocks are loosely-synchronized inside the server farm [43, 97]. Loose synchronization gives certainty about when it is safe to assume that all events belonging to an activity have been received by the analysis station (gap detection [12]). Moreover, it provides a reasonable bound on clock skew, simplifying the design of more precise skew correction for individual timestamps.

NTP [97] is used to achieve coarse-grained synchronization. Although NTP's clock synchronization is fairly tight on lightly-utilized networks (within 200 μs), experimentation with the prototype showed that it is not tight enough to order the communication and processing events within one activity properly. Heavy network loads aggravate the problem, creating skews as large as 20 ms. Moreover, NTP corrects clock skews only gradually, which is necessary to guarantee locally-consistent timestamps but prolongs clock convergence. Unfortunately, other clock synchronization mechanisms [80, 159] do not provide better clock-synchronization (within tens of μs) without incurring excessive overheads, which is needed for PMaps. To this end, the prototype implements an additional layer for event timestamp correction.

To fix the timestamps between two peer servers, the following sequence of events with an identical SID is matched repeatedly: SYN, SYN_RCVD, SYN_ACK, SYN_ACK_RCVD, ACK, ACK_RCVD, i.e., a three-way handshake between hosts A and B. If this sequence is found, one can obtain a fine-grained clock skew estimate between A and B by estimating the round-trip time (rtt) and skew between hosts A and B as

$$\begin{aligned} T(\texttt{SYN\_SENT}) + \frac{\texttt{rtt}}{2} &\approx T(\texttt{SYN\_RCVD}) + \texttt{skew} \\ T(\texttt{SYN\_ACK\_SENT}) + \texttt{skew} + \frac{\texttt{rtt}}{2} &\approx T(\texttt{SYN\_ACK\_RCVD}) \end{aligned} \tag{5.1}$$

where $T(x)$ denotes the timestamp of event x. The event sequence actually yields two rtt estimates and two skew estimates, which are averaged in the current prototype. The skew estimate is also averaged over time (a moving average).

To guarantee consistent skew adjustments across all sessions, it is necessary to apply the same clock skew fix-up to all events of a host that are waiting to be interpreted. This is necessary to avoid aliasing, i.e., two actions become indistinguishable since their events blend together due to inconsistent timestamp corrections. In fact, one must even guarantee that the transitive hull of clock skew adjustments (repeated squaring of the clock skew adjustment matrix) does not contain inconsistencies. Inconsistent clock adjustments — even over several tiers — are a frequent cause for incorrect service-model inference (in the case of imperfect traces) and plagued the implementation and experimentation process. Consistent skew adjustments are enforced by updating the analysis station's clock skew estimate matrix periodically using an independent clock skew estimator thread.

Communication edge statistics	
Measure	**Definition**
COUNT	Number of times a session between two communication endpoints has been observed
S	Aggregate duration of sessions identified by edge. It is labeled S^* when it is the duration of a front-end session.
I	Aggregate indication delay of sessions identified by edge. It approximately equals network delay.
A	Aggregate activation delay sessions identified by edge. The activation delay is the time between a connection being signaled at the server side until it is picked up by the server. This shows whether the server relies on the OS to manage pending connections. If it does, this measure is indicative of service load.
L	Aggregate linger delay of sessions identified by edge. It measures how long it takes for a connection to be fully closed after it has been closed on one side. An unusually large number may indicate an flawed shutdown mechanism in a server protocol or high server load.
B_{in}	Total bytes received by client via sessions identified by edge.
B_{out}	Total bytes sent by client via sessions identified by edge.
BLOCK^c	Blocking receives by the client.
BLOCK^s	Blocking receives by the server.
RECV_CNT^c	Total receives by the client.
RECV_CNT^s	Total receives by the server.
CHUNKS$_{in}$	Total chunks received by client via sessions identified by edge. Chunks represent data blocks that are sent as a unit by the application. Chunk information can reveal implementation problems, e.g., a mismatch between MTU and application block size. Usually small numbers of bytes transfered per chunk are problematic, large ones are not.
CHUNKS$_{out}$	Total chunks sent by client via sessions identified by edge.
R	Total message flow reversals, i.e., the number of time a session switched between being written to and read from by the client. When a session is recycled to send multiple requests, this number gives a more accurate estimate of the number of requests that a server received than the session count.

Table 5.3: PMap communication edge measurements

Service node statistics	
$\overline{W}$	Sum of all process' times during which a service has at least one runnable process processing work in the context of a given dependency map (Section 5.11).
W	Response time contribution of service that is not masked by the concurrent execution of any lower-tier services. This allows us to identify whether a back-end service contributes a significant amount of latency to an average request. A delay reduction process must start addressing the latency issues of the highest-numbered tier with $W > 0$ (Section 5.11, lack of pipelining).
C	CPU time consumed by a service.
FORK	Total forked processes.
EXIT	Total exit count. FORK and EXIT can be used to infer whether a service is pre-forking, forking on-demand, or handling processes independent of the request stream. This knowledge can be used to infer the service model.
FAN_{in}	Describes the set of services that depend on a specific service. The $\texttt{FAN}_{in}$ is only determined when a trace-back trigger is placed on a particular service.

Table 5.4: PMap per-service measurements

Even though the fix-up procedure cannot theoretically guarantee that an event chain is temporarily-ordered, experiments show that the subsequent parsing and analysis process hardly rejects any activity's event chain (less than $\frac{1}{10^5}$). Without it, however, a large number (approx. $\frac{1}{3}$) of events could not be interpreted correctly.

Model Inference: Once timestamps have been adjusted, the event stream is fed into a pattern matching module, which tracks incomplete activities, process, socket, and session-to-activity bindings. The reason why this parser operates in a streamed, online fashion is that it is necessary to weed out irrelevant events as early as possible to limit the amount of memory consumed by events waiting for analysis.

The inference process depends on a few important tables, such as mappings of process IDs, socket IDs, and session IDs to unfinished event chains. These allow new events to be routed to exactly those event chains with which they may be causally-related. Each event chain that receives new events is scheduled to be handled by the analysis thread for further model interpretation.

The analysis thread checks each event chain that received new events, and attempts to fit events into them in a manner that is compatible with the applicable processing models as explained in Section 5.7.

The current prototype implementation follows the specified algorithm but it is implemented in an object-oriented manner. The prototype provides separate event handlers in the event chain object, one for each event type (e.g., SEND and SES_ACCEPT). Each handler interprets the incoming event with respect to other events that are already recorded within the event chain. Events within the event chain object are indexed for fast retrieval. Events are consolidated as they are added to an event chain, for example, by combining adjacent communication events. Moreover, events that — in the light of the applicable processing behavior for the event chain — are inconsequential for the event chain are discarded. The main benefit of this procedure is that it weeds out irrelevant events as early as possible, thus conserving memory. Note that memory would quickly become exhausted since events can be streaming into the analysis station at rates of several Mbps.

Once an event chain is complete, typically indicated by the closing of the session with the

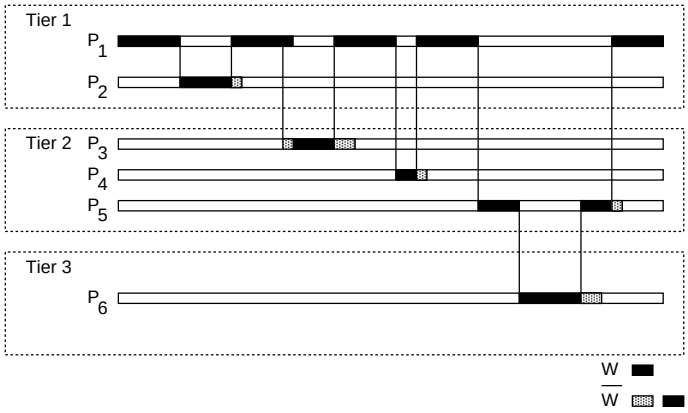


Figure 5.13: Computing $\overline{W}$ and W using a GANTT chart.

front-end server that caused the creation of the event chain, the event chain is marked as ready for harvest by a statistical analysis thread within a configurable delay. The reason why a delay is introduced is that some parts of the activity may still be executing asynchronously and some events may be forwarded after a slight delay.

Statistical Measures: Whenever an activity is completely parsed, the finalized event chain is harvested for statistical analysis. Harvesting includes the canonization of the event chain (Section 5.8) followed by measurement extraction (Table 5.3 and 5.4). Most performance metrics can be obtained easily because the event chain is already broken up by sessions, processes, sockets, and services during online model evaluation. Most metrics can be extracted from the event chain by counting or measuring durations between events. Only the measurement of overlapped processing between tiers and the computation of the activation sequence requires more sophisticated analysis on the GANTT chart of the event chain.

The GANTT chart (Figure 5.13) allows us to determine the amount of processing time that a particular tier contributes to the total processing of an activity. This is called the processing time contribution of a service, σ, its *blocking contribution*, W_σ (see Table 5.3 and 5.4). W_σ considers only those intervals during which no lower-numbered tier service than σ has any runnable processes. Figure 5.13 depicts the computation of, and the difference between, W and $\overline{W}$, i.e., a services total processing time. Obviously, a ratio $\frac{W_\sigma}{\overline{W}}$ close to 1 indicates that the front ends block on σ, whereas a ratio close to zero shows the opposite.

The computation of W uses an interval tree data structure to determine intersecting ON-time intervals and to correctly infer the lowest numbered tier that is active.

Upon extracting the above measurements from the event chain and GANTT chart, the dependency map and activation sequence are generated. Sessions are translated into edge data structures, with multiple sessions between the same two services being folded into the same data structure—multiplicity is accounted for in edge statistics (`COUNT`). The compiled dependency map for one particular event chain is passed to the PMap database/aggregation module. The database associates the generated dependency map with the service class of the activity that caused the event chain to be recorded and updates the existing dependency map for that service class. If no dependency map exists for the considered service class, then the newly-generated dependency map becomes the dependency map of that service class. The activation map is stored if no structurally equivalent chain has been recorded for the service class under consideration. Figure 5.16 shows an abridged dependency output of the PMap prototype (textual node and edge annotations are omitted for read-

ability purposes). The tool's output is formatted using `graphviz` [55]. The PMap database can be queried via a simple CORBA-based API or viewed as textual output.

5.11 Evaluation and Application of Performance Maps

The testbed used to study PMaps consists of two servers and three client machines connected by an SMC Tiger FastEthernet switch.

A simple UDP-based and a TCP-based single-tier service are used on kernels with and without PMaps to estimate an upper bound for PMaps impact on server applications. The server replies to a client by bouncing back the 1KB message it receives. The bar charts of Figure V.15(a) show increased latency when PMap monitoring is enabled. Since the service itself incurs only negligible processing overheads, the reported 6% (UDP) and 8% (TCP) penalties are more than what one would observe in a real server that requires a significant amount of processing.

To validate the the small performance overhead of PMaps for a more realistic service SpecWeb99 performance with and without PMaps are reported in Figure V.15(b). The dotted vertical line indicates the server's reported *SpecWeb99-Mark*. It is shown that PMap's presence has little impact on the performance of a WebServer.

The tests for the PMap prototype are executed in an example multi-tier setup, integrating a front-end HTTP server, a FastCGI middle-tier server, and a back-end Postgres database to mimic a simple E-Commerce site (Figure 5.14). SpecWeb99's dynamic requests are handled by an FCGI middle-tier server, which may trigger a `SELECT` operation in the back-end database (list shopping cart contents) or an `UPDATE` operation simulating a quantity change, with configurable probabilities p_s and p_u.

In the first measurement series PMaps' output is compared against the actual service configuration, and performance numbers measured inside the middle-tier FCGI server. PMaps are in all cases a reasonably accurate description of system performance. First, Figure 5.16 shows that PMaps accurately capture the dependencies between services. Second, the PMap reported relative request ratios — $p_s + p_u$ and the percentage of requests using the FCGI server— which are reported as edge weights for the dependency map, differ only minimally from those configured in the setup (less than 1% error).

The service time estimates are also very close to what is obtained by instrumenting the FCGI middle-tier itself (PMaps overestimate by less than 3%). PMaps were not expected to measure exactly the same values as service instrumentation because they observe the service from the outside. PMaps' relative measurement error decreases as service times increase.

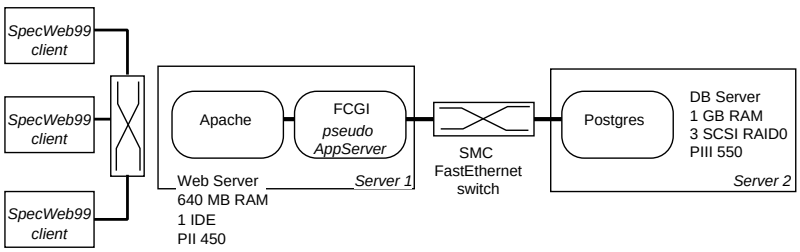
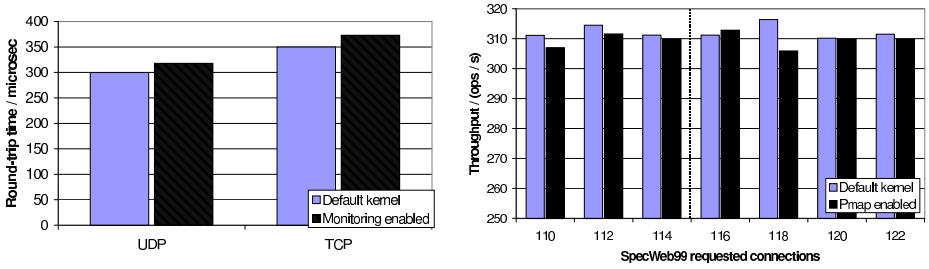


Figure 5.14: Setup of our pseudo E-Commerce site.



(a) Response time penalty for a simple ECHO server with PMap monitoring.

(b) Throughput penalty seen by SpecWeb-99 clients when enabling monitoring for all front-end services.

Figure 5.15: Performance overhead of PMap activation

5.11.1 Performance of the Analysis Station

The analysis station is a Pentium IV-based computer with a 1.7Ghz processor and 512MB of PC 800 RAMBUS memory.

The analysis station's software is implemented in C++, compiled with GNU's C++ compiler and not optimized for performance beyond choosing appropriate data structures and algorithms. For this reason, it is not so important to look at the absolute resource consumption of PMaps but the growth trend with increasing event load.

The performance of a single analysis host is sufficient for the analysis of small to medium-sized server setups. Figure 5.17 shows that its CPU consumption grows linearly with increasing request load on the servers, and reaches approximately 21% when mapping a front-end server that serves at a rate of 21 million daily hits. For the same event load, the analysis station consumes approximately 53 MB of RAM for its internal data structures, indices, and buffers, and possibly still undetected memory leaks. The fact that CPU load and memory consumption grow linearly with increasing load at the analysis station implies that PMap generation is not only possible but also practical. Since the analysis stations are very simple, requiring some memory, network connectivity, and a moderately powerful CPU, and require almost no configuration, they can be built from inexpensive commodity PCs and fully benefit from rapid increases in processor and memory technology.

5.11.2 Pinpointing Performance Problems with Performance Maps

To study PMaps' troubleshooting capabilities, the behavior of the middle-tier service and the workload are modified. Different performance problems are triggered by changing p_u, p_s, communication bandwidth, CPU consumption by the services, an increase in the number of requests

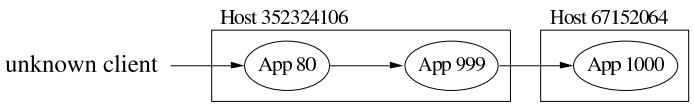


Figure 5.16: Automatically-generated PMap for one service class using the setup of Figure 5.14.

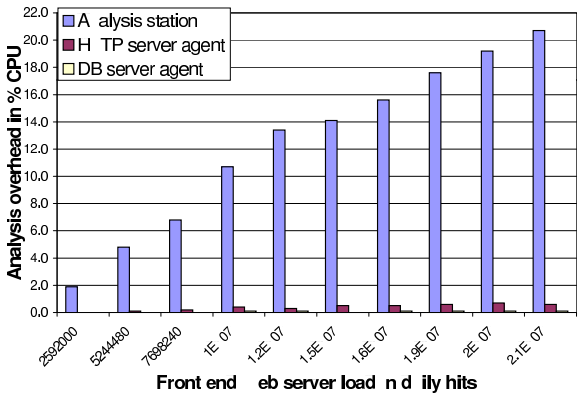


Figure 5.17: Processing overheads incurred by executing PMap on the servers and for the dedicated analysis station.

to the database server, as well as a relocation of the middle-tier from one server to another.

To demonstrate the expressiveness of PMaps, a number of diagnostic performance metrics are computed on the basis of PMaps basic measurements (Table 5.3 and 5.4). The metrics themselves are not part of the PMap tool but are directly calculated using its dependency maps. PMap users may create their own equivalent metrics. The main criterion in defining performance metrics for this evaluation is a strong, reproducible correlation with specific performance problems.

Overload: CPU-related overload detection for a specific service class is relatively straightforward. Without loss of generality, from now and on we will assume that all PMap measurements are 0 at time t_0 and that all measures are take after some time Δt. Service class j's CPU contribution, $\texttt{CC}_j$, to the total latency of j's requests is defined as

$$\texttt{CC}_j := \frac{C_j}{S^*} \tag{5.2}$$

(see Table 5.3 and 5.4 for definitions of the base measures collected by PMaps).

The network's contribution to request latency between two services, p and q, in the context of service class j, is defined as

$$\Omega_j(p,q) := \frac{I_j(p,q)R_j(p,q)}{\texttt{COUNT}_j(p,q)S_j^*}. \tag{5.3}$$

This metric expresses communication delays relative to the total service time of a request, flagging only significant delays. The combination of Ω, CPU contribution, and the dependency map for service class j shows whether j's performance is network- or CPU-bound. Future versions will also track the contribution of local I/O, thus providing a more accurate breakdown of response times.

Figures 5.18 and 5.19 demonstrate the usefulness of the measures defined above. The x-axis in Figure 5.18 shows an increase in the percentage of requests that reach the middle-tier and also require access to the back-end database. The response time increase coincides with an increase in the CPU contribution of the back-end database,[2] thus indicating that the back-end database is becoming a CPU bottleneck. The network metric indicates no anomaly (the value of Ω remains below 1%).

[2]Note that all reported response times are measured at the front-end HTTP server. Client-perceived response times are not under the control of a server farm operator.

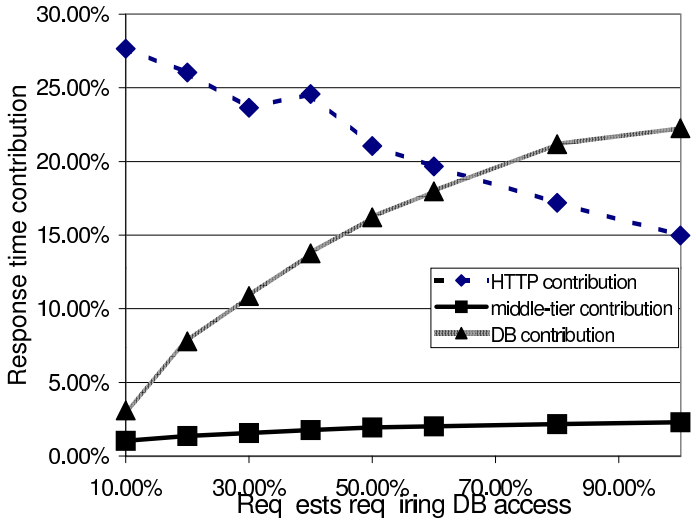


Figure 5.18: The impact of increasing the number of requests that require database access

The second measurement series focuses on network congestion. We create an artificial network bottleneck between the front-end HTTP and middle-tier FCGI server by limiting the full-duplex bandwidth between the two services using Linux's traffic shaping feature. Figure 5.19 shows the positive correlation between high response time and the newly-defined Ω metric. As the slow network also effectively disables available processing capacity by tying up processes with sending and receiving messages longer than they should be, the CPU contribution of the individual tiers actually decreases.

These measurements show that PMaps help pinpoint the performance problems caused by insufficient network or CPU bandwidth at arbitrary tiers. Of course, PMaps' service-centric statistics should be viewed in conjunction with component-level resource statistics to put the metrics' into perspective. The fact that the CPU contribution at a specific tier is great is not necessarily a prob-

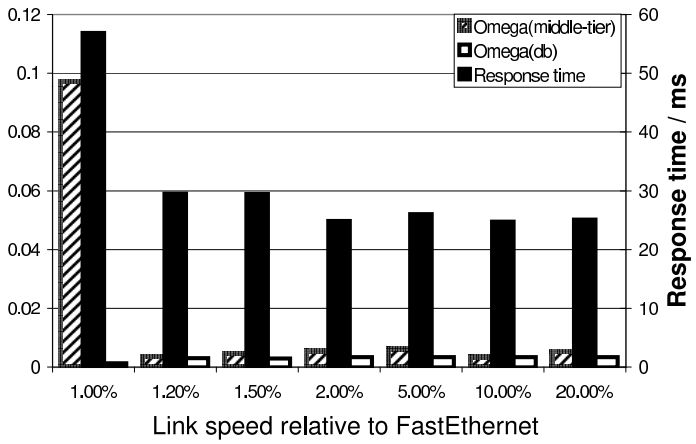


Figure 5.19: Reducing link speed between the HTTP front end and the FCGI middle-tier affects front end response time and middle-tier Ω.

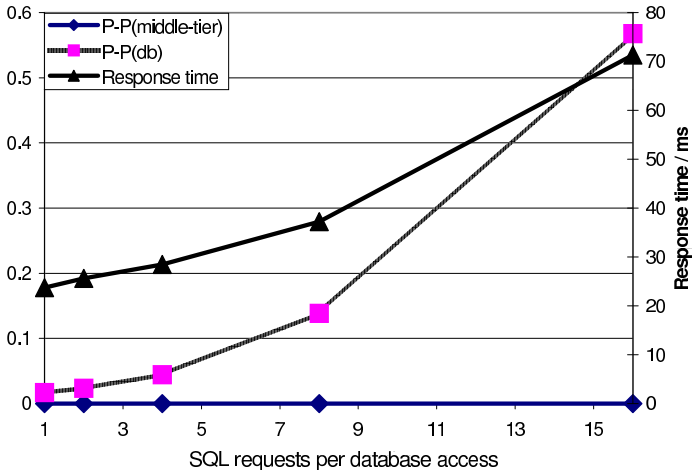


Figure 5.20: Ping-pong between middle-tier and database.

lem if the total resource contention at that host is low. This is the reason why traditional SNMP or host-based measurements (e.g., `vmstat`) are still important in determining the right course of action with respect to an observed performance anomaly. However, the following paragraphs will introduce a performance problem that is entirely unrelated to the metrics of traditional monitoring approaches: ping-pong communication.

Ping-Pong Communication: frequent, blocking, back-and-forth communication between peer-services is referred to as ping-pong communication (P-P). P-P is a service implementation or location problem. Unlike in the bandwidth-constrained problem, ping-pong does not overutilize network or CPU resources. It simply causes network latency to become the dominant component of overall response time. The P-P metric between two services, i (client) and j (server), in the context of executing in service class s.

$$\mathrm{PP}_s(i,j) := \max\left\{\frac{\texttt{BLOCK}_s^c(i,j)}{RECV\_CNT_s^c(i,j)}, \frac{\texttt{BLOCK}_s^s(i,j)}{RECV\_CNT_s^s(i,j)}\right\}\frac{R_s(i,j)I_s(i,j)}{\texttt{COUNT}_s(i,j)S_s^*} \tag{5.4}$$

To study the ping-pong problem, the number of simple SQL statements that are sequentially sent to the back-end database for every middle-tier request that require access to the database is gradually increased. The network bandwidth to the back-end is limited to 128KBps (full-duplex), which is only minimally more than the application's throughput requirements. Under this restriction, the sum of delays due to the application's ping-pong behavior results in poor response time — a delay is inserted for every request that accesses the back-end. Note, that results are sent in a bursty manner.

Only a small fraction of all requests (p_u = 0.05 and p_s = 0.1) access the database. Their corresponding SQL code is free of SQL aggregation statements to ensure that database processing is not the system bottleneck. Only one middle-tier-to-database connection is established per activity. Despite low processing and network requirements the response time grows steadily as the average delay between the database and the middle-tier increases.

Figure 5.20 clearly shows a significant increase in response time as the number of SQL statements per-request increases. Since neither the CPU contribution metric, defined in Equation (2), nor the network (`ntop`) indicates any overload, Figure 5.20's P-P metric points to a ping-pong problem between the middle-tier and the database.

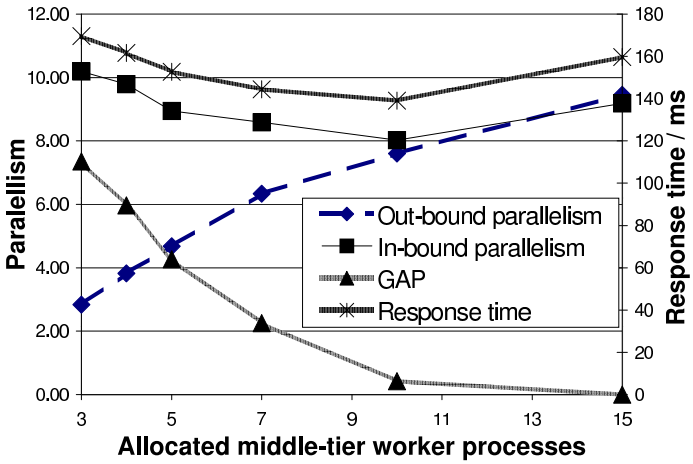


Figure 5.21: Measuring the middle-tier's degree of parallelism

The ping-pong problem can be alleviated by co-locating the middle-tier and the back-end service — thus reducing propagation delays — or by requesting the responsible programmer to improve the implementation's efficiency.

Parallelism: In addition to the ping-pong problem, services often suffer from too little parallelism or too much thereof. The response time curve of Figure 5.21 shows that both the excess and the lack of parallelism degrade performance. Most services leave it up to the administrator to determine the degree of parallelism. A service that exhibits too little parallelism may block on some slow resource and waste time waiting without making progress on any request. This effectively reduces server capacity without using any resources. On the other hand, excess parallelism wastes resources on scheduling and thread coordination overheads. While excess parallelism has less dramatic effects than a lack thereof, it can reduce server performance.

How does a system administrator estimate the degree of parallelism to allow for a network service? Seda [152] presents an interesting study about a self-calibrating Java framework to, among other things, address this particular problem. However, most applications are not self-scaling, and therefore, committed not to change applications, one must provide a means for system administrators to accurately assess the required degree of parallelism.

This leads to the GAP metric, which identifies the lack of parallelism but can also be used in reducing excess parallelism. The GAP metric takes advantage of the generated dependency map of services, and compares a service's concurrent incoming sessions against outgoing ones — the difference is the GAP. To determine the GAP one examines the number of processes that are blocked in tier n (waiting for services in tier $n+1$). If this number is much smaller than the fraction of tier $n-1$ requests for tier n services that will eventually propagate to tier $n+1$, we say that a parallelism GAP exists. The assumption is the tier n service is not dequeuing enough requests to match the incoming degree of parallelism. This is not always a problem. For example, if the server, on which the tier n service is hosted, is fully-loaded, then adding parallelism will not reap any benefits.

To compute the GAP metric accurately, one must consider that only a fraction of the requests received at tier n will cause one or more sessions to be initiated with services of tier $n+1$. This consideration leads to the following definition.

Let $\overline{\texttt{FAN}}_{in}(\sigma)$ and $\overline{\texttt{FAN}}_{out}(\sigma)$ be the sets of all services in service σ's fan-in and fan-out, respectively. The value Δt is the time elapsed since PMaps statistics were reset. Note that the GAP

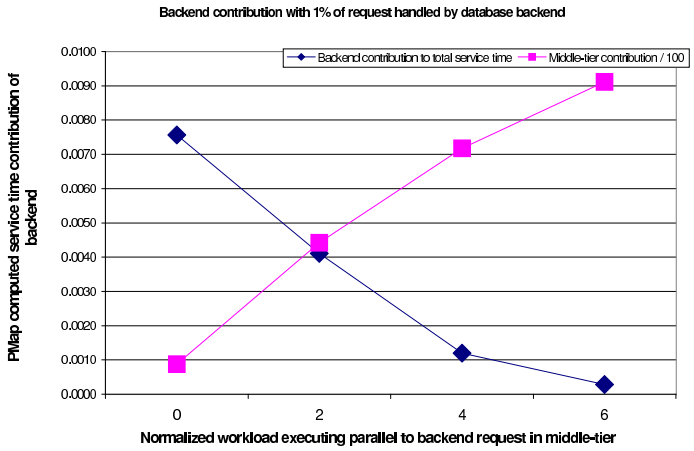


Figure 5.22: Overlap between middle-tier and back end.

measure is written without a service class index j. To compute GAP with respect to j one simply needs to add the subindex j to fan-in/out and every collected measurement. The subindex is omitted to simplify the equation.

$$\mathrm{GAP}(\sigma) := \max\left\{0, \frac{\sum_{\gamma\in\overline{\mathrm{FAN}_{\mathrm{out}}}(\sigma)} \mathtt{COUNT}_{\sigma,\gamma}}{\sum_{\gamma\in\overline{\mathrm{FAN}_{\mathrm{in}}}(\sigma)} \mathtt{COUNT}_{\gamma,\sigma}} \frac{\sum_{\gamma\in\overline{\mathrm{FAN}_{\mathrm{in}}}(\sigma)} \mathtt{COUNT}_{\gamma,\sigma}}{\Delta t} \frac{\sum_{\gamma\in\overline{\mathrm{FAN}_{\mathrm{in}}}(\sigma)} S_{\gamma,\sigma}}{\Delta t} - \frac{\sum_{\gamma\in\overline{\mathrm{FAN}_{\mathrm{out}}}(\sigma)} \mathtt{COUNT}_{\sigma,\gamma}}{\Delta t} \frac{\sum_{\gamma\in\overline{\mathrm{FAN}_{\mathrm{out}}}(\sigma)} S_{\sigma,\gamma}}{\Delta t}\right\}, \quad (5.5)$$

GAP is a unit-free estimate derived from Little's law [153] since it uses the arrival rate measurement and waiting time estimates to compute queue length estimates. The GAP metric essentially compares the queue composition at the front end with that at the back end and computes the difference between the measured queue length with what one would expect if the intermediary service had unlimited parallelism (e.g., fork-on demand).

This metric is evaluated by increasing the number of processes available in the middle-tier from 3 to 15, knowing that 3 processes would probably be too little parallelism and 15 too much to handle the given workload. Figure 5.21 proves this point. None of the previously-defined metrics reacts to this new problem. However, the GAP metric clearly indicates the lack of parallelism until the optimal degree of parallelism is reached at 10. Beyond this point, GAP goes to 0. Moreover, the GAP metric almost exactly identifies the number of processes that must be added if a service lacks parallelism; at 3 worker processes GAP measures 7, indicating that 10 processes would have been "optimal."

From this and other experiments with simple multi-threaded workload generators, one can establish the following engineering rule-of-thumb for determining the "optimal" degree of service parallelism: *adjust service parallelism so that the GAP metric lies between 0 and 1*. The rationale behind this rule is that if one lacks less than one process one can be sure that there is enough parallelism in the considered service to deal with the number of simultaneously-ready incoming connections. Note, however, that service parallelism should only be increased if real system resources are not overloaded (use `vmstat`, `ntop`, or equivalent).

Computing the GAP metric and determining the optimal degree of parallelism always requires complete knowledge of a service's fan-in and fan-out, which would be difficult or impossible to obtain without PMaps.

Asynchronous Communication: Multi-tier service programmers often use asynchronous com-

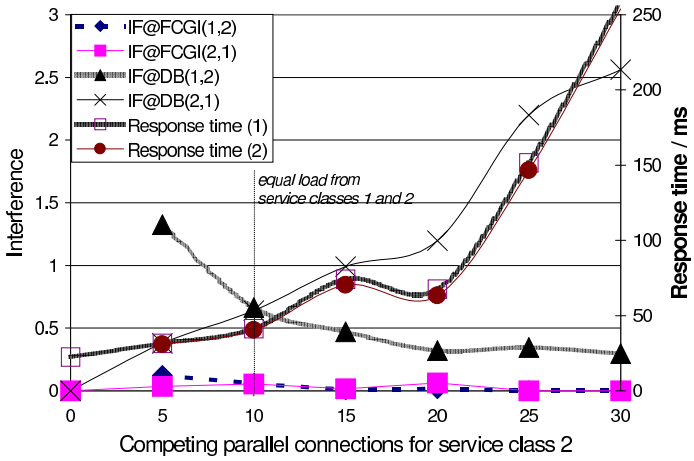


Figure 5.23: Interference between competing service classes.

munication to improve their implementations' latency and throughput. Asynchronous communication allows the middle-tier to do some useful work while waiting for a reply from the back end.

Conventional monitoring tools do not reveal if two tiers are proceeding with their work on the same request concurrently. So, how can one identify services that could benefit from asynchronous communication or should be co-located to hide the lack of asynchronous communication? The answer to this question requires two measures W and $\overline{W}$ that are introduced in Table 5.4.

In the context of one activity, the measure $\overline{W}_\sigma$ for service σ is defined as the sum of all times at which at least one of σ's processes that are associated with the activity is runnable. Figure 5.22, for example, shows how W_{FCGI} and W_{DB} respond to an increased amount of concurrent work in the middle-tier FCGI server.

Comparing $\overline{W}$ and W, it is possible to identify whether a multi-tier service could benefit from asynchronous communication. If $\overline{W}$ and W are close together for back-end service σ, the services in $\mathtt{FAN}_{\mathrm{in}}(\sigma)$ could potentially benefit from adding asynchronous communication or one could reassign σ to a host closer to services in $\mathtt{FAN}_{\mathrm{in}}(\sigma)$, in order to hide the lack of overlapped processing.

Service Interference: Service interference at a particular back-end service σ is assessed by tracing back accesses to σ. To determine whether some service class i interferes with the execution of another service class j at service σ, it is necessary to determine how much processing and communication service class i imposes at σ in relation to service class j. The interference of i with j at service σ is defined as

$$\mathtt{IF@}\sigma_{i,j} := \frac{W_\sigma^j}{2S_j^*}\left(\frac{W_\sigma^i}{W_\sigma^j} + \frac{\sum_{\gamma\in\mathtt{FAN}_{\mathrm{in}}(\sigma_i)} B_{\mathrm{in}}(\gamma,\sigma_i) + B_{\mathrm{out}}(\gamma,\sigma_i)}{\sum_{\gamma\in\mathtt{FAN}_{\mathrm{in}}(\sigma_j)} B_{\mathrm{in}}(\gamma,\sigma_j) + B_{\mathrm{out}}(\gamma,\sigma_j)}\right). \tag{5.6}$$

The above metric appears more complicated than it is. It captures the relative ratios of processing work and network traffic imposed by two competing service classes. As Figure 5.23 shows, this metric reveals that interference at the database server becomes significant as the number of competing requests from service class 2 increases while the request rate for service class 1 remains constant. Both request classes access the same middle-tier and back-end database through two different front-end HTTP sites.

The conclusion that interference is a problem would be very difficult to draw without a PMap-like mechanism, because it is impossible to determine the fan-in of a service with respect to competing service classes as the middle-tier would normally obscure any service-class-to-request mapping at the database server.

5.12 Comparing Performance Maps to Alternative Approaches

Current measurement tools for networks, clusters, and network services do not collect the information needed to diagnose multi-tier performance problems that are related to the interaction between peer services. This is due to the fact that current monitoring tools assume that the monitored system is a well-understood subject, they only quantify carefully defined and instrumented interactions. In contrast, PMaps' novel approach captures the dynamics of the system without prior knowledge of its components' interactions. Nevertheless, there are some profiling-oriented research prototypes for distributed application design that also infer the dynamic relationships between software components.

Parallel and distributed application optimization tools, such as Falcon [62], Jewel [83], Paradyn [96], and AIMS [158] instrument applications source codes to make them accessible to online profiling which, in turn, steers self-adaptation mechanisms inside the applications. These solutions assume the availability of application source codes. This assumption is usually not satisfied. Moreover, the above profiling tools are specially designed to support self-adapting code, so that their output is of little use for general system administration. They are geared towards tuning execution paths inside *one monolithic*, distributed application that conforms to one specific behavior or design model as opposed to inferring the interaction patterns between competing, loosely-coupled, multi-tier applications.

Non-invasive network service monitoring solutions implement various single-tier service monitoring approaches, but not multi-tier service monitoring since that would require correlating processing with network events. For example, the research prototypes Pandora [108] and Windmill [91] intercept in- and out-bound traffic of a server farm, parse service protocols, and infer appropriate performance metrics. Besides the fact that these solutions only apply to single-tier systems, they exhibit two additional disadvantages. First, they require protocol specialization for every possible front-end service. Second, they only let an administrator know that a performance failure occurred, without pinpointing complex interactions that caused the failure.

Commercial products: Tivoli's [138] performance measurement solutions and OpenView's [67] are able to provide accurate statistics and diagnostics with respect to an individual service. This is achieved by instrumenting individual applications using Tivoli's and OpenView's linked-in instrumentation frameworks. The basic idea is that applications need to signal the begin and end of an application-level transaction to the monitoring agent. In Tivoli, for example, this means that service implementors must embed service code between the measurement API calls, `arm_start` and `arm_stop`. The monitoring subsystem measures transaction times and aggregates them into a user-friendly console window. This approach can measure response times well, but fails to aid in problem cause identification if the transaction is carried out with the help of auxiliary processes; the monitoring middleware is unaware of their existence. Moreover, if a transaction involves subordinate transactions this basic approach does not identify whether any of the subordinate transactions are the bottlenecks. To account for the latter problem, Tivoli provides a correlation mechanism.

Tivoli's ARM framework [73] provides a "correlator object" to correlate dependent transactions across multiple tiers. To correlate two transactions, the front-end service must request a correlator object from the ARM framework—using a special API call—after beginning each new

transaction, and pass the correlator to every back-end service that it invokes on behalf of the ongoing transaction. When the back-end services receive the correlator object, they implicitly log the received correlator object identifier when calling `arm_start` and `arm_stop` with the correlator's ID as a function call argument. The main criticism of this approach is based on the fact that heavy source code transformation of the applications and communication interfaces between multi-tier services becomes necessary. The result of this transformation depends on the skill of the programmers who add the instrumentation and there is absolutely no guarantee that Tivoli will be able to correctly account for resource consumption for any service behaviors other than that of the single worker. Moreover, Tivoli cannot give any information regarding the impact that networking communication has on the hosted server applications. It will be difficult to integrate any legacy software into OpenView, Tivoli, and similar monitoring approaches.

There are areas in which PMaps overlap with other tools. Nevertheless, the PMaps approach to monitoring, a model understanding of event traces, sets it apart from the rest.

5.13 Summary and Conclusions

Considering the increasing popularity of multi-tier architectures in modern Internet server designs and the proliferation of different application service frameworks, model-based monitoring solutions, such as PMaps, will be indispensable in future multi-tier-aware UNIXes. Window's WMI [95] also needs to be updated for .NET with PMap-like monitoring support. Without abstract monitoring support for multi-tier services, it will become more manageable to pinpoint bottlenecks and configuration problems in large, outsourced multi-tier systems. Service interactions are too short-lived to be recognized by any simple status monitor add-on (e.g., SNMP, Tivoli, or HP OpenView). Moreover, such large systems change constantly so that system administrators may not be able to understand the dynamics of the system without a tool like PMaps.

PMaps implement fine-grained monitoring for multi-tier services and show, under reasonable assumptions, that

- they represent service behaviors accurately (albeit at a high level of abstraction),
- service class performance problems are diagnosed effectively even in the presence of shared back end services,
- modification of core service code is not necessary, and
- monitoring overheads are small and only incurred on-demand.

One drawback is that PMaps can only monitor services that are implemented according to one of its known service implementation models. Fortunately, most service implementations follow a few standard implementation patterns, thus allowing PMaps to be used in a large number of system scenarios. Moreover, there is no conceptual restriction on adding more service models, if needed.

The most significant contribution of PMaps is that service sharing and system consolidation problems can be reliably diagnosed for many typical service implementation architectures without requiring any changes to the component service's internal logic. Only when services implement their own alternative thread and communication handling libraries is it necessary to access third party code.

PMaps have shown the possibility of tracking resource usage across multiple tiers, thus providing information that is crucial for the configuration of proposed distributed resource management strategies, such as the propagation-based resource controls of Virtual Services (Chapter IV and [118]) or Cluster Reserves [10] without requiring application modification.

The collected measurements indicate that the generation of PMaps affects the performance of complex services only minimally and temporarily. Moreover, the collected measurements are good approximations of true, application-level performance. Consequently, PMaps that are constructed without direct application support are a viable alternative to today's and future distributed application instrumentation approaches.

CHAPTER VI

Lazy Virtual Service Calibration: Design and Limitations of Online Resource Allocation Adaptation in Multi-Tiered Systems

6.1 Introduction

Chapters IV and V introduced the basic components for online resource allocation adaptation [2]: resource provisioning and performance monitoring. The purpose of online resource allocation adaptation is to maximize or minimize an external objective function (e.g., utility or delay cost) by changing the resource allocation for competing workload classes. Online resource allocation attempts to address the problem of how one would determine resource quotas for competing VSs.

If service providers (SPs) commit to application-level performance contracts for their clients, then it is often possible to derive an implicit or explicit penalty of failure to achieve performance objectives (response time and throughput). Alternatively, one may derive a utility function that specifies the utility derived for each completed session and level of QoS received by the client. The penalty cost could either be a direct cost, such as a discount granted to an outsourcing client commensurate with the performance degradation experienced or an indirect cost, such as the loss of future outsourcing contracts. In the context of this chapter the process by which an appropriate cost or utility is determined is not considered. Instead, it is assumed that the economic cost of service degradation has been determined for each VS in the system. The question addressed here is how to adapt resource allocations in a multi-tiered system that can be controlled using VSs and monitored using PMaps subject to an external objective function.

The proposed solution is an incremental resource allocation calibration mechanism (Lazy Virtual Service Calibration or LVSC) that is designed to reduce both computational overheads and resource allocation overheads while minimizing delay cost or maximizing utility. This chapter addresses two key questions:

- What type of online resource allocation adaptation scales well to multi-tiered, multi-resource, and multi-workload-class systems?
- What are the limitations of resource allocation adaptation in a multi-tiered system?

The reason why these questions are of great interest is that, as was shown in the preceding chapters, resource allocation (VSs) is associated with delays (1.9-6 μs per call) plus a loss in total throughput. Moreover, the collection of performance statistics (PMaps) are associated with overheads (1.5% overhead for monitoring). This implies that continuous optimization may waste precious computing resources by reallocating resources and assessing the current state of the system. Moreover, the inherent delays of online resource allocation cause the resource allocation to

significantly lag the system state if workload turns over within a short time. This lag is amplified when resource allocations must be changed remotely at a large number of loosely-coupled hosts.

To address the question as to what type of resource adaptation would be practicable in a multi-tier server system, we differentiate between lightly-loaded, loaded, and overloaded systems. Obviously, adaptation strategies are of little value in lightly-loaded systems because the system has ample resources to achieve all performance targets. Systems that oscillate between heavy and light load (including most Web-driven server systems [44]) may benefit most from online resource allocation adaptation.

LVSC minimizes the cost of performance failure (or maximizes the utility of the system) by calibrating or tuning existing resource allocations. Other competing approaches typically completely reallocate resources to maximize or minimize the external objective function at every optimization step. By online tuning allocations the LVSC approach becomes less sensitive to possible resource allocation and monitoring delays. Moreover, the tuning approach requires resource allocations to change only in small increments. Such incremental resource allocation changes are easy to accurately track in the OS regardless of the resource allocation mechanism implemented in the OS.

Two LVSC approaches are introduced, one for short-lived workload, such as the Web-driven request load of the experiments in the earlier chapters, and the other for adapting long-lived session-based workload such as the streaming sessions of a multi-media server. These two models present different challenges and adaptation opportunities. In the short-lived workload scenario, such as quick stock quote queries that require at most a few milliseconds service time, any request-specific adaptation is bound to generate a relatively high overhead. Thus, it is most efficient to adapt the resource quota for all requests of a specific VS as a homogeneous class. When individual requests are relatively long-lived, e.g., a video streaming session, then it makes sense to handle each session individually to maximize overall utility. If a session is long-lived, it is possible to adapt the QoS of the individual session by changing, for example, the encoding of the transmitted signal [114]. Lazy calibration approaches are proposed for both long-lived and short-lived workload scenarios.

There are several differences between the approach presented in this chapter and other adaptive QoS approaches. For example, Real Audio [114] adapts resource allocations only with respect to resource availability, and does not optimize resource usage with respect to any objective function.

Adaptware [4] optimizes resource allocations with respect to an external utility function by dynamically assigning different QoS levels to different requests in the context of a single resource server. Rajkumar [85, 86, 112, 113] extends this approach to multi-dimensional resource scenarios. Both approaches pursue the same global optimization strategy: maximize aggregate utility received from all simultaneous sessions by using a theoretically-optimal resource allocation algorithm (knapsack). The primary objective of this approach is to show that resource allocation can be driven by external optimization objective functions and that utility derived from such a system improves. Adaptware is a feasibility study of online adaptation showing some potential benefits. Rajkumar's work addresses the complexity of the online allocation algorithm by proposing an approximate solution to the knapsack problem to scale the algorithm to multiple resources. The problem that has not been addressed to date is the fact that the system state may not be known instantaneously and that resource allocations cannot be enforced instantaneously. Furthermore, these approaches do not consider that changing resource allocations in itself incurs system overheads.

This chapter shows that aggressive optimization of utility or delay cost by changing resource allocations requires the resource scheduler to enforce a large number of resource allocation change operations without providing any (in the worst case) and only little (best case) cost or utility benefit over LVSC. This observation challenges existing online resource allocation adaptation approaches

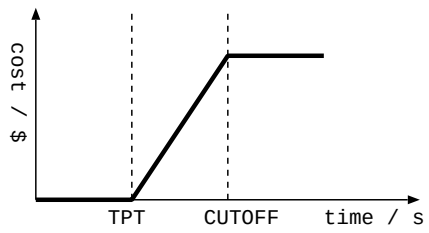


Figure 6.1: Delay cost profile for each request of a VS

that primarily focus on global utility maximization or cost minimization. Furthermore, this chapter shows that significant resource allocation delays, as encountered in a distributed system with a remote monitoring and administration console, are unlikely candidates for request-by-request online resource allocation adaptation.

Section 6.2 models the allocation optimization problem for short-lived requests such as those studied in the preceding chapters followed by a resource allocation adaptation algorithm and a comparative simulation study in Section 6.3. The resource allocation problem is updated in Section 6.4 to fit online resource adaptation strategies for the long-lived requests of a multi-media server. Section 6.5 presents the LVSC algorithm for long-lived session-based workload and a simulation study of its performance characteristics. Section 6.7 compares the LVSC approach to other proposed resource allocation adaptation schemes. A few details regarding the implementation of LVSC are discussed in Section 6.6. This chapter ends with conclusions in Section 6.8.

6.2 Basic System Model for Short-Lived Workload

The basic assumptions about the system and service implementation mirror those of the preceding chapters. This chapter makes an additional assumption that serving a request with delay incurs a quantifiable cost. This cost is incurred as follows. Up to a certain threshold of response time degradation the remote client remains indifferent and does not incur a cost. Once the threshold is crossed, cost increases linearly up to a maximal cost.

The threshold for request processing time is associated with each VS and called the *target processing time* (TPT). Similarly, a maximal cost is also associated with each VS. The maximal cost is incurred after the response time increases beyond a cut-off time (also VS-specific). The request is presumed to have failed or is aborted by the client once the request is delayed by more than the cut-off time. Such cut-offs can be client-imposed, application-specific, or server-side policies. For example, it is a rule-of-thumb that web users will most likely terminate web page downloads that take longer than 8 seconds, and many applications abort after a certain time interval. Figure 6.1 depicts the assumed delay cost function.

PMaps can be used to determine the hosts that are involved in processing the requests of a VS. Moreover, snapshots generated by PMaps can be used to determine the number of pending requests and their arrival times. In reality, these snapshots are only an approximation of the true system state because each snapshot of in-progress work is delayed (the current PMap prototype reports snapshots with a delay of $\sim$ 1s). However, to establish bounds on the best-possible of the introduced online algorithms we will assume that snapshots can be taken with arbitrary delays — including a zero delay.

The work associated with each request is scheduled as part of a specific VSs workload. In particular, all network packets and processing workload within one VS is treated equally in a

round-robin fashion. Between VSs resource quotas are enforced. Excess resources, i.e., unutilized VS allocation shares are allocated to best-effort workload. Resource shares that are left from best-effort processing are used in a round robin manner by all processes in the system — including processes that execute within a VS. This is exactly the implemented VS resource scheduling policy.

6.3 Adapting Resource Allocations for Short-Lived Request Workload

Our analysis uses the following notation: requests belonging to VS_i are submitted to a server and begin incurring a cost as soon as their sojourn time exceeds their target processing time, $\mathtt{TPT}_i$. Furthermore, no additional cost is incurred if a request's sojourn time exceeds its cut-off time, $\mathtt{CUT}_i$. Cost increases linearly up to a maximum total cost, $\mathtt{COST}_i$, for each VS_i.

The objective of resource-allocation adaptation is to assign resource vectors to VSs that minimizes the cost incurred by delayed requests (the dual approach of utility maximization will be considered in the context of long-lived workload in Section 6.4). Five algorithms are presented as solutions for the online allocation-adaptation problem: static (no adaptation), optimistic, amortized, bulk, and LVSC. The execution skeleton for the three dynamic, non-LVSC algorithms (optimistic, amortized, and bulk) is almost identical. All dynamic algorithms except LVSC dedicate some resource to the VS which, if delayed on the considered resource, adds the greatest cost estimate. This estimate changes as new requests arrive, as requests are finished, and as requests are forwarded from one server/service/resource to another. The main difference between the three dynamic scheduling algorithms is in their cost estimation method.

Optimization skeleton: On arrival of a new request or departure of a request at resource/service r do:

```
for all vs_i do
  if costEstimate(vs_i, r) > hi_cost_est then
    hi_cost_est := costEstimate(vs_i, r)
    hi_cost_vs := i;
  end if
end for
for all vs_i do
  if i == hi_cost_vs then
    allocate 100% of r to vs_i
  else
    allocate 0% of r to vs_i
  end if
end for
```

The computation of `costEstimate` for the dynamic resource allocation-adaptation algorithms is described below.

6.3.1 Static Allocation

The static allocation algorithm does not reallocate resources based on current resource demands. Instead, the system administrator allocates resources in accordance with average resource needs for different VSs. While this algorithm does not dynamically optimize cost, it still manages to capture some cost savings compared to a system without resource allocation because "high-cost workload" is effectively insulated from low-cost workload by VSs. This approach essentially takes

the statistics obtained by PMaps and translates those into VS resource allocations. This process is a pure one-shot resource allocation, which is used as the baseline for the following simulation study.

6.3.2 Optimistic-Cost Resource Allocation: Highest Real Cost Rate First

The optimistic-cost resource-allocation algorithm only incurs cost for a request r_j when the current delay $\delta(r_j) > \texttt{TPT}_{\texttt{vs}(r_j)}$. So, those VSs that are already violating target processing times will be associated with a non-zero cost estimate. If no VS is violating its TPT, then resources are allocated in a best-effort manner. This usually means that front-end servers are scheduled in a best-effort manner and back-end servers are scheduled in accordance with the specified cost function if the system is overloaded.

The benefit of this algorithm is that it will change resource allocations infrequently if TPT is violated infrequently. This leads to a relatively stable resource-allocation algorithm. The disadvantage of this algorithm is that only those VSs that are already incurring cost will be expedited, while VSs that are only likely to incur a cost, are not. Furthermore, this algorithm effectively puts the burden of QoS differentiation on the later request processing stages.

6.3.3 Bulk-Cost Resource Allocation: Highest Potential Cost First

The bulk cost estimation method is the pessimistic counterpart of the optimistic-cost estimation method. Instead of waiting to incur the cost until requests truly violate the TPT, the bulk-cost estimation algorithm assumes for every pending request that it will incur its maximal cost. Thus, for the scheduling of individual resources, the algorithm will dedicate the resource to those VSs that have the greatest possible cost penalty if all of its requests that need a particular resource fail.

At every resource this scheme leads to a modified weighted round robin scheme, in which the weights are determined by the arrival rate and delay cost of the VSs. The advantage of this scheme is that it balances cost-in-queue at every resource. The disadvantage of this scheme is that it cannot account for the fact that requests of different VSs may have different processing or communication requirements. This means that requests with large TPTs are essentially promoted because it is assumed that their penalty will be incurred although their TPT is unlikely to be reached.

The benefit of this algorithm is that it is stateless, i.e., it does not require any additional timing information for requests to determine the cost estimate. For this reason, this cost estimation method is very scalable for multi-tier deployment.

6.3.4 Amortized Cost Resource Allocation: Highest Cost Rate First

The amortized cost resource allocation divides the cost of VS_j, COST_j, over the entire lifetime of a request, i.e., from its arrival at t and its failure at $t + \texttt{CUT}_j$. This method assumes that a fraction of the cost is always incurred, but the fractional cost depends on the tightness of the absolute deadline $\texttt{CUT}_j$. This estimate favors VSs with relatively high value of $\frac{\texttt{COST}_j}{\texttt{CUT}_j}$.

This estimation improves on both optimistic and bulk-cost estimation. It is intuitively superior to the optimistic-cost estimation method because it considers the fact that cost is potentially incurred before exceeding the TPT. The algorithm improves on bulk-cost estimation because the cost may not be incurred, depending on the tightness of the TPT.

The main problem with this algorithm is that amortizing the cost over request lifetimes may be meaningless if service time distributions are highly variant (e.g., a service either incurs no cost or

the entire cost). This variance problem is also present in dynamic real-time scheduling algorithms for real-time systems [81] and has not been solved to date.

6.3.5 LVSC

Instead of attempting to precisely track the different system cost estimates as is done in the previously-introduced adaptation strategies, we propose Lazy Virtual Service Calibration (LVSC). The proposed algorithm is "lazy," because it ignores a number of system changes until it reevaluates resource allocations. The algorithm assumes that it is not possible to precisely track the system's cost metric, and therefore, changes allocations incrementally, thus attempting to calibrate resource allocations in order to balance the rate at which delay cost is incurred by all concurrent VSs (necessary condition for cost minimum).

The cost rate estimate is computed per VS (i) and resource (j) by multiplying the average cost rate for VS$_i$, $\frac{\texttt{COST}_i}{\texttt{TPT}_i}$, and the number of requests in VS$_i$ that are waiting for resource j. This product is referred to as $\texttt{COST}_{i,j}$ (equivalent to amortized cost estimate). This cost rate estimate applies to the system if resource j is not allocated to VS$_i$. The algorithm also assumes that VS$_i$ will not incur delay cost if it receives a 100% resource share. The rationale of this choice is that one could not expedite this VS any more. This assumption could be improved by computing the amount to which a resource is over-subscribed. For example, if each request takes 1s of service time, the TPT is 10, and there are 40 requests outstanding, one may assume — at some random snapshot time — that the best-case cost is 30, if requests are scheduled FIFO. Unfortunately, computing this improved estimate would require consideration of scheduling policies, and is, therefore, not applicable to heterogeneous multi-tier server systems, and most likely too fine-grained for the purposes of LVSC.

LVSC uses the cost rate estimate $\texttt{COST}_j(1 - v_{i,j})$ for VS$_i$ at resource j, where $v_{i,j}$ denotes the resource allocation for VS$_i$ at resource j. Given this estimate, LVSC chooses $v_{i,j}$ for all VSs i at each resource j to satisfy the expression

$$\forall k, l \texttt{ with } k \neq l \; \texttt{COST}_{k,j}(1 - v_{k,j}) = \texttt{COST}_{l,j}(1 - v_{l,j}) \vee \texttt{COST}_{k,j} = 0 \vee \texttt{COST}_{l,j} = 0 \quad (6.1)$$

The cost rate balancing computation creates a new proposed resource allocation, ϕ_i, for each VS$_i$. This allocation is not directly enforced. It is weighted against the old resource allocation for VS$_i$ with a factor ω ($0 \leq \omega < 1$) to compute the new allocation, v_i' for VS$_i$ as:

$$v_i' = \omega v_i + (1 - \omega)\phi_i \quad (6.2)$$

.

The performance of LVSC is affected by ω and the number of client requests that are accepted in between updating resource allocation according to the above procedure (OptCycle interval). The following sub-section studies the impact of these parameters.

The performance of different resource-allocation adaptation algorithms is compared by looking at the cost incurred for scheduling an identical random workload for each algorithm. This cost is computed as follows. The workload consists of requests, $r_0, r_1, \ldots, r_N$. The delay experienced by each request, r_j, is $\delta(r_j)$ and its VS is $\texttt{vs}(r_j)$. As was mentioned earlier, each VS, VS_i, is associated with its own $\texttt{TPT}_i$ and $\texttt{CUT}_i$. Thus, the total cost of an algorithm, $\mathcal{A}$, with respect to a given workload is computed as:

$$\texttt{COST}_{\mathcal{A}} = \sum_{n=0}^{N} \max\left\{0, \min\left\{\texttt{COST}_{\texttt{vs}(r_n)}, \frac{\delta(r_n)\texttt{COST}_{\texttt{vs}(r_n)}}{\texttt{CUT}_{vs(r_n)} - \texttt{TPT}_{\texttt{vs}(r_n)}} - \texttt{COST}_{vs(r_n)}\left(1 - \frac{\texttt{CUT}_{\texttt{vs}(r_n)}}{\texttt{CUT}_{\texttt{vs}(r_n)} - \texttt{TPT}_{\texttt{vs}(r_n)}}\right)\right\}\right\} \quad (6.3)$$

The optimization of this cost metric could be done off-line if one knew the exact resource requirements of every request and all release times. This cost-minimization scheduling problem is equivalent to distributed resource scheduling problems in real-time systems [81] and, therefore, also NP complete. For this reason, theoretically-optimal online cost optimization is not feasible. Instead, one must provide a reasonable online heuristic for minimizing the cost metric.

While the most obvious comparison criterion between LVSC and other algorithms is cost minimization, another metric regarding the algorithm's overheads should be considered. The main problem of optimal resource-allocation strategies observed in simulation studies is that optimal schedules require changing resource allocations, frequently changes. In particular, real-time scheduling typically ensure that the most urgent or earliest deadline tasks or requests receive service and resources before others. This leads to frequent and drastic resource allocation changes. Clearly, a greater number of resource-allocation changes per unit of time is a sign of an algorithm that will perform poorly in a network of loosely-coupled servers because resource-allocation changes will lag the system's dynamics (Section 6.3.6 supports this intuition). Moreover, it may be entirely impractical to implement an algorithm that requires frequent resource-allocation changes. To account for the amount of resource-allocation modification required by an algorithm the ADAPT metric is introduced.

The ADAPT metric sums up the absolute resource-allocation vector changes during the execution of a specific workload because the required amount of resource-allocation modification over time indicates the difficulty of tracking a resource-allocation schedule. For example, an algorithm that makes no changes to resource allocations is very easy to implement (`ADAPT = 0`) whereas an algorithm that changes the entire resource allocation within 10ms is difficult to implement. Assume the resource-allocation vector v_i for VS_i changes to v_i' due to the execution of a resource-allocation-adaptation step, then the absolute difference between these two vectors is computed as $\|v_i - v_i'\| = \sum_{j=1}^{R} |v_{i,j} - v_{i,j}'|$. Here R is the number of all resources in the system.

For each resource-allocation adaptation algorithm $\mathcal{A}$, one records the sequence of resource allocation vectors, $v_i^{(1)}, v_i^{(2)}, \ldots, v_i^{(N_i)}$, for all VS_i resulting from scheduling the workload $r_1, r_2, \ldots, r_M$. The $\text{ADAPT}_{\mathcal{A}}$ metric for algorithm $\mathcal{A}$ and a given workload is computed as

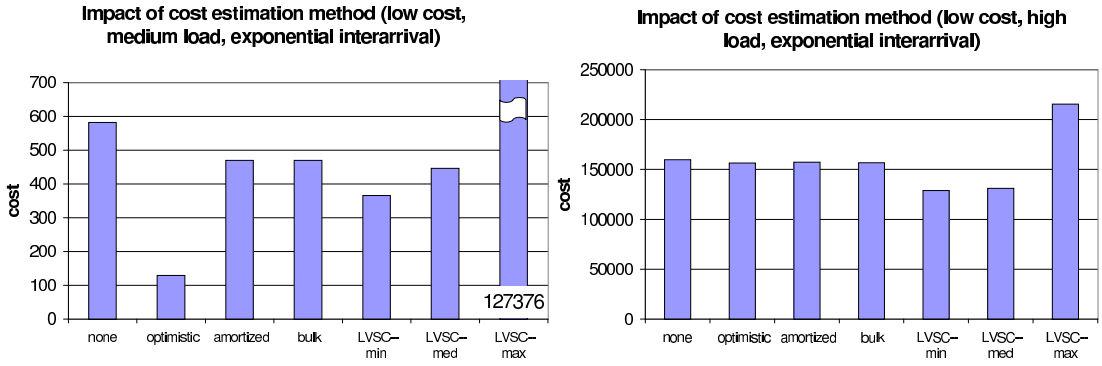
$$\texttt{ADAPT}_{\mathcal{A}} = \sum_{i=1}^{V} \sum_{j=2}^{N_i} \|v_i^{(j)} - v_i^{(j-1)}\|, \tag{6.4}$$

where V represents the total number of VSs whose resource allocation is subject to adaptation.

The problem captured by ADAPT is in fact a problem with all distributed online real-time scheduling algorithms, a fact which has thus far been cloaked by assuming perfect knowledge about the resource requirements of each request and the current system state at all processing and communication nodes. Moreover, previous research on resource-allocation adaptation generally neglects the fact that resource allocation changes are enforced with some delay unless the adaptation algorithm preempts all workload processing.

6.3.6 LVSC Simulation

The simulation studies a 3-tier scenario (client, front end, back end). In this server scenario, clients' requests belong to one of three different request classes (gold, silver, bronze). As described in the system model for short-lived requests, each VS is characterized by its own delay-cost func-



(a) COST of running a medium workload of exponential inter-arrival and service time distribution against the different algorithms

(b) COST of running a medium workload of exponential inter-arrival and service time distribution against the different algorithm

Figure 6.2: Comparing the different algorithms in terms of cost

tion, its arrival rate and different processing requirements at the servers. Arrival rates are generated at three load levels:

light: approximately 20% average utilization

medium: approximately 55% average utilization

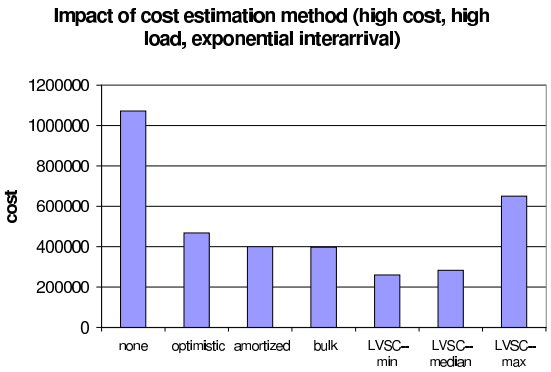
heavy: approximately 90% average utilization

To assess the contribution of the delay-cost functions relative magnitudes, we study a scenario in which the delay costs are 20, 10, and 5 for gold, silver, bronze, respectively (low cost scenario) and another scenario (high cost), in which the delay costs are 400, 40, and 4 for gold, silver, and bronze, respectively. This difference between delay costs obviously affects the benefits of online resource allocation adaptation. Whether delay costs are similar or vastly different for in real systems is outside the scope of this thesis; both cases are considered.

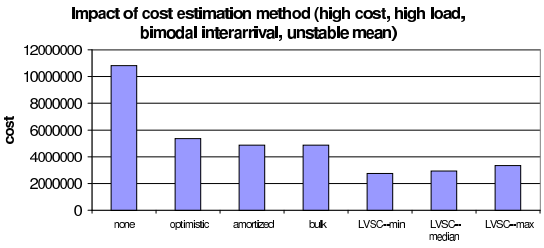
The performance of the different algorithms according to the `COST` metric

Figure 6.2 shows, there is still a noticeable performance difference among the different algorithms (30% at high load) even if the delay-cost differences between different request classes (VSs) are small. Overall, LVSC performs best. However, in the worst case, when LVSC parameters are badly chosen, the LVSC algorithm performs poorly; this can be easily avoided. One surprising result captured in the graphs shown is that the optimistic algorithm performs well in Figure 6.2(a). The reason for this outcome is primarily that the service time and inter-arrival time distributions in the experiment are exponential, i.e., memoryless, which favors optimistic-cost optimization because the likelihood of missing TPT increases only slowly as time progresses. In contrast, more tail-heavy distributions would have a suddenly increasing likelihood of TPT miss as time progresses.

The impact of different inter-arrival and service time distributions is shown in Figures 6.3-6.6. As these graphs show, LVSC outperforms the other algorithms with a greater margin when the distribution has a greater variance. The distributions of Figures 6.3(b) and 6.4(b) have unstable means. An unstable mean follows an independent random walk process. Every 1000 arrivals,



(a) Exponential inter-arrival and service time distribution



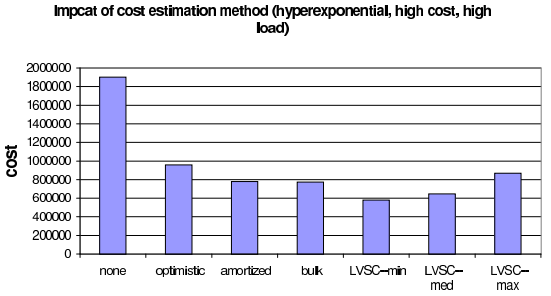
(b) Bimodal inter-arrival and service time distribution with unstable mean

Figure 6.3: Comparing the different algorithms at high cost, high load with respect to different inter-arrival and service time distributions

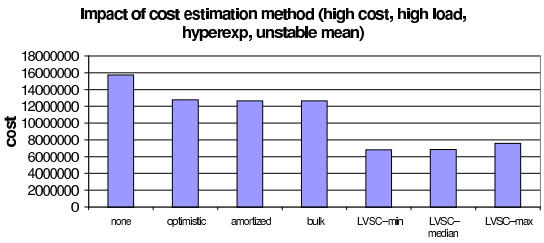
a random variable decides whether the mean will move up or down by a small amount. This movement is bounded within a fixed band around the mean. This behavior essentially increases the variance of the underlying distribution in an effort to mimic time-of-day load fluctuations.

Another series of simulation experiments analyzes the benefit of optimizing utility continuously, i.e., whenever the cost estimates change. This more aggressive optimization behavior would be beneficial for the optimistic algorithm because its cost estimate is based on actual cost that is being incurred. Figure 6.7 shows that the cost savings are only minimal when reallocating resources every time a request misses a TPT or exceeds the cut-off time. This minimal benefit suggests that aggressive optimization should not be of primary concern in the design of resource-allocation adaptation algorithm for short-lived workload.

Figure 6.8 shows the sensitivity of the LVSC algorithm to changes in its two main parameters. As expected, as the OptCycle interval (arrivals between resource-allocation optimizations) increases, cost grows. Moreover, while an increase in the weight of the old allocation (ω) makes less impact on total cost than increasing the OptCycle interval, its impact increases with increasing

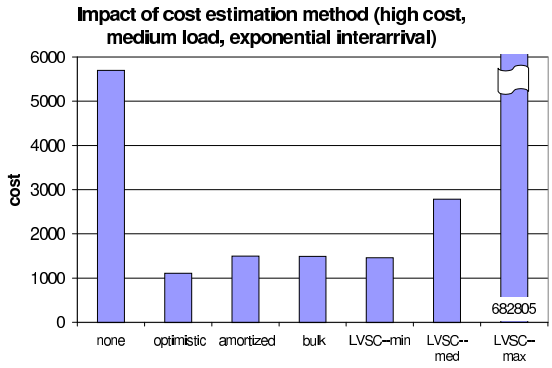


(a) Hyperexponential inter-arrival and service time distribution

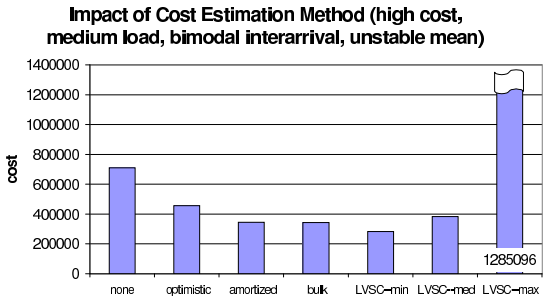


(b) Hyperexponential inter-arrival and service time distribution with unstable means

Figure 6.4: Comparing the different algorithms at high cost, high load with respect to different inter-arrival and service time distributions

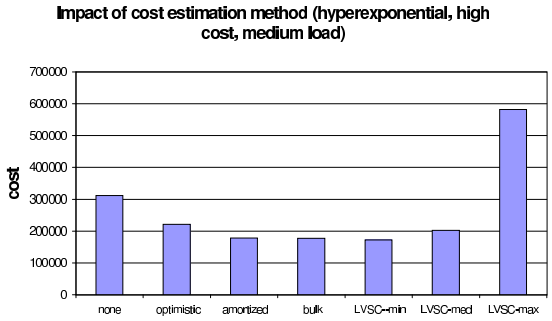


(a) Exponential inter-arrival and service time distribution

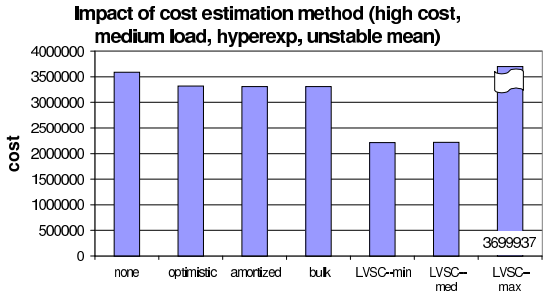


(b) Bimodal inter-arrival and service time distribution with unstable mean

Figure 6.5: Comparing the different algorithms at high cost, medium load with respect to different inter-arrival and service time distributions



(a) Hyperexponential inter-arrival and service time distribution



(b) Hyperexponential inter-arrival and service time distribution with unstable means

Figure 6.6: Comparing the different algorithms at high cost, medium load with respect to different inter-arrival and service time distributions

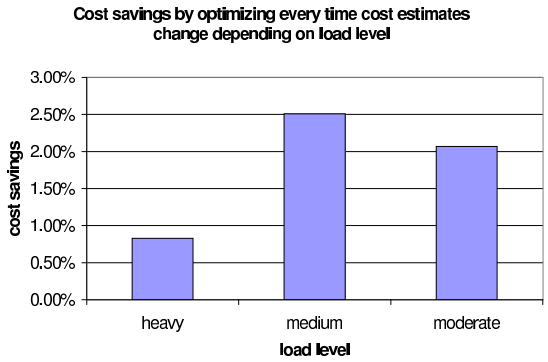


Figure 6.7: Reallocating resources aggressively, i.e., every time a TPT is violated or a cutoff timer expires is only of limited benefit

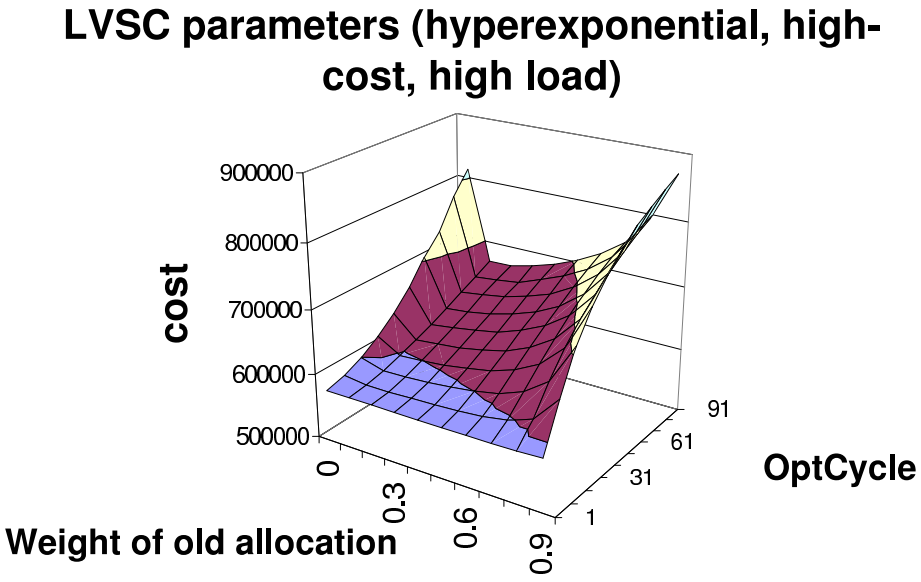
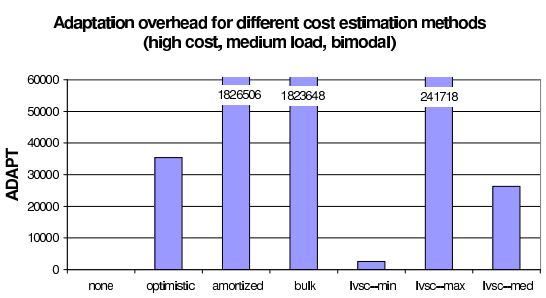


Figure 6.8: Impact of different LVSC parameter choices

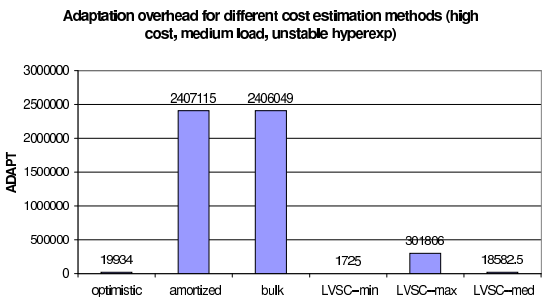
OptCycle values. The most important finding from this graph is that an instantaneous reaction to current demand without hysteresis or long-run averaging (expressed by the weight of the previous allocation) does not yield much additional benefit over lower overhead parameter choices. The LVSC-parameter-cost graphs look similar across all tested inter-arrival and service time distributions and load conditions. Figure 6.8 was chosen from the simulation results because it clearly shows all of the features of the relationship between cost and LVSC's configuration parameters.

Comparing the algorithms with respect to ADAPT

Figures 6.9(a) and (b) show that LVSC and the optimistic algorithm always outperform the bulk and amortized cost estimate-based cost minimization algorithms in terms of the ADAPT metric. Of the two low-overhead algorithms, LVSC requires less resource-allocation change than the



(a) Bimodal inter-arrival and service time distribution



(b) Hyperexponential inter-arrival and service time distribution with unstable means

Figure 6.9: Comparing the different algorithms at high cost, high load with respect to different inter-arrival and service time distributions in terms of the ADAPT metric

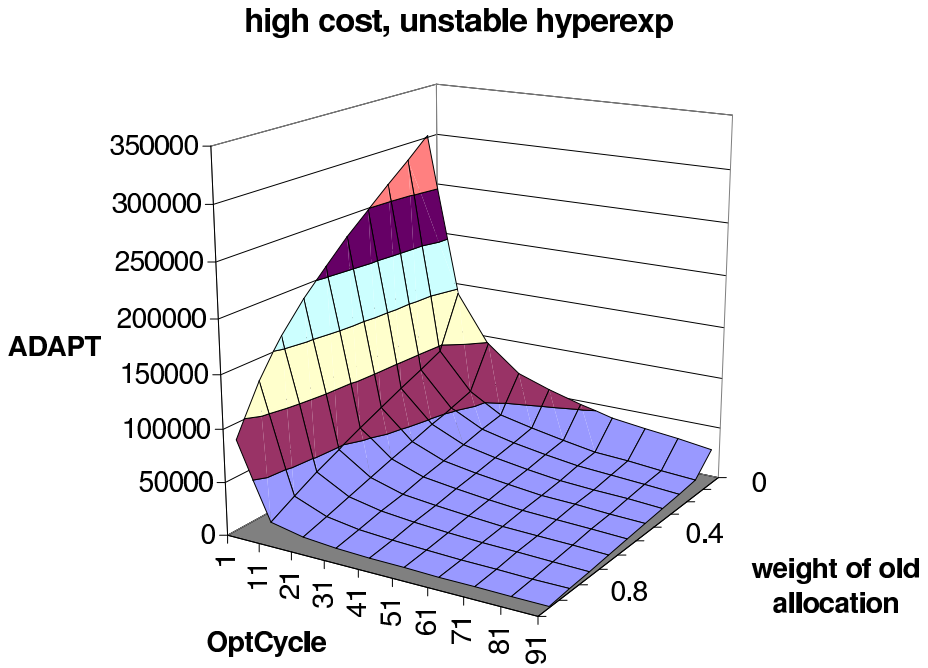


Figure 6.10: Impact of different LVSC parameter choices on ADAPT

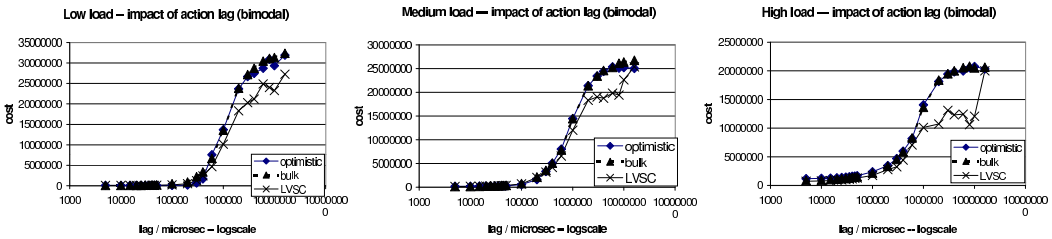
optimistic algorithm. If one combines this insight with the observation that the optimistic algorithm, overall, performs worst in terms of cost minimization, one may conclude that LVSC clearly outperforms its competition if one considers both COST and ADAPT metrics. Figure 6.3.6 shows that LVSC's required adaptation operation quickly decreases as the weight of the pervious allocation and the OptCycle interval increase. Comparing Figure 6.3.6 with Figure 6.8, one clearly sees the tradeoff between high adaptation overheads (large ADAPT values) and good cost optimization performance. The optimal parameters is system-dependent and cannot be exactly inferred by a theoretical means. However, choosing 0.4 as the weight of the old allocation and an OptCycle value of approximately 30 appears to be a good starting value for experimentation. These values incur only minimally more cost than the minimal cost incurred for all parameter choices. Also, the ADAPT metric for these values is only minimally higher than the absolute minimum for ADAPT on the given workload.

Comparing the Algorithms' Sensitivities to Action Lag

Instead of assuming instantaneous resource reallocation, we also examine the behavior of different resource allocation strategies in the presence of an action lag between taking a snapshot of the system's state, computing resource allocations, and the enforcement of updated resource allocations. Here we do not differentiate between the "amortized" and "bulk" cost estimate algorithms because their performance is very similar. This study makes several important points as follows.

1. Online resource-allocation adaptation only makes sense when action lags are relatively short.

2. As the lag between making a resource allocation decision and its enforcement increases, the cost incurred by the system increases. The cost increases follow a saturation function which

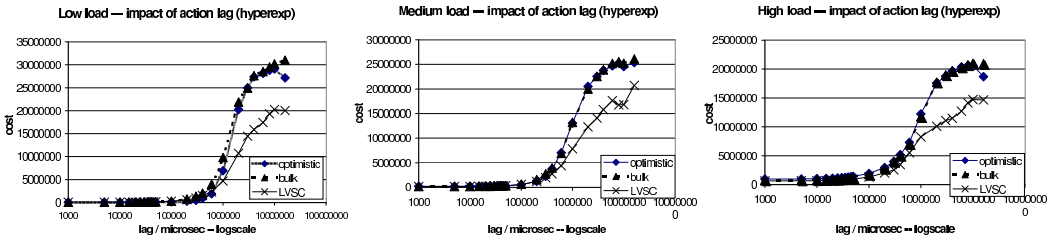


(a) Action lag impact under low load

(b) Action lag impact under medium load

(c) Action lag impact under high load

Figure 6.11: Comparing the different algorithms in terms of cost while changing the system's action lag (service time and arrival time are bimodally distributed)



(a) Action lag impact under low load

(b) Action lag impact under medium load

(c) Action lag impact under high load

Figure 6.12: Comparing the different algorithms in terms of cost while changing the system's action lag (service time and arrival time are hyper-exponentially distributed)

increases slowly at first, rapidly for a "short" range of lag values, and then slowly for very large action lags.

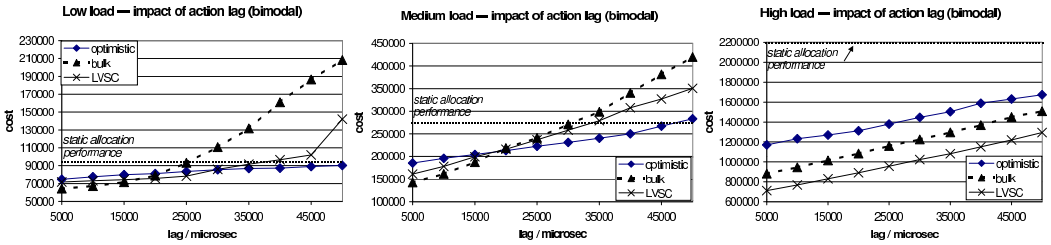
3. LVSC outperforms all other proposed algorithms consistently for long action lags. The margin of performance improvement by LVSC is greater for workload with greater variance of their inter-arrival and service time distributions.

4. The optimistic algorithm performs surprisingly well for small action lags under low-to-medium load. Under high load conditions LVSC performs best.

5. If the action lag is too large, the algorithms' performance will converge eventually (saturation).

Figures 6.11 and 6.12 show that LVSC performs comparatively well over the entire range of action lags (from 5ms to 16s). However, as Figure 6.12 shows its relative performance is better when the inter-arrival time and service time distributions are more variant (hyperexponential). For this reason LVSC should be favored for systems, in which the delay for resource-allocation enforcement is unknown.

Figures 6.13 and 6.14 show that the relative performance of the different algorithms for short action lags is surprisingly different from the previously-discussed overall performance. What makes this finding even more interesting is that it is only this short action lag range that is really

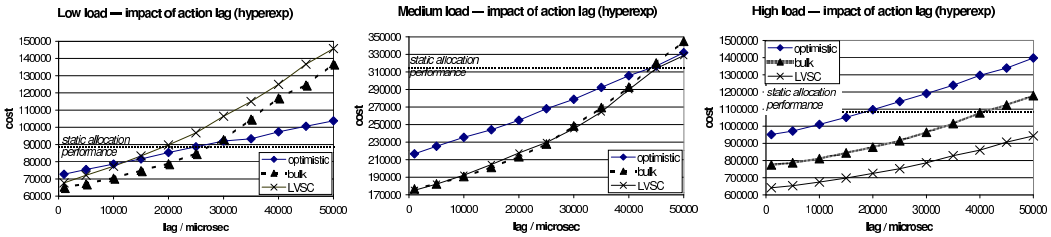


(a) Action lag impact under low load

(b) Action lag impact under medium load

(c) Action lag impact under high load

Figure 6.13: Comparing the different algorithms in terms of cost while changing the system's action lag — short lag < 50ms (service time and arrival time are bimodally distributed)



(a) Action lag impact under low load

(b) Action lag impact under medium load

(c) Action lag impact under high load

Figure 6.14: Comparing the different algorithms in terms of cost while changing the system's action lag — short lag < 50ms (service time and arrival time are hyper-exponentially distributed)

relevant because beyond the cut-off line (*static allocation performance*), a static resource allocation based on long-run average resource requirements performs better. Under high load conditions LVSC consistently outperforms all other dynamic resource reallocation alternatives. However, under medium and light loads and in case of bimodal inter-arrival and service time distribution (Figures 6.13(a) and (b)), cost minimization based on the optimistic-cost estimation method performs surprisingly well. The reason for this is that the workload does not exhaust the system's resources. This means that the optimistic-cost estimation method will not interfere with best-effort processing of the workload. Best-effort is a good policy if overload conditions are very transient and the action lag for resource allocation enforcement is large. However, if workload exhibits more variability, load bursts during which significant workload queues build become more likely, making the optimistic-cost minimization method disadvantageous (Figure 6.14).

The negative impact of the reaction time lag on optimality is more pronounced when the interarrival and service time distributions exhibit greater variances. This means that the different algorithms track each other more closely performance-wise. For example, in the case of a bimodal interarrival and service-time distribution, LVSC outperforms static allocation by a factor of 4 for short action lags. However, if service times and interarrival times follow hyperexponential distributions, then the performance improvement is reduced to only 30%. The relative performance among the dynamic resource allocation algorithms remains relatively unchanged under high load.

LVSC provides a 15-20% improvement over the next best algorithm ("bulk" and "amortized").

Final Assessment of Dynamic-Resource Allocation Adaptation for Short-Lived Request Workload

This study showed that the performance of the proposed cost-optimizing algorithms are not as drastic as one may expect—differences are largely within the same order of magnitude. All cost-optimizing algorithms clearly outperform static resource allocation when action lags are negligible. Under high load, LVSC performs 30% better than the competing algorithms presented in this chapter. However, the dynamic resource-allocation adaptation algorithms (optimistic, bulk, amortized) require a multi-tier system in which resources can be rapidly (re)allocated. This conjecture is supported by their performance in terms of the ADAPT metric as well as their performance in the presence of action lags.

If allocation choices are made locally, then action lags are likely within ranges that lead to good performance of online resource-allocation algorithms. However, if resource allocation is adapted via a remote resource allocation and monitoring station, then action lags are likely to be large, thus completely offsetting the benefit of online resource-allocation adaptation. This affects especially any solution to the resource-allocation adaptation problem that relies on global schedule optimization.

It was shown that monitoring the progress of computations without application cooperation (PMaps) consumes significant resources at the analysis station. If each server was monitoring its applications' states without their cooperation, then each server's capacity would be reduced significantly. Therefore, we conclude that the feasibility of resource allocation adaptation depends on whether the applications cooperate with a decentralized resource-allocation adaptation algorithms or not. This cooperation would entail announcing to the local resource-allocation agent how many requests of a particular VS are in progress and when they were received.

Accounting for different possible arrival and service time distributions, LVSC permits a good tradeoff between minimizing cost when enforcement delays are low, and provides competitive cost optimization with increasing delays. When resource-allocation delays are assumed to be very large ($> 1s$), LVSC is no longer an appealing resource-allocation adaptation choice because static resource allocation performs better.

6.4 Service Model for Long-Lived Sessions

The system model needs to be updated for long-lived session workload since a long-lived session may consist of many dependent requests or simply be streaming quasi-continuous media. Especially in multimedia streaming type workload it is not obvious how one would associate a delay cost with individual video or audio frames. The per-request delay cost model introduced earlier is too fine-grained for this type of workload. In the long-lived workload scenario it is more natural to assume that a client perceives a certain (continuous) utility while the service is delivered at a specific QoS level. If the server implements the same service at multiple levels of QoS, then the server can adapt to changes in demand and resource availability by switching the QoS-level for active client sessions [2]. If one simply changed resource quotas without any application coordination, the service could easily fail.

The server's ability to maintain a given QoS level for an individual session depends on the availability of necessary resources and the availability of an implementation (e.g., a CODEC) that implements the desired QoS. The concept of *service modes*, i.e., the aggregation of machine code

that needs to be executed to achieve a certain level of QoS along with the amount of resources that are required to execute the code, will be used to abstract from service implementation details.

Every adaptable service must support multiple service modes and the client must be able to decode the different service modes. This requirement is already implemented in various adaptable multimedia servers, e.g., Real and Windows Media Players provide client software that can digest media streams with varying bit-rates and sometimes different CODECs.

Definition 4 (Service Mode) *For a service that supports b service modes, each service mode $sm_i \in \{0, 1, \ldots, b-1\}$ contains information about which code/process to execute at the front and the back ends. Furthermore, the service mode specifies the corresponding resource-consumption vector (obtained using PMaps). In case of k resources and b service modes, the function* **rescon** *is defined as*: $\{0, 1, \ldots, b-1\} \rightarrow [0, 1]^k$*, with* $rescon(i) = (x_1, x_2, \ldots, x_k)^t$*, where* $rescon(i)$ *represents the (distributed) resource vector that must be allocated to one session to deliver service to a client in service mode i.*

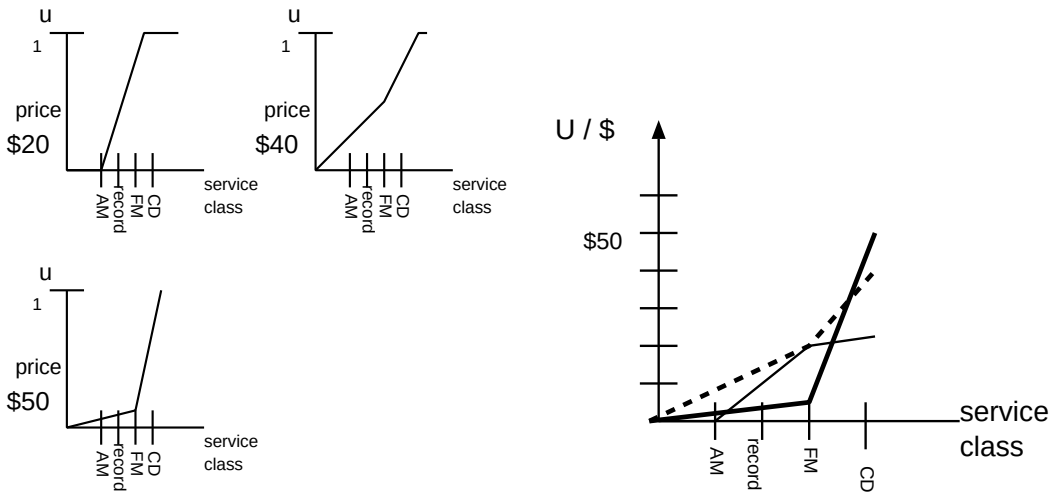
Note that the definition of a service mode alone does not imply anything about the user-perceived QoS, which can be achieved in many different ways. For example, to transmit a video, one may use more CPU cycles to process (e.g., compress or smooth) the video signal, thus reducing network-bandwidth requirements, or use less CPU cycles, thus requiring more communication bandwidth. This conceptually complicates the dynamic resource adaptation problem because the same utility could be achieved in different ways.

6.5 Reward Model for Long-Lived Workload

In the long-lived workload scenario, e.g., video-on-demand sessions, the basic model of reward and penalty of Section 6.2 is modified. As before, each client session is viewed as part of one VS, which is associated with its own utility function. However, since the session is long-lived utility is accrued throughout the lifetime of the session. Response time is not of interest, but the level of service received during the session, i.e., the codec and bit-rate. For example, a network client may evaluate the quality of an audio on-demand server as a function of what percentage of time he received a high-fidelity signal.

The client-perceived utility model associates a client-perceived utility with each service mode. Since the customer is likely paying for a service, it is possible to infer a utility function from his willingness to pay and his sensitivity to changes (degradations) in service mode (i.e., service-quality). The objective is to introduce a resource allocation algorithm for on-demand streaming servers that take into account client-perceived utility to determine a utility-maximizing resource allocation for the servers and VSs involved in serving the streaming requests.

Each VS_i is characterized by a maximal utility $\bar{u_i}$ (analogous to the maximal cost in the short-lived workload case) typically equal to its subscription price, and a relative sensitivity function s_i with $\texttt{Image}(s_i) \subset [0, 1]$ and the domain of all possible service modes (e.g., different CODECS and bit-rates). The sensitivity function captures a client's reaction to service mode changes. Thus, one can define the absolute utility function for VS_i, U_i (Fig. 6.15) as the product of $\bar{u_i}$ and s_i. The reason why this collapsed representation was not introduced right away is that the two components, sensitivity and maximal utility are independent. For example, two VSs that identify different client classes may have the same utility, but their sensitivity to reduced QoS may be different. To maintain a configurable model, we keep $\bar{u_i}$ and s_i separate.



(a) VS definition (b) Derived absolute utility

Figure 6.15: Subscription class and absolute utility.

Example: A SP (service provider) may decide to offer service to three different client classes, each of which is identified by its own VSs: CD quality, FM quality, opportunistic FM which upgrades to CD whenever possible. Each of these will be priced differently. Assume the SP charges $50 per month for CD quality, $20 per month for FM quality and $40 per month for the opportunistic VS. This defines the $\bar{u_i}$'s for the server setup. Based on user-feedback and/or observation and testing the SP may define a sensitivity function for each VS (Fig. 6.15(a)). Note that the sensitivity function was chosen to boost each client's absolute utility beyond the actual subscription price to indicate that a client may still perceive additional benefit even if he has not paid for it.

Definition 5 (VS Utility) *Suppose there are n active clients and the server farm offers k VSs, $vs_i, i \in \{0, \ldots, k-1\}$, and each client session j, $j \in \{0, 1, \ldots, t-1\}$ is associated with exactly one VS, $\mathtt{vs}(j)$. Each VS$_i$ is characterized by its maximal utility $\bar{u}(i)$ and its corresponding sensitivity function s_i. This sensitivity function is defined for all possible service modes that the server can provide.*

6.5.1 Online Adaptation of Resource Allocations for Long-Lived Sessions

An *allocation* is a mapping of active sessions to service modes, which must satisfy the following two conditions:

AC1: Each client receives service in exactly one service mode at a time. No client (active front-end session) that has already been admitted to the system is dropped. The function *mode* maps a client to a service mode and takes the client's ID as a parameter. This implies that online adaptation of streaming sessions is relevant whenever there is a change in the number of active sessions. This fact distinguishes it from short-lived workload. In the case of short-lived workload, resource allocations were subject to change whenever a request moved from one service to the next, when a new request arrived, when a request was finished, and possibly when requests missed their target TPT. This fine grain does not apply to long-lived sessions, which consume resources periodically.

AC2: In case of t clients, the following resource constraint must be satisfied:

$$\sum_{i \in \{0,1,\ldots,t-1\}} rescon(mode(i)) \leq (r_1, r_2, \ldots, r_k)^t$$

where $\leq$ is a vector comparison in which the left-hand side is less than or equal to the right-hand side in every component and the vector r represents the resource vector that is allocated for use by multi-tier services. In a dedicated system the vector **r** would equal $(1, 1, \ldots, 1)$.

The above conditions ensure that (i) an allocation specifies a service mode for every client and (ii) resources will not be over-booked.

The following discussion of the mapping algorithm focuses mainly on QoS adaptation to client arrivals and departures, but the model is general enough to accommodate other sources of change, such as an increase in the number of competing services on the same host or resource failure. In case of a resource failure, **AC2** may be violated and online adaptation has to guarantee that **AC2** will be met as soon as possible, upon detection of the failure. The same would happen if the service provider reduced the resource share that is available to all active services, e.g., by reducing the size of a resource partition.

The resource-allocation problem is to maximize aggregate utility over all client sessions subject to **AC1** and **AC2**. This problem is in fact only a slight modification of the well-known *knapsack optimization problem* [92]. Since the general knapsack problem is NP-complete, it is important to carefully examine the optimization problem to provide a solution that closely tracks the performance of the knapsack algorithm without its overheads. It is impractical to solve a multi-dimensional knapsack problem every time a client enters or leaves the system.

Formally, the optimization problem for t clients is stated as

$$\max_{mode} \sum_{j \in 0,1,\ldots,t-1} U_{sj}(mode(j)). \tag{6.5}$$

The maximization of utility is subject to **AC1** and **AC2**.

If dynamic programming is used, the time complexity is typically in $O(\text{SIZE} * M)$ where M is the number of items from which to choose and SIZE is the size of the knapsack. The items in the knapsack problem correspond to the service modes and their size corresponds to their required resource vectors. The objective function is the aggregate of all clients' perceived absolute utilities. An additional constraint is added to the knapsack problem in that each client must be assigned to exactly one service mode.

The size of the knapsack is obvious in the single resource case. If the resource is allocated at the granularity of $1/x$ ($x \geq 1$) relative to the total amount of a resource available, one can dedicate the resource to each client at *x* different consumption levels. The size of the knapsack would be *x*, which is usually not very large (< 10000), but as additional resources are added, the size of the knapsack grows multiplicatively. Assuming addition of one resource, one must multiply the reciprocal of the granularity of the first resource by that of the newly-added resource to get the size of the knapsack for the two-resource problem. This implies an exponential growth of the problem in the number of resources or servers considered. The exponential growth in the optimization space, as the resource-allocation granularity becomes finer, makes the problem intractable for online use in a multi-tiered server farm.

A Heuristic Approach to Solving the Allocation Problem

In the one-resource case, the proposed LVSC solution is the based on the traditional dynamic programming knapsack solution [87]. In the case of multi-dimensional resource vectors LVSC focuses on optimizing the utility with respect to the bottleneck resource. The bottleneck resource can be determined using PMaps. However, while packing the knapsack at the bottleneck resource, the algorithm must ensure that the chosen mapping of sessions to service modes does not violate resource constraints at the non-bottleneck resources. This heuristic works quite well if there are no resource-substitutes, e.g., processing at one host versus processing at another; a comparison of the optimal knapsack algorithm and the heuristic version shows less than 1% difference in long-run aggregate utility for random client populations. Even the naive method of choosing an arbitrary resource for which the resource allocation is to be optimized works relatively well under the assumption of utility being monotonically increasing as resource consumption increases in any resource vector component.

Since the problem is primarily due to the (theoretical) possibility of resource trade-offs, one may wonder how one could reduce the complexity that resource trade-offs add to the allocation problem. One solution is to consider resources that are perfect substitutes as a unit. For example, if processing at hosts A and B are perfect substitutes, then one may just consider allocating resources as a percentage of the combined virtual resource (processing at A + processing at B). This is essentially what load balancing devices do (Chapter II).

Complex resource trade-offs between, for example, networking and processing are generally absent from multi-tier services because servers are fine-tuned to deliver the best possible performance for the services they host. Changing the relative consumption of resources for any particular service will most likely lead to performance deterioration because the server would no longer be fine-tuned for its changed workload. Therefore, it is reasonable to assume that in a multi-tiered server scenario resource trade-offs are either between substitutes or resource trade-offs are not feasible for a carefully-tuned system. These considerations inspire the following algorithm:

LVSC Long-Lived Workload OptCycle Algorithm (*OptCycle*):

1. Pick the bottleneck resource for which the allocation will be optimized.

2. Pack the resource optimally for the first client under all levels of resource availability. Make sure that AC2 holds for every solution computed. Remember the maximum utility achieved for each allocation level of the resource considered.

3. Add next (n-th) client to the allocation, and try to serve him in a manner that maximizes the sum of utilities obtained by the n-th client and the $n-1$ clients, who can only utilize the resources left over by the n-th client. Determine the "optimal" allocation for all levels of resource availability. The utility achieved and service mode choice for each level of resource availability is recorded in a table (Figure 6.16). The utility for a service mode choice is computed by examining how much utility $n-1$ clients were able to gain from utilizing the resource share left over after allocating the n-th client at a specific service mode.

4. If there are more clients then go to Step 3 else output the allocation.

Step 1 of the *OptCycle* algorithm is not trivial but as pointed out earlier, this problem is solved by PMaps.

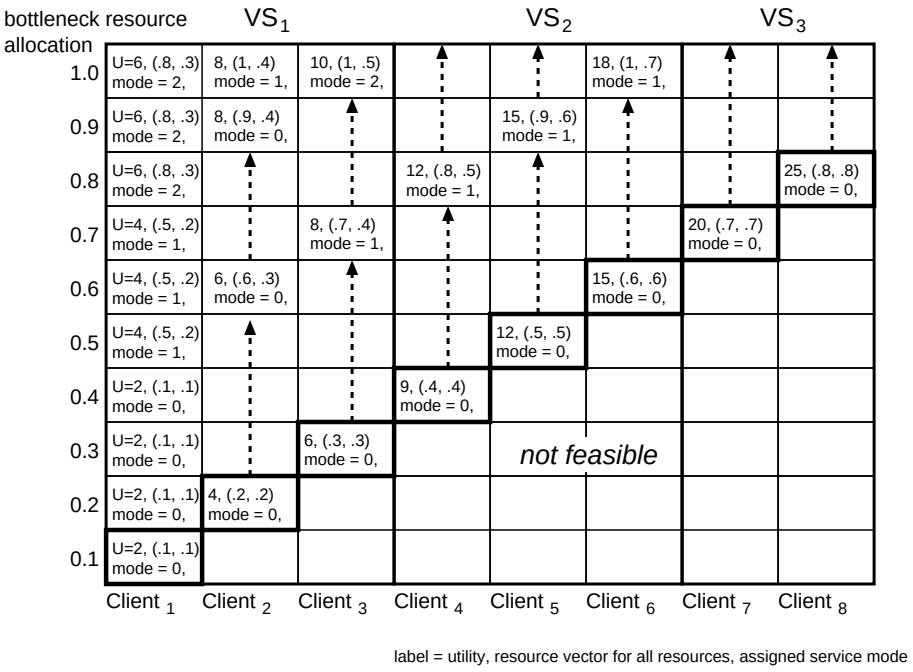


Figure 6.16: Computation of optimal resource allocation using dynamic programming.

Despite the significant complexity reduction from a problem that was originally exponentially complex in the number of resources, to one that is linear in the number of clients, one may still question whether this core algorithm will operate efficiently if executed for every change in the system's client make-up. There are some shortcuts that can be taken when implementing this algorithm. For example, the system's aggregate utility can only change if the client under consideration in the inner-loop of the above algorithm can be moved to a higher service mode or the utility changes for the previous clients while allowing the current client to be allocated at its lowest service mode. This shortcut minimizes the squares of the table that need to be computed (in Figure 6.16 the reduction is from 80 to 25). Nevertheless, the algorithm still has problems. First, it is still fairly complex. Second, it may require large changes in resource allocations in reaction to only minimal change in system state. To obtain a more practical solution, it is necessary to deviate even further from the optimal multi-dimensional knapsack optimization algorithm.

6.5.2 A Feasible Online Solution to the Session-Based Resource-Allocation Adaptation Problem

The first hurdle to overcome in adjusting the above algorithm for actual system deployment is that the resource-allocation algorithm itself takes time to execute. Should a client session be delayed while the resource-allocation adaptation executes? The answer is to introduce a *short-term repository* (STR), from which new clients draw some resources until the algorithm assigns the new incoming session to a service mode–possibly updating other sessions in the process. The resources needed for the STR are reserved in advance and subtracted from the resource limit. The STR for long-lived sessions is proposed for two reasons. First, it allows the resource allocation to take time without delaying the client request. Second, some client sessions may be too short-lived to justify optimization and reallocation. This is especially true when session lifetimes are

bimodally distributed.

The second problem is that frequent resource-allocation changes take a toll on optimality (as shown for short-lived work) and introduce significant overheads. The short-lived workload version of LVSC already solved this problem by deriving updated resource allocation from previous allocations. In the context of short-lived workload, this rationale was captured in computing the new configuration as a weighted average between the old configuration and a new proposed allocation. This fractional, averaging allocation is possible in the short-lived requests scenario because resource-allocation changes do not trigger service mode changes; the applications do not have strict resource requirements. The only adaptation criterion was delay, and all performance requirements were soft. This is not true in the case of streaming sessions for which delay between subsequent frames may render the service useless. For session-based workload, even a minimal change to the resource level of one ongoing session may require end-to-end service mode changes. This is a possibly expensive operation that should be avoided. Therefore, we derive a new allocation from existing allocations by introducing placeholder sessions in place of sessions that have just been terminated (Free-List [FL]). Each VS maintains a FL of recently-closed sessions. Resources tied up by these placeholder sessions remain unused until a new session of the VS arrives or until a time expires. Upon timer expiration, the resources consumed by a placeholder session are returned to the global resource pool.

Each new session arrival can now be satisfied by allowing it to take the place of a client of the same service class who has just left. This avoids triggering a time-consuming optimization process or changes to the existing resource allocation.

Whenever a new session arrives, the most convenient way to admit it is to take resources from its VS's FL. So, checking the FL is the first step in the admission algorithm. If this step is successful, the algorithm takes the resources from the FL, allocates them to the new client session and, and exits without updating any other sessions configuration.

If the admission algorithm fails to allocate the new client by using the FL, it allocates resources from the STR to it, which will suffice to immediately start serving the client at the lowest-quality service mode. What follows is a local optimization. Active client sessions are degraded for the new client until there are no clients who, if degraded, lose less utility than is gained by upgrading the new client to the next higher service class and also free sufficient resources for the upgrade. This local upgrade and degrade operation quickly gets stabilized and requires only a few resource allocation changes (less than the number of service modes). This behavior resembles the cost-rate balancing across VSs of the LVSC version for short-lived workload.

The local optimization strategy may obviously lead to inconsistencies and sub-optimality because the underlying optimization problem is discrete and requires global optimization. For this reason, it is important to reset the resource allocation from time to time to an allocation that is close to optimal. For this reason, one must periodically (every OptCycle arrivals) check whether the allocation is still close to optimal. This means execution of the LVSC long-lived workload version of the *OptCycle* algorithm. This resembles the approach taken by LVSC for short-lived requests. In this scenario, too, the performance of the resource-allocation algorithm did not suffer much by adapting resource allocations every OptCycle-th arrival. However, running the *OptCycle* algorithm too infrequently degrades utility.

Whenever the *OptCycle* algorithm is run, all resources that are tied up on FLs are released and OptCycle is executed to reseed the allocation to one that is close to optimal. The period of this check is an important design parameter, and is defined in terms of how many adaptation operations the greedy optimization strategy may perform before OptCycle must be run again.

serv. mode ID	res. cons. vector
1	(0.0078125, 0.00390625, 0.00390625)
2	(0.015625, 0.0078125, 0.0078125)
3	(0.0234375, 0.015625, 0.015625)
4	(0.03125, 0.01953125, 0.01953125)

Table 6.1: Service mode description

6.5.3 Simulation Results for Long-Lived Session-Based Workload

Various design aspects are studied in an event-driven simulation of the the system model running the LVSC algorithm. Each simulation data point is derived from the simulation of one day of server activity.

Simulation Parameters

To give a rough estimate of the performance of the proposed algorithm in realistic settings, the modeled server approximates a RealSystem 5.0 [114], Real Network's audio-on-demand server. This widely-used system supports four service modes: 14.4, 28.8, ISDN, and Dual ISDN. According to Real Networks, it is possible to support more than 1,000 sessions per server PC at 14.4 kbps when broadcasting live contents. With individual streams the performance is significantly lower. Due to the unavailability of any more accurate information about the number of streams supported, the server was assumed to support up to 128 sessions at 14.4 kbps each. This requires a bandwidth of 230 KB/s, which is manageable by a single workstation server and coincides with the capacity of single server when running the WebSpec99 benchmark. Table 6.1 shows the modeling parameters. Only one bottleneck resource is considered, since the bottleneck-determination problem is tackled in the previous chapter. Furthermore, the time granularity for the simulation is fixed at 2 ms.

To get a handle on client inter-arrival times and session durations, the simulation assumes that they are exponentially-distributed. This assumption is consistent with most queuing models of interactive applications. The exponential parameter for session durations is the same for all subscription classes in all simulations: 6×10^{-7} (a mean of 45 min). The performance of LVSC is tested under high, low, and very light loads. The corresponding VS simulation parameters are given in Table 6.2.

The server resources considered in the simulation are CPU, hard disk, and communication (in that order of appearance in the resource vector). The CPU was considered the bottleneck resource because it is involved in data copying, data retrieval, and protocol processing. Note that the bottleneck resource depends on the server setup. This simulation did not add distributed resources because it would have only increased the dimensionality of the resource vector without adding any additional insight.

The lowest quality is near AM radio with a 14.4 kbps connection. At 28.8 kbps one receives quality between AM and FM radio. Finally, ISDN facilitates true FM quality. Using both channels of ISDN guarantees near-CD quality. Each of these QoS levels is represented by a service mode. Clients are assumed to be capable of receiving service at all of these levels.

The modeled server offers three different VSs. The lowest quality might be of interest to those who want to access voice transmissions, such as radio learning at 14.4 kbps for $15. The price is used to generate a canonical utility function under the assumption that all clients are equally

VSID	$	Timeout/min	utility vector
1	15	45	(0.95, 0.98, 0.99, 1.0)
2	30	45	(0.3, 0.6, 0.98, 1.0)
3	50	45	(0.1, 0.6, 0.8, 1.0)

Table 6.2: VS description

Load	VS1/s	VS2/s	VS3/s
light	0.005	0.02	0.01
low	0.02	0.07	0.03
high	0.035	0.1225	0.0525
heavy (overload)	0.05	0.2	0.1

Table 6.3: Arrival rates for different VSs at different load levels.

sensitive to quality changes, however, placing a different value on quality. The second subscription class targets at those who want to listen to music at FM quality, ideally receiving service at 64 kbps. The cost is $30. The last and most expensive class is for CD quality, and served best at 128 kbps; it costs $50. See Table 2 for the exact utilities used in the simulation. The generated conditions for the simulation were mainly modeled after the motivating example in Section 6.5.

LVSC Enhances Service Availability

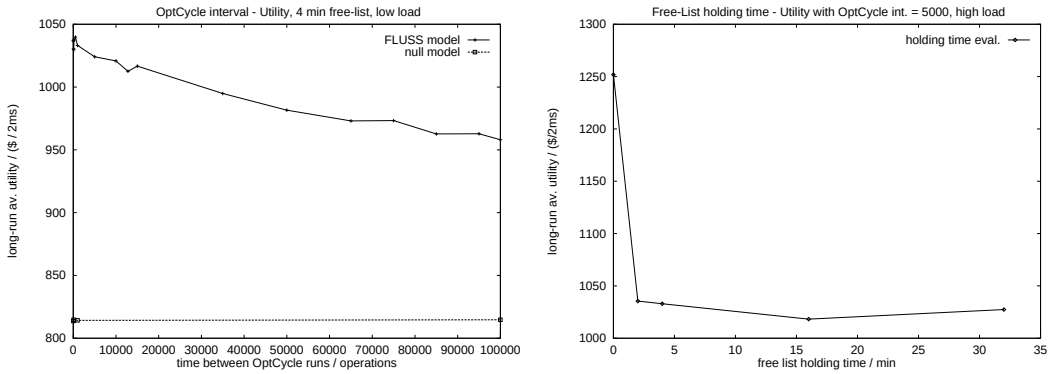
The LVSC model is compared against today's common practice in session management for servers that admit clients as long as there are sufficient resources to set up sessions for them. The server does not distinguish between clients, and simply tries to serve each client as best as it can. Furthermore, there is no concept of adaptation. Because of this model's simplicity it will be referred to as the Null model.

Intuitively, LVSC will outperform the Null model in terms of availability because it has the option of degrading clients' QoS, which is not supported by the Null model. Table 6.4 shows that Null does not scale well to high-load environments whereas LVSC does. The lack of adaptability to load variations is handled by excessive over-design in current multimedia systems that are not adaptive. By contrast, flexible QoS allows the service providers to tailor their systems more to their real needs, thus drastically decreasing the cost of the server and eventually the clients' cost.

Moreover, LVSC is found to be able to serve 30% more clients under low load than the Null model (40 as opposed to 30). Under high load this gap widens even more to 233% (70 versus 30). On approaching the overload region, the two policies drift apart even farther. The number of clients served in the LVSC model is sensitive to the removal of the short-term repository. Its

Load	LVSC	Null
low	0.00	0.12
high	0.00	0.27

Table 6.4: Average rejection rates: LVSC versus Null



(a) 4 min free-list, low load

(b) *OptCycle* interval 5000, high load

Figure 6.17: Tradeoff between optimality, OptCycle interval, and FLs (counterpart of short term averaging factor ω)

removal decreases the average number of clients served by 10% and results in a non-zero rejection rate. This negative impact is amplified as load increases. Clearly, adaptation and the ability to admit clients on a conditional basis improve the server availability significantly. Note that the STR was not required in short-lived request workload because the algorithm only adapted resource quotas for VSs and had no impact on the scheduling of individual sessions.

Allocation Improvements

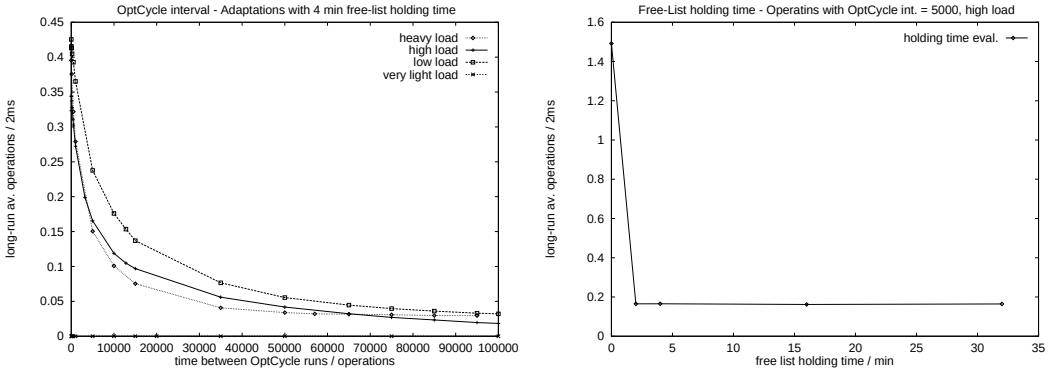
Since LVSC is utility-based, it is important to analyze how the aggregated utility of allocations improves through the use of LVSC. LVSC is compared against Null and the impacts of its design parameters are studied.

As expected, LVSC outperforms the Null model (Figure VI.17(a)) by 15–25%, depending on the choice of system design parameters. There is a clear anti-proportional relation between the inter-*OptCycle* interval and the achieved aggregated utility. This is because the local strategy might get stuck at local extrema if the solution is not optimized from time to time by a global strategy. The decline in the aggregated utility is larger under higher load, to an extent that even the Null model achieves greater aggregated utility if *OptCycle* is not run frequently enough.

As expected, FLs have a negative impact on the achievable aggregated utility, as they were not introduced to increase utility but to decrease overhead. The aggregated utility with FLs ranks approximately 20% lower than without them (Figure VI.17(b)). This is significant with respect to the utility gain achieved so far, so one must find a compelling reason for using FLs. The immediate decline of utility in the presence of FLs is due to the high arrival rates considered in the experiments. The lower the arrival rate the flatter will be the slope of the decline in utility in Figure VI.17(b).

Trading Allocation Optimality for Less Overhead

Decreasing overhead effectively means improving the aggregated utility, since the resources saved on adaptation operations can be used to admit more clients to the server, which would then increase the aggregated utility. The overhead per session adaptation is not negligible, since it requires to change both the resource reservation and the QoS level of server threads, potentially at



(a) 4 min free-list

(b) *OptCycle* interval = 5000, high load

Figure 6.18: Tradeoff between number of adaptation operations, OptCycle interval, and FLs

multiple hosts. Moreover, the fact that resource-allocation adaptations will be implemented with some delay results in a loss of adaptation benefit.

Knowing that aggregated utility decreases linearly with the time between *OptCycle* runs, it would be desirable to confirm that the number of adaptation operations decreases faster than utility so that one can effectively trade optimality for overhead reduction.

The decrease in the number of adaptation operations with the increase of inter-*OptCycle* intervals resembles a decay function, meaning that its decline will be larger than a linear function at zero and less as $x \rightarrow \infty$. If one increases the time between *OptCycle* runs from 0 to 15,000 adaptation operations, there is a 10-fold difference in the number of adaptation operations (Figure VI.18(a)). The aggregated utility for the same difference only changes by less than 10%, thus confirming the rationale of trading optimality for a reduction in adaptation overhead.

As mentioned above, the existence of FLs negatively affects utility (see Figure VI.17(b)). However, this effect is compensated by the fact that, while achievable aggregated utility decreases by 20% with FLs, the number of adaptation operations is reduced by approximately 87% (Figure VI.18(b)). Every time a client is admitted through FL, adaptation can be skipped. Extending the time to keep resources on the FL does not decrease the number of operations any further beyond the optimal point.

Assessment of Resource-Allocation Adaptation for Long-Lived Workload

The conclusion of the LVSC design for long-lived workload is that near-optimal resource allocations can be achieved even if one aims to minimize the number of resource-allocation changes. The principles used to arrive at such an algorithmic design are the same as those used for the design of LVSC for short-lived request-reply-type workload: retaining a history of previous resource allocations and occasional optimization as opposed to continuous optimization. Furthermore, long-lived workload introduced the need for the *drop list* (DL) of connections that have overstayed. The problem with long-lived workload is that an unfortunate lifetime distribution of active sessions can affect the utility derived from a server for a possibly very long time. Thus, the impact of "runaway" sessions is much larger than in short-lived workload. Experiments with a DL in LVSC for short-lived requests showed almost no cost savings over the version of LVSC without a DL.

We also observe that the utility increase for the simulated scenario due to online resource-

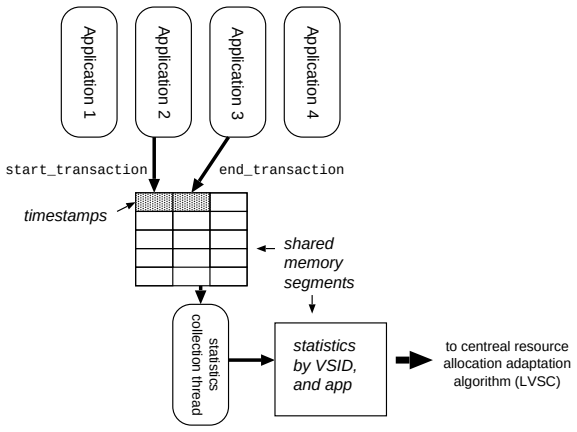


Figure 6.19: Simple response time and throughput monitoring

allocation adaptation is approximately 20% over best-effort resource allocation. While a 20% benefit may be considerable in some deployment situations, it is questionable if LVSC for long-lived workload would be implemented in a real system solely because of its utility benefit. The true benefit of LVSC is that it adapts gracefully to an overload situation by readjusting VS resource quotas to better match incoming workload characteristics.

6.6 Implementation

6.6.1 External Adaptation — for Short-Lived Requests

Since PMaps were not completely implemented when the LVSC prototype was built, it relied on application support for response time measurement. To this end, a simple transaction monitoring library (`xactmon`) similar to Tivoli's ARM [138] was implemented atop the VS-enabled Linux implementation. An application first calls `init_xactmon` to be attached to the monitoring memory segment. Whenever an application-level transaction begins, the application calls `xactmon_start_transaction` and it calls `xactmon_end_transaction` whenever the transaction is completed.

The monitoring framework also records implicit information, such as the VS with which the calling thread is associated, in order to compile the information required for the execution of LVSC (pending requests and their arrival times). As shown in Figure 6.19, transaction information is recorded inside a shared memory segment, which is evaluated by an external server thread. The server thread compiles statistics for each application and VS, which are also exported via a shared memory segment.

The system administrator specifies the TPT and cutoff times and maximal cost values in a configuration file and a simple daemon program keeps track of the current client set and resource situation at the server and readjusts VS CPU quotas as suggested by the LVSC algorithm.

The classification of requests is achieved by the previously-described VS workload classification and classification-propagation mechanisms. The implementation of LVSC for short-lived workload is relatively simple because it does not propose a new performance metric or design approach. In contrast, LVSC for long-lived, session-based workload does introduce service modes, which require a more sophisticated implementation approach.

```
adaptive struct timeval *getNextSampleTime(void)
{
        BODY_1
}

{
        BODY_2
}
```

Figure 6.20: Writing alternative function bodies for service mode construction

6.6.2 LVSC Service Builder Toolkit for Long-Lived, Session-Based Workload

The core piece of the LVSC implementation for session-based workload is a thread library, which implements scheduling, i.e., automated service-mode management.

Service modes were introduced in the context of online adaptation to provide alternative service implementations with different resource requirements. Thus, adapting the resource consumption of a session translates to adapting its service mode. To allow for continuous service-mode adaptation, applications must support the notion of a service mode and export it to an external adaptation thread that switches the service modes of different threads to adapt their resource consumption (see Figure 6.21).

To simplify the design of adaptive services, a C/C++ language extension was implemented that allows for "moded" execution of code. By placing the additional modifier `adaptive` in front of a C function, the programmer may define any number of function bodies, in increasing order of QoS-levels (Figure 6.20). Depending on the current service mode of the executing thread, different versions of the function implementation will be called.

The current prototype of the intermediary LVSC service mode compiler creates a mode-enabled C++ program, which binds to core LVSC thread library. The core library ensures that every thread that is created within the C program will be associated with a service mode in a manner that is transparent to the application logic. A separate service mode management thread adjust the service modes of concurrent threads. The adaptive program will execute different functions depending on

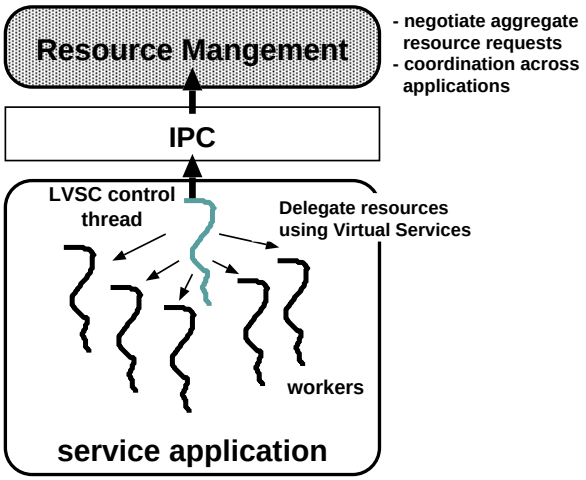


Figure 6.21: LVSC's Mater-Slave Thread Model

its execution mode. The execution mode of an executing mode-enabled thread is controlled by an outside mode-manager master thread (see Figure 6.21). The prototype assumes that it is possible to change the service mode of an application at any time. This requires the application programmer to be especially careful with respect to the equivalence of different function bodies that are implemented as alternatives.

A separate configuration file for a program (associated with the program by its execution path), describes the number of service modes and the resource requirements of each service mode. Moreover, each VS may specify a utility function and session lifetime for each application. If no utility function is defined, it is assumed that this VS will execute a specific service in best-effort mode, i.e., without any resource guarantees.

LVSC's service-mode adaptation management thread is started with the relevant parameters, i.e., OptCycle and FL holding time. Currently, the application must pass these parameters to the management thread. In future implementations the control thread will retrieve those values also from the configuration file. The STR is currently not implemented because unclassified workload is directly mapped to the system's best-effort workload class, i.e., workload without VS classification.

The classification of incoming sessions is not implemented in the service-mode adaptation library, but it is directly based on the VS framework. Whichever classification a session assumed via the VS classification mechanism is assumed to be applicable for the purposes of online adaptation. Of course, the application can also override this classification.

6.6.3 Alternative QoS Mode Adaptation Strategies

Applications that are developed in a manner that is not service-mode-aware cannot be adapted continuously. Nevertheless, sometimes the system may implement two alternative versions of the same service in separate service applications. While their network protocols may be identical, their internal implementation and source data repository may differ. For example, one may run two HTTP servers, one configured to serve high-resolution graphics and the other to serve text-only web pages [2]. These two implementations effectively implement two service modes, however, not inside a single service instance. If the OS rerouted incoming session requests to alternative service instances based on the incoming sessions' VS classifications (see Figure 6.22), one could easily implement a one-shot adaptation version of LVSC. The adaptation would be to reconfigure the VS-to-service instance routing function, depending on the resource situation at the server. Such an external mechanism for alternative service implementations is described as an example in Chapter III.

6.7 Related Work on QoS Adaptation

LVSC is one of several approaches that look at the resource-allocation problem from the perspective of cost minimization or utility maximization. Waldspurger [151] *et al.* introduced a resource-share enforcement scheme that directly uses a high-level economic model (lottery tickets) to drive resource allocation. In the proposed approach, every process is allocated lottery tickets for each resource time-slot. These tickets allow the process to participate in a resource lottery, which replaces traditional scheduling procedures. If a certain process' lot is drawn, it is granted access to the resource for one time-slice. This basic mechanism is used to allow distributed resource allocation based on bartering between processes in Spawn [150]. In Spawn, each process participates in a virtual barter market, in which it "sells" tickets for resources that it does not need, and "buys" tickets for needed resources. The main problem is that applications must collaborate in

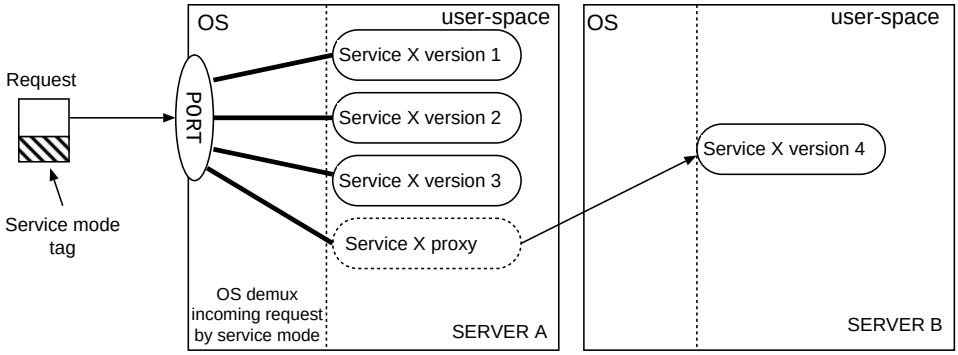


Figure 6.22: OS-level service mode implementation

trading resource tokens across hosts. Furthermore, the approach of bartering for time-slices when workload changes quickly is not feasible because the negotiation between processes has quadratic message overheads (every process may have to negotiate with every other process). Finally, the author [150] failed to provide a convincing argument based on cost and benefits for their complex resource-bartering algorithm. After some initial interest in this solution approach within the OS community, this approach has fallen into oblivion.

Abdelzaher and Shin [2–4] introduced two related dynamic resource-adaptation solutions in Adaptware, CLIPS, a single-host resource-management library similar to WebQoS (introduced in the next paragraph) and RTPOOL, a distributed resource-allocation adaptation scheme. In the context of this chapter, RTPOOL is the more relevant approach. RTPOOL was first to suggest that a distributed real-time application may potentially adapt to fluctuations in resource availability, e.g., in response to component failures. RTPOOL is concerned with scheduling tasks of a single application, in which individual application tasks are mapped to different QoS modes depending on resource availability (similar to the QoS modes introduced in this chapter). The paper presents a pseudo flight control application that is implemented with the help of RTPOOL. Unfortunately, the Adaptware approach does not seem to be applicable to generic multi-tier services, because it assumes a periodical real-time workload, such as the workload generated by sensor sampling. In contrast, multi-tier server environments face a workload that is non-deterministic due to the random nature of requests and their arrivals. The fact that units of work arrive randomly in a multi-tiered server system requires resource-allocation algorithms to be run more frequently, i.e., every time the system state changes. This fact places a greater emphasis on finding feasible online solutions to the resource-allocation optimization problem as opposed to applying direct translations of knapsack packing algorithms.

Spawn and RTPOOLs are the only two approaches that attempt distributed resource-allocation adaptation. Unfortunately, both of them fail to go beyond demonstrating the possibility of online resource adaptation. This chapter adds to this prior research by investigating the inherent overheads of resource-allocation adaptation and its value in the context of a multi-tiered server scenario. Both Spawn and RTPOOL consider predictable workload scenarios that are very different from the workloads experienced by Internet-driven servers. When workload is unpredictable, both approaches will suffer from the problem that it takes time to optimize allocations, additional time for resource allocations to take effect, and finally that it takes non-zero time to obtain a snapshot of the system's state. Moreover, both approaches require heavy application changes. LVSC for short-lived workloads can operate without application modifications.

HP's WebQoS [66] takes a more practical approach towards resource-allocation adaptation that does not require changes to existing applications. WebQoS recognizes requests in a middleware socket library interposition, in which it measures throughput and changes the order in which incoming sessions are forwarded to the managed application(s). Incoming session requests are classified (by source IP and target port) into service classes (equivalent to VSs). Each service class may be associated with absolute (reqs/s) or relative throughput targets. This approach is similar to the adaptive traffic-shaping extension for VSs that was introduced in Section 4.10. The WebQoS approach cannot control response times and even its ability to achieve throughput targets heavily depends on the arrival process. Furthermore, WebQoS does not optimize an external objective function, it simply attempts to enforce a given resource allocation.

Steere *et al.* [133] present a study of online resource-allocation adaptation and stabilization for interactive applications at the OS-layer. Their OS-layer adaptation provides much firmer performance controls than anything that has been discussed so far, because all applications are subject to the QoS adaptation and management strategy regardless of whether they bind to any specific support libraries or not. Secondly, in theory it would be possible to make their approach application-transparent. The key idea in [133] is to use application queue length as a feedback for resource allocations. The objective is to allocate resources in such a way that application work queues do not fluctuate or grow. The authors contend that this will lead to less bursty best-effort processing. This conjecture, however, is questionable because they do not consider how queues are affected by the workload arrival process. While the approach improves the predictability of best-effort execution times of straight line code and simple services, the feedback-driven scheduler is unable to optimize an external objective function that differentiates workload by value. Moreover, Steere's approach is not applicable to multi-tiered systems.

While the practical approaches presented thus far fail to reduce the complexity of multi-resource or multi-server adaptation, there is some theoretical work on multiple-resource QoS adaptation problems by Rajkumar *et al.*, attempting to address the complexity of solving multi-dimensional optimization problems online [85, 86, 112, 113]. However, the work does not provide any concrete implementation model nor takes into account the computer systems for which the algorithms are proposed. The proposed algorithms are essentially approximate solutions of the knapsack problem, which could be used as a drop-in replacement for the OptCycle algorithm proposed in the context of long-lived session-based workload. In particular, the Rajkumar *et al.* do not consider the costs of changing resource allocations and acquiring system snapshots. This chapter shows that the resource-allocation adaptation algorithm for a multi-tier system should be carefully engineered to dampen the oscillations of resource allocations. A straightforward application of theoretical algorithms that perform near optimally with respect to cost minimization or utility maximization fail to account for technical limitations, such as the cost of acquiring system snapshots and the delays associated with changing a resource allocation.

6.8 Summary and Conclusions

Building on the ability to assign resource quotas to activities in a multi-tiered system (VS) and the ability to obtain snapshots of the load state in a multi-tiered system (PMaps), this chapter shows that, with respect to dynamic resource-allocation adaptation schemes, LVSC offers a definite advantage over other conceivable schemes: slowly-changing resource allocations without sacrificing much utility or incurring extra costs. This was validated in the context of short-lived request-reply type client-server workload and long-lived session-based workload. This chapter also shows how the adaptation models for short- and long-lived workloads differ.

Unfortunately, short-lived and long-lived sessions are too different to be managed by the same algorithm. Short-lived requests are treated as continuous flows, whereas long-lived requests are treated as discrete resource consumers. Therefore, it was not possible to propose LVSC as a single resource management algorithm for all possible workloads as was expected at the outset of this research. Instead, two different LVSC versions that are based on the same principles were introduced. This means that requests would have to be classified into short- and long-lived ones.

Flexible resource allocation may offer a great advantage in building commercial server systems that are exposed to demand fluctuations and that can associate a real economic cost with request processing. This conclusion is not specific to LVSC but shared by all resource-allocation adaptation approaches. LVSC's specific contribution is that it makes online resource-allocation adaptation feasible for distributed server scenarios by greatly reducing the absolute number and amounts of resource-allocation changes.

The high service availability for high cost or important clients of LVSC-administered services, allows service providers to host their services on systems with less over-designed capacity because the system adapt gracefully to overload even if it operates near its capacity limit. Using VSs alone, it would only be possible to confine the impact of an overload situation to the VS that is causing it. VSs do not take into account the current demand situation at the server and cannot achieve utility-maximizing or cost-minimizing resource allocations.

Even though LVSC is a much more applicable approach for multi-tier resource-allocation adaptation than previously-proposed solutions for online resource-allocation adaptation, this chapter also presents challenges in online resource-allocation adaptation. First, the differences between handling short-lived request-reply workload and long-lived session-based workloads lead to complex system-management infrastructures that are likely to be misconfigured by typical system administrators. Second, this chapter introduced a realistic constraint into the discussion of resource-allocation adaptation: action lags.

Past research was usually based on the assumption that resource-allocation changes can be implemented instantaneously. Especially in large multi-tiered systems, this assumption does not hold, so one must consider the impact of non-zero resource-allocation enforcement delays. This chapter shows that non-zero resource-allocation delays are a serious challenge to the usefulness of dynamic resource-allocation adaptation, especially if adaptation is aggressive,

Another problem affecting online adaptation of long-lived, session-based workload using LVSC is that the specification and implementation of distinct service modes or QoS levels within one application is both laborious and error-prone. Hence, the cost of software development would increase drastically. If one considers the decreasing productivity of programmers as the size of an application grows, this challenge may become insurmountable. Without an automated approach for generating different service modes, online adaptation for long-lived sessions — other than bit-rate changes — is unlikely to be implemented. Moreover, the difficulty in identifying exact cost benefits of different levels of resource allocation may lead to malformed utility function descriptions, which is an additional obstacle for the adoption of online resource-allocation adaptation.

CHAPTER VII

Conclusions and Future Directions

7.1 Conclusions

This thesis has shown that resource-management concepts that have been successfully applied in single-tiered OS (resource partitioning and service monitoring) can be extended to multi-tiered systems, thus implementing basic OS functionality for a network of loosely-coupled servers. The approach taken here differs significantly from previous approaches to distributed system management, which have largely focused on creating a single system image abstraction. This thesis completely abandons the single-system objective and focuses instead on creating a configurable infrastructure (SDI) that allows limited coordination of OS functionality across loosely-coupled tiers.

The first contribution of this thesis is the analysis of typical client-server computing models as they are implemented in today's multi-tiered systems. From this design analysis, it is possible to derive a few basic models of client-server interaction, server-side processing, and interactions between tiers. The other contributions are based on the service models that were derived from this research.

The second contribution is to show the feasibility of policing server activities even if they propagate across different component services, system abstractions (e.g., processes), and machine boundaries. This objective can be achieved if one installs an appropriate workload-tracing and policing subsystem in the OS. The coupling between different machines is reduced to a simple Virtual Service ID (VSID) and low-level monitoring functionality that ensures that resource partitions are being enforced as configured. The complete system, consisting of resource partitioning, workload tracking, and overload shedding, is able to provide good insulation between competing applications that share services.

The third contribution is to show that it is possible to learn the workload-propagating behavior and tier-relationships of multi-tiered services without requiring their cooperation. This ability to learn how the components of a system interact and possibly how they interfere with each other will allow system administrators to diagnose problems, plan system capacity, and help with the configuration of VSs. PMaps strengthen the VS solution because they allow VSs to be applied to black-box services that fit the introduced service and communication models.

The fourth contribution is to show the limits of monitoring and resource control in a multi-tiered system. By examining the performance of online resource-allocation adaptation algorithms, it is shown that lags in the enforcement of resource allocations and lags in the recognition of the system's state adversely affect the performance of online resource allocation optimization. Furthermore, it is also shown that algorithms that seem as if they would track resource-allocation

optimality well, do not, or only marginally, outperform calibration-based resource allocation that changes resource allocations minimally as load conditions on the server change.

The fifth and most important contribution is to identify the fundamental principles that apply to a large class of OS extensions for multi-tiered systems and that are not yet addressed by any existing OS abstraction. Stateful Distributed Interposition (SDI) is the embodiment of these principles. Its design asserts that state propagation and state-dependent interposition are crucial for the integration of loosely-coupled servers if one pursues a system-wide management objective. Chapter III also shows how SDI can be implemented in today's OSs. Since SDI is a catalyst for system extensions, it effectively turns this thesis research into an open-ended challenge for system designers and researchers to utilize SDI or a similar mechanism to port other functionality, e.g., security mechanisms, that are well-established in single-tiered systems, to multi-tiered systems.

The proposed solutions, models, analysis, and conclusions of this thesis are built on the basic assumption of loosely-coupled and potentially heterogeneous multi-tiered server environments as they exist today. Therefore, our results of are highly applicable to current server OSs because this thesis does not propose any radically new OS paradigms, machine definitions, or application designs. This thesis also assumes that a multi-tiered infrastructure is utilized by multiple applications and client classes. A change of this assumption (single client class and single application environments) will render many of the conclusions irrelevant. However, a consideration of today's multi-tiered deployments suggests that the assumptions of this thesis are likely to hold for the foreseeable future.

7.2 Future Work

There are still several open research questions relating to the automatic identification of workload classes, i.e., VSs. A VS partitioning of the system is not always clear from the application domain. There are many system-management objectives that could use automated VS generation. For example, one may want to automatically differentiate between clients who enter the system via different front-end services. Furthermore, one may want to distinguish between clients based on the types of back-end services they utilize. It should become possible to generate such rules automatically by using the output of PMaps.

Both PMaps and VSs will need to be applied to various runtime systems, e.g., Java and `pthreads`. This porting effort will show that the core logic of PMaps is not only designed to be independent of a runtime system, but also to validate it again experimentally. Having Java and `pthread` instrumentations for PMaps would also enhance the real-world applicability of the implemented prototype.

SDI provides opportunities for numerous OS extensions for multi-tier systems. The most appealing future direction for SDI is the design of secure computing infrastructures, since today's multi-tiered systems are generally unsafe. The main weakness of today's server designs is that an attacker typically only needs to break one service or server in order to corrupt the entire system. This is due to the fact that systems are generally fortified only against outside attacks, not against attacks from the inside. So, once an attacker has intruded into a server network, there is often no further obstacle for him to compromise the entire system. SDI could be used to propagate immutable source IDs with all activities in order to indicate which IP address caused a request, even if the attacker tries to hide his tracks by logging into intermediary systems.

The SDI μ-language may need to be extended by adding a "set of context objects" abstraction, select ($\exists$) and assert ($\forall$) operators. Thus, the logic expressiveness of SDI guards would fall in line with first-order logic and potentially allow new kinds of applications. The impact of widening the

expressiveness of the SDI μ-language in this way is not yet understood.

The parsing of the SDI μ-language is also the subject of future work. Currently, SDI policies are evaluated in linear order. To speed up their evaluation, it would be advantageous to keep statistics regarding the likelihood of different guard clauses to be true, and to build an optimized decision tree instead of using linear evaluation order. This extension would enable SDI to allow possibly thousands of simultaneously-installed policies. As a result, it would become feasible to install dynamic rules, for example, policies that will apply only to one active communication session. A simple example use of such a capability would be to use SDI as a drop-in replacement for network address translation code.

A good future application of SDI that fits well with the objectives of this thesis is to use it to resolve namespace pollution in shared services. For example, a service may be used by two independent applications whose namespaces overlap, such as, two file server clients with directories named `/usr`. This problem is currently resolved by not sharing the service, but one could instead use SDI to build a thin interposition layer that is transparent to the shared application and thereby resolve possible namespace conflicts, for example, by prefixing names with the client's VSID on-the-fly.

Finally, an interesting question is whether it is possible to integrate PMaps and VS for offline system-capacity planning instead of online resource-allocation adaptation. Despite the fact that the prospects for online resource-allocation adaptation using VSs and PMaps are not promising, it may still be possible to use the output of PMaps to optimize system allocations on a coarser time-grain. One could use PMaps to build a simulation model of the multi-tiered system. Then, system administrators would be free to install VS-based and system-based performance controls in the simulated system, add extra capacity, and accurately assess the impact of the proposed changes. In this way, system administrators would be able to make more informed system-management and upgrade decisions.

APPENDICES

APPENDIX A

Survey of Service Implementations

A.1 Client-to-Server Communication

Request-per-message: The only request-per-message protocols that are directly or indirectly used by multi-tier applications are DNS, NFS, DCOM, and some custom applications that rely on datagram-based RPC.

DNS messages utilize a client-generated 16-bit message identifier to correlate request and reply packages. Thus, it is possible to uniquely identify requests and replies without modifying the applications, simply by matching the first two bytes of DNS request or reply packet [135].

Similar to the DNS protocol, RPC messages also carry client-generated transaction ID to correlate request and reply packets (the first 4 bytes of a UDP datagram) [135]. Due to problems with RPC's recovery from message loss and its poor performance on slow networks, most applications use the equivalent TCP transport method (request-per-connection). Only older NFS versions tend to rely on RPC over UDP. Both block requests and replies are sent as single UDP datagrams.

Microsoft's DCOM protocol is a separate layer of indirection layered atop the OSF RPC protocol [61]. It explicitly uses UDP-based RPC.

Request-per-connection: RPC is not only an example for a request-per-message protocol but also for the request-per-connection model [135]. When RPC is used with TCP as its transport layer, then each request is submitted to the server using a separate TCP connection. While this procedure incurs greater delays due to connection-establishment times, the TCP version of RPC is more robust in the presence of packet loss and slow networks. Moreover, TCP is the only choice for non-idempotent RPCs because UDP does not guarantee datagram delivery.

The most popular request-per-connection protocol is HTTP 1.0 without the HTTP keepalive option [20]. Here, too, request and reply are sent and received via the same connection. The connection is closed by the server as soon as it has sent its reply.

Connection recycling: HTTP implements a keep-alive feature [20] to avoid the overhead of connection establishment for every HTTP GET request. This is useful since some some web pages require tens of requests to be sent to the server. This feature can be used to interleave multiple requests and replies over the same connection. One important drawback of this protocol is that the server is only required to handle one outstanding request at a time. Similarly, to ensure proper processing of the replies, the client will only accept one reply at a time. This leads to a ping-pong-like communication pattern between client and server.

This back-and-forth communication pattern can also be found in low-end database client-to-server communication. If the database does not properly frame requests (e.g., Postgres SQL [34] provides no request framing), the client needs to wait for its query's result before submitting the next query. One benefit of this behavior is that the number of requests sent over a connection

is proportional to the number of times the client switches between sending and receiving on that connection. This observation is useful in policing and monitoring multi-tiered applications.

This ping-pong behavior also results naturally, when the client software is programmed in a sequential manner, i.e., without the use of asynchronous I/O. Due to the programmer's experience using multi-threading and asynchronous I/O on various platforms, custom-designed code is likely to be implemented in a sequential manner, thus exhibiting the ping-pong communication pattern that is typical of connection recycling even if the protocol permits pipelining.

Request pipelining: To avoid the added latency of having to hold back additional pending requests until the server replies, many client/server protocols — most importantly IIOP [102] and RMI [140] — allow request pipelining. Pipelining is made possible by framing each request and reply package. There are in fact two sub-categories of pipelined protocols: those that allow out-of-order processing (i.e., allowing a request B that was submitted after request A to finish before A) and those that don't (e.g., HTTP 1.1).

In HTTP 1.1 [53] the client sends a number of independent request messages (GET or POST) towards the server without waiting for any answers to arrive in between. To the outside observer of this client-to-server communication scheme it seems as if the client were submitting one very large request. The server, responds to the client requests by sending its reply messages (properly framed) over the back-channel of the same connection. Since HTTP 1.1 does not include a request ID for each request message, the server must reply to the client by sending all replies in the same order as the requests were sent.

Sun's Java RMI protocol also allows clients to send multiple consecutive requests to a server. Similar to HTTP 1.1, it requires replies to be sent in the same order as requests (a.k.a., remote method invocations) because calls and arguments are transmitted back-to-back without any additional wrapping in Sun's RMI protocol. RMI uses TCP because it must support all method operations, including non-idempotent ones, without forcing programmers to change their programming-style due to the network nature of RMI.

IIOP has become the *de facto* standard for client/server communication in modern multi-tiered applications because it provides a powerful system-independent communication framework which allows for object-orientation. In fact, even Sun's own Java distribution includes Corba bindings, which can be used instead of RMI. IIOP is a very sophisticated protocol, allowing even for disconnected operation. However, all commercial implementations use the bi-directional connection-based Object Request Broker (ORB) architecture: a full-duplex TCP connection is established between client and server. Clients initiate a connection with the server's ORB. Once the connection is established the client my send various request types to the remote ORB. Each request is identified by a unique request ID to allow the client to demultiplex replies that it will receive over its reply channel. CORBA allows out-of-order processing and request cancellation. However, the default behavior of most ORB implementations is FIFO request processing on a per-client basis. The ORB does not guarantee any specific execution order. Consistency must be enforced by the applications.

FTP: The traditional active FTP communication model [111] does not fit any of the communication models introduced in this section because it creates a separate connection from the server to the client for each reply, i.e., file transfer. The reason for the peculiar FTP protocol scheme is that replies are very long-lived, implying that the server would be non-responsive to control commands submitted by the client during an already-initiated upload. From a modeling perspective it is unfortunate that FTP is implemented using connection-pairs. For modeling purposes one would aggregate these two TCP/IP sessions into one logical, application-level session. This application-level session would then fit the pipelined model with connection recycling. Another

positive observation is that FTP is almost never used within multi-tier service implementations.

A.2 Service Processing Architectures

Apache 1.x: Version 1.3 of the popular Apache web server [33] is the most widely-used HTTP server in the entire Internet. Its processing model is designed to work well for servers of different sizes and is compatible with a variety of OSs. Apache may follow either the forked worker model (0 spare processes) or the process pool model (MaxClients equal to spare processes). Intermediary modes are possible. The best-performing configuration is the process pool model because it avoids process-creation overheads. Apache features one listener thread that waits for incoming connection requests and dispatches them—via a shared memory area called *scoreboard*—to an idle worker thread, or if none is available and the server still has less workers than MaxClients, it will dynamically fork a new worker process. CGI programs are executed inside a separate process that is forked on-demand.

Apache 2.x: Apache 2.x [35] is an updated version of Apache that, instead of relying on processes for its concurrency, utilizes threads (pthreads). Thread creation requires less overhead than process creation and most importantly, switching between threads of a shared memory space does not require expensive cache flush and translation look-aside buffer flushes in the CPU, thus improving overall application throughput. Except for the change from processes to threads the processing model remains the same as its previous version.

IIS: Microsoft's IIS [42] is a popular HTTP server alternative for Windows-based servers. Overall, the designs of IIS and Apache 2.x workload management are quite similar. The key difference between the two is that IIS uses an I/O completion port (a Windows request queuing abstraction) to communicate requests from the listener to the workers. Besides this technical detail, IIS implements a thread pool processing model like Apache.

FastCGI: FastCGI [103] is an extension for HTTP servers to improve throughput for dynamic content. Traditionally, CGI scripts were used to generate dynamic Web pages. However, this method causes process creation overhead for every CGI request. Instead of creating a new process for each request, the FastCGI model externalizes the binary inside its own separate (third tier) server application, to which the HTTP worker thread connects either by using a UNIX domain socket (UNIX), I/O Completion port (Windows), or a TCP/IP connection in general systems when the FastCGI server program executes on a different server than the Web server. FastCGI provides a simple plugin for a web server, which is invoked by the worker thread whenever the URI of a FastCGI script is encountered. The FastCGI script itself is very similar to a standard CGI script to ease the transition from CGI to FastCGI.

The FastCGI server has a configurable number of threads waiting for incoming connections from the Web server, thus implementing a thread pool model. Moreover, each remote FastCGI request from the Web server is submitted via its own separate connection (request-per-connection). FastCGI is far more efficient than traditional CGI script invocation.

Servlets: Servlets [68, 134] are the Java equivalent of FastCGI. The Jakarta/Tomcat Web server plugins are used in most HTTP servers that provide servlets.

There are three reasons for using servlets. First, invoking CGI scripts causes high overheads. Second, those overheads are even greater when the CGI script requires the instantiation and initialization of a JVM. To allow customizing Web sites with dynamic Java-based programs, the servlet mechanism provides a persistent JVM inside the HTTP server (i.e., within the Jakarta module). Third, web pages can use the Java Server Pages (JSP) language to refer to include dynamic calls to servlet routines, to generate part of a web page, thus providing an easy-to-use way to combine

static and dynamic content in a web page. Requests for servlets, dynamic objects are communicated to the persistent JVM, which translates the request to the appropriate method invocation using either RMI or IIOP. The object's output is then relayed to the Web client in the same manner in which a FastCGI script's output is relayed to the web clients. There is typically a one-to-one correspondence between the Web-servers worker thread that is waiting for the completion of the servlet call and the JavaThread within the JVM that executes the servlet's code.

The servlet mechanism is the basis of increasingly popular application server solutions, such as BEA Weblogic and IBM WebSphere.

ProFTP: The popular ProFTP [39] server for Unix-based OSs operates a forked-worker model. A main thread accepts incoming connections. As long as the FTP server's maximal connection limit is not reached, it forks a worker process to service the individual FTP session. Once the remote client disconnects from the FTP server, the worker process exits. The overhead of forking a worker process is negligible for an FTP server since its connections are relatively long-lived.

WuFTP: Wu-FTP [40], another popular FTP server, implements the same server behavior as ProFTP, i.e., a forked worker. However, this server does not implement a maximal remote connection limit, which makes it slightly more vulnerable to overload.

OpenSSH: OpenSSH [38] is an open source implementation of the secure remote login environment called Secure Shell (SSH). The operation of this login command resembles that of all other remote access programs, e.g., FTP and Rlogin. Since sessions are long-lived, the per-connection setup overhead tends to be negligible. So, until the server has reached its connection limit each incoming connection in SSH generates a new connection handling process using fork. The forked worker process handles authentication of the client (e.g., public key challenge or password) and forks another worker process, which will take over the processing of the connection within a separate user privilege process context. The authentication process exits. This processing model was referred to as the staged worker model, which is implemented using the the basic forked worker mechanism. The forked worker finally executes a shell program (a forked helper thread) whose output it relays to the remote client.

Linux mount.d: Linux's implementation of the mount.d service–inside the kernel follows the single worker scheme (using only one waiting thread) [36], which is also referred to as the iterative server model.

PostgreSQL: The Postgres DBMS [34] implements the fork-on-demand model. A dispatcher forks a new worker process for each incoming connection until a maximal number of processes is reached.

Sybase: Sybase introduces an additional layer of indirection in its service implementation when compared to other database implementations [143]. Sybase's Adaptive Server Enterprise DB has a dispatcher, which waits for incoming connections. The dispatcher does not read from the incoming connection. Instead it hands the incoming connection over to one of a number of listener threads that reside in a thread pool. Listeners are dispatched in round-robin order. For each new incoming connection the listeners create a new light-weight thread that handles all aspects of the communication with the client. Each connection-handling thread may create additional threads to handle certain aspects of client queries in parallel. The light-weight thread library is a proprietary, Sybase-specific threading library. The reason why Sybase does not use standard thread libraries is most likely due to the fact that the legacy of the current database version predates the availability of thread packages for different computing platforms.

DB2: DB2 [71] supports two different access modes: one for local clients and another for remote clients.

Local clients connect to the DB2 database via a special virtual memory-based communication

library. This communication library essentially provides a shared memory-based full-duplex message queue. The client appends its messages to the tail of its outgoing queue and the server reads from the head. Replies are enqueued by the server in a second FIFO queue. The communication mode is transparent from the perspective of the API used by DB2-dependent applications. Each new shared memory queue is handled like an accepted TCP connection: it is recognized by a dispatcher (for local requests), which activates an agent for every incoming in-memory request queue. If there are no more free agents, the dispatcher will wait for one to become available. Although the DB2 documentation does not explicitly describe the mechanism used for communication between the dispatcher and the agents, which execute queries and communicate directly with the clients, it is likely that this communication is based on the same communication abstraction that is used in the communication with local clients, i.e., a shared memory-based FIFO queue.

Remote clients connect to a dispatcher via a well-known TCP port. The dispatcher also takes the incoming TCP connection and passes the connection to an agent process, which handles the requests sent by the remote client.

DB2 uses actual processes to implement its worker processes (a.k.a. agents). The reason for this design choice is that it increases the databases robustness even if the database itself encounters an unexpected condition — only one worker process will die instead of taking down the entire DBMS. Since query processing is naturally a high overhead operation, the additional cost of disjoint address spaces between agent, dispatchers, and clients does not have much impact on DB2's overall performance.

Like all other databases, DB2 has numerous background processes to reorganize databases, reclaim buffer space, manage lock contention, and the like. These processes are not synchronized with the processing of the actual client-induced workload.

MS-SQL: Microsoft's SQL server implements a somewhat peculiar processing model [19, 45]. Incoming connections are forwarded via an IOCompletionPort to a free worker from a thread pool. If there are no free workers left, the incoming connection is forwarded to a worker thread that is already serving another connection. This thread will be multiplexed between two concurrent client connections. The recommended configuration is to set the number of simultaneous client sessions equal to the number of workers to avoid thread multiplexing. Nevertheless, the system degrades more gracefully when connection pooling is enabled for worker threads (incoming connections will not be denied). This model was chosen in previous versions of the SQL server because each process consumes a fixed amount of memory and there is only a fixed number of threads that each Windows kernel instance can manage. IIS's thread multiplexing is a complex implementation (service becomes a large state machine) only to hide some of older Windows version's limitations.

The more recent MS-SQL 2000 version supports fibers, a Windows version of LWPs (lightweight processes), which can be allocated freely–one per incoming connection because each fiber only consumes minimal resources. Fibers also exhibit lower task switching overheads and enabling fiber support for SQL 2000 is strongly recommended. In this case, IIS's actual processing model would be that of a thread or fiber pool (in Microsoft terminology). MS-SQL server's design is gradually moving towards the processing models employed by other DBMS systems with a longer history. In its current stage it most resembles Sybase, which utilizes application-level threads. Both Oracle and DB2 heavily utilize OS processes instead of application level threads.

Oracle: Client connections to the Oracle [29, 104] database server are recognized by a listener process, which accepts the connection and either passes the socket directly to a dispatcher process or redirects the client to directly connect to a dispatcher at a specified port. The reason for this redirection scheme is that it simplifies building a clustered database server. Each dispatcher handles a number of client connections in parallel. For each incoming connection the dispatcher creates a

so-called "Virtual Circuit," which it uses to communicate the requests generated by its clients to a thread pool of worker threads. Whenever a worker thread reads from a particular Virtual Circuit, it will execute an SQL statement and send its results through the same Virtual Circuit. The dispatcher reformats the result for transmission over the network and pushes it out to the client. Oracle uses LWP and processes as supported by the underlying kernel. Unlike MS-SQL server, Oracle avoids a state-machine like approach by introducing Virtual Circuits, which are easier to monitor and debug.

Weblogic: Weblogic [17, 18] is an application server. An application server is essentially a collection of library routines that simplify building custom Web sites that integrate a variety of data sources, such as the file system and databases. Application servers typically provide scheduling, request queuing, authentication, and persistent object support. Typical application servers, such as Weblogic, either integrate an HTTP server to allow network clients to connect or they hook into an existing HTTP server.

Weblogic includes its own Java-based HTTP server, thru which clients connect to remote Java object. Clients may also access the Java objects directly using Java's RMI or IIOP. The front end (proxy) maintains a Java thread for every incoming client connection. Client requests for specific Java back-end objects are submitted via a special request queue (a simple Java object similar to Oracle's Virtual Circuits) and communicated to the back-end Java object via one pre-established network connection. The benefit of this implementation detail is that it eliminates connection-establishment delays and that it reduces the number of connections required to allow network clients to access back-end Java objects; for c clients and n objects the number of simultaneous connections required for a traditional connection-per-request model would be in the order of nc as opposed to n with Weblogic's connection-per-object model.

The communication model between front-end and back-end servers, although TCP-based, is of the request-per-message kind. The front-end server employs a typical request pool with up to a maximal number of threads serving client connections. The back-end implementations are theoretically free to choose arbitrary threading models. However, if the back-ends are also implemented in the Weblogic processing model, they are implemented with a thread per object. Each back-end object has a listener thread that waits for incoming connections on a specific port and services the incoming requests.

WebSphere: The WebSphere [70, 72] application server is equivalent to the Weblogic application server. It also attempts to create a homogeneous programming environment for Web-enabled services that require access to persistent data and transactional semantics. The designs of WebSphere and Weblogic are very similar. The front end of WebSphere is a derivative version of the Apache HTTP server using the Jakarta servlet engine to drive WebSphere. As in Weblogic, WebSphere accesses remote objects to satisfy certain servlet requests.

WebSphere does not invent any novel communication mechanisms. It relies on CORBA IIOP and RMI. Incoming connections that are accepted by the HTTP server are handled by the Jakarta servlet engine if they require access to a Java object. If the Java object is external to the Jakarta server then the front end places the request into an application-level request queue. The request is then picked up by a thread from a pool of worker threads, which carry out the remote invocation using either IIOP or RMI. The worker thread blocks waiting for its peer to reply. The object of this intermediate step is to limit parallelism in the server farm to prevent overload.

The back-end servers of WebSphere are standard JVMs that host objects and provide no concurrency control; they simply operate on a worker thread per-connection basis.

SAP R/3: SAP R/3 [28] is essentially a mediator between client-side applications, e.g., client masks (terminal or Web-based) and back-end database systems. Each incoming connection is

accepted by a listener/dispatcher process, which hands the connection to a worker thread residing in a thread pool. If no worker is available, the connection is enqueued in a connection queue, from which the next available worker thread will dequeue it. Internally, the worker summarizes and consolidates the client's transaction requests and forks a special helper thread (a LWP called task handler) to execute back-end database transactions whenever database transactions are ready to be executed. Thus, the worker remains responsive to the client. When the client disconnects, the worker waits for all outstanding database transactions to complete and returns itself to the thread pool.

Orbix: While CORBA's IIOP protocol [102] has become the *de facto* standard for communication between objects in a multi-tiered server setup, standalone CORBA architectures have not achieved the same level of success. Nevertheless, native CORBA applications exist. Of those native CORBA applications, most use IONA's ORBIX [75, 76] implementation of CORBA.

The Object Request Broker (ORB) is the key component of a CORBA implementation. The ORB is responsible for unmarshaling the requests received from a client (e.g., a servlet engine) and activating the appropriate method on the target object. The communication between client and server is governed by the Inter-ORB protocol (usually IIOP). This protocol allows for pipelined requests or simple connection recycling. Since most client applications invoke remote objects in the same way they would invoke local objects (location transparent), the access to remote objects is typically sequential and the connection is only recycled (without pipelining).

IONA's ORBIX ORB implements the object server using the dispatcher model. First, every object server contains one listener thread, which listens for incoming requests to the objects that reside in its address space. Each request received by the the listener is translated into a request data structure and enqueued in a thread-safe queue object that operates as a buffer between incoming requests and a worker thread pool. Threads from the thread pool dequeue requests from the thread pool and invoke the requested method on the target object. If the target object itself is implemented as a single-threaded object, the worker from the thread pool actively carries out the request by invoking the requested method. If the target object is implemented as a multi-threaded object, then the thread from the thread pool enqueues the request in an object-specific request queue, from which it is retrieved by the object implementation's main thread. This latter model requires more complex object implementations, which are not only functional but also control their own request scheduling. The implementation is best described as a staged worker model, in which the client's request is partially processed in several stages until it is finally finished by the target object. The stages are thread pools that are connected by thread-safe request queues that are provided by the ORBIX middleware.

OmniORB: OmniORB [37, 89] is a high-performance CORBA implementation. Surprisingly, the thread pools of the ORBIX CORBA implementation are absent from OmniORB. It implements a simple thread-per-connection model instead. More specifically, incoming connections are accepted by a listener which creates a worker thread for each incoming connection. Thus, OmniORB implements a simple forked worker model. Despite OmniORB's simple processing model it has been shown to outperform competing ORB implementations [46]. This observation suggests that the overhead of setting up a worker for each incoming connection is only a small fraction of CORBA's overall per-request overhead.

APPENDIX B

Detailed Trace Characteristics of Selected Service Implementations

In discussing the identification of different processing models, it is assumed that every needed system event can be generated and timestamped with a consistent global timestamp. As will be explained later in Section 5.10, the mechanics of achieving these prerequisites are not trivial and require significant implementation effort (see Section 5.10).

B.0.1 Single Worker

The single worker model is the simplest workload handling behavior that can be implemented for a service. The service may consist of a single thread that simply accepts one incoming connection at a time and finishes it completely before moving on to the next request or connection. Alternatively, the single worker service may consist of a (fixed or dynamically-sized) pool of threads which await incoming connections using `accept`. The OS assigns each incoming connection to a thread, which handles all of the work requested by the client via the accepted connection. This model is recognized by observing that a thread that accepted a connection will also read it and write to it without forking or dispatching any intermediary helper process. Due to its poor performance and lacking scalability [136] this model is not used in commercial products and only in early stage open source projects.

B.0.2 Forked Worker Thread

The forked worker thread model consists of a *delegating parent* and a *connection-handling child*. We generally assume that the parent's resource consumption between its resumption of processing (return from `select`) to its return to `select` or `accept` should be charged to the activity that caused the forking behavior. Any processing that occurs between the parent's entry to `select` and its successful return from `select` should not be charged to any activity but be counted as background processing overheads.

As Figure B.1 shows, there are essentially two distinct forking behaviors for network processes. The first model applies to both TCP- and UDP-based servers, whereas the second model is only implemented in TCP-based servers, which constitute the vast majority of all network services used in multi-tier systems. The first model calls `select` (or `poll`) to wait for a new connection. In the second model the parent calls `accept` directly relying on the OS to wake it up. The select-approach is often chosen when the server thread has other maintenance responsibilities or when the server waits for data on a UDP socket. The observation of the underlined events reliably identifies the forked worker model.

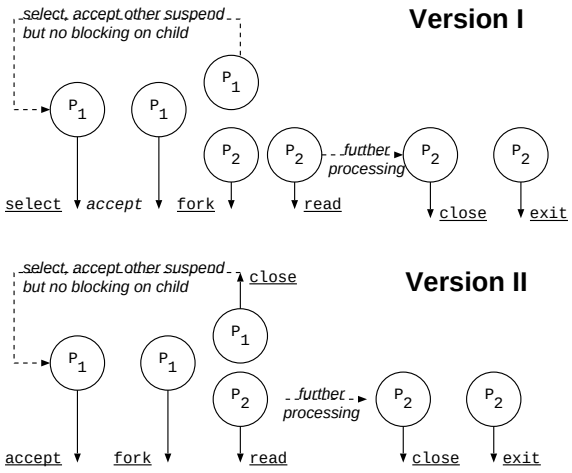


Figure B.1: The forked worker processing model

To identify a forked worker, it is necessary that all features of the model are recognized, i.e., the waiting parent, `accept`, the return of the parent to a waiting system call, and the child's termination which concludes the incoming session. If any of the trace features that are underlined in Figure B.1 are missing, the trace is simply identified as unrecognized (each failure is logged). The reason for this rigid interpretation is that the models will begin to become indistinguishable if one allows for guessing errors. Compared to other recognition problems known from Artificial Intelligence (AI) research, such as image recognition, the behavior recognition problem of PMaps is marked by a relatively small number of distinguishing features. Model detection techniques from AI that use similarity typically rely on larger sets of characteristic features.

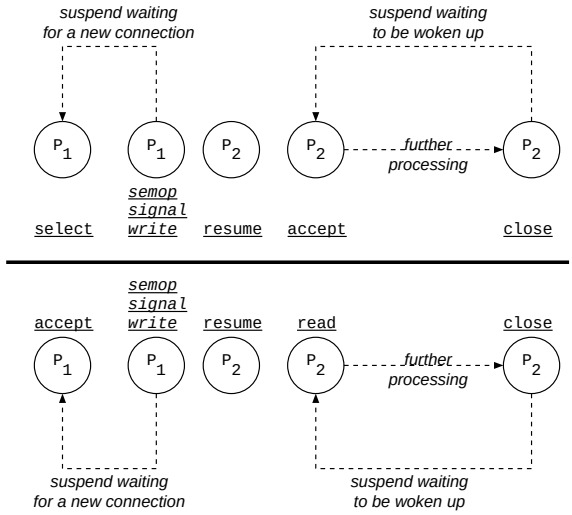


Figure B.2: The dispatcher processing model

B.0.3 Dispatcher

The dispatcher model is another asymmetric processing model in which the dispatcher does little work. It simply checks for a pending request and dispatches a worker thread or process to complete the work. In contrast to the forked worker model, one does not observe the creation of a new process or thread in response to an incoming request and processes do not exit immediately, i.e., within several μs, after finishing a request (i.e., `close` or `shutdown` of the corresponding socket). In this model the process set management is generally separate from the handling of connections and requests.

One may argue that many real server applications are hybrids of dispatcher-style and forked-worker behaviors. The worker threads are first forked as in the forked worker model but stay alive after finishing the client request. Fortunately, the initial startup phase (forked worker) is in general statistically irrelevant. For this reason the startup behavior model is omitted from the statistical PMap tool.

The typical indicator for the handoff of work is that the dispatched process or thread uses the accepted socket. The reason why one cannot totally depend on synchronization between worker and dispatcher is that there are many different ways in which dispatcher and workers may synchronize. The most common calls such as `semop` and `write` to a FIFO are easily observed. However, tracking synchronization using file locking mechanisms is more difficult to achieve. This is why the use of synchronization between processes should never be used as the key characteristic of a model.

Experimentation showed that it is possible to reliably identify the dispatcher model by observing that the dispatched process resumes its processing and `read`s from a socket or `accept`s a connection on a socket that was previously touched by the dispatcher thread. The trace features that are required to positively identify the dispatcher model are:

1. The worker does not exit upon closing the connection with its client but it suspends processing and

2. there must not be a `fork` event without a corresponding `exit` between the dispatcher's recognition of the pending connection and the read or accept by the worker (see Figure B.2).

One may be able to enhance the ability to recognize the dispatcher model by adding positive identification of the actual handoff event. It appears, however, that it is possible to correctly account for the dispatcher model without identifying the actual wake-up-handoff sequence because the dispatch operation and the worker's first use of the connection are typically only a μs apart.

With respect to statistics the dispatcher's activity should only be counted towards the activity that causes an incoming connection as long as it can be associated with that activity, i.e., until it reenters `select` or `accept` waiting for the next connection.

B.0.4 Helper threads

The helper thread model is commonly used to increase the degree of parallelism in services. Especially in I/O intensive services, it can be very helpful to handle potentially-blocking operations spawned by one activity in their own independent thread, thus maximally utilizing the server's resources.

Forked and dispatched threads are essentially specializations of the helper thread model, because the parent uses a helper for each incoming connection to increase the number of connections that are handled in parallel. This thesis distinguishes helper threads from forked and dispatched

worker by stating that the helper's parent or dispatcher keeps working on behalf of the same activity as the child or the dispatched thread. In fact, it typically `wait`s for the child or resynchronizes with it. Moreover, the helper never closes the connection that is associated with the current activity. This is also the simplest way to recognize a helper thread: it is a dispatched thread or forked worker that *does not* close the socket over which the activity's request was received later than its parent (dispatcher). Oftentimes, the helper thread will not even touch the socket or connection that is associated with the incoming activity. The activity of a helper thread is to be counted fully towards the activity.

If the helper is not forked but dispatched from a thread pool, it is important to observe the activation of the worker, i.e., an IPC mechanism, lock, or message, because there will be no `read` or `accept` on the socket that identifies the ongoing activity. As soon as the worker suspends on the same synchronization primitive from which it was released, it is to be disassociated from the ongoing activity. The dispatched helper is to be disassociated from the activity, which caused its release no later than the end of that activity plus a small time interval to allow for some possible cleanup in the worker. This helper model is frequently used in databases. Here the connection-managing process suspends to wait for one of several I/O threads to take and complete its record retrieval command. Once the I/O thread resynchronizes with the parent or dispatcher, the parent/dispatcher resumes and finishes processing the activity. If the helper resynchronizes without waking up the dispatcher that caused its most recent release, then the helper must be flagged as *multiplexed*. In this case the helper is associated with a number of activities simultaneously and its resource consumption must be charged in equal parts to all activities that are currently associated with the thread. In general, this mode of operation is the most difficult to capture in an automated online-monitoring tool since it requires careful implementation and good data structures to capture rapidly changing "works-for" relationships. Moreover, the PMap beyond such a multiplexed thread can no longer be broken down by service class, unless the application's multiplexing mechanism is made transparent to the PMap tool by instrumenting the appropriate application or middle-ware libraries.

To account for dispatched helper threads one must assume that competing dispatchers are served in FIFO order and that dispatched helpers eventually resynchronize with their parents to mark the end of their activity.

B.0.5 Staged worker

Work is passed from stage n to stage $n+1$, which can be observed by either intercepting communication or synchronization between the stages. Typically, the process at stage n waits for work from stage $n-1$ by synchronizing on a semaphore or calling a receive primitive (e.g., FIFO read). It is unblocked by a process from stage $n-1$, which is working on a request different from the one that stage n was working on prior to its unblocking. Upon unblocking the stage n worker, the worker in stage $n-1$ closes its reference to the incoming client connection. Eventually, the last stage will close the session associated with the request under consideration.

The staged worker model is in fact only the multi-tier extension of the dispatcher model. The recognition and measurement specifics for child processes, sessions, and resource consumption do not change from the previously-described two tier processing model (dispatcher). Notice that this model does not require distinct hosts.

B.0.6 Nested worker

The nested worker model is also a multi-tiered processing model. However, the difference between the nested worker and staged worker is that the dispatcher or parent thread does not sever its association with an ongoing activity once it dispatches a nested worker. For example, a worker thread may dispatch a nested worker without accepting or waiting for another request, or closing the socket over which it received the request. The nested worker model is the *n-tier* extension of the helper thread model, in which the parent thread or dispatcher remains in charge of the incoming connection but waits for the next stage handle a sub-request.

B.0.7 Peer relationships

Thus far, the relationships between processes were confined to parent-to-child dependencies and dependencies that are created by host-based synchronization (messaging, semaphores, etc.). In multi-tier systems, processes that reside on distinct hosts create peer relationships by establishing network connections between them. The relationships manifest themselves in the existence of a communication channel between the communicating processes of some network protocol. Network-connection-based dependencies between two processes almost always resemble nested worker-type processing models, in which a front end process connects to a back end process.

The back-end's process pool is decoupled from the front end, i.e., front-ends have no control over the processing model, resource allocation, and implementation of back ends. This is the natural point for composition of models. In general, a model identification procedure should expect both client and server side of a communication channel to implement their own behavior models. The peer relationship ties the model of the front-end service to that of a back-end service, thus reducing complexity. Each service's behavior can be identified by events generated on the server. By identifying peer services it is possible to say that two independent services collaborate and their interaction can be characterize by describing network communication between them.

The communication model is of interest for peer services because network performance can drastically alter the performance of client-server processing. For example, the amount of data exchanged between peers, the communication delay, and possibly back-and-forth communication patterns (ping-pong) are very important if one attempts to understand the performance of a multi-tiered, multi-host service.

B.0.8 Combinations

The above models are composed in multi-tier systems. For example, the Apache HTTP server implements a dispatcher model. However, its processing of CGI scripts follows the helper thread model. The CGI script may in turn rely on a back end database, to which it connects via a TCP connection (using ODBC), thus implementing a nested worker model. The back end database's receiving process typically acts as a dispatcher, while the database's worker threads may utilize numerous dispatched helper threads (e.g., in parallel queries).

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. A Calculus for Access Control in Distributed Systems. *ACM Transactions on Programming Languages and Systems 15*, 4 (September 1993), 706–734.

[2] ABDELZAHER, T. QoS Adaptation in Real-time Systems. Tech. rep., Ph.D. Thesis, University of Michigan, 1999.

[3] ABDELZAHER, T., AND SHIN, K. Qos provisioning with *q*contracts in web and multimedia servers. In *IEEE Real-Time Systems Symposium* (Phoenix, AZ, December 1999).

[4] ABDELZAHER, T. F., ATKINS, E. M., AND SHIN, K. G. QoS Negotiation in Real-Time Systems and its Application to Automated Flight Control. In *Real-Time Technology and Applications Symposium* (Montreal, CA, June 1997), IEEE.

[5] ALVISI, L., BHATIA, K., AND MARZULLO, K. Causality Tracking in Causal Message-Logging Protocols. *Springer Distributed Computing 15*, 1 (2002).

[6] AMAN, J., EILERT, C. K., EMMES, D., YOCOM, P., AND DILLENBERGER, D. Adaptive Algorithms for Managing Distributed Data Processing Workload. *IBM Systems Journal 36*, 2 (1997), 242–283.

[7] AMD M. F. KAASHOEK, A. S. T., VAN RENESSE, R., AND BAL, H. The amoeba distributed operating system: A status report. *Computer Communications 14* (July 1991), 324–335.

[8] ARMSTRONG, J., VIRDING, R., WILKSTRÖM, C., AND WILLIAMS, M. *Concurrent Programming in Erlang*. Prentice Hall International, Herfordshire, HP2 7EZ, U.K., 1996.

[9] ARNOLD, J. Q. Shared Libraries on UNIX System V. In *Proceedings of USENIX Summer Conference* (1986), pp. 395–404.

[10] ARON, M., DRUSCHEL, P., AND ZWAENEPOEL, W. Cluster Reserves: A Mechanism for Resource Management in Cluster-based Network Servers. In *ACM SIGMETRICS* (Santa Clara, California, June 2000).

[11] BABAOGLU, O., AND MARZULLO, K. *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms*, 2 ed. Frontier Series. ACM Press, New York, NY, 1995, ch. 8, pp. 199–216.

[12] BABAOGLU, O., AND MARZULLO, K. *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms*, 2 ed. Frontier Series. ACM Press, New York, NY, 1995, ch. 4, pp. 55–96.

[13] BADGER, L., STERNE, D. F., SHERMAN, D. L., WALKER, K. M., AND HAGHIGHAT, S. A. A Domain and Type Enforcement UNIX Prototype. In *5th USENIX Security Symposium* (Salt Lake City, Utah, June 1995).

[14] BANGA, G., AND DRUSCHEL, P. Lazy Receiver Processing LRP: A Network Subsystem Architecture for Server Systems. In *Second Symposium on Operating Systems Design and Implemenation* (October 1996).

[15] BANGA, G., DRUSCHEL, P., AND MOGUL, J. Resource Containers: A New Facility for Resource Management in Server Systems. In *Third Symposium on Operating Systems Design and Implemenation* (February 1999), pp. 45–58.

[16] BANJERJI, A., TRACEY, J. M., AND COHN, D. L. Protected Shared Libraries - A New Approach to Modularity and Sharing. In *15th ACM Symposium on Operating System Principles* (December 1995).

[17] BEA SYSTEMS. Making Component-based Systems Scale with BEA WebLogic Enterprise. http://www.bea.com/products/weblogic/enterprise/paper_components.shtml, 2002.

[18] BEA SYSTEMS. WebLogic Server Performance and Tuning: Tuning WebLogic Server. eDocs BEA, http://edocs.bea.com/wls/docs70/perform/WLSTuning.html#1136287, 2002.

[19] BERENSON, H., AND DELANEY, K. Microsoft SQL Server Query Processor Internals and Architecture. MSDN Library, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsql7/html/sqlquerproc.asp, January 2000.

[20] BERNERS-LEE, T., FIELDING, R., AND FRYSTYK, H. Hypertext transfer protocol – http/1.0. IETF — RFC 1945, May 1996.

[21] BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M., BECKER, D., EGGERS, S., AND CHAMBERS, C. Extensibility, Safety and Performance in the SPIN Operating System. In *15th ACM Symposium on Operating System Principles* (December 1995), pp. 267–284.

[22] BERSON, A. *Client/Server Architecture*. McGraw Hill, New York, N.Y., U.S.A., 1996.

[23] BIRRELL, A., NELSON, G., OWICKI, S., AND WOBBER, E. Network Objects. In *14th Symposium on Operating Systems Principles* (Ashville, NC, December 1993), ACM, pp. 217–230.

[24] BRENTON, C. *Mastering Network Security*. Sybex, Alameda, CA, U.S.A., 1998.

[25] BRIN, S., AND PAGE, L. The Anatomy of a Large-Scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems 30* (1998), 107–117.

[26] BRUNO, J., BRUSTOLONI, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. Retrofitting Quality of Service into a Time-Sharing Operating System. In *USENIX Annual Technical Conference* (June 1999).

[27] BRUNO, J., GABBER, E., OZDEN, B., AND SILBERSCHATZ, A. The Eclipse Operating System: Providing Quality of Service via Reservation Domains. In *USENIX Annual Technical Conference* (New Orleans, Louisiana, U.S.A, June 1998).

[28] BUCK-EMDEN, R., AND GALIMOW, J. *Sap R/3 System : A Client/Server Technology*. Addison-Wesley, New York, 1996.

[29] BURLESON, D. K., AND BURLESON, D. *Oracle9i UNIX Administration Handbook*. Osborne Media. McGraw-Hill, New York, N.Y., U.S.A., 2002.

[30] CAO, P., FELTEN, E. W., AND LI, K. Implementation and Performance of Application-Controlled File Caching. In *First Symposium on Operating System Design and Implementation* (1994), ACM.

[31] CARRIERO, N., AND GELERNTER, D. The S/Net's Linda Kernel. *ACM Transactions on computer Systems 4* (May 1986), 110–129.

[32] CISCO INC. Local Director (White Paper) http://cisco.com/warp/public/cc/cisco/mkt/scale/locald/tech/lobal_wp.htm. 2000.

[33] COLLABORATIVE OPEN SOURCE EFFORT. Source Code Apache HTTPD. http://www.apache.org/dist/httpd/old/apache_1.3.19.tar.gz, 2001.

[34] COLLABORATIVE OPEN SOURCE EFFORT. Source Code PostgreSQL. ftp://ftp.postgresql.org/pub/source/v7.1.3, November 2001.

[35] COLLABORATIVE OPEN SOURCE EFFORT. Source Code Apache HTTPD. http://www.apache.org/dist/httpd/httpd-2.0.43.tar.gz, 2002.

[36] COLLABORATIVE OPEN SOURCE EFFORT. Source Code Linux. ftp://ftp.kernel.org/pub/linux/kernel/v2.2, 2002.

[37] COLLABORATIVE OPEN SOURCE EFFORT. Source Code OmniORB. http://sourceforge.net/project/showfiles.php?group_id=51138, 2002.

[38] COLLABORATIVE OPEN SOURCE EFFORT. Source Code OpenSSH. ftp://ftp.openbsd.org/pub/OpenBSD/OpenSSH/openssh-3.5.tgz, 2002.

[39] COLLABORATIVE OPEN SOURCE EFFORT. Source Code ProFTPD. ftp://ftp.proftpd.org/distrib/source/proftpd-1.2.7.tar.gz, 2002.

[40] COLLABORATIVE OPEN SOURCE EFFORT. Source Code WuFTP. ftp://ftp.wu-ftpd.org/pub/wu-ftpd/wu-ftpd-current.tar.gz, July 2002.

[41] COMER, D. E., AND STEVENS, D. L. *Internetworking With TCP/IP Volume III: Client-Server Programming and Applications, Linux/POSIX Socket Version*. Prentice-Hall, New Jersey, 2001.

[42] CORPORATION, M., Ed. *Microsoft Internet Information Server Resource Kit*. Microsoft Press, Washington, U.S.A., 1998.

[43] CRISTIAN, F. Probabilistic Clock Synchronization. *Distributed Computing 3* (1989), 146–158.

[44] CROVELLA, M., AND BESTAVROS, A. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proceedings of SIGMETRICS'96: The ACM International Conference on Measurement and Modeling of Computer Systems*. (Philadelphia, Pennsylvania, May 1996).

[45] DELANEY, K. *Inside Microsoft SQL Server 2000.* Microsoft Press, Washington, U.S.A., 2000.

[46] DEPARTMENT OF SOFTWARE ENGINEERING. CORBA Comparison Project: Final Report. Tech. rep., Charles University, Malostranske namesti 25, Prague, Czech Republic, June 1998. http://www.kav.cas.cz/ buble/corba/comp/report.pdf.

[47] DRAVES, R. P., BERSHAD, B. N., RASHID, R. F., AND DEAN, R. W. Using Continuations to Implement Thread Management and Communication in Operating Systems. In *13th Symposium on Operating Systems Principles* (1991), ACM, pp. 122–136.

[48] EGGERS, S. J., AND KATZ, R. H. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture* (1989), ACM, pp. 2–15.

[49] ELLISON, C. The Nature of a Useable PKI. *Computer Networks 31*, 8 (1999), 823–830.

[50] EMMERICH, W. *Engineering Distributed Objects.* John Wiley & Sons, New Jersey, U.S.A., 2000.

[51] ENSIM CORP. *ServerXchange (White Paper).* Mountain View, California, 2000. http://www.ensim.com/products/wpaper_fr.html.

[52] F5 CORP. http://www.f5.com/bigip/. 2000.

[53] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext transfer protocol – http/1.1. IETF — RFC 2616, June 1999.

[54] FORD, B., BACK, G., BENSON, G., LEPREAU, J., LIN, A., AND SHIVERS, O. The Flux OSKit: A Substrate for OS and Language Research. In *16th ACM Symposium on Operating Systems Principles* (Saint Malo, France, October 1997).

[55] GANSNER, E., KOUTSIFIOS, E., NORTH, S., AND VO, K. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering 19* (1993), 214–230. Homepage http://www.research.att.com/sw/tools/graphviz/.

[56] GELERNTER, D. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems 7*, 1 (1985), 80–112.

[57] GEORGE COULOURIS, J. D., AND KINDBERG, T. *Distributed Systems: Concepts and Design*, 3rd ed. Addison-Wesley, New York, 2001.

[58] GHORMLEY, D., RODRIGUES, S., PETROU, D., AND ANDERSON, T. SLIC: An Extensibility System for Commodity Operating Systems. In *USENIX Annual Technical Conference* (June 1998).

[59] GOYAL, P., GUO, X., AND VIN, H. M. A Hierarchical CPU Scheduler for Multimedia Operating Systems. In *Second Symposium on Operating Systems Design and Implementations* (Seattle, Washington, U.S.A, October 1996), ACM, pp. 107–122.

[60] GRAY, C. G., AND CHERITON, D. R. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. *ACM Operating Systems Review 23*, 5 (1989), 202–210.

[61] GRIMES, R. *DCOM Programming.* Wrox Press, Birmingham, U.K., 1997.

[62] GU, W., EISENHAUER, G., KRAEMER, E., SCHWAN, K., STASKO, J., VETTER, J., AND MALLAVARUPU, N. Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation* (McClean, VA, 1995), pp. 422–429.

[63] HALL, C. L. *Technical Foundations of Client-Server Systems.* John Wiley & Sons, New Jersey, U.S.A., 1994.

[64] HAND, S. M. Self-Paging in the Nemesis Operating System. In *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation* (New Oreleans, Lousiana, February 1999), USENIX, pp. 73–86.

[65] HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. Recovery Management in QuickSilver. *ACM Transactions on Computer Systems 6*, 1 (1988), 82–108.

[66] HEWLETT PACKARD. WebQoS Technical White Paper. http://www.internetsolutions.enterprise.hp.com/ webqos/producllllts/overview/wp.html, 2000.

[67] HEWLETT-PACKARD COMPANY. *HP OpenView Data Extraction and Reporting (White Paper).* California, 1999. http://openview.hp.com/sso/getdoc?doc/products/operations/tech_whitepaper/ovops_twp2.pdf.

[68] HUNTER, J., AND CRAWFORD, W. *Java Servlet Programming.* O'Reilly & Associates, California, U.S.A., 2001.

[69] HYDRAWEB TECHNOLOGIES, INC. Load balancing (White Paper). www.hydraweb.com, 1999.

[70] IBM CORP. http://www-4.ibm.com/software/webservers/appserv/whitepapers.html. 2001.

[71] IBM CORP. *IBM DB2 Universal Database Administration Guide : Performance (Version 8).* IBM Corp., N.Y., U.S.A., 2002.

[72] IBM CORP. IBM WebSphere Application Server,Version 5 Monitoring and Troubleshooting. ftp://ftp.software.ibm.com/software/webserver/appserv/library/wasv5base_pd.pdf, November 2002.

[73] IBM REDBOOKS. *Tivoli Application Performance Management Version 2.0 and Beyond.* IBM Corp., Armonk, NY, 2002.

[74] INTEL CORP. Yahoo!: Lessons Learned. http://www.intel.com/eBusiness/casestudies/-yahoo/lessons.htm, 2002.

[75] IONA TECHNOLOGIES PLC. Orbix c++ administrator's guide. February 1999.

[76] IONA TECHNOLOGIES PLC. Orbix c++ programmer's guide. February 1999.

[77] JEFFAY, K., SMITH, F., MOORTHY, A., AND ANDERSON, J. Proportional Share Scheduling of Operating System Services for Real-Time Applications. In *Proceedings of the 19th IEEE Real-Time Systems Symposium* (Madrid, December 1998).

[78] JONES, M. B. Interposition Agents: Transparently Interposing User Code at the System Interface. In *14th Symposium on Operating Systems Principles* (December 1993), ACM, pp. 80–93.

[79] KENT, S., AND ATKINSON, R. Security Architecture for the Internet Protocol. IETF RFC 2401, November 1998.

[80] KOPETZ, H., AND OCHSENREITER, W. Clock Synchronization in Distributed Real-Time Systems. *IEEE Trans. Comput. 36*, 8 (1987), 933–940.

[81] KRISHNA, C. M., AND SHIN, K. G. *Real-Time Systems.* McGraw-Hill, New York, N.Y., U.S.A., 1996.

[82] LAMPSON, B. W., AND REDELL, D. D. Experience With Processes and Monitors in Mesa. *Communications of the ACM 23*, 2 (February 1980), 106–117.

[83] LANGE, F., KROEGER, R., AND GERGELEIT, M. JEWEL: Design and Implementation of a Distributed Measurement System. *IEEE Trans. Par. Dist. Sys. 3*, 6 (December 1992).

[84] LAURIE, B., AND LAURIE, P. *Apache: The Definitive Guide*, 2nd ed. O'Reilly & Associates, February 1999.

[85] LEE, C., LEHOCZKY, J., RAJKUMAR, R., AND SIEWIOREK, D. On quality of service optimization with discrete qos options. In *Real-time Technology and Applications Symposium* (June 1999), IEEE.

[86] LEE, C., LEHOCZKY, J., SIEWIOREK, D., RAJKUMAR, R., AND HANSEN, J. A scalable solution to the multi-resource qos problem. In *IEEE Real-Time Systems Symposium* (December 1999).

[87] LEISERSON, T., CORMEN, C., AND RIVEST, R. *Introduction to Algorithms.* McGraw-Hill, New York, N.Y., U.S.A., 1994.

[88] LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W., GUPTA, A., HENNESSY, A., HOROWITZ, J., AND LAM, M. The Standford Dash Multiprocessor. *IEEE Computer 25* (1992), 62–79.

[89] LO, S.-L. The omniorb2 version 2.5. Tech. rep., Olivetti & Oracle Reseach Laboratory, U.K., February 1998.

[90] LOSCOCCO, P., AND SMALLEY, S. Integrating Flexible Support for Security Policies into the Linux Operating System. In *USENIX Annual Technical Conference (FREENIX track)* (Boston, MA, U.S.A., June 2001).

[91] MALAN, G. R., AND JAHANIAN, F. An Extensible Probe Architecture for Network Protocol Performance Measurement. In *SIGCOMM* (1998), pp. 215–227.

[92] MARTELLO, S., AND TOTH, P. *Knapsack Problems.* John Wiley & Sons Ltd., Chichester, U.K., 1990.

[93] MCILROY, M. D., AND REEDS, J. A. Multilevel Security in the UNIX Tradition. *Software–Practice and Experience, Wiley & Sons 22*, 8 (August 1992), 673–694.

[94] MERCER, C., SAVAGE, S., AND TOKUDA, H. Processor Capacity Resevers: Operating System Support for Multimedia Applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems* (May 1994), IEEE.

[95] MICROSOFT CORP. WMI and CIM Concepts and Terminology, http://www.microsoft.com/hwdev/driver/WMI/WMI-CIM.asp.

[96] MILLER, B. P., CARGILLE, J., IRVIN, R. B., NEWHALL, T., CALLAGHAN, M. D., HOLLINGSWORTH, J. K., KARAVANIC, K. L., AND KUNCHITHAPADAM, K. The Paradyn Parallel Performance Measurement Tool. *IEEE Computer* (November 1995), 37 – 46.

[97] MILLS, D. L. Network Time Protocol (Version 3): Specification, Implementation and Analysis. RFC 1305, 1992.

[98] MINEAR, S. E. Providing Policy Control Over Object Operations in a Mach Based System. In *5th USENIX Security Symposium* (April 1995).

[99] MULLENDER, S., GUIDO VON ROSSUM, TANENBAUM, A., ROBBERT VON RENESSE, AND HANS VON STAVEREN. Amoeba: A Distributed Operating System for the 1990's. *IEEE Computer 14* (May 1990), 365–368.

[100] NEEDHAM, R. M. *Distributed Systems*, 2 ed. Frontier Series. ACM Press, New York, NY, 1995, ch. 12, pp. 315–327.

[101] OIKAWA, S., AND RAJKUMAR, R. Linux/RK: A Portable Resource Kernel in Linux. In *Work in Progress 19th IEEE Real-Time Systems Symposium* (Madrid, December 1998).

[102] OMG, Ed. *The Common Object Request Broker Architecture and Specification 2.2*. OMG, February 1998.

[103] OPEN MARKET INC. FastCGI: A High-Performance Web Server Interface. http://www.fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm, April 1996.

[104] ORACLE CORP., Ed. *Oracle9i Net Services Administrator's Guide Release 2 (9.2)*. Oracle Corp., California, U.S.A., 2002. Part Number A96580-01.

[105] OUSTERHOUT, J., CHERENSON, A., DOUGLIS, F., NELSON, M., AND WELCH, B. The Sprite Network Operating System. *IEEE Computer 21*, 2 (1988), 23–36.

[106] PAI, V. S., DRUSCHEL, P., AND ZWAENEPOEL, W. Flash: An Efficient and Portable Web Server. In *Proceedings of USENIX Annual Technical Conference* (1999).

[107] PARDYAK, P., AND BERSHAD, B. Dynamic Binding for an Extensible System. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation* (Seattle, WA, October 1996), pp. 201–212.

[108] PATARIN, S., AND MAKPANGOU, M. Pandora : A Flexible Network Monitoring Platform. In *Proceedings of the USENIX 2000 Annual Technical Conference* (San Diego, June 2000).

[109] PLAINFOSSÉ, D., AND SHAPIRO, M. A Survey of Distributed Garbage Collection Techniques. In *Proceedings International Workshop on Memory Management* (Kinross, Scottland, September 1995).

[110] POSTEL, J. Internet Protocol Darpa Internet Program Protocol Specification. IETF RFC 791, September 1981.

[111] POSTEL, J., AND REYNOLDS, J. File transfer protocol (ftp). IETF — RFC 959, October 1985.

[112] RAJKUMAR, R., LEE, C., LEHOCZKY, J., AND SIEWIOREK, D. A resource allocation model for qos management. In *IEEE Real-Time Systems Symposium* (December 1997), IEEE.

[113] RAJKUMAR, R., LEE, C., LEHOCZKY, J., AND SIEWIOREK, D. Practical solutions for qos-based resource allocation problems. In *IEEE Real-Time Systems Symposium* (December 1998), IEEE.

[114] REAL-NETWORKS, I. Real Networks, the Home of Real Audio, Real Video, Real Flash, http://www.real.com/. WWW information page, 1997.

[115] RENESSE, R. V., BIRMAN, K., AND MAFFEIS, S. Horus: A Flexible Group Communication System. *Communications of the ACM 39*, 4 (April 1996).

[116] REUMANN, J., AND JAMJOOM, H. Qguard: Protecting internet servers from overload. Tech. Rep. CSE-TR-427-00, The University of Michigan, 2000.

[117] REUMANN, J., JAMJOOM, H., AND SHIN, K. Adaptive Packet Filters. In *Globecom* (San Antonio, TX, U.S.A., November 2001), IEEE.

[118] REUMANN, J., MEHRA, A., SHIN, K., AND KANDLUR, D. Virtual Services: A New Abstraction for Server Consolidation. In *USENIX Annual Technical Conference* (June 2000).

[119] RICHTER, J., AND CLARK, J. *Programming Server Side Applications for Win 2000.* Microsoft Press, Redmond, WA, 2000.

[120] RODRIGUES, R., CASTRO, M., AND LISKOV, B. BASE: Using Abstraction to Improve Fault Tolerance. In *18th Symposium on Operating System Principles* (October 2001), pp. 395–404.

[121] RODRIGUES, R., CASTRO, M., AND LISKOV, B. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 18th ACM Symposium on Operating System Principles* (Banff, Canada, October 2001), pp. 15–28.

[122] ROZIER, M., ABROSSIMOV, V., ARMAND, F., BOULE, I., GIEN, M., GUILLEMONT, M., HERRMAN, F., KAISER, C., LANGLOIS, S., LEONARD, P., AND NEUHAUSER, W. Overview of the Chorus Distributed Operating System. In *Usenix Workshop: Micro-Kernels and Other Kernel Architectures* (April 1992), pp. 39–69.

[123] SAITO, Y., BERSHAD, B., AND LEVY, H. Manageability, Availability, and Performance in Porcupine: A Highly Scalable Internet Mail Service. In *Proceedings of 17th Symposium on Operating Systems Principles* (Kiawah Island, SC, 1999).

[124] SALTZER, J. H., REED, D. P., AND CLARK, D. D. End-to-End Arguements in System Design. *ACM Trans. Comput. Syst. 2*, 4 (November 1984), 277–288.

[125] SCHNEIDER, F. B. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys 22*, 4 (1990), 299–319.

[126] SCHWARZ, R., AND MATTERN, F. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing 7*, 3 (1994), 149–174.

[127] SHAKKOTTAI, S., AND SRIKANT, R. How Good are Deterministic Fluid Models of Internet Congestion Control? In *IEEE Infocom 2002* (New York, U.S.A., June 2002).

[128] SIRER, E. G., GRIMM, R., GREGORY, A. J., AND BERSHAD, B. N. Design and implementation of a distributed virtual machine for networked computers. In *17th ACM Symposium on Operating System Principles* (Charleston, SC, December 1999), pp. 202–216.

[129] SPATSCHECK, O., AND PETERSON, L. L. Defending Against Denial of Service Attacks in Scout. In *Third Symposium on Operating Systems Design and Implemenation* (February 1999), pp. 59–72.

[130] SPENCER, R., SMALLEY, S., LOSCOCCO, P., HIBLER, M., ANDERSEN, D., AND LEPREAU, J. The Flask Security Architecture: System Support for Diverse Security Policies. In *8th USENIX Security Symposium* (August 1999).

[131] STALLINGS, W. *Snmp, Snmpv2, and Cmip : The Practical Guide to Network-Management Standards*. Addison-Wesley Publishing Company, New York, 1993.

[132] STANDARD PERFORMANCE EVALUATION CORPORATION. *SPECWeb99 (White Paper)*. http://www.spec.org/osg/web99, 2001.

[133] STEERE, D. C., GOEL, A., GRUENBERG, J., MCNAMEE, D., PU, C., AND WALPOLE, J. A Feedback-Driven Proportion Allocator for Real-Rate Scheduling. In *Third Symposium on Operating Systems Design and Implementation* (New Orleans, LA, USA, Feb 1999), pp. 145–58.

[134] STEPHANIE BODOFF SUN MICROSYSTEMS, INC. Java Servlet Technology . http://java.sun.com/j2ee/1.4/docs/tutorial/doc/Servlets.html, November 2002.

[135] STEVENS, W. R. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, New York, U.S.A., 1994.

[136] STEVENS, W. R. *UNIX Network Programming, Volume 1, Second Edition: Networking APIs: Sockets and XTI*. Prentice Hall, New Jersey, 1998.

[137] STEVENS, W. R. *UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications*. Prentice Hall, New Jersey, 1999.

[138] STEWART, F., DAWOOD, M. S., DOBLAS, I., GRIFITH, G., AND NELSON, D. *Using Tivoli's ARM Response Time Agents*. IBM International Technical Support Organization, North Carolina, U.S.A., 1998.

[139] SUN MICROSYSTEMS. Java(TM) 2 Platform, Enterprise Edition. 2001.

[140] SUN MICROSYSTEMS. Java Remote Method Invocation. http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html, 2002.

[141] SUN MICROSYSTEMS INC. *Solaris Resource Manager 1.0 (White Paper).* Palo Alto, California. http://www.sun.com/software/white-papers/wp-srm/.

[142] SUN MICROSYSTEMS INC. *Sun Enterprise 10000 Server: Dynamic System Domains.* Palo Alto, California. http://www.sun.com/servers/white-papers/domains.html.

[143] SYBASE INC. Scalability Enhancements in Sybase Adaptive Server Enterprise. http://was.sybase.com/products/databaseservers/ase/whitepapers/scalability.pdf, 2002.

[144] TANENBAUM, A. S. *Distributed Operating Systems.* Prentice Hall, New Jersey, U.S.A., 1995.

[145] THAI, T., AND LAM, H. *.NET Framework Essentials* . O'Reilly & Associates, CA, U.S.A, 2001.

[146] VERMA, D. *Supporting Service Level Agreements on IP Networks.* Macmillan Technical Publishing, 1999.

[147] VMWARE INC. White Paper GSX Server. http://www.vmware.com/pdf/gsx_whitepaper.pdf, 2000.

[148] VOIGT, T., TEWARI, R., AND FREIMUTH, D. Kernel Mechanisms for Service Differentiation in Overloaded Web Servers. In *USENIX Annual Technical Conference* (Boston, MA, U.S.A., June 2001).

[149] VON EICKEN, T., CULLER, D., GOLDSTEIN, S., AND SCHAUSER, K. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture* (1992), ACM.

[150] WALDSPURGER, C. A., HOGG, T., HUBERMAN, B. A., KEPHART, J. O., AND STORNETTA, W. S. Spawn: A Distributed Computational Economy. *Software Engineering 18*, 2 (1992), 103–117.

[151] WALDSPURGER, C. A., AND WEIHL, W. E. Lottery scheduling: Flexible proportional-share resource management. In *First Symposium on Operating System Design and Implementation* (November 1994), ACM.

[152] WELSH, M., CULLER, D., AND BREWER, E. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *18th ACM Symposium on Operating System Principles* (Banff, Canada, October 2001).

[153] WOLFF, R. W. *Stochastic Modeling and the Theory of Queues.* Prentice Hall, Englewood Cliffs, NJ, 1989.

[154] WORLD WIDE WEB CONSORTIUM. Simple Test of Amount of System calls in Jigsaw. http://www.w3.org/Protocols/HTTP/ Performance/System/SysCalls.html, 2002.

[155] YAGHMOUR, K. Linux trace toolkit, http://www.opersys.com/LTT .

[156] YAGHMOUR, K., AND DAGENAIS, M. R. Measuring and Characterizing System Behavior Using Kernel-Level Event Logging. In *Proceedings of USENIX Annual Technical Conference* (2000), USENIX.

[157] YAMAMURA, S., HIRAI, A., SATO, M., YOMAMOTA, M., NARUSE, A., AND KUMON, K. Speeding Up Kernel Scheduler by Reducing Cache Misses - Effects of cache coloring for a task structure - . In *Proceedings of 2002 USENIX Annual Technical Conference (FREENIX)* (Monterey, CA, June 2002), USENIX.

[158] YAN, J., SARUKKAI, S., AND MEHRA, P. Visualization and Modeling of Parallel and Distributed Programs Using the AIMS Toolkit. *Software Practice and Experience 25*, 4 (April 1995), 429–461.

[159] YANG, Z., AND MARSLAND, T. A. Annotated Bibliography on Global States and Times in Distributed Systems. *Operating Systems Review* (1993), 55–74.

[160] YOUNG, M., TEVANIAN, A., RASHID, R., GOLUB, D., EPPINGER, J., CHEW, J., BOLOSKY, W., BLACK, D., , AND BARON, R. Qos provisioning with *q*contracts in web and multimedia servers. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (November 1987), pp. 63–76.

ABSTRACT

Model-Based System Management for Multi-Tiered Servers

by
John Reumann

Chair: Kang G. Shin

Internet services are increasingly designed as multi-tiered, i.e., composite services linked by a fast and reliable communication network infrastructure. Even though multi-tiered systems are common, their problems are only insufficiently addressed by current operating systems (OS) designs and existing literature.

This thesis specifically addresses performance management problems in multi-tiered server deployments and proposes a generalized framework which facilitates system-level management of multi-tiered activities as an add-on for standard OSs.

The problem of interference between co-located multi-tiered services is addressed by proposing *Virtual Services* (VSs) to control the performance of shared back-end services. Since the configuration of VSs requires at least a model-based understanding of the multi-tiered system, Performance Maps (PMaps) is introduced. PMaps infer the dependencies between interacting services in a multi-tiered system through observation and online service model understanding. This thesis demonstrates that metrics based on PMaps can be used to identify several performance problems that occur in multi-tiered setups.

This thesis also addresses the question of whether it would be beneficial to integrate VSs and PMaps into an online resource allocation adaptation approach that attempts to optimize an external cost or utility function by changing resource allocations. Two calibration-based approaches are proposed and shown to perform nearly as well as, or better than, aggressive rescheduling of system resources. However, it is also shown that resource allocation enforcement delays negatively affect the performance of online resource allocation adaptation, thus limiting its usefulness.

This commonalities between VSs and PMaps are captured by the proposed system support layer called *Stateful Distributed Interposition* (SDI). It is designed to simplify the adaptation of single-host systems for their use in multi-tiered setups. SDI supports the addition of arbitrary state to OS entities without requiring any kernel recompilation. SDI automatically propagates this state according to system-specific propagation rules (alongside multi-tiered activities) from one tier to another. This attached state can be used to trigger and control OS plugins, such as PMaps and VSs. A prototype of SDI is implemented and shown to add only slight ($<2\%$) performance overhead to Linux.