Stateful Distributed Interposition

JOHN REUMANN and KANG G. SHIN
The University of Michigan, Ann Arbor

Interposition-based system enhancements for multitiered servers are difficult to build because important system context is typically lost at application and machine boundaries. For example, resource quotas and user identities do not propagate easily between cooperating services that execute on different hosts or that communicate with each other via intermediary services. Application-transparent system enhancement is difficult to achieve when such context information is obscured by complex service interaction patterns. We propose a basic mechanism for sharing contextual information across the tiers of multitier computations to support system enhancement for multitier servers and applications.

This article introduces generic, cluster-wide context as a new, configurable abstraction for the OS. System administrator- or application-specified context tracking rules determine how context is associated with system processes, sockets, messages, how it is relayed along the interapplication communication channels, and how it is to be interpreted by system interpositions, thus realizing Stateful Distributed Interposition.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; C.5.5 [**Computer System Implementation**]: Servers; D.4.m [**Operating Systems**]: Miscellaneous

General Terms: Design, Management

Additional Key Words and Phrases: Distributed computing, component services, operating systems, multitiered services, server consolidation, distributed context

1. INTRODUCTION

Monolithic network services of the mainframe era are quickly being replaced with modular, lightweight multitier services. This trend is accelerated by emerging standard components for modular application design, such as application servers, servlets, CORBA, and .NET. For example, today's Internet services are composed of multiple service modules: front-end Web interfaces, middle-tier application servers, name servers, back-end databases, and storage

The work described in this article was supported in part by a Graduate Research Fellowship from IBM Corporation and by the National Science Foundation under grant CCR-0216977.
Authors' present addresses: J. Reumann, IBM T. J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; email: reumann@us.ibm.com; K. G. Shin, The University of Michigan, EECS Dept., 1301 Beal Avenue, Ann Arbor, MI 48109-2122.

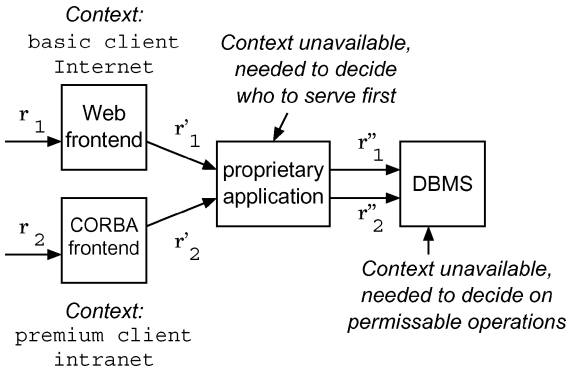


Fig. 1. Context information is lost as requests r_1 and r_2 propagate across shared intermediaries. Neither preferred processing nor effective access control can be implemented at the system level.

servers. Such modularity has brought great flexibility, reusability, and potential application logic sharing to service hosting environments and the service design process.

Unfortunately, in this respect applications have advanced far beyond the support of operating systems (OSs), thus leaving the implementation of important OS functions, such as resource quotas, user identities, and environment variables, to the applications. This increases the potential for errors. Moreover, it complicates the implementation of typical state-dependent functionality for multitiered systems, such as scheduling, access control, and tracing.

From our experience in resource management for multitier systems, propagating information across middle-tier applications that are oblivious to the fact that they are shared by multiple competing front-end services is a problematic issue (as shown in Figure 1). For example, to achieve performance isolation between competing front ends in spite of shared back-end services, it is necessary to propagate a resource context from the front end to the back-end machines [Reumann et al. 2000b; Aron et al. 2000] and to enforce it in the back ends' resource schedulers. *Virtual Services* [Reumann et al. 2000b] associate each incoming request with a Virtual Service (VS) structure, which defines a number of resource limits. This VS affiliation propagates 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 be changed at numerous places, and the network stacks must be modified to piggy-back VS IDs onto all communication messages between cooperating services, thus demanding a substantial implementation effort to achieve a relatively simple goal.

Other problems resulting from the loss of contextual information at tier boundaries range from trivial but annoying problems to serious security issues. We have all encountered the trivial, but nonetheless annoying problem that `telneting` 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 at tier boundaries is far more serious when system security and system integrity are to be preserved in a multitier 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 simply handles it as raw data. Hence, 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 [Spencer et al. 1999] and DTE [Badger et al. 1995] attempt to address this dilemma by proposing distributed security attribute propagation to manage system integrity and security in a consistent and completely application-transparent fashion.

Despite substantial differences among DTE, Flask, and VSs, we observe significant design similarities:

(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 article generalizes the above observations into a generic, multitier context service called *Stateful Distributed Interposition* (SDI). SDI addresses the following requirements.

Keep State. Provides 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. Provides a mechanism that automates the generation of context, that is, a classification facility.

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

Utilize Context. Uses dynamic contextual information to trigger interpositions on standard system interfaces. Stateful interposition allows system behavior to be influenced by context. This is an important advancement from prior interposition schemes, which operate only on fixed and existing system state.

This article is organized as follows: We define our system model in Section 2 before outlining the overall approach and the system-level architecture of SDI in Section 3. Section 4 argues for the high-level design and concepts of SDI. Important implementation aspects and practical design issues based on our experience in implementing SDI on the Linux OS are discussed in Section 5. Implementation and usage examples of our prototype system are presented in Section 6. Readers approaching SDI from an application-oriented angle may want to jump to Section 6 after reading Sections 2 and 3, to obtain a feel for the use of SDI before returning to the theory of operation and design in Section 4. Section 7 evaluates the performance of our SDI prototype. Section 8 describes how SDI generalizes previous approaches to system design by interposition and how it generalizes previous domain-specific solutions that incorporate distributed context. The article ends with concluding remarks in Section 9.

2. SYSTEM MODEL

Our system design choices of SDI are based on the following assumptions. Multitier systems consist of many servers, which are connected via a fast, reliable server area network (SAN), for example, switched gigabit Ethernet or Myrinet [Boden et al. 1995]. Each individual server hosts an arbitrary set of services, a number of which act as so-called *front-end* services. Front ends handle requests received from the outside network. They typically depend on back-end services within the server farm, such as DNS, DBMS, payment, and application servers to implement their functionality. Back-end services may be utilized by multiple independent front-end services simultaneously.

Cooperating front- and back-end services rely on OS communication primitives to exchange requests and replies. In particular, back ends receive requests as network or IPC communication messages. We assume that there are no communication channels between front and back ends that are hidden from the OS.

As argued in Section 1, cooperating tiers must share contextual state to achieve certain system management objectives. This can be accomplished without application support only if one assumes that each kernel-level thread or process executes on behalf of at most one request at any given moment. Otherwise, the application's user-level thread library or event-handler must be modified to reveal when it switches between different requests, in order to allow the OS to transfer correct contextual information along the interapplication communication channels. We assume that minor modifications to thread libraries can be made, if necessary.

SDI assumes a typical layered OS design. This assumption allows the placement of interception points for multitier 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.

Some of the parameters passed into OS functions are called *system objects*. What sets a system object apart from a simple 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. SDI adds a distributed context reference facility to system objects.

3. ARCHITECTURAL OVERVIEW

3.1 The Context Abstraction

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 at multiple hosts simultaneously—a necessity in distributed systems where multiple processes, sockets, and kernel threads may work on the same request or request class simultaneously. To facilitate coordination of policies for distributed activities in a multitiered system, we provide context, a flexible extension of the execution environment concept, as a network object.

3.2 Managing Context Dynamically

Since context objects are separate from other system objects, it is necessary to provide a mechanism by which system objects can be bound to a particular (new or existing) context object, that is, tagging of system objects. The initial context for "context-free" messages that are received from the outside of the network are tagged using classifiers (Figure 2). Context classifiers parse incoming communication messages to infer their appropriate context binding. This initial classification is only preliminary, that is, it can be modified in later stages of request processing. Classifiers extract as much useful information from incoming messages as is possible and use it in making a preliminary context determination. Processes may also be classified manually, by user-level scripts, or by the applications themselves.

The context of a packet or a process may affect 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. Context attributes affect OS behaviors at so-called tap points, which are shown at layer transitions for the Linux OS in Figure 3. The layers may differ slightly across OSs. However, equivalent features can be identified in all OSs and wrapper layers may provide common interfaces [Ford et al. 1997].

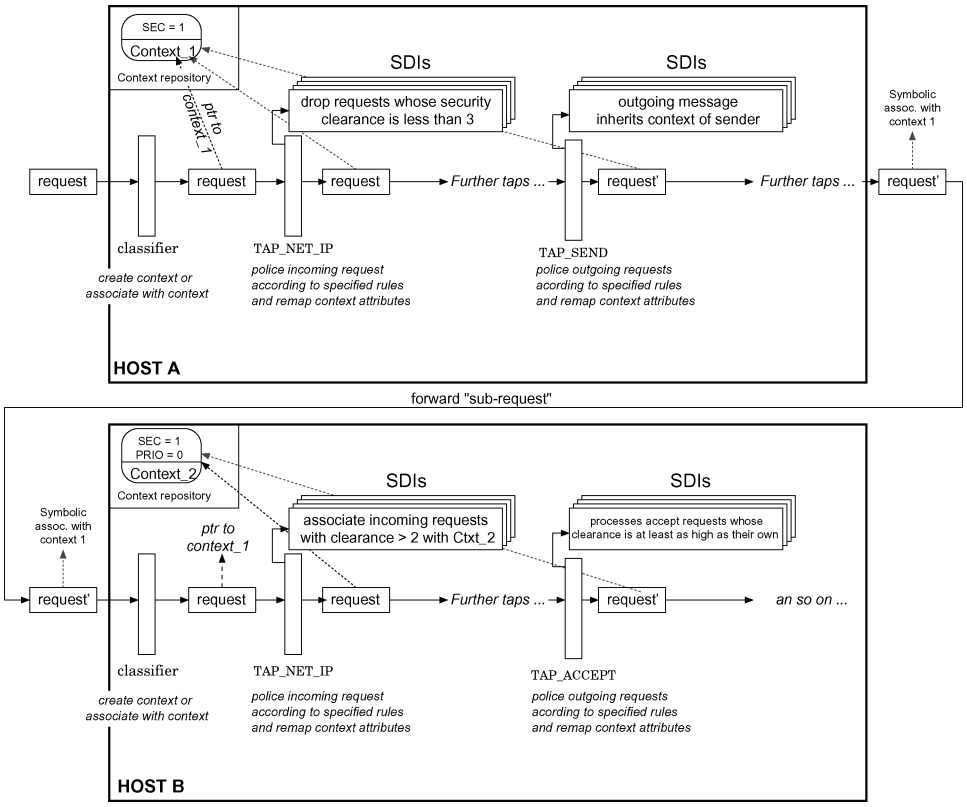


Fig. 2. SDI provides mechanisms to associate additional state with incoming messages, and propagates it according to SDI rules as request processing progresses.

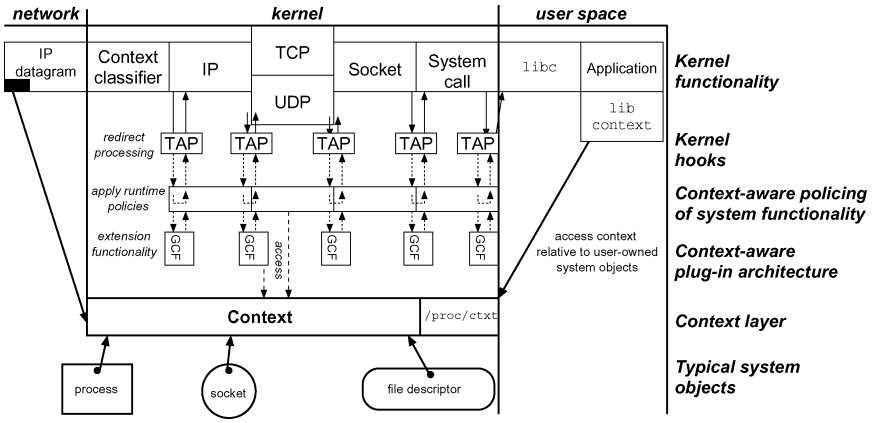


Fig. 3. Architectural overview of SDI-Linux integration. The abbreviation GCF stands for Guarded Context-Dependent Function, that is, a plug-in function that can be triggered by an SDI rule.

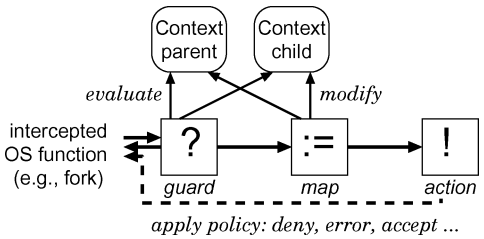


Fig. 4. The structure of SDI rules.

Taps intercept the control flow of multitiered applications at the OS-layer, and as is shown in Figure 2, can apply Stateful Distributed Interposition rules (SDIs), to intercepted computations. SDIs are rules that are dynamically installed by the system administrator or the applications. 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 tracking for multitiered applications in a standardized syntactic format (Figure 4). SDI rules are triggered if certain conditions that are specified in the guard clause of an SDI are met. These guard clauses may refer to context attributes that are associated with any of the intercepted system objects. For example, it would be possible to specify an SDI that is triggered at the `fork` system call, if the `contaminated` attribute in the calling process' context is set. If an SDI rule is triggered, the rule may modify context attributes or tag system objects with references to different context objects.

The last feature of SDI rules is a means for policing intercepted multitier computations. Policing is a recurrent theme across security, performance management, and fault isolation mechanisms. To this end, SDIs include an action directive, which may apply a standard policy, such as `DENY`, to an intercepted computation which caused the guard clause to become true. In addition to a number of predefined standard actions, SDIs also permit the invocation of arbitrary system extension functions called *Guarded Context-Dependent Functions* (GCFs), which execute after the guard and assignment have been executed. For example, one could implement encryption and decryption GCFs that are called upon arrival of messages whose respective contexts indicate `MSG_ENCRYPTED = 0` and `MSG_ENCRYPTED = 1`. Context must be implemented as an OS-level abstraction, because it may be used by OS interpositions that are transparent to the applications.

3.3 Application-Level Integration

Applications may use context as a substitute or in addition to traditional environment variables. Context is application-accessible via a simple library interface (Figure 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 access controls for context and context is extended to system objects other than processes.

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 SDIs. 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 multimedia 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. 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 and attach the socket context to every message that passes through it. If the receiving host had an SDI at the `IP_IN` tap that instructs 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 smaller latency. One could not have achieved this objective if the `FastPath` attribute were implemented at the application level. The `FastPath` example can be expanded to implement Active Messages [von Eicken et al. 1992].

4. STATEFUL DISTRIBUTED INTERPOSITION CONCEPTS

4.1 What is Context?

Context is essentially an aggregate of attribute-value pairs, 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. Each context object can be made accessible to the entire cluster. 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. Without recompiling or restarting the kernel, it is possible to associate a context object that specifies the appropriate security attribute with each process owned by the unprivileged user. Context should be understood as dynamically extensible state for system objects.

To take advantage of context information in user programs or kernel modules that may query the attributes of its process, sockets, and file descriptors, 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, the Windows registry, SNMP monitoring attributes, and will most likely be resolved in the same manner, that is, 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.

Naming Attributes. While numerous attribute naming schemes are imaginable, the following scheme is proposed. Any attribute prefixed with private., which can be encoded as a single bit, is considered to have a meaning that is applicable only within a specific cluster or system deployment. Those private attribute names have no meaning across installations and should therefore not be used in generic applications or setup-independent policies.

Some attributes that are considered to be universally useful, for example, security_clearance and priority, are not prefixed. The University of Michigan's Real-Time Computing Laboratory (RTCL) will control this global attribute name space until it is taken over by an independent standards organization. All other attribute names that are created either by other administrative bodies, concurring standard committees, hardware, or software vendors are to be prefixed with the creators Internet domain name or their SNMP prefix. This scheme reflects the effective decentralized naming schemes of Java packages and that of SNMP attributes.

Addressing Context. In order to be able to identify a specific context object whose attributes are to be queried, 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, we propose that it be 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.

Eventually it is always necessary to address context objects by absolute, global context references as shown in Figure 5:

—Global references are needed when resolving relative references to actual memory objects (possibly locations at remote hosts).
—Global references allow mapping multiple system objects to the same shared context object.
—Template context objects must be referenced through global references because they are not associated with any system object.

Template context is important when individual per-request, per-message, or per-user 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 the template is not associated with any particular system object.

The addressing mode requirements for context objects clearly distinguish SDI context from local POSIX environments, which can only be evaluated relative to the current process. Context objects can be addressed from within an

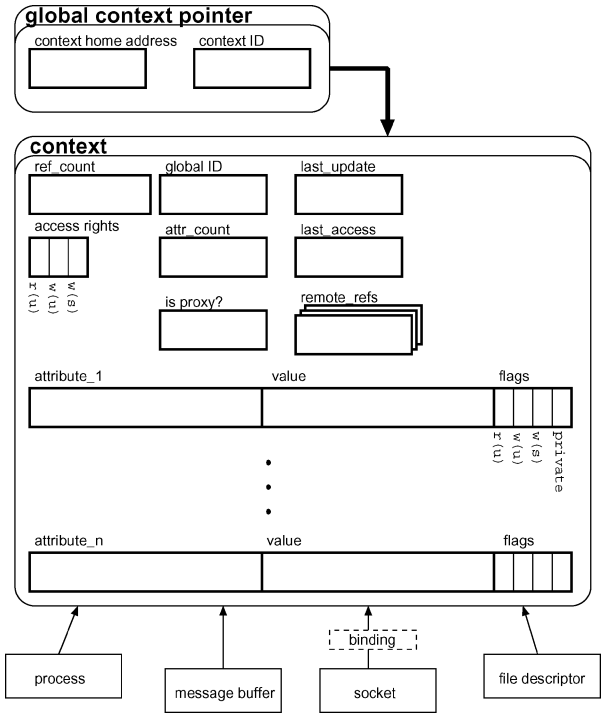


Fig. 5. Context objects may be associated with multiple system objects (sockets, processes, etc.) possibly on different hosts via global context references.

entire multitiered server cluster. This raises the question how to design its distributed reference.

Distributed objects can be referred to directly [Birrell et al. 1993; OMG 1998], via broadcast [Carriero and Gelernter 1986], or by name via a name service [Needham 1995]. 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 provide context in today's multitier environments that are often driven by 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, that is, host ID in combination with a local context handle is the only global reference method that scales to high-load multitiered service deployments.

At this point, we have defined context as a network-addressable container for attribute-value pairs that can be used to extend the state of existing OS abstractions. The features of this container object are its addressing relative to existing system objects and the fact that it allows extending system objects' states on-the-fly. For example, by associating a context object with a socket, one

can easily extend the state that is maintained for each socket. Furthermore, it is possible to share this additional state across system abstractions. For example, it is possible to associate an incoming TCP connection that is headed for port 80 with the HTTP server's context object. Thus, it becomes possible to process the incoming message using the receiving application's priority, resource quota, or application ID.

Context Propagation. It is important to note that context propagation is not essential to the nature of the context object itself, but it is fundamental to the provision of a multitiered execution context based on the context abstraction. As services interact with other services or system services that execute under different kernel executives, context must be exchanged between the cooperating kernels (Figure 2) to facilitate distributed priorities, access controls, and the like.

Context propagation is also relevant when two applications on the same host collaborate. It should be possible for a local daemon process to inherit all or some of a client's attributes without rewriting the applications by simply tracking the exchanged messages over IPC and other local messaging abstractions. This would greatly enhance system management capabilities. Traditional OS mechanisms do not address the system management problem of polcing a daemon application that works only on behalf of other applications, for example, the DNS cache. Within the daemon all requests are typically processed using the context or environment of the DNS caching process.

To achieve context propagtion it is necessary to tag system objects, such as messages, file descriptors, network packets, and processes with context references. The context references can be decoded on the receiver side and appropriate tagging rules specify how to tag the receiver of the message.

The default tagging rule should track the control and data-flow of a composite service. For example, if a process, P_1, that is associated with a context, X, writes a message to an IPC message queue, the message receives the context of the writing process, that is, X. After the message is consumed by a reading process, P_2, which, prior to reading the message from the message queue, was associated with context Y, P_2 may be associated with the context of the message that it just consumed, X. If P_2 communicates with another back-end process P_3, then P_3 will also inherit the association with context X. This behavior is quite useful for policing many multitiered applications.

Unfortunately, there are also many scenarios in which simply copying and inheriting context alongside the data-flow between applications (from writer to reader) is not appropriate. For example, a database process may not be permitted to operate in the context of a client if the client process has some very high privileges (e.g., super-user) noted in its context. In general, one needs a conditional, selective propagation mechanism that allows transferring some attributes from the front-end process, for example, the remote client's IP address, while discarding others. This requires a more flexible context rewriting framework, which handles context at control-flow and data-flow intersections between the multitiered applications and the OS. SDI serves this dynamic context management purpose.

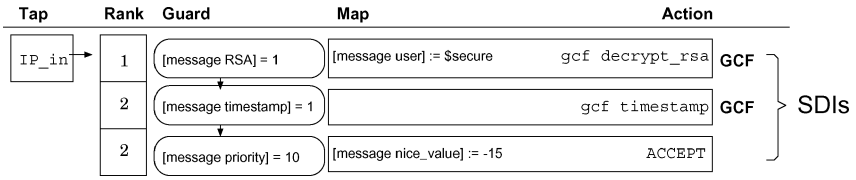


Fig. 6. The structure of SDIs: a context-dependent guard triggers attribute value remapping, context rebinding, interpositions, that is, guarded context-dependent functions (GCFs), and policies.

4.2 Stateful Distributed Interposition

The name "SDI" is used for both the framework presented in this article as well as for the individual SDI rules that allow for dynamic context rewriting, binding, policing, and interposition. This section discusses the SDI rules.

SDIs (Figure 6) are simple rules for the modification of contextual state based on previous contextual state and the interactions of computations with and within the OS. First, they provide a general mechanism to update context attributes and context bindings to system objects based on the OS operations that multitiered 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 [von Eicken et al. 1992], SPKI [Ellison 1999], Virtual Services [Reumann et al. 2000b], and future system enhancements for multitiered systems.

The guarded clause structure of SDIs (see Figure 6) leads to the definition of the SDI μ-language of Figure 7, which implements the features needed for propagating context across system interfaces and across tier boundaries. Moreover, this grammar features context rewriting capabilities and the ability to trigger actions when specific context-dependent conditions are met at interception points (a.k.a., taps).

First, Figure 7 shows that each SDI clause is applicable to a specific system interface, that is, a *tap*. Second, SDI rules execute sequentially in *rank* order because each SDI 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 [Armstrong et al. 1996] and Linda [Gelernter 1985]. This similarity is not surprising, since SDI, Erlang, and Linda are all designed to tackle distributed programming problems in a simple, extensible manner.

Tap Points specify where in the kernel a rule is applicable. Tap point implementations interpret the entire SDI rule and resolve relative references to context objects that are used in the rule language with respect to the context objects that are associated with the actual system objects that are intercepted at the tap point. Tap point implementations glue the SDI framework into the OS.

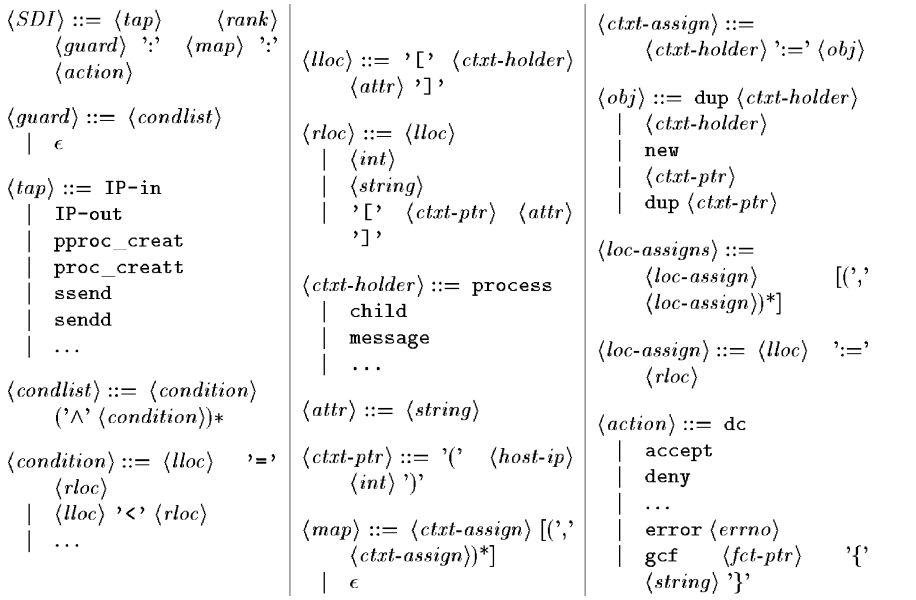


Fig. 7. Abridged SDI grammar: duplicate first and last letters of a system call name specify event-interception taps before and after the execution of the default system action, respectively. The word ϵ is the empty word. A completely empty guard always evaluates true. *Context holders*, for example, `socket`, always refer to the canonical object at the tap. For example, `socket` would refer to the sending socket in an SDI that is interposed on `send`.

The high degree of OS-dependence in each tap point implementation raises the question of whether their implementation is too difficult, thus rendering SDI impractical for real world OSs. Fortunately, this 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 SDI rules takes place in the tap point implementations, but it is largely tap-point-independent. It is done 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 system call or system function.

Guards determine whether or not an SDI is applicable, thus making context rewriting and 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_tcp_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. The final term `gcf` in the above rule represents Guarded Context-Dependent Function and instructs the SDI interpreter to jump into the specified function. This function is typically implemented as

a plug-in module for the kernel. Such interception of a specific system functionality and redirection of the OS's control flow is typical of all interposition approaches. However, SDI's ability to restrict the interposition to execute only if the calling process' security clearance, a runtime-specified attribute, is below 5 is unique to SDI. This is a clear advantage over previous models of interposition [Bershad et al. 1995; Jones 1993; Ghormley et al. 1998], which provide little control over the conditions under which a specific interposition should be invoked.

The second key element of SDIs is attribute value **remapping** and the **rebinding** of context. As mentioned earlier, this is necessary because of the potentially heterogeneous, multitiered 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 [Aman et al. 1997]. 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 that could have subsumed those operations were not used because each atom of a nonatomic 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 and should be implemented within plug-in `gcf` actions.

The term, context rebinding, refers to the fact that it may be necessary to, for example, bind a process to a different context depending on the context of the connection that it accepted most recently. The rebinding feature is typically used to track distributed activities.

The third and final key 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 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. This action allows administrators to optimize rule sets. For example, by installing catch rules for interactions that should not be policied, the checking of SDIs can be terminated faster. The `accept` shortcut action is especially useful when interactions between the applications and the OS that are not to be policed can be described in a very concise manner (with one or two rules). 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.

Whether and when the `deny` action is almost impossible for the system administrator to determine. The needed feedback 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 before denying the intercepted functionality.

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 solutions, 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`. The reason for the introduction of these conditioning actions is that many proposals for performance management in network servers rely on admission control. 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. Since SDIs can be specified dynamically, one can instantiate SDIs whenever a specific traffic class requires admission control.

Finally, the `gcf` action achieves unrestricted extensibility for the SDI framework. Whenever the generic action codes are not powerful enough to force multitier services to behave in a certain manner or to infer the correct context or attributes to be applied to intercepted system objects, generic interposition code may be invoked. 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, for example, 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, for example, 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.

4.3 Initial Context Creation and Association

One difficulty within SDI that we have not addressed thus far is the creation of context for incoming requests that are not already associated with a context. The introduced language is capable of tracking context, modifying context, and taking action based on the contents of context objects. The question as to how one would set context attributes for incoming requests remains to be addressed.

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 similar to firewalls. They reside in the bottom layers 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. They also parse packets for specific patterns to infer a context tag for the message. A classifier may evaluate an incoming packet's source address, destination address, protocol, and destination port. Based on these, the classifier consults a map and 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 network-based classifiers:

$\langle CLASSIFIER \rangle \ ::= \ \langle matches \rangle \longrightarrow \langle map \rangle$

$\langle matches \rangle \ ::= \ \langle match \rangle, \ \langle matches \rangle$

$\quad | \ \langle match \rangle$

⟨*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` ⟶ `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.

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 multitier 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

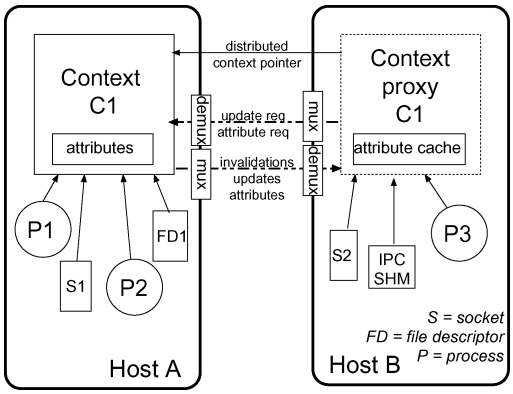


Fig. 8. The relationship between primary context, its proxies, and references from system objects.

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 each the returned file descriptor and the calling process.

4.4 Handling Distributed Context Efficiently

The language encourages the use of context for the policing of system functionality regardless of where the actual context object was created. Since the context attributes that SDIs refer to may be located remotely, repeated accesses to the same context can incur high latency penalties. The standard remedy for this problem is caching, which has already been addressed in a rich body of work in distributed and multiprocessor shared memories [Eggers and Katz 1989; Lenoski et al. 1992]. In line with these earlier results—especially Birrell et al. [1993]—distributed access to context attributes proceeds through a caching proxy object (see Figure 8).

The second efficiency 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 [Plainfossé and Shapiro 1995] 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, that is, 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 for regularly used context objects.

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 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 multitier 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, that is, 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 5). Upon detecting a failed host, the context management subsystem reduces the reference count for each primary context object. The reverse problem, the failure of the host that hosts a primary context, is discussed as a failure mode in Section 5.6.

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

4.5 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 on their own, such as a security clearance, or that would 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 more specifically 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.

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 already 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 secured 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 superuser privileges are required for the installation of SDIs on any individual host. For more centralized control, one can easily build a daemon process that executes on every host of the multitiered system under the local `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—only the superuser can install them.

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, we believe that it can and should be addressed by using IPSec [Kent and Atkinson 1998], which ensures network authenticity and privacy regardless of the message traffic. A system administrator who assumes that hosts may illegally connect to a multitiered system, should not only be concerned about protecting context but also about all other traffic between applications, which would be subject to tampering. IPSec addresses both of these problems.

4.6 Context at the Application-Level

Different context abstractions have been invented for distributed application frameworks, such as CORBA [OMG 1998], J2EE [Sun Microsystems 2001], and WebSphere [IBM Corp. 2001], thus emphasizing the need for a generic distributed context abstraction. 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. Context is lost if an intermediary service does not propagate this application-level context. 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. We export system-level context 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 the context via a 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 its recipient) generates a miniature version of a distributed thread, thus satisfying the requirements of many simple sequential applications, for example, the propagation of user IDs. More complex applications that require remapping (e.g., if certain front-end users are mapped to different back-end users) 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 the proposed 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 correct context-to-process-bindings automatically. The solution is to allow applications to specify explicitly which context objects are currently applicable. To this end, the system allows processes to export a request ID, which is tracked by SDI like a process descriptor, that is, 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.

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 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

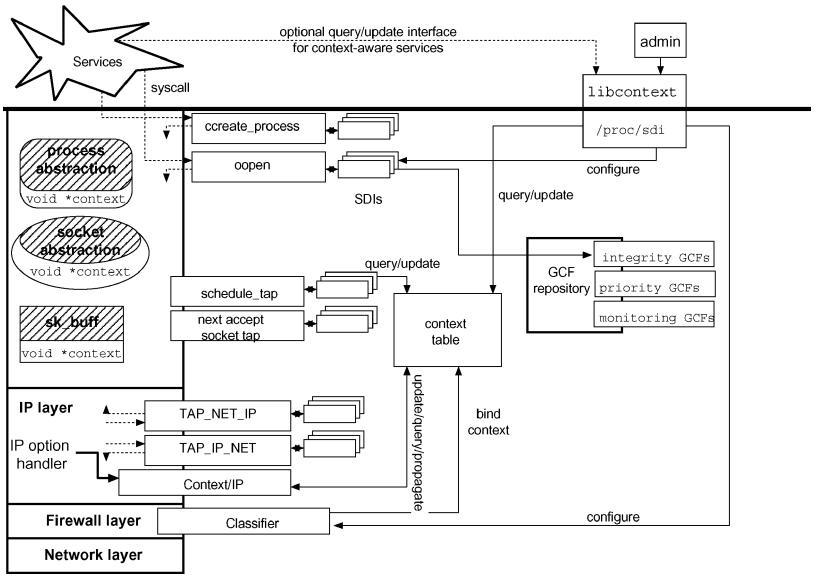


Fig. 9. Detailed architecture of the Linux-based prototype.

implementations. Tap points and GCFs are usually implemented as individual kernel modules, which are interposed at runtime.

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. These numbers 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.

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-size 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 process. In practice, this feature may be unnecessary because today's servers are equipped with large memory, 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

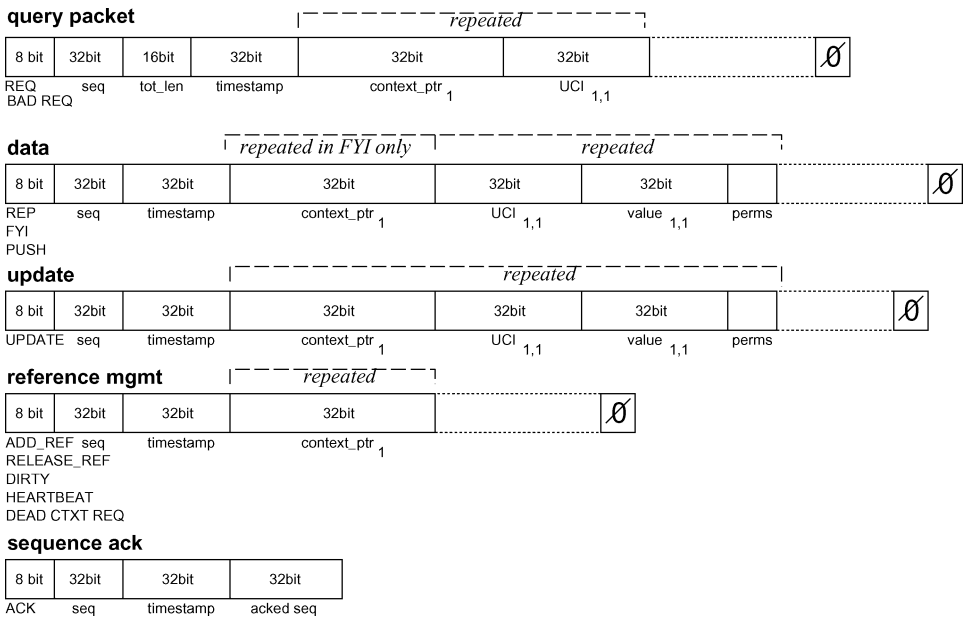


Fig. 10. Context is maintained using pre-defined message formats. The message formats leave implementors freedom to deploy and experiment with caching, consistency guarantees, and various reference management strategies.

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 10. The protocol is purposely designed in such a way that different proxy behaviors (e.g., update propagation vs. invalidation) are left configurable at runtime. 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 goal to reduce communication overheads, the reply is sent via a FYI packet. FYI packets contain only the requested attributes (see Figure 10).

Disposal of context objects and transmission 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 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 [Gray and Cheriton 1989], they would inevitably increase context access latency. Strict transactional semantics require distributed locks and multiphase 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 (see Section 8.3).

The context abstraction must consider an important trade-off to allow many short-lived context objects and those 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. Fast lookup for potentially a large number of context objects requires index structures. However, index structures typically require memory allocation and substantial initialization costs (see Section 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 honored 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 timeframe. 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 1 second (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 contexts. 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, for example, `sk_buffs`, 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.

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 [Brenton 1998] 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 }
```

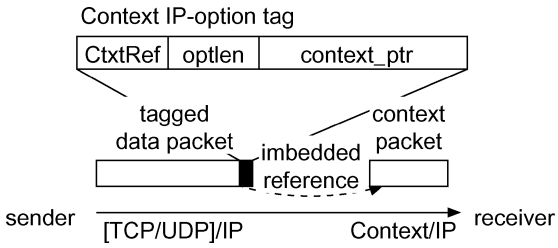


Fig. 11. Senders may push context ahead of data packets to initialize a proxy before packet receipt.

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 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 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 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 multitier server installations—with or without the endorsement of end-to-end argument advocates—their smooth 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).

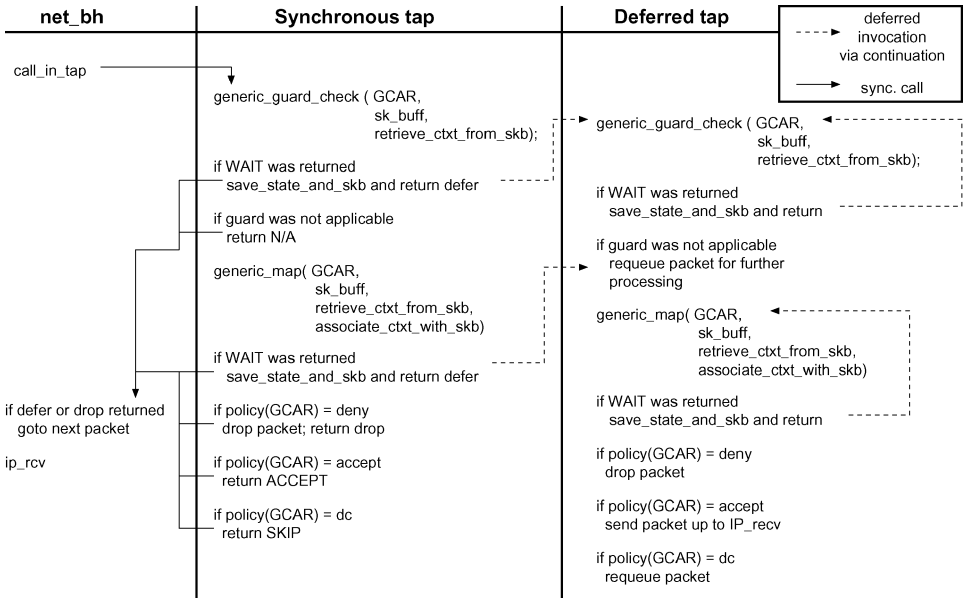


Fig. 12. A sketch of the prototype's tap at the in-bound IP interface. The tap links into the Linux firewall `call_in` chain. Most other taps are of a similar structure.

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. In accordance with to the IP specification [Postel 1981], 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, SANs are often switched, not routed, networks. Therefore, the presence of an IP option will have only negligible impact on packet forwarding times.

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 12). If this fails, the tap records its current state in a continuation structure [Draves et al. 1991], requests the nonlocal 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 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

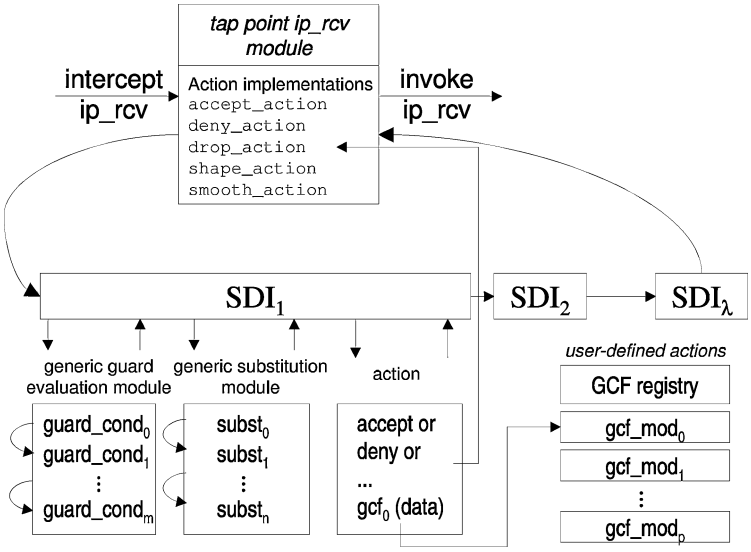


Fig. 13. Modular structure of taps, SDIs, and GCFs in the current prototype.

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 multitier services. Tap points must intercept all communication activity (`send`, `recv`, `read`, `write`) and the creation of new system objects (`fork`, `socket`, `open`). Additional tap points are optional but can be very helpful in system management tasks.

Figure 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 ($SDI_1 - SDI_\lambda$). 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 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 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 13 bottom right).

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 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 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 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 6.3).

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.

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 nonexisting attribute, its value is assumed to be `NULL`. If a context and attribute remapping directive attempts to assign a value from a nonexistent attribute to another context attribute, only that substitution clause is skipped; subsequent clauses are unaffected. Finally, GCFs may request nonexistent contexts or nonexistent 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 losses 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 losses, SDI retransmits requests for which an answer has not been received within a specified timeout (default 10 ms). 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, that is, 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 ($r = 1$ 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 times out (three 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.

6. THREE EXAMPLES OF USING SDI

6.1 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 colocated. 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 14.

6.1.1 *Implementation Steps.* The implementation steps required for SDI to enable the IPC message queue abstraction are most likely a superset to those required for SDI-enabling 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.

The second step is to define the interception points `TAP_IIPC_MSGSND`, `TAP_IPC_MSGSNDD`, `TAP_IIPC_MSGRCV`, and `TAP_IPC_MSGRCVV`. These definitions,

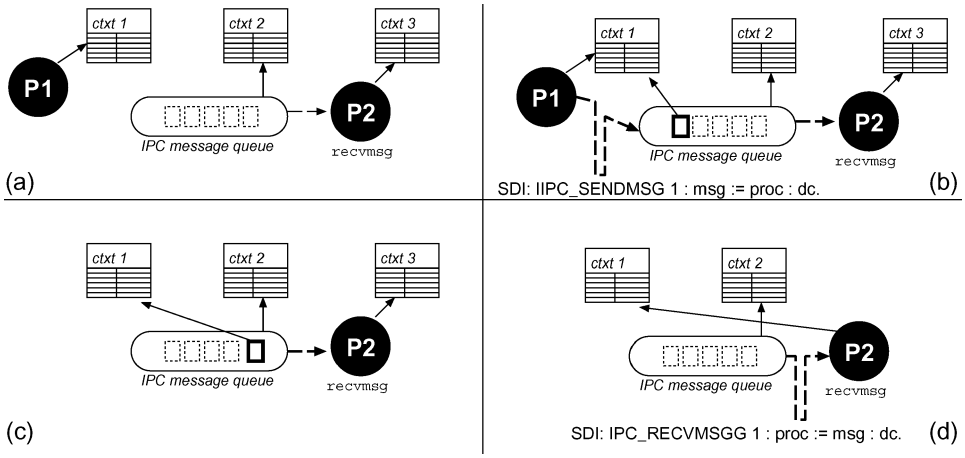


Fig. 14. This figure shows the process of context propagation through the SDI-enabled IPC message queue mechanism: (a) the sender is associated with context object 1 prior to sending the message, the queue with context 2, and the receiver with context 3; (b) the created message inherits the sender's 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.

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 an SDI registered 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 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 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, for example, to a remote message queue on a different host, one would insert a `redirect` GCF. An example of a redirection GCF is given in Section 6.3.

```
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;
...
}
```

Fig. 15. This figure shows most of the work that is required inside the kernel for the invocation of the SDI framework from IPC `msgsnd`.

6.2 Prioritized Request Handling

The first example illustrating the use and functionality of SDI, VS-SDI, is a greatly simplified implementation of Virtual Services [Reumann et al. 2000b]. The purpose of this example is not to reimplement VSs but to show how a new layer of OS functionality for a multitier system can be implemented atop 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 a 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 access to 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

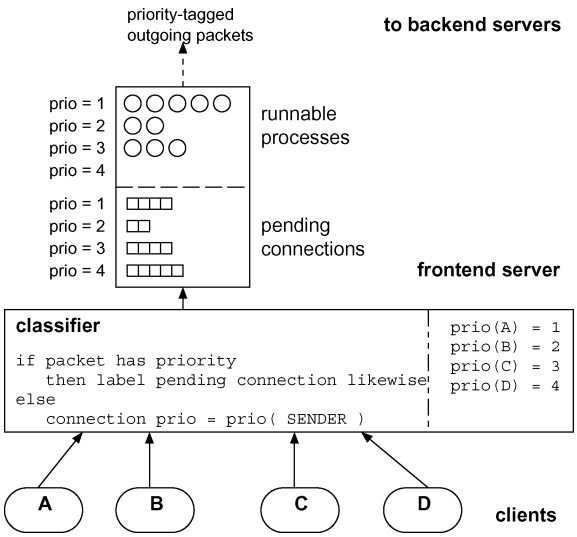


Fig. 16. VS-SDI consists of a scheduler and `accept` extension. The classifier labels incoming requests with system administrator specified priority attribute mappings, which are enforced by the scheduler and the `accept` of pending connections.

prioritize request handling throughout the server farm, that is, 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 (see Figure 16). 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
```

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.

```
/* 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;
}
```

Fig. 17. Code snippet of our context-aware scheduler interposition.

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 17 and 18 show, creating multitier-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 propagate 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;
}
```

Fig. 18. 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 19 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. To avoid this throughput drop, one would have to shed load at the front end once the system's capacity is reached [Reumann et al. 2000a].

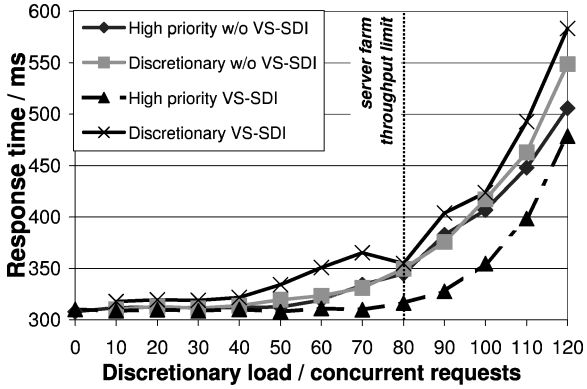


Fig. 19. The performance of a multitier server farm serving high- and low-priority clients with and without the SDI-based priority mechanism.

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.

6.2.1 *Increasing Classification Complexity.* Instead of differentiating between high- and low-priority clients at the Web server, one may decide 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, that is, 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 were submitted by the network clients. SDI solves this problem.

This example configuration requires the creation of three 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.
```

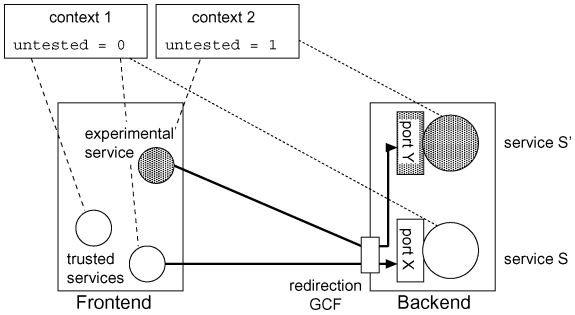


Fig. 20. Back end request redirection based on the requester's untested attribute.

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.

6.3 Protecting Back-End Service Integrity

One of the concerns when services are co-hosted in a shared services infrastructure is that new services may corrupt existing ones. 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 20). Once the new service has proven reliable, the system administrator simply deletes SDI classification rules, and requests submitted 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 [Spencer et al. 1999] 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.

7. EVALUATION

The objective of our evaluation is to determine whether SDI is an efficient mechanism for system extension. This requires fast access to context attributes, fast rule interpretation, and minimal overheads for policing multitiered applications.

Our performance study of the SDI prototype for the Linux OS indicates that SDI can be implemented efficiently. We present both micro- and macrobenchmark 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.

7.1 Micro-Benchmarks

The first set of measurements is taken on a single 450-MHz Intel Pentium II computer. The results, shown in Figure 21, demonstrate the relevance of the

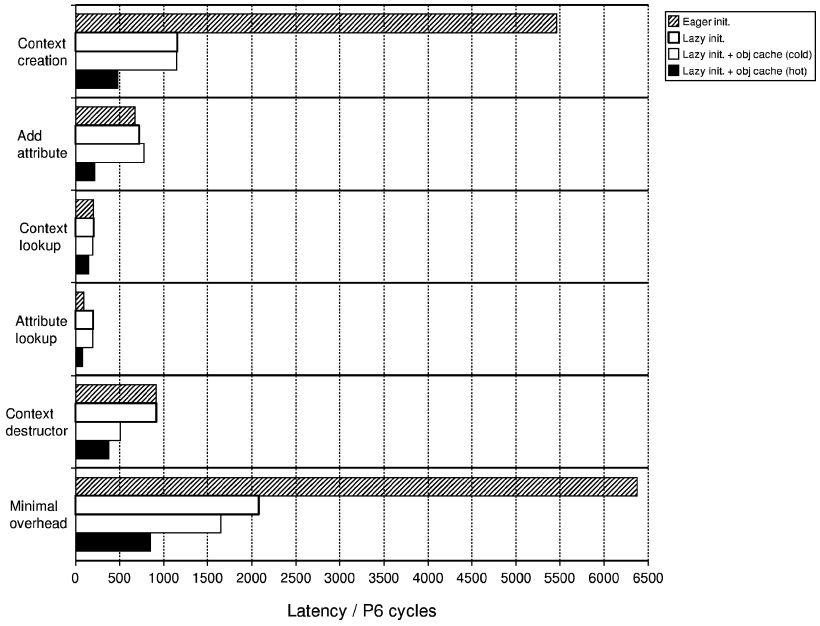


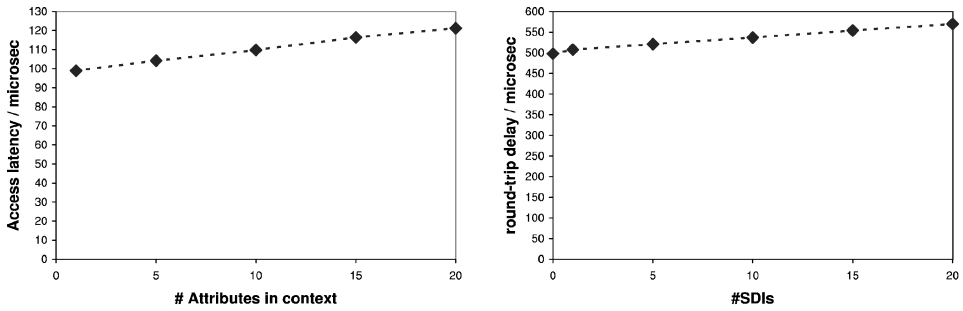
Fig. 21. Base overhead for context operations and comparison of the performance optimizations for fast context creation described in Section 5.

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 wipe out 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 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 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



(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

Fig. 22. Micro-benchmark results.

current prototype implementation, will add only negligible additional latencies to multitier services.

Figure 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 22(b)). Each additional SDI adds approximately 3 μs of latency. These delays are too small to noticeably increase 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 complexity. This has been noted in earlier interposition-based research projects [Ghormley et al. 1998; Reumann et al. 2000b]. 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'

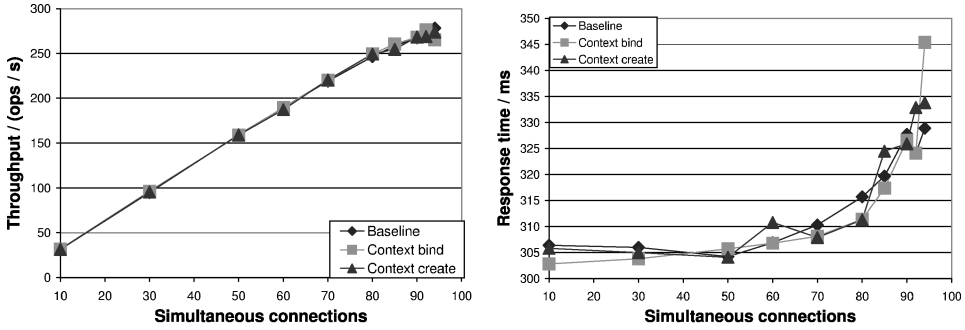


Fig. 23. Througput comparison between a system without SDI (baseline), binding incoming requests to existing context and to newly created context.

total processing time [World Wide Web Consortium 2002]. Services spend most of their time executing application code and heavy-weight `send`, `read`, `recv`, and `write` system calls.

7.2 Macro-Benchmarks

To obtain a more realistic assessment of SDI's impact on true server performance, we evaluate SDI's effect on a Web server executing the SPECWeb99 [Standard Performance Evaluation Corporation 2001] benchmark. 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 Figures 23(a) and 23(b), respectively, are reported for 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. Our measurements show that neither response time (see Figure 23(b)) nor throughput (see Figure 23(a)) of an HTTP server is affected by the presence of SDI. The lines labeled "context bind" and "context create" in Figures 23(a) and 23(b) represent the binding of incoming requests to existing context and the creation of a new context object for each incoming request, respectively.

8. RELATED WORK

The proposed SDI mechanism is the first to integrate extensible, distributed system state with interposition into a highly extensible 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, for example, 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 [Gelernter 1985] 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. Moreover, 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.

8.1 Application Frameworks

CORBA [OMG 1998], J2EE [Sun Microsystems 2001], and WebSphere [IBM Corp. 2001] are environments for the design of multitiered 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 multitiered 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.

8.2 Interposition

Since its proposal as a generic system extension mechanism by Jones [1993], interposition has gained significant support. The SPIN OS [Bershad et al. 1995; Pardyak and Bershad 1996] effectively promotes interposition as the standard way in which system functionality is to be achieved. SLIC [Ghormley et al. 1998] pursues similar goals for commodity OSs. The basic objective of interposition can be summarized as calls to existing system and service interfaces that are intercepted and redirected to interposed wrapper layers that improve or augment the intercepted interface's semantics.

The above approaches adopt an event-based dispatcher scheme [Pardyak and Bershad 1996] 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 alone 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 multitier systems cannot assume that state is always local; it must be stateful.

8.3 Domain-Specific Context Solutions

Active Messages [von Eicken et al. 1992] 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 [McIlroy and Reeds 1992; Badger et al. 1995; Ellison 1999; Abadi et al. 1993; Minear 1995] includes some of the basic features of SDI with respect to security attribute propagation, attribute remapping, and policing. However, these papers fail to abstract from the concrete problem of, say, 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 [Badger et al. 1995] stresses the need for flexible security attribute propagation along the path of interapplication communication. To address this need, this 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 [Spencer et al. 1999] 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 6).

The need for propagation of state information has also been noted in recent work on resource management. The Scout OS [Spatscheck and Peterson 1999] and Lazy Receiver Processing [Banga and Druschel 1996; Banga et al. 1999] emphasize the importance of processing incoming workload in the right resource context. To this end, they provide proprietary resource-binding mechanisms. Scout provides a processing path abstraction, which automatically propagates resource reservations across traditional OS abstractions. LRP binds incoming requests to a Resource Container [Banga et al. 1999], which can be utilized by arbitrary system objects. Both approaches fail to provide proper resource isolation when competing processes relay work to shared, remote processes. Virtual Services [Reumann et al. 2000b] solve this problem by propagating explicit resource reservation handles along with all interapplication message exchanges. Cluster reserves [Aron et al. 2000] provide similar functionality at the application-layer by using Resource Containers in combination with application modification and a resource management daemon application.

9. CONCLUDING REMARKS

We have introduced SDI as a generic, low-overhead improvement of OSs for hosting multitier services. SDI associates state with multitier computations and propagates state as computations spread to multiple machines and subservices without mandating application modification. SDI achieves the same extensibility and customizability for multitier, 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 communication 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 [Haskin et al. 1988].

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 has significantly reduced implementation complexities. We believe that SDI can generally simplify the design of system software enhancements for multitiered systems. We are currently exploring more example applications of SDI.

Despite the prototype's promising performance, there is still ample room for future research. Some obvious extensions of SDI, such as chains of SDIs and hierarchically-nested context, have to be evaluated with respect to their additional expressive power, performance benefits, and their overheads. One may also wish to provide context security in the absence of secured network links. In this case it is necessary to generate private, unforgeable context reference tags that are placed inside IP packet headers and to secure the exchange of context data between hosts.

A concrete 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, that is, 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 abstractions of context (e.g., `identd` and CORBA). In order to provide the next generation context service for IP-based multitier systems, it will be necessary to consider and address the main points raised in this article.

ACKNOWLEDGMENTS

We want to thank the referees for their helpful suggestions that have significantly improved this article.

REFERENCES

ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. 1993. A calculus for access control in distributed systems. *ACM Trans. Program. Lang. Syst. 15*, 4 (Sept.), 706–734.

AMAN, J., EILERT, C. K., EMMES, D., YOCOM, P., AND DILLENBERGER, D. 1997. Adaptive algorithms for managing distributed data processing workload. *IBM Syst. J. 36*, 2, 242–283.

ARMSTRONG, J., VIRDING, R., WILKSTRÖM, C., AND WILLIAMS, M. 1996. *Concurrent Programming in Erlang*. Prentice-Hall International, Herfordshire, U.K.

ARON, M., DRUSCHEL, P., AND ZWAENEPOEL, W. 2000. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proceedings of ACM SIGMETRICS* (Santa Clara, Calif.). ACM, New York.

BADGER, L., STERNE, D. F., SHERMAN, D. L., WALKER, K. M., AND HAGHIGHAT, S. A. 1995. A domain and type enforcement UNIX prototype. In *Proceedings of the 5th USENIX Security Symposium*. (Salt Lake City, Utah).

BANGA, G. AND DRUSCHEL, P. 1996. Lazy receiver processing (LRP): A network subsystem architecture for server systems. In *Proceedings of the 2nd ACM Symposium on Operating System Principles*. ACM, New York.

BANGA, G., DRUSCHEL, P., AND MOGUL, J. 1999. Resource containers: A new facility for resource management in server systems. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*. 45–58.

BERSHAD, B. N., SAVAGE, S., PARDYAK, P., SIRER, E. G., FIUCZYNSKI, M., BECKER, D., EGGERS, S., AND CHAMBERS, C. 1995. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating System Principles*. 267–284.

BIRRELL, A., NELSON, G., OWICKI, S., AND WOBBER, E. 1993. Network Objects. In *Proceedings of the 14th ACM Symposium on Operating System Principles*. ACM, New York. 217–230.

BODEN, N. J., COHEN, D., FELDERMAN, R. E., KULAWIK, A. E., SEITZ, C. L., SEIZOVIC, J. N., AND SU, W.-K. 1995. Myrinet: A gigabit-per-second local area network. *IEEE Micro 15*, 1, 29–36.

BRENTON, C. 1998. *Mastering Network Security*. Sybex, Alameda, Calif.

CARRIERO, N. AND GELERNTER, D. 1986. The S/Net's Linda Kernel. *ACM Trans. Comput. Syst. 4*, 110–129.

DRAVES, R. P., BERSHAD, B. N., RASHID, R. F., AND DEAN, R. W. 1991. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating System Principles*. ACM, New York. 122–136.

EGGERS, S. J. AND KATZ, R. H. 1989. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th Annual Internation Symposium on Computer Architecture*. ACM, New York. 2–15.

ELLISON, C. 1999. The nature of a useable PKI. *Comput. Net. 31*, 8, 823–830.

FORD, B., BACK, G., BENSON, G., LEPREAU, J., LIN, A., AND SHIVERS, O. 1997. The flux OSKit: A substrate for OS and language research. In *Proceedings of the 16th ACM Symposium on Operating System Principles* (Saint Malo, France). ACM, New York.

GELERNTER, D. 1985. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1, 80–112.

GHORMLEY, D., RODRIGUES, S., PETROU, D., AND ANDERSON, T. 1998. SLIC: An extensibility system for commodity operating systems. In *USENIX Annual Technology Conference*.

GRAY, C. G. AND CHERITON, D. R. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM Oper. Syst. Rev. 23*, 5, 202–210.

HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. 1988. Recovery management in quick-silver. *ACM Trans. Comput. Syst. 6*, 1, 82–108.

IBM CORPORATION. 2001. http://www-4.ibm.com/software/webservers/appserv/whitepapers.html.

JONES, M. B. 1993. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating System Principles*. ACM, New York, 80–93.

KENT, S. AND ATKINSON, R. 1998. Security architecture for the internet protocol. IETF RFC 2401.

LENOSKI, D., LAUDON, J., GHARACHORLOO, K., WEBER, W., GUPTA, A., HENNESSY, A., HOROWITZ, J., AND LAM, M. 1992. The Standford dash multiprocessor. *IEEE Comput. 25*, 62–79.

McIlroy, M. D. and Reeds, J. A. 1992. Multilevel security in the UNIX tradition. *Softw. Pract. Exper. 22*, 8 (Aug.), 673–694.

Minear, S. E. 1995. Providing policy control over object operations in a mach based system. In *Proceedings of the 5th USENIX Security Symposium*.

Needham, R. M. 1995. *Distributed Systems*, 2 ed. Frontier Series. ACM, New York, Chap. 12, 315–327.

OMG, ed. 1998. *The Common Object Request Broker Architecture and Specification 2.2*. OMG.

Pardyak, P. and Bershad, B. 1996. Dynamic binding for an extensible system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation* (Seattle, Wash.). 201–212.

Plainfossé, D. and Shapiro, M. 1995. A survey of distributed garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (Kinross, Scottland).

Postel, J. 1981. Internet protocol DARPA internet program protocol specification. IETF RFC 791.

Reumann, J., Jamjoom, H., and Shin, K. G. 2000a. QGuard: Protecting internet servers from overload. Tech. Rep. CSE-TR-427-00, The University of Michigan.

Reumann, J., Mehra, A., Shin, K., and Kandlur, D. 2000b. Virtual services: A new abstraction for server consolidation. In *USENIX Annual Technical Conference.*

Spatscheck, O. and Peterson, L. L. 1999. Defending against denial of service attacks in scout. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation*. 59–72.

Spencer, R., Smalley, S., Loscocco, P., Hibler, M., Andersen, D., and Lepreau, J. 1999. The flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*.

Standard Performance Evaluation Corporation. 2001. *SPECWeb99 (White Paper)*. Standard Performance Evaluation Corporation, http://www.spec.org/osg/web99.

Sun Microsystems. 2001. Java(TM) 2 Platform, Enterprise Edition.

von Eicken, T., Culler, D., Goldstein, S., and Schauser, K. 1992. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*. ACM, New York.

World Wide Web Consortium. 2002. Simple test of amount of system calls in jigsaw. http://www.w3.org/Protocols/HTTP/ Performance/System/SysCalls.html.

Received September 2001; revised July 2002 and July 2003; accepted August 2003