Sapphire: Statistical Characterization and Model-Based Adaptation of Networked Applications

Abhijit Bose, Mohamed El Gendy, and Kang G. Shin

Abstract—Many modern networked applications require specific levels of service quality from the underlying network. Moreover, next-generation networked applications are expected to adapt to changes in the underlying network, services, and user interactions. While some applications have built-in adaptivity, the adaptation itself requires specification of a system model. This paper presents *Sapphire*, an experimental approach for systematic model generation for application adaptation within a target network. It employs a nearly-automated, statistical design of experiments to characterize the relationships of both application and network-level parameters. First, it applies the Analysis of Variance (ANOVA) method to identify the most significant parameters and their interactions that affect performance. Next, it generates a model of application performance with respect to these parameters within the ranges of measurements. The key benefit of the framework is the integration of several well-established concepts of statistical modeling and distributed systems in the form of simple APIs so that existing applications can take advantage of it. We demonstrate the usefulness and flexibility of *Sapphire* by generating a performance model of an audio streaming application. We show that many existing multimedia and QoS-sensitive applications can exploit a statistical modeling approach such as *Sapphire* to incorporate application adaptivity. The approach can also be used for feedback control of distributed applications, tuning network and application parameters to achieve service levels in a target network.

Index Terms—Application-aware adaptation, measurements, statistical analysis, performance analysis.

1 Introduction

Today's enterprise networks comprise a variety of access, edge and core subnetworks with different levels of bandwidth, delay, loss, and jitter characteristics. For example, it is common to have a combination of wired, satellite, and wireless segments in an enterprise network of an organization. The degree of tolerance or sensitivity to each of these parameters varies widely from one application to another. Emerging applications such as home networking, online gaming, intelligent appliances, factory supply-chain networks, as well as the vast majority of multimedia applications require certain levels of service quality from the underlying network. Many ERP applications are routinely accessed by both mobile users using their PDAs or notebooks, as well as desktop clients. Service differentiation is achieved by allocating different amounts of network resources, such as link bandwidth and buffers, to different types of traffic traversing the network's intermediate routers along the end-to-end (e2e) path. In our previous work [1], we presented a characterization of per-hop QoS as the building block of the e2e QoS perceived by users that can be measured and monitored. However, a crucial step in providing application-level QoS is to generate a model of the application adaptation behavior in a given enterprise environment. This is environment-specific because enterprise networks are configured with many different devices and topologies.

This critical requirement of measuring and monitoring the e2e performance of enterprise applications has led to many commercial tools [2], [3]. However, most of these tools use only well-known ports to analyze traffic between end-hosts, and provide only basic information about application-network interactions. Several studies [4], [5], [6], [7] have addressed the problem of provisioning and mapping the e2e QoS requirements of multimedia applications to available network resources. For example, Nahrstedt and Smith [5] proposed negotiation and information exchange between applications and networks at the call/connection boundary. The mapping is facilitated by a broker that can perform bidirectional mapping between applications and the network-level parameters. However, they assumed that the application-specific mapping models for a given network are available prior to invocation of the application. In fact, this is an implicit assumption used in many proposed QoS and application adaptation frameworks.

There are adaptation models developed for specific classes of applications. For example, Bolot and Vega-Garcia [8] developed mechanisms to limit the impacts of jitter and packet loss on the clients, as well as to limit the sending rate to the capacity of the connection. MPEGTool [9] is an X Windows-based MPEG encoder and statistical analysis tool that can characterize variable bit-rate MPEG video traffic in ATM networks. In this paper, we present a

- *A. Bose is with Michigan Grid Research and Infrastructure Development (MGRID), The University of Michigan, 2356D Duderstadt Center, 2281 Bonisteel Blvd., North Campus, Ann Arbor, MI 48109. E-mail: abose@umich.edu.*
- *M. El Gendy is with Cisco Systems, Inc., 170 West Tasman Dr., Bld SJ-20, San Jose, CA 95134. E-mail: mgendy@eecs.umich.edu.*
- *K.G. Shin is with the Real-Time Computing Laboratory, Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109-2122. E-mail: kgshin@eecs.umich.edu.*

Manuscript received 16 Jan. 2005; revised 3 Oct. 2005; accepted 27 Dec. 2005; published online 25 Oct. 2006.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0033-0105.

general-purpose experimental framework called *Sapphire* that provides interaction models between a given networked application and the network in which it is deployed. The key contribution of *Sapphire* is the integration of several well-studied concepts in statistical modeling, event management in distributed systems, and application and network-layer instrumentation to provide an automated system for generating such interaction models. It provides application developers and network engineers with a set of APIs (Application Programming Interfaces) for instrumenting an application, along with several kernel modules for setting network-layer parameters. Events corresponding to changes in the levels of target parameters in the instrumented layers are logged by *Sapphire*. The resulting traces are collected along with timestamps and details about these events. Then, the events are correlated with network measurements using statistical analysis, which leads eventually to a set of models that describe the interaction of the application and network parameters. The models are then incorporated in the application at runtime to adapt to the varying network conditions based on a feedback control loop. Note that Sapphire models are *application* and *network-specific*, i.e., they can be used for the specific applications and the deployed network that were used to build the models. This is due to the fact that the performance of many applications varies considerably between testbed and deployed network environments. The most suitable environments for *Sapphire* are, therefore, enterprise networks that allow better control and setting of both application and network-level parameters.

Since it is not possible to provide deterministic guarantees to networked applications, we propose an adaptation framework for the application performance based on statistical mapping. There are several challenges to overcome. Different classes of applications have very different traffic characteristics and require different types of networked resources. This is true for individual participants ("clients") of a given application as well. Distributed applications use a variety of network interactions—such as client-server, peer-to-peer, and distributed multicast—that make characterization of such applications very difficult. *Sapphire* probes and associated APIs make instrumentation of these applications relatively easy and the model-building step allows one to automatically build interaction models for the application in a given network. *Sapphire* does not depend on particular network and switching topologies. For example, it can be used in tightly-coupled cluster environments to generate interaction models of messaging-passing parallel applications. Moreover, the application-level parameters in *Sapphire* are completely user-defined and, therefore, interaction models at different levels of granularity can be generated using the framework.

The workflow of *Sapphire* and its components are explained in Section 3. The core of the framework tracks individual processes of a networked application and injects specific events (as part of instrumentation) to control levels of user-defined application parameters and the parameters in the network stack during runtime. It can also set up related network configuration parameters at the intermediate routers of the target network so that coordinated experiments are possible. While the statistical modeling techniques such as ANOVA and polynomial regression used in *Sapphire* are well-established, the capability of automatically generating application-network interaction models for an application is a unique feature of the framework and, therefore, it can be useful to a large community of users.

The rest of this paper is organized as follows: First, in Section 2, we provide an example of an audio streaming application and motivate the need for model-based application adaptation. Next, we present the *Sapphire* framework in Section 3, and describe the main components and the associated work-flow. The statistical methods used in the framework—design of experiments, ANOVA, and regression—are briefly discussed in Section 4. We provide experimental results in Section 5 to demonstrate the flexibility of our framework and an example of model-based adaptation in the context of the audio-streaming application. Section 6 reviews related literature and, finally, we conclude the paper in Section 7 with a discussion on possible extensions of *Sapphire*.

2 Motivation

The primary goal of *Sapphire* is to develop an *integrated* and *automated* framework for the generation of interaction models of an application with respect to its underlying network. Both heuristic and statistical methods can be applied to generate such models. We have implemented an approach based on the design of experiments and polynomial regression to generate the application-network interaction models. To illustrate this further, let us consider an audio streaming application using the popular Pulse Code Modulation (PCM) encoding technique. PCM is a digital encoding scheme for analog data and is used in many multimedia applications such as full-motion video, telephony, music and virtual reality. In PCM, an analog signal is sampled at a regular frequency, called the *sampling rate* (in Hz), which is typically several times the maximum frequency of the analog waveform. The samples are quantized in one of the predetermined levels, where the levels are represented by a fixed number of power of 2 bits. The output of PCM is a binary series of the original analog signal in digital form. The parameters that determine the quality of the audio stream are the sampling rate, number of bytes per sample, and duration of audio per packet. An audio streaming application consists of a PCM transmitter (or server) that reads a set of audio files, encodes them into PCM, and sends the streams to a set of remote clients. The clients use a PCM decoder and play them using an audio device. The audio traffic is sensitive to packet loss, jitter and available bandwidth, and the background traffic in the network may affect the playback quality. Network-level QoS for such traffic is usually offered by the Differentiated Services (DiffServ) [10], [11], bandwidth reservation, etc. The traffic from different applications are assigned to different service classes depending on the application requirements. Most importantly, they are protected from background (mostly best-effort) traffic in the network. The parameters affecting the performance of the audio-streaming application are PCM encoding parameters as well as those of the DiffServ building blocks used in the construction of the network. In this example, *Sapphire* is used to determine which of these parameters are the most significant and how they are related so that an application-network interaction model can be generated.

The audio streaming can also be performed more efficiently by incorporating adaptivity in the server. For example, the server may adapt to different sampling rates and sample sizes based on feedback from the different clients and the corresponding network configurations. While this is desirable for any networked application, the difficulty lies in deciding the ranges by which the application must adapt its parameters to satisfy the e2e QoS requirements. *Sapphire* provides this model in the form of an algebraic polynomial of the most significant parameters. In practice, such a model can be incorporated within an application (both server and client sessions) and activated via periodic feedback using RTCP (RTP Control Protocol) [12]. RTCP is based on the periodic transmission of control packets to all participants of an application session using the same distribution mechanism as data packets. A typical usage of RTCP is control of adaptive encodings in audio and video streaming applications. The diverse requirements of real-time applications are supported in RTCP in the form of application profiles and associated payload format specifications. We envision *Sapphire* to be useful to application programmers since they do not have to learn RTCP details. Using *Sapphire*, applications can define which parameters and associated events will be monitored during runtime, and the transport of these parameter values and events between server and clients can be handled by RTCP. The translation of *Sapphire* parameters and events to corresponding RTCP packets can be generated automatically via a compiler.[1]

One of the advantages of *Sapphire* is to allow hierarchical parameters. To illustrate this, consider the problem of compressing video data for delivery over the Internet. A lower keyframe rate will reduce the required bandwidth, but it will also result in a lower-quality image. At the same time, when motion increases, one must increase the keyframe rate and the frame rate since high-motion video clips require additional uncompressed keyframes to be encoded in the video file. To develop an interaction model of this application using *Sapphire*, it is possible to assign a high-level parameter describing the degree of motion (e.g., perceived speed of an object as computed from a video game) that is implicitly related to the keyframe rate. When a model is generated, it links the degree of motion parameter directly to network bandwidth. *Sapphire* thus enables us to link any network-sensitive application-level parameter directly to network-level parameters such as bandwidth, delay, loss, or jitter. This hierarchy also applies to evaluation of "perceptual QoS" [13], [14] parameters that determine the user's perception of quality for a networked application. Examples of user-level perceptual parameters are image quality, video rate, video smoothness, audio quality, etc. The corresponding application-level parameters are pixel resolution, frame rate, frame rate jitter, sampling rate, and number of bits per sample, respectively. While the subjective quality assessment based on "mean opinion score" (MOS) is reliable, it is often time-consuming and expensive to determine the MOS scores of a network-sensitive application. Therefore, estimation techniques for the subjective quality of specific applications such as VoIP have been proposed [13] by measuring the physical characteristics of end-hosts and networks. The authors of [14] proposed an experimental methodology to calculate perceptual QoS parameters based on arbitrary packet traces and successively matching the encoded speech sample with all possible trace fragments. In both examples, *Sapphire* can be used to identify the hierarchical mapping relationships among the user-level perceptual QoS parameters, application-level parameters, and network-level parameters.

1. However, we have not implemented this in our current prototype.

Finally, our framework is suitable for enterprise networks rather than the Internet because the design parameters required in the model-building process can be controlled more tightly in the former. The role of the network service provider (NSP) and the application service provider (ASP) for the enterprise in this context is important. While the service-level agreements (SLAs) with NSPs are standard for most enterprise environments, the SLAs for hosting applications and services are not. Enterprises may depend on ASPs for some of their applications, while running other applications in-house. Any QoS adaptation model for an application must be supported with appropriate SLAs by the respective NSP and ASP (if the application is hosted by a third party). The *Sapphire-generated* adaptation models assume that the underlying network and application-level SLAs support the ranges of permitted adaptation.

Fig. 1 shows the steps of the framework which are detailed in the following section.

3 The *Sapphire* Framework

The *Sapphire* framework provides the necessary tools and methods for instrumenting an application (server and clients), the underlying network stack, as well as the intermediate network elements such as routers. Then, it uses well-established methods for experimental design and statistical analysis to run coordinated experiments based on the configurable instrumented parameters. The data from these experiments are used to generate an interaction model, which can then be used by the application in order to systematically adapt itself to varying conditions in the network. By design, such a framework is suitable for "model-based adaptation." In what follows, we elaborate on the goals, the core components, and the workflow of the framework.

3.1 Design Goals

In the design of *Sapphire*, we would like to achieve the following goals:

- **Supporting coordinated experimental analysis.** To enable efficient exploration of the parameter space, *Sapphire* creates a design of experiments from user-specified parameters by coordinating all instrumented layers with their corresponding levels for each experiment. Such coordination among various processes of a networked application and different protocol stacks of an end-host is achieved via a synchronization protocol to be described shortly. Therefore, an application instrumented with *Sapphire* can be configured in various degrees of freedom (monitored parameters) across its multiple protocol layers. Additionally, this configuration is coordinated with the intermediate network path to provide synchronization of network interactions. The last step requires that enterprises have control over these network-level parameters.
- **Supporting reliable delivery of events.** Since injected application events are correlated with network measurement data, *Sapphire* must log and

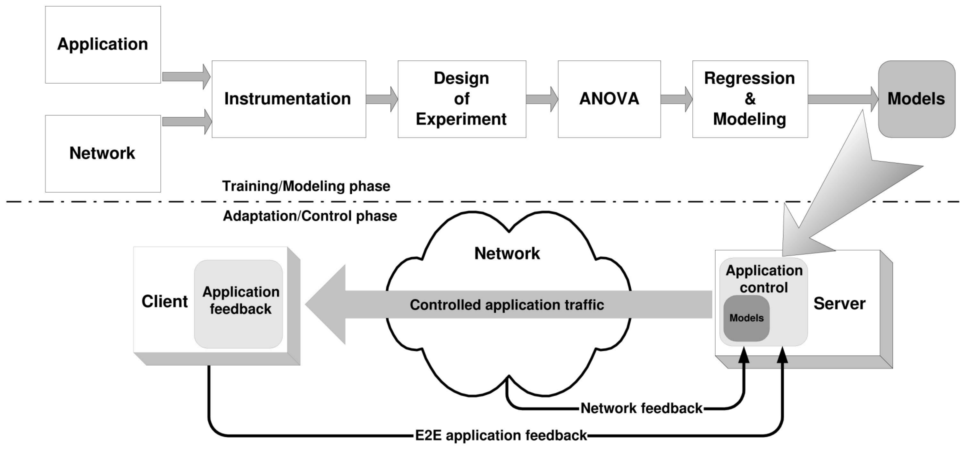

Fig. 1. Workflow of *Sapphire*.

deliver all events reliably. This is achieved by using ARDP (Asynchronous Reliable Delivery Protocol) [15]. Once a set of parameters (in any of the instrumented layers) have been chosen, *Sapphire* keeps track of any change of their levels and logs appropriate event messages entirely transparent to the application by using separate message channels.

- **Statistical modeling.** The goal of providing deterministic QoS guarantees to networked applications is extremely difficult and, therefore, *Sapphire* chooses a statistical modeling approach. The outcome of *Sapphire* experiments is a set of statistical models of the given application in the deployed target network. Two well-established statistical methods, ANOVA and multiple polynomial regression, are integral to the framework and will be presented shortly.
- **Lightweight APIs.** The instrumentation of an application using *Sapphire* is meant to be straightforward —only a small set of functions are needed to initialize, synchronize and perform a set of experiments. Hence, the overall processing and communication overheads are low. Complexities such as synchronization and coordination among multiple processes (e.g., clients) of an application during a *Sapphire* experiment are handled by the framework transparent to the application sessions and the user.

3.2 Workflow of *Sapphire*

As shown in Fig. 1, *Sapphire* works in two phases. First, the training or the modeling phase which involves instrumenting both the application and the underlying network to be able to inject events and control parameters as dictated by the design of experiments. A factorial design of experiments is conducted to investigate the interactions among the different parameters affecting the performance of the application with respect to the underlying network. The first statistical analysis method used is called ANOVA, which extracts the most significant factors affecting performance as described in Section 4. Based on these significant factors, regression-based modeling is performed to relate the user-specified application parameters to the parameters of the network. In the second phase, the models extracted are used in adapting the application performance according to measurements from the network as well as e2e measurement from the application parameters themselves in the feedback loop as shown in Fig. 1.

3.3 Components of *Sapphire*

Fig. 2 illustrates the main components of the framework. Each component in *Sapphire* essentially builds an abstraction for a particular capability, and the components communicate with each other via messages. Such an approach allows us to incorporate new measurement methods and introduce new hardware (network nodes, devices, end-hosts) without changing the overall framework. Each of these components is detailed next.

3.3.1 Application Instrumentation and Application-Level Trace Collection

The heterogeneity of application-level objects and parameters requires a flexible approach for runtime application-level parameter configuration, measurement and collection. The simplest mechanism is needed when most of the configurable parameters are available in the command-line interface (CLI) or via a runtime configuration file for the application. In this case, there is no need to instrument the application—*Sapphire* simply generates timestamped events corresponding to the set of configuration parameters for each experiment so that they can be correlated with the corresponding network-level traces during the analysis step. One can also instrument the message channels between all the processes of the application, e.g., the socket API in the kernel. This enables a suitably-instrumented end-host to collect information about the application-level data that are exchanged between the instrumented end-hosts.

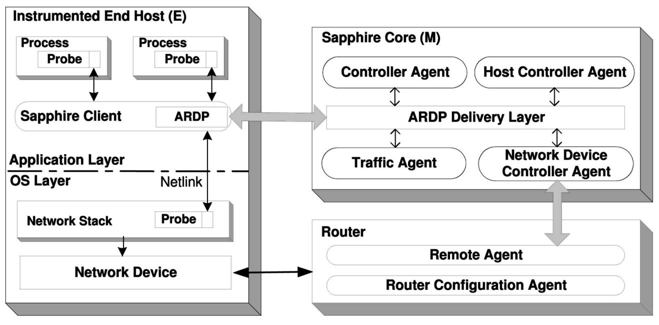

Fig. 2. Components of *Sapphire*.

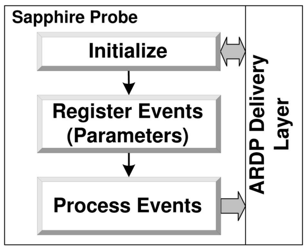

Fig. 3. Components of a generic *Sapphire* probe.

The latter approach is, of course, coarse-grained. The synchronization among the end-hosts is achieved by a group management protocol. In fact, a combination of CLI and message channel monitoring may be sufficient for most applications without ever instrumenting the application itself. The most extensive mechanism is an explicit instrumentation of the application to insert events. Any change in the level of a factor is encapsulated in an event and captured by *Sapphire*. The basic mechanism for instrumenting an application is a *Sapphire* probe. Probes are used for both active and passive monitoring of parameters of a networked application. An active *Sapphire* probe can set the level of a parameter during runtime, thereby affecting the performance of the application. On the other hand, a passive *Sapphire* probe simply monitors the levels of parameters of instrumented protocol layers and generates appropriate event messages.

3.3.2 Sapphire *Probe*

A *Sapphire* probe, shown in Fig. 3, provides four functionalities:

1. synchronizing the instrumented process with other processes of the networked application at the beginning of an experiment,
2. registering a set of user-defined parameters to be monitored and modified during the experiment,
3. providing a logical ordering of all events (changes in levels of monitored parameters) via Lamport's vector clocks [16], and
4. forwarding events to other probes and the core agents, as necessary.

Although not implemented in our current version, a *Sapphire* probe can also be inserted in protocol layers other than the application itself. For example, probes can be used to control various parameters of the transport layer (e.g., TCP send and receive buffer sizes, MTUs, and number of parallel streams) to study the interaction of an application with respect to changes in transport-layer settings in the end-hosts. This flexibility in *Sapphire* makes it a general-purpose characterization approach across many application domains.

3.3.3 APIs

Table 1 lists the primary APIs available for application instrumentation. An application can define and manipulate events at a lower level by calling methods available in the *Sapphire* core (Table 2).[2] The event monitoring and application-level APIs are designed to be general-purpose. For example, we have instrumented a network stack and the PCM audio streaming application for the results presented in Section 5. As mentioned earlier, users are free to include any application-level parameter during the model-generation phase. The parameters can also be hierarchical in case of complex networked applications that use multiple components and services.

3.3.4 Event Management

The events of a networked application can be broadly classified as internal and external events. The internal events are generated within a process and do not interfere with other processes. In *Sapphire*, one can set internal events as fine-grained as tracking changes to individual variables, as well as higher-level objects such as functional blocks and state changes. For applications that require large-scale computations or network resources, it is becoming more acceptable to introduce such instrumentation for performance analysis and monitoring as long as the measurement overhead is a small fraction of the overall resource usage. During *Sapphire* initialization, an event structure is created for each class of variables and each event class is assigned an *event_type*. Associated with each class of events, there is an event sequence string, or *event_seq*. The event sequence strings identify events within a class of similar events. The individual fields in the *event_seq* string are defined as follows: (*Ipaddr: PortNum*) are the IP address and port number for the originating process, respectively. The *PID* is the process identification number—it is included to accommodate multiple processes on a single device or end-host. *TSTamp* is a globally-ordered timestamp using vector clocks, and the field *Dir* is a string denoting a location of the event data. Therefore, an *event_seq* object is represented as the tuple: ((*Ipaddr:PortNum*), *PID*, *TSTamp*, *Dir*). The event definition format containing *event_type* and *event_seq* uniquely identifies any event in a given process group. The event notification messages are delivered using these two items only. The values of the event objects themselves are cached locally on the originating end-host/device and are not exchanged. This approach has two advantages: 1) the format of the event messages to be exchanged is fixed no matter how large or small the actual event objects are, and 2) by having a fixed format, one can always bound the amount of network resources to be used by the core agents. Further, having a fixed format for event notification and message ordering makes *Sapphire* general-purpose since it can be used irrespective of the type of messages and events generated by different applications. As an example, consider two entirely different classes of applications: distributed simulation of molecular structures and video-conferencing. The first application (a class of scientific computing) is typically a collection of processes that are highly CPU-intensive, with occasional message passing to exchange large volumes of data. In contrast, the second application (a class of multimedia applications) requires soft real-time guarantees and frequent exchange of audio and video packets via the H.323 protocol. Both classes of applications can be instrumented using the same set of APIs in *Sapphire*—the only difference is how probes interpret and set different parameters and objects in the respective application. Hybrid applications such as virtual scientific collaboratories may incorporate both classes of applications. Using our framework, the events from different applications (running as different sets of processes) may have different *event_types* and *event_seqs* and, therefore, do not require any changes to the underlying core libraries.

2. Security issues of modifying parameters across multiple protocol layers by the application are not addressed here, but will be investigated in the future.

TABLE 1
The Programming Interfaces for Application Instrumentation and Event Definition

Function	Arguments	Description
sa_init( )	sa_t * handle, procinfo *p	initialize with current process information (*p)
sa_define_event( )	sa_t *handle, event_t *event	define an event of type event_t and register it with the core
sa_generate_event( )	sa_t *handle, event_t *event, remote_addr *dst int sendrecv	generate an event of type event_t destined for remote address dst
sa_cache_event( )	event_t *event, char *loc	cache event data in location loc
sa_finalize()	sa_t *handle	close connection

TABLE 2
Primary Functionalities of the Sapphire Core Agents

Function	Arguments	Description
sam_init()	sa_t *q	initialize service
sam_get_event()	sa_t *q, event_t *event	receive an event from a process
sam_init_tstamp()	sa_t *q, tstamp *t	initialize the clock algorithm
sam_increment_tstamp ()	sa_t *q, tstamp *t	increment the local counter
sam_update_tstamp ()	sa_t *q, tstamp *local, int remote	update the clock based on the 'remote' value
sam_get_local_tstamp ()	sa_t *q, tstamp *t	get the local counter of the clock
sam_close_tstamp ()	sa_t *q, tstamp *t	terminate the clock algorithm
sam_save_event()	event_t *event, char *loc	save an event notification message
sam_send_event()	sa_t *q, event_t *event, remote_addr *dst	send an event to remote process dst
sam_recv_event()	sa_t *q, event_t *event, remote_addr *src	receive an event from a remote process src
sam_enqueue_event()	sa_t *q, event_t *event	enqueue an event to a FIFO queue
sam_dequeue_event()	sa_t *q, event_t *event	dequeue an event from a FIFO queue
sam_close()	sa_t *q	close service

Fig. 2 shows a schematic of the *Sapphire* core components. An end-host (E) is instrumented for individual applications (processes) as well as for the network stack. A client process in the end-host coordinates all the local probes of E and performs all necessary synchronization with the *Sapphire* core. The core of the framework (M) is a collection of several agents: An overall *Controller Agent* performs global coordination among all end-hosts, processes, and probes during the experiment. At the beginning of an experiment session, it reads in an experiment scenario file which contains runtime levels of the parameters being studied. An application defines a set of events of interest that are monitored for the runtime of the experiment. For portability, the event definitions can be exported out of the application using a markup language such as XML [17]. The core loads a runtime module to register these event definitions so that an appropriate data structure can be set up for them. An application-level event is generated by calling `sa_generate_event()` (see Table 1) and by passing an appropriate event structure to it.

3.3.5 *Synchronization of Instrumented* Sapphire *Clients*

The controller agent assigns a timestamp to an event based on the notion of *vector clocks* [16] to denote causality. In the original vector clock scheme, all event massages are tagged with a timestamp of size n, the number of processes in the instrumented application, to maintain the notion of vector time. Obviously, when n is large, the amount of timestamp data that has to be attached with each event message may become unacceptably large. To avoid this situation, we adopt the scheme proposed in [18], which reduces to a single scalar value instead of a vector of n elements. The core services include two additional agents: the *Host Controller Agent* performs end-host synchronization and correct installation of parameters at the beginning of an experiment session, whereas the *Network Device Controller Agent* is responsible for communication to network-level agents (see below) so that correct network-level (e.g., the network stack of the end-hosts) parameters are installed in the network under study. One of the functionalities of the core agents is to initialize a group of instrumented clients on behalf of the application. The initialization is performed using a three-phase protocol as shown in Fig. 4. First, clients C_1, C_2, and C_3 send requests for joining the core service. The server replies by sending acknowledgment messages along with the relative rank of each client (currently based on the arrival time of the initial request) in the process group. When either a new request has been received at the core or a preset time period has expired, the server sends one final message (*start*) to each of the clients indicating that a new experiment with its specified level of parameters is about to

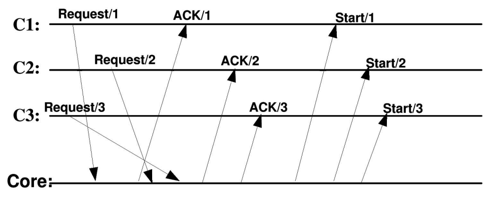

Fig. 4. Synchronization of instrumented *Sapphire* clients.

start. At this point, control is returned to the application for normal execution.

3.3.6 Network Instrumentation

Sapphire also includes a set of agents for configuring individual nodes of a network. These agents have also been used in our previous work in [1]. This is necessary for correct handling of the network-level parameters in the overall design of experiments. Note that such administrative access to individual nodes (i.e., routers, switches, etc.) of a network is not always possible. In such a case, an emulation for the target network can always be created in conjunction with instrumented application processes.

- ***Remote Agents.*** These are a set of distributed agents placed in the network, typically one at each intermediate router, and are controlled by a *Network Device Controller Agent* (below) which executes and keep track of the experiment steps. The Controller Agent sends periodic messages to the remote agents residing on the remote network elements according to the scenario of each experiment. It reads in a scenario file that defines the network configuration parameters (factors) and their values (levels).
- ***Network Device Controller Agent.*** It is a domain-level agent, responsible for configuring the traffic conditioning elements in a given network domain. It receives information from the Controller Agent in the core about each scenario to be performed for a set of experiments and sends messages to each of the routers under its administrative control with instructions for the set of traffic controls (router configuration) parameters to be installed for this scenario. These parameters are often the configuration parameters for setting up appropriate queueing disciplines, buffer sizes, traffic classes, and filters.
- ***Network Configuration Agents.*** These agents are placed on the individual routers in the network that participate in the experiments. They perform the actual configuration of the traffic control blocks on each router and are, therefore, device-dependent.

4 Design of Experiments and Statistical Modeling

Central to the model-building step are two statistical methods, namely, ANOVA and regression analysis. A detailed description of these methods can be found in [19], [20]. ANOVA partitions the variances (i.e., sums of squared deviations from the overall mean) among the factors and their interactions. From the percentages of variations, we can identify the most important factors. The factors with small or negligible contributions to the total variation of the output can be removed from the model. This process essentially eliminates the large number of parameters that the user may choose during the model-building step.

For any three factors (i.e., $k = 3$) denoted as A, B, and C with levels a, b, and c, and with r repetitions of each experiment, the response variable y can be written as a linear combination of the main effects and their interactions:

$$y_{ijkl} = \mu+\alpha_i+\beta_j+\epsilon_k+\gamma_{ABij} + \gamma_{ACik} + \gamma_{BCjk} + \gamma_{ABCijk} + e_{ijkl}$$
$$i = 1, \ldots, a; \quad j = 1, \ldots, b; \quad k = 1, \ldots, c; \quad l = 1, \ldots, r, \tag{1}$$

where y_{ijkl} = response in the lth repetition of experiment with factors A, B, and C at levels i, j, and k, respectively.

μ = mean response = $\bar{y}_{....}$.

α_i = effect of factor A at level $i = \bar{y}_{i...} - \mu$.

$\bar{y}_{i...}$ = average response at the ith level of A over all levels of other factors and repetitions.

β_j = effect of factor B at level $j = \bar{y}_{.j..} - \mu$.

γ_{ABij} = effect of the interaction between A and B at levels i and $j = \bar{y}_{ij..} - \alpha_i - \beta_j - \mu$.

γ_{ABCijk} = effect of the interaction between A, B, and C at levels i, j, and $k = \bar{y}_{ijk.} - \gamma_{ABij} - \gamma_{BCjk} - \gamma_{ACik} - \alpha_i - \beta_j - \epsilon_k - \mu$.

e_{ijkl} = error in the lth repetition at levels i, j, and k and so on.

Squaring both sides of the model in (1), and summing over all values of responses (cross-product terms cancel out), we get:

$$\sum_{ijkl} y^2_{ijkl} = abcr\mu^2 + bcr\sum_i \alpha^2_i + acr\sum_j \beta^2_j + abr\sum_k \epsilon^2_k$$
$$+ cr\sum_{ij}\gamma^2_{ij} + br\sum_{ik}\gamma^2_{ik} + ar\sum_{jk}\gamma^2_{jk}$$
$$+ r\sum_{ijk}\gamma^2_{ijk} + \sum_{ijkl} e^2_{ijkl},$$

which can be written as:

$$SSY = SS0 + SSA + SSB + SSC + SSAB + SSAC$$
$$+ SSBC + SSABC + SSE.$$

The total variation of y, denoted as the sum of square total or SST, is then:

$$SST = \sum_{ijkl}(y_{ijkl} - \mu)^2 = SSY - SS0.$$

The error in the kth repetition is $e_{ijkl} = y_{ijkl} - \bar{y}_{ijk}$, and the sum of squared errors (SSE) is equal to:

$$SSE = \sum_{ijk} e^2_{ijkl} = SST - SSA - SSB - SSC - SSAB$$
$$- SSAC - SSBC - SSABC.$$

The percentages of variation can be calculated as $100 \times (\frac{SSA}{SST})$ for the effect of factor A, $100 \times (\frac{SSAB}{SST})$ for the interaction between A and B, and so on. From these percentages of variations, we can identify the most important factors. The factors with small or negligible contributions to the total variation of the output can be removed from the model. Using ANOVA also allows us to calculate the mean square error (MSE) and compare it with the mean square of the effect of each factor to determine the significance of these effects against the experimental errors. This is called the *F-test* in

ANOVA and usually leads to the same conclusion if we compare the percentage of variations of the factors with those of errors. The linear model used in ANOVA is based on the following assumptions [19]: 1) the effects of the input factors and the errors are additive, 2) errors are identical and independent, normally-distributed random variables, and 3) errors have a constant standard deviation. Therefore, an important step after the ANOVA analysis is to validate the model by inspecting the results. This can be done using several "visual tests": 1) the scatter plot of the residuals (errors) versus the predicted response should not demonstrate any trend and 2) the normal quantile-quantile (Q-Q) plot of residuals should be approximately linear (after removing the outliers). The ANOVA method itself does not make any assumption about the nature of relationship between the input factors and the response variables [20]. However, in some cases where the residual errors are non-i.i.d., one can use the family of "mixed models" and bootstrapping [21] instead of the straightforward application of ANOVA. This can be easily accommodated in the software framework of *Sapphire*.

Once the most significant factors have been identified, we use regression models [20] to capture possible relationships between each output response variable and the most significant input factors. Basically, we use a variant of the multiple linear regression model called the *polynomial regression* for this purpose. The justification for this is that any continuous function can be expanded into piecewise polynomials given enough number of terms. We use a number of simple transformations [19] such as inverse, logarithmic, and square root, to capture nonlinearity in these relationships and convert them into linear ones. However, more complex transformations [19], [20] can be used for complex models. We choose the transformation that best satisfies the visual tests, minimizes the error percentage in ANOVA, and maximizes the coefficient of determination (R^2) in the regression model. For a stable and statistically valid application of the regression model, the confidence intervals must be smaller than the respective parameter values. To check the accuracy of the model, we divide the collected experimental data into training and verification sets and make sure that most of the parameters in the verification set are within 95 percent confidence interval of the values estimated from the training data sets. These results are presented in Section 5.

4.1 Factors and Factor Levels

The choice of factors depends entirely on the type of characterization, type of application, and network nodes being used for an experiment. Additionally, choosing levels of such parameters requires either knowledge of the traffic control components of the node or the use of vendor-supplied specifications. In our experiments, we used network routers based on the open-source Linux operating system and, therefore, we were able to access all the configuration parameters as well as their implementation details. The Linux traffic control module provides a flexible way to realize various traffic differentiation schemes with the help of a number of queueing disciplines and traffic conditioning modules. However, other network hardware can also be instrumented via serial access. The choice of application-level parameters depends entirely on the type of application under study. An example of a PCM audio streaming application is provided in Section 5 in which we study the effect of parameters such as frame size, PCM sampling rate, bits/sample, drop probability, and inter-packet jitter. It is also possible to define a set of higher-level application parameters such as throughput (an effective share of network bandwidth), latency (e2e delay), frame delay variation, reliability (frame loss, atomicity of operations), etc., for a particular service such as VoIP and, then, relate these to protocol-specific parameters (e.g., ITU and ETSI GSM speech codecs). For VoIP, one should also consider the number of simultaneous sessions as well as echo/round-trip delay levels. Similarly, the codec parameters of MPEG-2 (HDTV), H.323, can be related to the above high-level parameters for digital video transmission. We expect that such application-specific parameters that affect the e2e performance of an application are known before an approach such as *Sapphire* can be used.

An important issue is the choice of the levels for the factors in designing a set of experiments, which covers the entire range of the expected performance of an application. In some cases, intuition can reveal the relationships between the factors and the response variables. For example, if the configured service rate of the forwarding engines along the path is higher than the total input traffic rate, the output throughput converges to the input rate. On the other hand, if one considers delay, it is not obvious a priori as to how it will be affected by the relative difference between the input and configured rates. Our approach provides a generic solution for this issue, in which polynomial models are derived to represent such complex relationships. It may not be possible to cover all possible ranges and various modes of operation in this manner. However, the experiments should capture the expected ranges of operation for the application traffic running on a given network.

4.2 Factorial Design

A *full factorial design* utilizes every possible combination of all the factors [19] at all levels. If we have k factors, with the $i\text{th}$ factor having n_i levels, and each experiment is repeated r times, then the total number of experiments to perform will be $\prod_{i=1}^{k} n_i \times r$. One of the drawbacks of the full factorial analysis is, therefore, the large state space of the factors. To reduce the number and the execution time of experiments, we can use a combination of *factor clustering* and *iterative experimental design* techniques. In factor clustering, the input and configuration parameters having similar effect on the output are grouped together. This is similar to [22] in which the authors clustered 10 congestion and flow control algorithms in TCP Vegas into three groups, according to the three phases of the TCP protocol. We used this iterative design technique to investigate three distinct types of network provisioning for per-hop QoS [1]. Nevertheless, even with both experiment-reduction techniques used, the number of experiments can still be large and therefore, an automated approach such as the one used in *Sapphire* is very useful.

5 Experimental Results and Validation

As an example of generating interaction models via *Sapphire*, we present experimental results for an audio streaming application. Note that there are many other potential applications that can be instrumented with

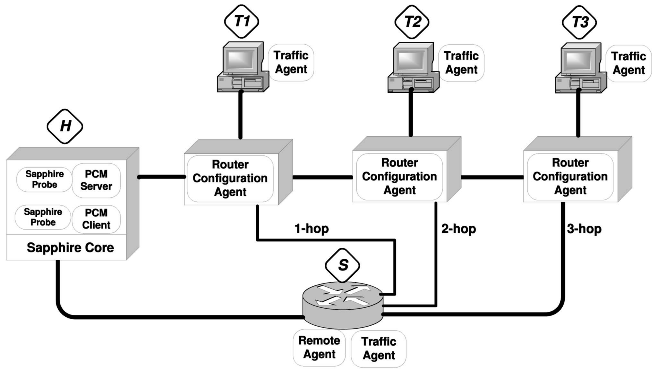

Fig. 5. Multiple-hop testbed for PCM audio application.

Sapphire, most notably, video streaming and other multimedia applications. We chose audio streaming as an example application due to its simplicity and small number of encoding parameters affecting its performance. Once we derive a model from the experiments, the resulting model is incorporated back in the application so that the server can adapt its streaming parameters (using this model) in response to varying traffic in the network. This is an example of using *Sapphire* for generating a model that can then be incorporated in a feedback loop. Through this set of experiments, we demonstrate the flexibility of the framework and the potential of using *Sapphire* to introduce model-based adaptivity to other networked applications.

Fig. 5 shows the network testbed used for our experiments. We use three routers, $R1$, $R2$, and $R3$, as the QoS-enabled network path for the audio application traffic. The routers support two classes of traffic: premium application (high priority) traffic and best-effort background (lower priority) traffic. The application traffic is always generated from host H and ends at the same host in a ring topology. Hosts $T1$ through $T3$ are used to generate the background (best-effort) traffic, that shares the links with the application premium traffic. The router S installs different traffic control configurations according to the experimental scenario. One can envision two different operating modes for this network configuration with respect to the premium input traffic: The network can be *overprovisioned* (OP) or *underprovisioned* (UP). In the OP mode, the total premium input traffic rate is less than its allocated rate, whereas, for the UP mode, it is larger than the allocated rate. We vary the background traffic parameters along with those for PCM encoding to generate the interaction model.

5.1 Model Generation for the PCM Audio Streaming Application

To generate the model, we instrument both the server and client processes of the PCM audio streaming application with *Sapphire* probes (see Fig. 5) so that specific application parameters can be monitored and configured as part of the design of experiments step. We also introduce a background traffic from hosts $T1$, $T2$, and $T3$ to determine the sensitivity of the audio traffic to other traffic in the network.

Table 3 lists the interaction model factors and their levels. The factor a_j is an application-level parameter in the PCM server that can selectively introduce per-packet jitter based on feedback received from the clients about the quality of the audio traffic. The factors a_s, a_b, and a_{ms} are related to the PCM encoding technique and refer to sampling rate (Hz), number of bits per sample, and number of milliseconds of audio per packet, respectively. The first two parameters provide the range and quality of audio content, e.g., CD-quality audio requires encoding at 44.1 KHz and 16 bits for levels. The MPEG layer 3 encoding (MP3) requires even a larger number of bits (128 bits) per sample. As shown in Table 3, the audio quality is varied from telephony (8KHz/8bit) to CD-quality (44KHz/32bit stereo). Intuitively, both the sampling rate and the number of bits per sample affect the bandwidth requirement of an audio application—we verify this as well as model the relationship as part of the model generation step. The rest of the factors are for the background traffic from hosts $T1$, $T2$, and $T3$. The allocated rate for the application traffic is 3.33 Mbits/sec.

TABLE 3
Factors for PCM Audio Streaming Experiment

Factor	Symbol	Levels
Sampling rate (KHz)	a_s	8 10 25 44
Number of bytes/sample	a_b	1 2 3 4
Audio per packet (msec)	a_{ms}	10 50 100
Average jitter	a_j	0 1 10
Background packet size (Bytes)	b_{pkt}	200 1000 1470
Background rate (Kbits/sec)	b_r	50 200 1000

Table 4 shows the percentage variations of the effects due to the factors and their interactions. As expected, the measured bandwidth of the application depends entirely on the sampling rate (a_s) and the number of bits per sample (a_b). Since the network is over-provisioned, the response variables (*Throughput* or *BW*, *Delay*, and *Jitter*[3]) are not affected by the background traffic in the same aggregate.[4] The delay is entirely dependent on the size of the audio packets, as given by the parameter a_{ms}. For a given sampling rate and quantization level, the size of the PCM-encoded packets is linearly proportional to the milliseconds of audio encoded in them, and therefore, the experimental results for delay are valid. The characterization for jitter, however, shows strong first-order interactions between application-induced jitter (a_j) and a_{ms}. The error analysis and scatter plots for the response variables confirm the linearity of the ANOVA model (e.g., see Figs. 6 and 7 for jitter). We also confirmed that the errors are normally-distributed. There is no trend in the residual versus predicted response scatter plot. Moreover, the errors are normal since the normal Q-Q plot is mostly linear. The tests for throughput and the delay show linearity as well.

Next, we calculate the polynomial regression models for the response variables in terms of the most significant factors as follows.

3. Jitter is calculated $J = J + (|D(i-1,i)| - J)/16$, where $D(i-1,i)$ is delay variation between packets i and $(i-1)$.

4. There is a small effect of the background traffic packet size on delay as can be expected, but it is less than 2 percent and, hence, omitted in the table.

TABLE 4
Anova Results for PCM Audio Experiment

Response Var	Transformation	a_s	a_{ms}	a_b	a_j	(a_s,a_b)	(a_{ms},a_j)	Error
BW	linear	59%	≈ 0%	26%	≈ 0%	11%	≈ 0%	0%
Delay	linear	≈ 0%	99%	≈ 0%	≈ 0%	≈ 0%	≈ 0%	0%
Jitter	linear	≈ 0%	11%	≈ 0%	57%	≈ 0%	18%	3%

$$\begin{aligned} BW &= 18386.51 - 31173.72a_b - 1.24a_s + 12.04a_ba_s \\ &\quad + 10877.14a_ba_b - 1.20a_sa_b^2 - 1554.36a_b^3 \\ &\quad + 0.1542a_sa_b^3, \\ Delay &= -0.0002 + 0.0010a_{ms}, \\ Jitter &= 0.35 - 0.0036a_{ms} + 0.2a_j - 0.0077a_{ms}a_j \\ &\quad + 0.03a_j^2 - 0.0001a_{ms}a_j^2. \end{aligned} \tag{2}$$

The surface plots for BW and jitter corresponding to this model, along with their 95 percent confidence intervals, are shown in Figs. 8 and 9. Note that confidence intervals are intimately tied to sample size. The larger the sample size, the narrower the confidence interval. Therefore, one should have a reasonable operating range of the factors to sample from while developing the interaction model.

Next, we extend the testbed to study the effect of multiple hops as well as multiple background traffic sources on the audio streaming traffic. The testbed allows the number of hops (n_{hop}) to be varied from 1 to 3 as an additional factor. Due to lack of space, we only show the results of the ANOVA analysis in Table 5. The regression models for bandwidth, delay, and jitter for multiple hops include the previously significant factors as well as a number of higher-order interactions with a maximum polynomial degree of 4. It is important to observe that bandwidth is independent of the number of hops (as can be expected), but depends on sampling rate, bits per sample, and the background traffic characteristics. However, both delay and jitter are now affected by the number of hops, as well as packet sizes.

5.2 Model-Based Adaptation of PCM Audio Streaming Application

We now use these models as an example of providing application-level adaptation. We modify the audio streaming application to include server-side adaptation based on periodic sampling of the response variables and the introduction of a feedback loop between the server and the client. For simplicity, we only adapt the sampling rate (a_s) and the audio per packet (a_{ms}) parameters while keeping the other parameters fixed. In general, one chooses a set of parameters to be adapted based on the percentage variations in the ANOVA table. There are a number of approaches a server can adapt based on periodic feedback. Our primary goal in this section is to demonstrate the usefulness of the *Sapphire*-generated models in identifying

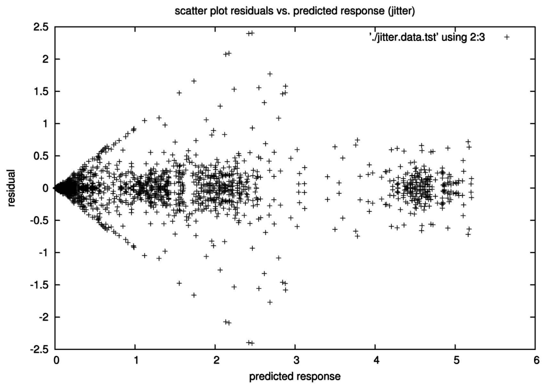

Fig. 6. Residuals versus predicted response for jitter.

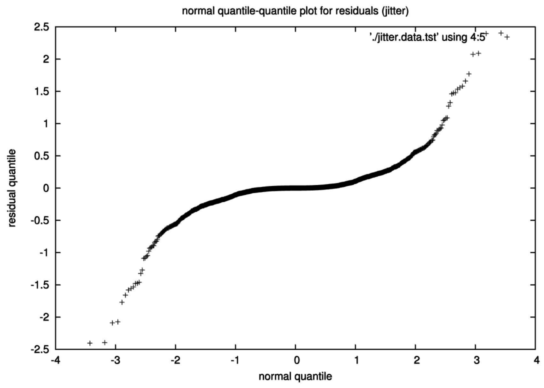

Fig. 7. Normal Q-Q plot for jitter residuals.

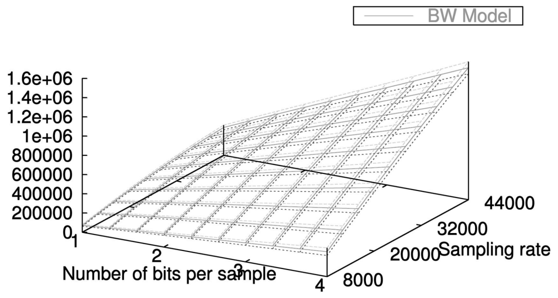

Fig. 8. PCM bandwidth model and 95 percent confidence interval.

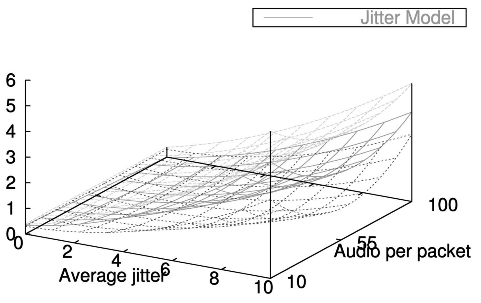

Fig. 9. PCM jitter model and 95 percent confidence interval.

TABLE 5
ANOVA Results for PCM Audio Streaming over Multiple Hops

Response	a_s	a_{ms}	a_b	b_{pkt}	n_{hop}	(a_s,a_b)	(a_{ms},b_{pkt})	(a_{ms},b_{pkt},n_{hop})	(a_s,a_{ms},a_b,b_{pkt})	*Error*
BW	43.1%	$\approx$ 0%	28.2%	4.1%	$\approx$ 0%	9.1%	$\approx$ 0%	$\approx$ 0%	$\approx$ 0%	0.21%
Delay	$\approx$ 0%	58.5%	$\approx$ 0%	$\approx$ 0%	$\approx$ 10.2%	$\approx$ 0%	8.1%	4.2%	$\approx$ 0%	1.6%
Jitter	$\approx$ 0%	10.9%	$\approx$ 0%	4.3%	$\approx$ 0%	$\approx$ 0%	$\approx$ 0%	$\approx$ 0%	5.2%	2.7%

the most significant parameters and in providing the interaction model. Once the models are available, one can develop efficient adaptation frameworks for an application depending on the levels of QoS guarantees needed. The following are two simple mechanisms for the PCM audio adaptation that were easy for us to set up in the testbed environment. For other application domains such as video-on-demand or video-streaming, more sophisticated approaches are available in the literature.

- *Greedy approach*. In this approach, the sender transmits longer audio content per packet (twice the previous size) as long as the delay is below a given threshold. When the delay threshold is reached, the sender decreases a_{ms} to half the previous value and starts the process again. Similarly, for bandwidth, it increases the sampling rate until a threshold (set by a specified rate for the audio traffic) is reached, at which point the sender throttles itself. It is obvious that the Greedy approach, by design, will be oscillatory in terms of delays experienced by the application and, therefore, may not be practical in real-life deployments.
- *Baseline approach*. In this approach, the average delay is held constant while increasing the bandwidth slowly up to the specified limit for the application, provided there is enough bandwidth available in the network path. This approach enables more stable adaptation since the bandwidth changes are gradual.

Figs. 10 and 11 show measured values of bandwidth and delay for these two approaches, respectively. Note that, for each value of bandwidth and delay, the corresponding values of the application parameters, a_s and a_{ms}, are calculated from the models that were generated during the model-generation phase of *Sapphire*. These computed values for the sampling rate and audio per packet are shown in Figs. 12 and 13, respectively, during a typical audio streaming session. By comparing the two approaches, the Baseline adaptation appears to be a smoother process than the Greedy approach. As mentioned above, this is to be expected since the baseline adaptation slowly increases the bandwidth while maintaining a constant e2e delay for the audio traffic. Many other

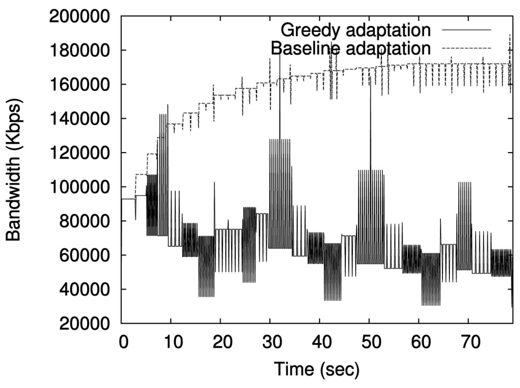

Fig. 10. Bandwidth response by adapting the sampling rate (a_s).

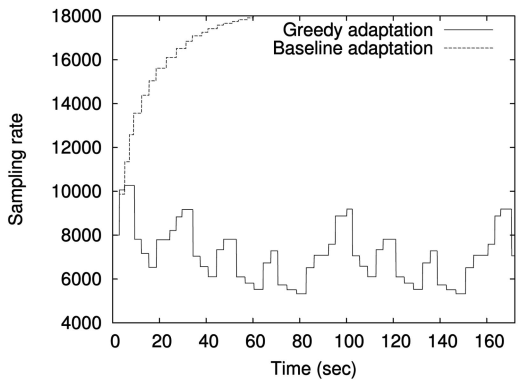

Fig. 12. Adaptation of sampling rate (a_s).

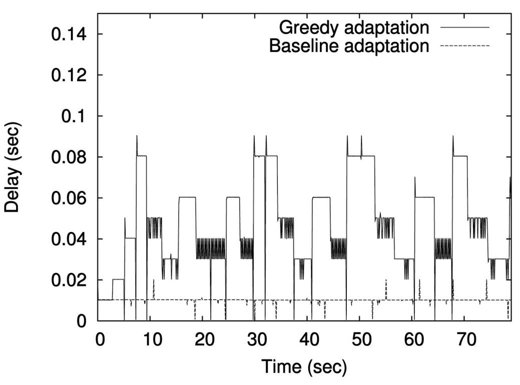

Fig. 11. Delay response by adapting the audio per packet (a_{ms}).

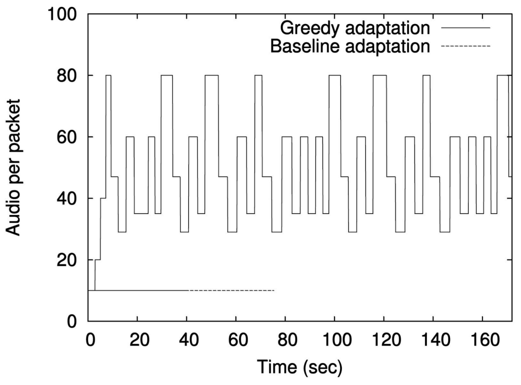

Fig. 13. Adaptation of audio per packet (a_{ms}).

adaptation models can be developed depending on the type of application traffic and user interactions in the given network.

We provide this section on model-based adaptation as an example of taking an existing application, instrumenting it with *Sapphire* probes, generating an interaction model, and, finally, using this model in the application itself to provide adaptation. This flexibility of nearly-automated model generation for existing networked applications is a key feature of our work.

6 Related Work

There has been a renewed interest in the area of application adaptation and control, especially based on feedback from runtime middleware [23], [24]. While classical control-theoretic approaches are increasingly being applied to control performance of Web-based services, multimedia applications currently use a variety of ad hoc methods to adapt to changing network and server-side conditions. *Sapphire* provides a robust model for such adaptation. The design goals of *Sapphire* are closely related to the adaptation framework developed by Chang and Karamcheti [25] for automatic configuration and runtime adaptation of distributed applications. The authors of [25] present two novel components: a tunability interface for providing alternate configurations of runtime application parameters and a virtual execution environment emulating application execution under diverse resource availability. The specification for application tunability requires development of an application interaction model (called application profiles). The authors use the virtual execution environment running on top of a static distributed system to emulate a wide range of resource availability scenarios and, thus, generate the application profiles. Our framework achieves this as part of the ANOVA analysis. Note that the most important parameters for a networked application in [25] need to be known a priori for the tunability interface, whereas, in *Sapphire*, these are identified as part of the profiling step.

The USA (User Service Assistant) system [26] is a QoS management framework that provides assistance in negotiating the required levels of services, provided these are known for an application. It does not estimate the resource requirements and only provides information about the session quality to enable users make informed decisions. By design, such a framework requires the users to know the application-network interaction model. Abdelzaher [27] presented the automated profiling subsystem approach that can estimate application resource requirements by applying recursive least-squares and adapt the application transparently to the resouce capacity of the underlying platform. The key aspect of this study is profiling server-side applications, such as Web servers. In [28], the authors describe the design and implementation of ControlWare, a middleware QoS-control architecture based on well-established linear feedback control theory. The framework is best suited for server applications and a system model for the server is assumed to be available to the controller.

The BRISK application instrumentation system [29] is targeted for distributed real-time systems. The system writes all records to a shared memory in the network. The information is then sent to an instrumentation manager for a graphical display of all events. The main limitation of this system is the use of a global clock—while BRISK employs some degree of global clock synchronization, global clocks are inherently problematic due to their inability to accurately reconstruct a global state. Taylor et al. describe the Prophesy system [30], which also provides internal application instrumentation. An interesting aspect of Prophesy is that the user can automatically instrument via a set of tools provided in Prophesy. However, this system is not designed to study application-level protocols since it was developed for examining the performance of a specific algorithm. Nonetheless, the idea of automatic instrumentation is attractive for large-scale distributed applications. The authors of [31] present compiler techniques and code extensions in which the compiler attempts to predict performance for an application using a task graph approach. Gunter et al. [32] describe the NetLogger system for the collection of events from instrumented applications. However, NetLogger shares the same problem as BRISK, in that it also requires a global clock. Another problem is that no attempt is made in NetLogger to provide any clock synchronization in the presence of skewed clocks and, therefore, the system is useful primarily for event logging. The Emulab [33] testbed allows coordinated experiments with distributed and emulated resources, and it may be possible to use the framework for performance characterization with suitable instrumentation. In IQ-RUDP, described by He and Schwan [34], the transport-layer itself is instrumented, and by using various coordination measures, the application and transport protocols adapt their own behaviors. The approach assumes that an a priori model of the application at various performance levels is available. Malan and Jahanian [35] present Windmill, a passive network protocol performance measurement tool. Windmill allows the reconstruction of application-level network protocols and the underlying protocol layers' events. The idea of probing multiple protocol layers in *Sapphire* is similar to network-layer probing in Windmill, but there are important differences between the two. In *Sapphire*, one can perform coordinated experiments at both application and network layers and develop fine-grained statistical models of application performance. With support for both active and passive probing, *Sapphire* is used as a framework for application adaptation similar to IQ-RUDP. *Sapphire* also borrows ideas from large-scale event correlation techniques. The goal of event correlation is to relate events back to their probable cause. The authors of [36] describe a system called InCharge that can be used to isolate and handle network faults in near real time. The system attempts to automatically reconfigure itself in response to a dynamic network topology. It divides the network into domains, each managed by a domain manager. Fault injection systems are often used to test the implementation of distributed protocols, and as such, can also be used to test the effectiveness of event correlation. Dawson and Jahanian [37] describe Orchestra, a portable fault injection environment that attempts to minimize intrusiveness on target protocols. However, the fine-grained performance of any application-level protocol cannot be understood without explicit instrumentation. Chandra et al. [38] describe Loki, a state-based fault injection system capable of injecting faults with only a partial view of the global state of a distributed system.

7 Conclusion

We have presented *Sapphire*, a framework for the generation of application-network interaction models within a given networked environment. The framework constructs statistical models of application performance in the target network environment by performing a set of designed experiments and by identifying the parameters that have the most significant effect on the performance. The design of *Sapphire* allows emulation of the target network so that alternative configurations of network-level parameters can be analyzed before deploying QoS-sensitive services. As an example, we have used *Sapphire* successfully to characterize and model an audio-streaming application. The resulting model can be incorporated, as we have presented, in the application to enable adaptation with respect to changes in the most important parameters affecting its performance. *Sapphire* can be used in a number of contexts such as feedback control of application-level parameters in response to changing network conditions, and tuning application and network-level parameters to meet service-level guarantees. The *Sapphire* core and probes presented in this paper can be extended to more complex systems, such as applications deployed in an enterprise network to characterize their behavior under different resource-provisioning scenarios. More sophisticated statistical methods such as iterative system identification and control can also be incorporated to generate robust adaptation models.

Acknowledgments

The authors thank the anonymous reviewers for their valuable suggestions and improvements to the paper.

References

[1] M. El-Gendy, A. Bose, H. Wang, and K.G. Shin, "Statistical Characterization of Per-Node QoS," *Proc. 11th IEEE Int'l Workshop Quality of Service (IWQoS),* June 2003.
[2] Fidelia NetVigil Version 3.5, http://www.fidelia.com, 2006.
[3] "Mercury Interactive Topaz Application Performance Management," http://www.mercuryinteractive.com, 2006.
[4] L.A. DaSilva, "QoS Mapping Along the Protocol Stack—Discussion and Preliminary Results," *Proc. IEEE Int'l Conf. Comm. (ICC),* June 2000.
[5] K. Nahrstedt and J.M. Smith, "An Application-Driven Approach to Networked Multimedia Systems," *Proc. 18th Ann. Conf. Local Area Computer Networks,* pp. 361-368, 1993.
[6] J. Huard and A. Lazar, "On QoS Mapping in Multimedia Networks," *Proc. 21st IEEE Int'l Computer Software and Application Conf. (COMPSAC),* 1997.
[7] B.H. Liu, P. Ray, and S. Jha, "Mapping Distributed Application SLA to Network QoS Parameters," *Proc. 10th IEEE Int'l Conf. Telecomm. (ICT),* 2003.
[8] J. Bolot and A. Vega-Garcia, "Control Mechanisms for Packet Audio in the Internet," *Proc. INFOCOM,* pp. 232-239, 1996.
[9] T. Urabe, H. Afzal, G. Ho, P. Pancha, and M. El Zarki, "MPEGTool: An X-Window-Based MPEG Encoder and Statistics Tool," *Proc. ACM Multimedia Conf.,* pp. 259-266, 1993.
[10] S. Blake, D. Black, M. Carlson, E. Davis, Z. Wang, and W. Weiss, "An Architecture for Differentiated Services," *RFC 2475,* IETF, Dec. 1998.
[11] K. Nichols, V. Jacobson, and L. Zhang, "A Two-Bit Differentiated Services Architecture for the Internet," *RFC 2638,* IETF, July 1999.
[12] Schulzrinne, Casner, Frederick, Jacobson, "RTP: A Transport Protocol for Real-Time Applications," *IETF RFC 3550, Network Working Group,* 2003.
[13] N. Kitawaki, "Perceptual QoS Assessment Methodologies for Coded Speech in Networks," *Proc. IEEE Workshop Speech Coding,* 2002.
[14] F. Hammer, P. Reichl, and T. Ziegler, "Where Packet Traces Meet Speech Samples: An Instrumental Approach to Perceptual QoS Evaluation of VOIP," *Proc. 12th IEEE Int'l Workshop Quality of Service (IWQOS),* pp. 273-280, 2004.
[15] B.C. Neuman and S.S. Augart, "Prospero: A Base for Building Information Infrastructure, " *Proc. Int'l Networking Conf. Internet Soc. (INET '93),* 1993.
[16] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, no. 7, 1978.
[17] Extensible Markup Language (XML), http://www.w3.org/XML/, 2006.
[18] R. Schwarz and F. Mattern, "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail," *Distributed Computing,* vol. 7, no. 3, pp. 149-174, 1994.
[19] R. Jain, *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, Inc., 1991.
[20] J. Neter, W. Wasserman, and M. Kutner, *Applied Linear Statistical Models: Regression, Analysis of Variance, and Experimental Designs.* R.D. Irwin, Inc., 1985.
[21] R.Y. Liu, "Bootstrap Procedures under Some Non-i.i.d. Models," *Annals of Statistics,* vol. 16, 1998.
[22] U. Hengartner, J. Bolliger, and T. Gross, "TCP Vegas Revisited," *Proc. IEEE INFOCOM '00,* Mar. 2000.
[23] B. Li and K. Nahrstedt, "A Control-Based Middleware Framework for Quality of Service Adaptations," *IEEE J. Selected Areas in Comm.,* Sept. 1999.
[24] L. Abeni and G. Buttazzo, "Hierarchical QoS Management for Time Sensitive Applications," *Proc. IEEE Real-Time Technology and Applications Symp.,* 2001.
[25] F. Chang and V. Karamcheti, "Automatic Configuration and Run-Time Adaptation of Distributed Applications," *Proc. Ninth IEEE Int'l Symp. High Performance Distributed Computing (HPDC),* 2000.
[26] B. Landfeldt, A. Seneviratne, and C. Diot, "User Service Assistant: An End-to-End Reactive QoS Architecture," *Proc. Sixth IEEE Int'l Workshop Quality of Service (IWQoS),* May 1998.
[27] T.F. Abdelzaher, "An Automated Profiling Subsystem for QoS-Aware Services," *Proc. IEEE Real Time Technology and Applications Symp.,* pp. 208-217, 2000.
[28] R. Zhang, C. Lu, T. Abdelzaher, and J. Stankovic, "Controlware: A Middleware Architecture for Feedback Control of Software Performance," July 2002.
[29] M.W. Mutka, A. Bakic, and D.T. Rover, "BRISK: A Portable and Flexible Distributed Instrumentation System," *Proc. 13th Int'l Parallel Processing Symp.,* 1999.
[30] V. Taylor, X. Wu, and R. Stevens, "Design and Implementation of Prophesy Automatic Instrumentation and Data Entry System," *Proc. 13th IASTED Int'l Conf. Parallel and Distributed Computing and Systems (PDCS '01),* 2001.
[31] V. Adve, V.V. Lam, and B. Ensink, "Language and Compiler Support for Adaptive Distributed Applications," *Proc. ACM SIGPLAN Workshop Languages, Compilers, and Tools for Embedded Systems,* 2001.
[32] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer, "Dynamic Monitoring of High-Performance Distributed Applications," *Proc. 11th IEEE Int'l Symp. High Performance Distributed Computing (ISHPDC),* 2002.
[33] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An Integrated Experimental Environment for Distributed Systems and Networks," *Proc. Fifth Symp. Operating Systems Design and Implementation,* pp. 255-270, Dec. 2002.
[34] Q. He and K. Schwan, "IQ-RUDP: Coordinating Application Adaptation with Network Transport," *Proc. 11th IEEE Int'l Symp. High Performance Distributed Computing,* 2002.
[35] G.R. Malan and F. Jahanian, "An Extensible Probe Architecture for Network Protocol Performance Measurement," *Proc. ACM SIGCOMM,* 1998.
[36] S. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie, "High Speed and Robust Event Correlation," *IEEE Comm. Magazine,* vol. 34, 1996.
[37] S. Dawson and F. Jahanian, "Probing and Fault Injection of Protocol Implementations," *Proc. Int'l Conf. Distributed Computer Systems,* May 1995.
[38] R. Chandra, R.M. Lefever, M. Cukier, and W.H. Sanders, "Loki: A State-Driven Fault Injector for Distributed Systems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* 2000.

Abhijit Bose (M'03) is a PhD candidate in computer science and engineering at The University of Michigan, Ann Arbor. He is a research scientist and associate director of Michigan Grid Research and Infrastructure Development at the University of Michigan, Ann Arbor. His research interests are network security, distributed resource scheduling and computational modeling of biological processes. He has published more than 20 technical papers in international journals and conference proceedings. His research is funded by the US National Science Foundation (NSF), the University of Michigan, and industry. He is a recipient of the best paper award (2003, IEEE IWQoS), a TI Fellowship (1995), and a National Merit Scholarship (1984-1986).

Mohamed El Gendy (S'94-M'03) received the BSc degree in electrical engineering and the MSc degree in computer engineering from Cairo University, Egypt, in 1995 and 1998, respectively. He received the PhD degree from the Department of Electrical Engineering and Computer Science at the University of Michigan, Ann Arbor, in September 2005. He has been working at Cisco Systems, San Jose, California, in the 7600 router group since October 2005. While at the University of Michigan, he worked on QoS modeling and control for real-time applications over the Internet and his thesis described several ways of realizing a model-based QoS control framework. His research interests are computer networks, operating systems, network QoS, real-time systems, and distributed systems. Dr. El Gendy is a winner of the best student paper award from the International Workshop on Quality of Service (IWQoS) in 2003 and the best master thesis award from Cairo University in 1999.

Kang G. Shin (S'75-M'78-SM'83-F'92) received the BS degree in electronics engineering from Seoul National University, Seoul, Korea, in 1970, and the MS and PhD degrees in electrical engineering from Cornell University, Ithaca, New York, in 1976 and 1978, respectively. He is the Kevin and Nancy O'Connor Professor of Computer Science and the founding director of the Real-Time Computing Laboratory in the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor. He has supervised the completion of 55 PhD theses and authored or coauthored approximately 630 technical papers. His current research focuses on QoS-sensitive networking and computing, as well as on embedded real-time OS, middleware, and applications, all with an emphasis on timeliness and dependability. Dr. Shin is a fellow of the ACM and a member of the Korean Academy of Engineering.