Narnia: A Virtual Machine for Multimedia Communication Services

Mauricio Cortes
Bell Labs, Lucent Technologies
101 Crawfords Corner Rd
Holmdel, NJ 07733 USA
+1 732 949 9088
mcortes@dnrc.bell-labs.com

J. Robert Ensor
Bell Labs, Lucent Technologies
101 Crawfords Corner Rd
Holmdel, NJ 07733 USA
+1 732 949 7533
jre@dnrc.bell-labs.com

Abstract

Narnia is middleware that helps programmers build multimedia communication services. This middleware uses a collection of familiar programming abstractions—including events, event handlers, resources, sessions, and user roles—to provide both a service development environment and a service execution environment. The run-time environment—the Narnia virtual machine—provides means for creating, transmitting, and consuming events as well as means for creating, loading, and executing sessions and event handlers.

The Narnia middleware has served as the development and execution platform for a few applications—including an audio/video chat application and a SIP proxy simulator. The paper reports on how well these applications handle various user request workloads. These early system performance measurements indicate the effectiveness of Narnia in supporting development and deployment of scalable network multimedia services.

Keywords
Distributed multimedia applications, events, sessions, roles, performance measurements, Session Initiation Protocol (SIP), enhanced chat room

1. Introduction

Narnia is middleware designed to aid the development and execution of distributed applications. In particular, it helps create applications that can handle high request loads and can meet strict reliability requirements. To support such applications, we gave our middleware a distinctive set of characteristics: a highly efficient event-based communication platform, a transport-independent event addressing scheme, direct support for code updates, and familiar, but extensible programming abstractions.

To handle high request loads, a server must be able to retrieve and act upon messages quickly—that is, the server must be built upon a high performance communication system. Narnia provides an event-based communication platform. The events are semi-structured, with fixed header formats and variable body lengths, making them easy to write and read. Furthermore, the events are delivered to precisely specified event handlers; Narnia itself has no event dispatchers (though applications can implement them). Narnia invokes one event handler to process each event, according to the event’s priority and user roles (an access control mechanism).

To be portable, a server must be built upon a network agnostic communication platform. Narnia event addresses are network independent; they correspond to resources, which are similar to objects found in some programming languages. Thus, Narnia addresses can be mapped to various underlying transport network addresses. While it presently uses IP networks, Narnia could make use of other packet networks, e.g., ATM.

To be highly available (an aspect of reliability), a server must recover from its faults and undergo upgrades with little down time. Narnia helps support server availability by allowing code changes during system run-time. More specifically, event handlers can be loaded and unloaded during system execution.

Narnia supports program development through its primary programming abstractions—events, event handlers, resources, sessions, and user roles—and its run-time environment, which is based upon the same set of abstractions. These abstractions are extensible using C++ programming language. Applications are built as collections of sessions. Sessions provide rendezvous mechanisms through which users can share resources. These resources can represent a wide variety of computer or network entities such as video server, audio bridges, network management servers, etc.
We have built several Narnia-based applications including a session directory server, a multimedia chat room, a computer load-monitoring server, and a SIP proxy simulator. This paper briefly describes the chat room application and the SIP proxy simulator, and presents measurements of their performance under various workloads.

2. Programming abstractions

Several familiar programming abstractions are collected together in the Narnia middleware. The Narnia abstractions provide a base for the development of control applications running on NVM- our execution environment. A description of these abstractions follows.

2.1 Events

Informally, we say that Narnia-based applications communicate through exchange of events. More precisely speaking, Narnia-based applications send and receive Narnia messages. A Narnia message is a serialized representation of part of a Narnia event object, and the Narnia virtual machine serializes and deserializes each message at its sending and receiving sites, respectively.

As shown in the Figure 1, a Narnia message is composed of a header and a payload. The header has a fixed length of 48 bytes. It contains a sender address, which identifies the resource that generated the event, and a destination address, which identifies the resource that should process or consume the event. The source and destination addresses occupy half of the event header’s space.

Narnia’s resource address space supports more than $2^{96}$ (or $10^{28}$) source and destination addresses. (By way of comparison, the IP Version 4 address space is $2^{32}$.) A Narnia address is composed of an application identifier, a version number, an instance identifier, a session identifier, and a resource identifier. This address uniquely defines a Narnia resource.

One specialized Narnia application is the session and resource directory (SRD). This application keeps track of session and resource addresses for a collection of applications. When an application creates a public resource, it registers the resource address with its serving SRD so that other applications can find and post events to the new resource. The SRD maintains correspondence between each session/resource and its IP address and port number. When an application needs to send an event to (a session or resource within) an application instance, it queries the appropriate SRD to retrieve the application instance’s IP address and port. The sending application can then create a connection to the other application. This new connection will allow these applications to exchange one or more events. Since an application can contain many sessions and resources, the SRD maintains a many-one mapping between session/resource addresses and IP address: port number (of an application).

In addition to the source and destination addresses, the Narnia message header contains the following attributes: a device identifier, an event identifier, an event processing priority, a creation timestamp, and the length of the message payload.

The device identifier is currently derived from the MAC address of the machine on which the message originates. It uniquely identifies the physical machine that originated the message.

The event type field contains a number that is unique within a resource abstraction. Therefore, the concatenation of the destination address (12 bytes) and the event type (2 bytes) form an application unique event identifier. These 14 bytes determine the event handler that will be invoked to process a particular event.

An event message includes a suggested processing priority, which the destination application can take into account if it receives two or more events at about the same time. Our virtual machine maintains an event priority queue for each session, but does not preempt the execution of low priority events on behalf of a high priority event.

The payload length indicates the number of bytes included in the message payload. The payload is an application-level byte stream; it is not interpreted by any Narnia internal component. Currently, Narnia messages are transmitted using TCP. We are currently working on a new version that includes support for other transport protocols such as UDP or SCTP.

![Figure 1. Narnia message format](image)
2.2 Event handlers

An event handler specifies how an incoming event is processed. Each event handler is associated with a single event type. When an event arrives to a Narnia application, the NVM will invoke the event handler associated with the type, i.e., the destination address and the event identifier fields, of the incoming event.

Programmers can create new event handlers by extending the Narnia event handler class. Each new event handler class should include a new `executeHandler` method. This method receives the incoming event as one of its parameters. The `executeHandler` method can access the event’s header and payload. In particular, this method should check for the payload syntax, and implements the semantics of the incoming event.

Programmers can develop one or more event handlers to process a particular event type. However, one and only one event handler per event type can be active within a resource of an application instance.

Event handlers can be replaced at any time in a Narnia application. Programmers can build configuration applications that can request to a Narnia application to replace its event handlers. This feature enables the construction of highly available applications that can be upgraded without service interruptions. For example, some telecommunication applications are required to have 5.25 minutes or less of downtime a year (i.e. 99.999% available). Instead of stopping and restarting a critical application, operators can replace the event handlers at runtime.

2.3. Shared Resources

Narnia shared resources implement the control mechanisms to manage information in multi-user environments. One or more users can create, access, update, and destroy shared resources. Narnia allows programmers to define the mechanisms that control how this information is being shared. These control mechanisms are implemented by one or more event handlers.

For example, the programming schedule of a TV channel can be controlled by a Narnia shared resource. All users share the channel schedule. While all users can browse the programming schedule, only a small number of privileged users can change its content. A Narnia shared resource specifies these rules to share the channel schedule.

A Narnia shared resource comprises a unique identifier, (control) state, and a set of methods represented by event handlers. The control state is stored in a list of name-value pairs. The resource state is available to all event handlers defined in the shared resource. Access to this list is thread-safe. Thus, two or more event handler instances can update the shared resource state without compromising its integrity.

Every event handler instance is associated with exactly one event identifier, i.e. destination address + event type. Thus, each incoming event triggers one and only one event handler.

2.4. Sessions

Conceptually, a session is a rendezvous point where two or more users can exchange information in a shared context. Thus, we treat a session as a specialized shared resource—one that has been augmented with session state and control mechanisms to coordinate the exchange of information between session users.

Our session class extends the resource class described above. In addition to the base resource attributes, e.g., the lists of name-value pairs and event handlers, the session abstraction contains a list of shared resources, a list of participants, and list of user roles. Therefore, a session object requires additional event handlers to administer these new lists. Programmers can develop handlers to control how users can join, query, and leave the session, users can add and delete resources to a session, and users can add, modify, and delete user roles.

Analogous to processes running in an operating system, a session object is independent of other session objects. For example, the event handlers loaded in a session cannot update information stored in other sessions. Similarly, events provide the inter-process communication mechanisms to exchange information between two Narnia sessions. Narnia supports both inter- and intra-session communications. The former allows two sessions running in different machines or processes to send events using a TCP socket. Intra-session communication allows the exchange of events between two sessions running in the same process. Narnia has been optimized for intra-session communication bypassing the TCP stack and posting the events directly to the destination resource.

A session’s state can be saved in stable storage. This information can be used to restart the session later. The default persistence algorithm is to store all the session information including the list of devices, event handlers, shared resources, and lists of name-value pairs. However, some sessions do not need to store all this information. Programmers can override the default behavior and store a subset of session information.

2.5. User roles

The role abstraction provides an access control mechanism to define user privileges in multi-user
applications. In these applications, a group of users might have restricted access to some shared information, while another group of users might have full access to all information.

Formally, a role is defined as a set of event identifier. A user assigned to a role can send events with any event identifier included in the role definition. Users can assume more than one role. Narnia calculates a new set of event identifiers for this user by taking the union of each role.

The role abstraction helps programmers define these user roles in the context of a Narnia session. Each session can support up to 16 different user roles. Once the role definitions are loaded in the session, the NVM checks every incoming event before triggering the corresponding event handler.

3. NARNIA Virtual Machine

The Narnia virtual machine is the runtime environment that supports Narnia applications. This virtual machine includes the following low-level abstractions: event readers, a session manager, a connection manager, a memory manager, an event handler loader, and a module loader. It provides each session with an event priority queue and a pool of execution threads.

3.1. Event reader

The event reader is responsible for getting incoming events from a remote application to the appropriate session in the local application. Each reader retrieves events from one or more sockets.

The expected event rate determines the number of event readers for each application. Our experience with Sun Solaris machines has been to have one event reader for every 4000 events per second.

3.2. Session manager

The session manager is responsible for creating and destroying sessions. This manager maintains control information about every session running in an application instance. A single session manager is present in an application instance.

The session manager is implemented as a special session. Like other sessions, the session manager has a list of event handlers. This list includes event handlers to create and destroy sessions. Therefore, the creation of new sessions can be requested by any Narnia application. Local or remote application instances can send a Narnia event to create a session. It will be up to the session creation event handler to authenticate, authorize, and create the session.

We have developed default event handlers that create and destroy sessions upon receiving the corresponding event. Programmers can override these event handlers to implement the required session management policy.

3.3. Connection manager

The connection manager handles all the incoming and outgoing connections. A single connection manager is present in an application instance. This manager waits for clients to connect to a control port. Currently, the NVM only supports TCP ports, thus the manager waits to accept a new socket connection. Currently, we are developing a solution to support other transport protocols such as UDP.

This manager maintains information of all the opened connections. The manager gets report from other Narnia entities whenever any connection is broken. In this case, the connection manager frees the resources assigned to the connection.

The application instance becomes public when the control port is registered in SRD. Applications can query and retrieve from the SRD, the IP address and control port of another application instance. Then, the program can request the connection manager to open a connection with the other Narnia applications. The connection manager will open a new connection only if no socket is active between the two applications. If a socket between the two applications does exist, the connection manager will not create a new connection.

3.4. Memory manager

This manager controls the allocation and deallocation of memory needed for incoming and outgoing events and other data structures. The memory manager optimizes memory usage by allocating a large memory block from the heap. This memory is partitioned into three memory
pools. Event objects, name-value pairs, and strings are allocated from their own pool.

Each pool administers memory chunks of the same size. Recall that events have variable size with a maximum of 65448 bytes. It can be argued that the vast majority of the (control) events are small (e.g. <128 bytes). We have defined a default size of 128 bytes for event objects. However, each Narnia application dictates the size of their incoming and outgoing events. Narnia includes a runtime configuration mechanism that allows programmers specify the number of event objects and their size.

3.5. Event and module loaders

The event and module loaders are used to include new event handlers and programming modules, respectively. Their responsibility is to retrieve, instantiate, and link dynamic libraries at runtime.

The event loader checks that the newly loaded class is a valid event handler class. This loader extends this class to load event handler factories. This class has a single method, loadEventHandler that takes one URL as its sole parameter. This URL points to the location (i.e. protocol + machine address + path) where the module is to be retrieved. The module will be loaded and linked to the local application by this method.

Similarly, programmers can extend the virtual machine by loading new modules at runtime. One or more event handlers can call these newly loaded modules.

4. Performance measurements

Narnia is the basis for several applications. We are conducting performance measurements on a selected group of these applications. The sections below report some of our early performance results. The performance measurements in this paper focus on server load. The applications described are a Narnia baseline example, a multimedia chat room application, and a SIP proxy simulator.

In all three applications, we report the response time measured at the client site. The response time is measured from the time when the client sends the initial event and until the final response arrives to the client. Thus, the response time includes the time an event spends in the operating system buffers, the network cards, and other network elements.

The applications were tested in local area network with one Riverstone RS2000 router. Narnia server applications and their clients executed on dual-processor Sun Enterprise 450 machines.

4.1. Example 1: Narnia baseline

The baseline example implements a ping application where thousands of clients send events to a server. In turn, the server triggers an event handler that sends a reply back to the original sender. Our client application is an event load generator that can simulate thousands users. This load generator opens a TCP socket with the server for each simulated user. Thus, the server must read incoming events from all the opened sockets and post the events to the corresponding session. The execution thread in each session will execute the ping event handler that sends back a reply to the sender.

4.1.1. Implementation

We configured the server application to run one event reader and each session to run one execution thread. We ran several configurations to show how Narnia scales in terms of the number of sessions, users, and events. We selected four different session loads: 7, 100, 500, and 1000 sessions. For each session load, we ran multiple event loads from 1000 events per second (eps) to 8000 eps, with intervals of 1000 eps.

The load generator posts and stores every message sent to the Narnia application server. The server will process the message by sending a reply back to the load generator. When the generator reads the reply event, it retrieves the original stored event and calculates its roundtrip time.

4.1.2. Performance

Figure 3 shows the performance results of 28 session and event load configurations. For each session load, we only plot the event loads with roundtrip times under one second. For instance, the application server with 7 sessions can run all 8 event loads with a response time of 45 milliseconds or less. However, the application can
only support 4000 eps for 500 and 1000 sessions with an average roundtrip time of 1 second or less.

The maximum event load supported by this Narnia application server is 8000 eps while running 7 sessions. The maximum number of sessions that a Narnia application instance can support is currently 1000 sessions. Notice that the response time in all cases is under 45 milliseconds.

The number of sessions has a significant impact on the server performance. As we increase the number of sessions, the total number of threads increases. The overhead of switching among these threads has an important impact on the application performance. The roundtrip time under the same event load is significantly higher as we increase the number of sessions. For example, the roundtrip times range from 20 to 38 milliseconds for a 4000 eps load as the number of sessions increases from 7 to 1000 sessions.

4.2. Example 2: Chat room application

The multimedia chat room system is a client/server application that uses Narnia’s abstractions to support thousands of active chat rooms and tens of thousands of users simultaneously. The client application is a Java application that uses the Narnia protocol to join, interact, and leave chat room sessions. The chat room server is a Narnia application that controls and coordinates actions sent by all the client applications.

Client applications can enter chat rooms, exchange messages with other room participants, and leave the chat rooms. The server applications notify all the room participants of any newcomer. It also notifies the remaining participants of any user leaving the chat room. Client applications can send private messages to one participant and public messages to all chat room participants.

Figure 4 depicts the chat room application architecture including a session directory, a chat server, and one or more chat clients. The session directory application stores addresses of public sessions for applications to retrieve. The chat server is the rendezvous point for users to meet and exchange information. Each chat client is connected to a chat server as well as the session directory. Initially, the chat clients retrieve from the session directory the address of the entry session running in the chat server. Each client opens a connection with chat server and can join one or more chat rooms.

The chat room server is also a Narnia application that supports the creation of thousands of sessions. Chat rooms are implemented as Narnia sessions. Each session keeps track of the list of logged users and the list of public messages. When a user enters a chat room, an event handler will notify all the ongoing users in this session of the newcomer. Likewise, when a user leaves a session the corresponding event handler will notify all the ongoing users of this event.

We have included event handlers to allow users exchange instant and bulletin board messages with other users in the same session. These messages are always sent to the chat server, which in turn forwards these messages to the appropriate clients. An instant message is sent by a chat client to another chat client. In contrast, a bulletin board message is sent by one chat client, and broadcasted to all clients in the same chat room.

The chat room server keeps a queue of the last 20 bulletin board messages. This message queue enables new users to have background information of the ongoing conversation.

4.2.1. Implementation

During the load tests, we configured our load generator to simulate thousands of chat clients. The load generator opens one socket with the chat room server for each simulated client. Then, the client logs in to one chat room and starts sending at a rate of 1 event every 4.5 seconds.

The chat room server has been configured to create up to 1000 sessions. Each new session will start one execution thread to process incoming events. Given that the event rate is low, a single event reader is sufficient to handle the incoming load.

4.2.2. Performance

We will present performance results for bulletin board messages only. These results are a strict lower bound compared to the performance of instant messages because of the number of events and users involved in each new incoming message.

Figure 5 shows six configurations of the chat room application. The bars show the configuration in terms of the number of sessions, users, and input and output events. For each configuration, we performed three runs. We add 166 chat room sessions on each new configuration. The event rate per user and the number of users per session are constant in all configurations. Therefore, as we increased the number of sessions, the total number of users in the system increases. This implies that the total number of incoming events per second in the system also increases.
Each incoming event is broadcasted by the session to all the session participants. Since each chat room has 4 users, the chat server generates 4 outgoing events for every incoming bulletin board message. In addition, the chat room server generates additional control events to keep the number of messages at each client at 20. In average, the chat room server is generating 7.5 messages for each incoming bulletin board message.

We performed three runs for each configuration. The average the response time for these runs is plotted in the Figure 5. Users can expect to send a message to all his/her peers in 120 milliseconds while 4000 other users are sending 900 bulletin board messages a second to 1000 chat rooms.

4.3. Example 3: SIP proxy simulator

The SIP proxy simulator reproduces the message flow supported by a SIP proxy as found in Henning [[13]]. This simulator’s performance has been measured in an environment built from Narnia-based servers and clients.

Session Initiation Protocol (SIP) is used to discover the IP address and port number of a session endpoint. This protocol allows endpoints to describe the multimedia encodings of the traffic that will flow between endpoints. This encoding description is not interpreted by any SIP entity. Therefore, we decided not to include this description in our Narnia events.

Client applications will exchange SIP messages with one or more of the following SIP entities: a registrar, a proxy server, and user agents. Client applications include a user agent component. This component can register the application with a registrar for other applications to find them. The caller’s user agent can discover the IP address of the callee’s user agent by sending a request to a proxy server. The proxy server will find the callee’s IP address and port, forward the caller’s request to one or more proxies or contact the callee directly.

We have simulated the message flow between two user agents through a proxy server to establish and teardown a SIP connection. As specified by the SIPStone benchmark, we have measured the performance of the proxy server from the initial INVITE message to the final OK message. A proxy server needs to process at least 7 messages and generates at least 8 messages to establish and teardown a SIP connection.

The number of call attempts per second (caps) that commercially available SIP proxies can handle varies from 40 caps to 150 caps. As reported by Miercom[[5]], the latter number of connections per second was achieved with a cluster of four Sun SPARC stations.

4.3.1. Implementation

Our simulation mimics the message flow described in SIPStone benchmark using Narnia events for the 200 OK test. Thus, four processes (proxy, registrar, caller user agents, callee user agents) are executed in each run. Each process ran in a separate machine. We develop a load generator that simulates caller user agents by sending INVITE, ACK, and BYE events to the proxy. A Narnia server simulates a SIP proxy server using eight event handlers to process all the incoming events. We used our resource directory as the SIP registrar. The callee user agent simulator was developed to store information of 10,000 users in the SIP registrar as the potential number of callees.

The load generator creates twice the number of caller user agents than the target caps for each run, so that half of the callers can initiate a call every second. Each simulated caller user agent establishes and tears down a call every two seconds.

4.3.2. Performance

We examined the behavior of the SIP proxy under different call loads from 50 to 700 caps at intervals of 50 caps. For each load, we ran the simulation three times.
Figure 6 plots the average response time (solid line) for each call load. The proxy server scales linearly up to a load of 550 caps. The response time is approximately equal to \((0.5 \times \text{caps}) + 575\). The response time to establish a new call is well below the 2-second limit specified in the SIPStone proposal. Even at high loads, our SIP message flow simulation has a response time under 1.2 seconds. CPU utilization (dash line) shows the percentage of the CPU usage to process each workload. CPU utilization grows linearly with the load up to 550 caps.

5. Related work

The Narnia middleware has a unique combination of properties: a highly efficient event-based communication platform, a transport-independent event addressing scheme, direct support for code updates, and familiar, but extensible programming abstractions. Clearly, other middleware platforms possess some attributes in common with Narnia. In this section, we examine other systems and their relationship to Narnia. Most of these comparisons are based on platforms that represent classes of middleware. We focus on systems that support sessions, events, and telephony call models. Since Narnia is designed for high performance, we also measure it against performance of a well-known session protocol.

The Touring machine[1] was a software control infrastructure providing abstractions for session control, resource allocation, and network management. It included services such as directory and security. Narnia include session and resource abstraction as well as the directory service. In contrast with the Touring Machine, Narnia is a high performance virtual machine. In the near future we intend to address network management issues for IP networks.

DCWPL[4] is a coordination programming language that allows programmers to develop control programs to manage computational resources. Both DCWPL and Narnia share that control applications can be upgraded at runtime. Unlike DCWPL, Narnia support high workloads.

Several middleware platforms for the support of multimedia services have been described in the literature. Some of these platforms include communication primitives similar to those provided by Narnia, and we have selected representatives of these platforms for comparison to our present work.

CINEMA[11] defines sessions as the basis for access to remote services. A session represents the collection of resources needed to provide a service. To provide a service, a CINEMA session locates and reserves the service’s required resources. Then it establishes the needed flow of data among the resources. Narnia sessions, in contrast, do not have inherent flow-building capabilities (these are programmed as part of the event handlers in the applications) and are not limited to a single user interaction (they can persist without active users).

Several event-based middleware platforms, such as JEDI[5], Siena[3], SIP[9], and CORBA events[6], decouple event producers and consumers. The SIP Event notification proposal and the CORBA services required the interested party to register with an event dispatcher, specifying the event types they are interested in processing. In SIP, an agent sends one or more events to the dispatcher, which in turn forwards them to all the consumers registered for that event type. Narnia does not include such an event dispatcher. Rather, programmers can build such infrastructure using Narnia primitives. One would simply develop an application with the dispatcher’s functionality. Other applications can register their interest in sets of events with the new dispatcher controller. Narnia does not provide a specification language to describe how an event handler should forward events.

JAIN[15] is a collection of Java APIs to help developers build applications for multi-network (i.e. IP, ATM, and PSTN), multi-protocol and multi-party environments. JAIN is based on the JTAPI[14] call model. In contrast, our control applications are based on sessions that have no knowledge of the underlying networks and types of endpoint devices. It is possible to develop a Narnia application to use JAIN-based applications as resources.

SIP is an application layer protocol that helps an application instance to discover the IP address and port of another endpoint. Rosenberg, et.al.[11] describe how applications are supported by service applications like SIP registrars, proxies, and redirectors to find an endpoint address. In Example 2, we describe a Narnia application equivalent to a stateful SIP proxy. Furthermore, Narnia’s SRD was used to simulate the SIP registrar functionality. In addition to SIP’s discovery mechanism, Narnia provides a session abstraction to support the interaction of a group of users. This abstraction is complemented by the event handler abstraction that allows programmers implement policies to manage computational and network resources.

Like Narnia, SIP endpoints[10] can represent an end user application, a server application (e.g. audio mixer), or multicast session. SIP specifies a strict order how these applications can exchange messages. In contrast, Narnia does not impose a message flow between applications instances. It is up to each instance to establish the order of the incoming Narnia messages.

6. Conclusions

Narnia middleware is designed to support application servers that must respond to high request loads. Narnia’s support is created by its programming abstractions and by
its run-time execution environment, the Narnia virtual machine.

Our experience building applications with Narnia suggests that Narnia’s collection of familiar programming abstractions provides effective support for the creation of communication services. Our measurements of the performance of these applications indicate that the Narnia virtual machine provides an efficient execution environment for network based applications.

In particular, our performance measurements show that the Narnia communication subsystem is capable, on one CPU, of placing several thousand events per second into priority queues for processing by event handlers. This performance is capable of supporting the high server loads found in core telephony service elements and in large Web servers. For instance, the performance of our multimedia chat room server allows one server instance to support thousands of users. The performance of our SIP proxy simulator is an order of magnitude better than commercially available SIP proxies.

7. References