# Java Active Extensions: Scalable Middleware for Performance-Isolated Remote Execution

Travis Newhouse, Joseph Pasquale

*Department of Computer Science and Engineering*
*University of California, San Diego, La Jolla, CA 92093-0114, USA*

**Abstract**

We present the design and implementation of a highly scalable and easily deployed middleware system that provides performance-isolated execution environments for client and server application functionality. The Java Active Extensions system allows clients or servers to "extend" their operation by hosting portions of their codes, called extensions, at network vantage points for improved performance and reliability, and by providing them with qualities of service in the form of rate-based resource reservations. This is especially useful for wireless resource-limited clients, which can remotely locate filters, caches, monitors, buffers, etc., to act on their behalf and improve interactions with servers. Servers also benefit by moving some of their services close to their clients (e.g., those near a common base station) to reduce latency and improve bandwidth. In both cases, the client's or server's extended functionality executes with a specified fraction of the (remote) system's processor. The system design is based on a scalable distributed architecture that allows for incremental hardware growth, and is highly deployable as it runs entirely at user level, including its rate-based scheduling system.

*Key words:* Remote execution, Resource control, Quality of service, Mobile code

## 1 Introduction

While advances in wireless networks promote untethered access to information, several challenges remain before we can realize truly ubiquitous and seamless access from mobile devices. For example, consider a scientist using a mobile device to view a large 3-dimensional data set made available by a remote data

*Email addresses:* `newhouse@cs.ucsd.edu` (Travis Newhouse), `pasquale@cs.ucsd.edu` (Joseph Pasquale).

repository. When accessing a remote data source, the high latency and limited bandwidth of the wide-area network might prevent satisfactory interactive use, even if the local wireless network has sufficient capacity. To counteract these problems, an intermediate point of control that executes atop more powerful hardware in close proximity to the mobile device can, for example, cache data to reduce the effects of latency and transfer times. This point of control might also perform portions of the rendering pipeline, such as hidden surface removal, to reduce computation, network bandwidth, and power consumption on the device.

Remote services that support wireless clients can also benefit from the ability to place functionality at points between clients and servers. For example, a content distribution network can use an infrastructure of distributed computational resources to dynamically form a distribution tree by placing content caches where demands are highest. Another example is a group of friends traveling together who decide to pass time in an airport by engaging in a multi-player game against one another using their mobile devices. Multi-player wireless-networked game-play improves by reducing the latency to the game server. To reduce latency, a game server may move functionality to a point in the network that is closer (and ideally equidistant) to all, or a majority, of the players. In the case of the travelers, in which they may all be connected to the same base station, the game server may be able position functionality at or near the base station itself and take advantage of the broadcast capability of the wireless network to simultaneously send updates to multiple players. Or, this functionality might even be placed on one of the travelers' machines, assuming it were powerful enough.

The above examples illustrate the benefits of the ability to select a node between two endpoints and dynamically load application-specified functionality to that node for execution. The ability to dynamically position such functionality supports both mobile clients that connect from different locations throughout the network, and services that need to respond to unforeseeable demands. Moving functionality close to the consumer of a service can reduce the latency caused by physical distance and congestion in wide-area networks. The inserting endpoint gains a second point of control in the network that allows it to reduce the limiting effects of the Internet's best-effort design. One can take a step beyond these fairly well-established ideas by including support for improved quality of service, whereby a node that provides computational resources can guarantee resource availability for the hosted functionality. Indeed, the application should be able to specify the level of service that it requires, and thus reserve it for its execution (perhaps as a result of some form of payment).

In this paper, we present the design and implementation of Java Active Extensions, a scalable, user-level middleware system for remote execution with

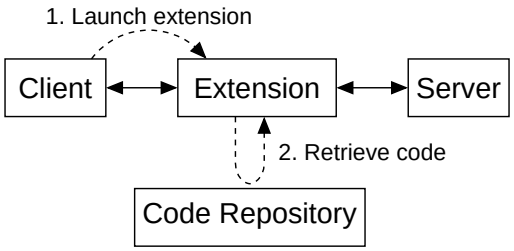


Fig. 1. The Java Active Extension model of remote execution.

explicit support for processor quality of service. The system provides a general mechanism by which applications and services can deploy functionality to network vantage points. The system architecture allows for incremental hardware growth, and its user-level implementation promotes practical deployment. Providing quality of service guarantees at user-level is challenging in terms of providing accuracy and limiting overhead because a user-level scheduler does not have total control of the processor. As a result, we adopt a resource model that lends itself to being supported by user-level scheduling mechanisms we have developed (and which we describe), while allowing applications to express different quality of service requirements.

The rest of the paper is organized as follows. In Section 2, we present the extension execution model. In Section 3, we explain the system architecture, including our resource model for processor quality of service. We describe key details of our implementation in Section 4. We present a programming example in Section 5 and a performance evaluation in Section 6. In Section 7, we discuss related work, and present conclusions in Section 8.

## 2 The Extension Execution Model

The basis for Java Active Extensions is an *extension model* for remote execution in which a remote point of control extends a client or server application (Fig. 1). We will use the term *endpoint* to refer to either a client or a server that makes use of this remote execution model. An *extension* is a unit of code that implements the functionality that an endpoint can request to have executed remotely. An *extension system* is a remote service that provides computational resources to endpoints by loading and executing their extensions. For maximum flexibility, the extended endpoint chooses what code executes at an extension system, and the extension system dynamically loads the code at runtime. The compiled format of extension code is the machine-independent Java bytecode, and our user-level implementation takes advantage of the widely available Java Runtime Environment (JRE) to provide a homogeneous execution environment [1,2]. The portability of Java bytecodes enables endpoints to deploy extensions to extension systems that run atop a variety of platforms.

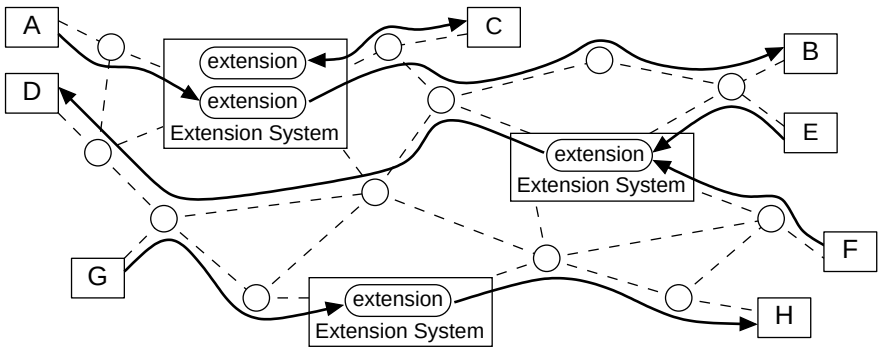


Fig. 2. Multiple extension systems operate throughout the network. Clients A and G access servers B and H, respectively, via extensions executing at intermediately located extension systems. Endpoint C uses an extension as a private server. Server D positions a portion of its services in the form of an extension at an extension system that is close to its expected clients E and F.

This simple extension model facilitates the construction of higher-level distributed computing structures, including the following canonical models: client-extended, server-extended, and private server. In the client-extended model, extensions spawned by clients can filter or customize data, cache data, bridge protocols, monitor and react to conditions, or provide anonymity. Client extensions can help mobile devices adapt to the existing Internet infrastructure by supporting a device-specific protocol between the device and extension, while the extension communicates with existing servers using a standard protocol. In the server-extended model, extensions spawned by servers can promote scalability of Internet services by enabling the service to dynamically distribute functionality to strategic nodes in the network. In the private server model, an extension spawned by an endpoint acts directly as a server rather than as an intermediary (between two endpoints). It operates in tight coordination with its creating endpoint to provide additional resources or resources with different characteristics than those locally available. For example, a mobile device can reduce power consumption by offloading computation to a private server. A desktop application may use a private server to gain access to storage resources with higher reliability than a personal computer.

One of our main goals is to promote ease and practicality of deployment. Thus, our decentralized design allows for the existence of numerous extension systems operating throughout the network. Endpoints can select an extension system that meets their requirements for resource availability and network position (Fig. 2). To launch an extension, an endpoint reserves some processor resources from an extension system and specifies the location of the extension's code. The extension system then loads the extension code and begins its execution.

Interaction between an endpoint and an extension system consists of 4 phases: discovery, resource allocation, extension deployment, and execution (Fig. 3).

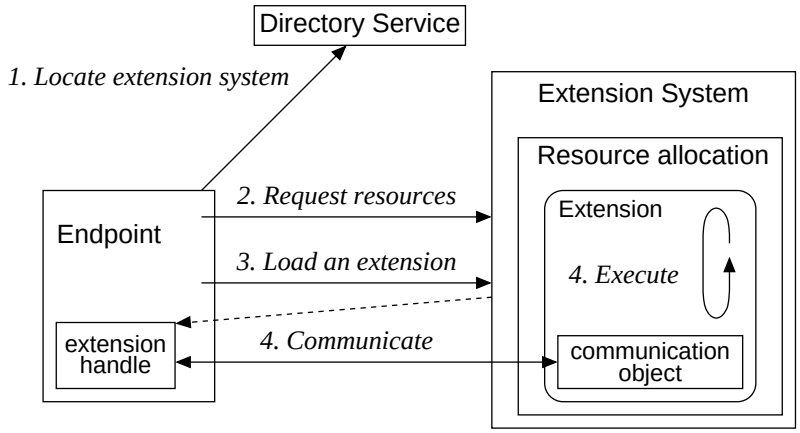


Fig. 3. Phases of interaction between an endpoint and an extension system.

In the discovery phase, an endpoint locates an extension system. Our design does not dictate how an endpoint initially locates an extension system. For example, two possible techniques, both of which we have implemented, include the following. A decentralized and highly scalable approach uses Jini Network Technology [3]. Jini provides a multicast discovery protocol to locate directories of registered services. An extension system registers itself with any "nearby" directories. Endpoints then use the same multicast protocol to locate nearby directories that can be searched for a registered extension system service. A simpler more centralized approach uses sockets and an out-of-band information publishing scheme, such as a Web page that lists the addresses and port numbers of extension systems. Other discovery methods are certainly possible.

The resource allocation phase supports endpoints that require more than best-effort quality of service. Before an endpoint launches an extension to an extension system, the endpoint must first request a share of the extension system's processor resources with which the extension will run.[1] When loading an extension, the endpoint binds the extension to an allocated share of processor resources. Multiple extensions can be bound to the same resource allocation. An extension system ensures that all extensions bound to the same allocation do not consume more processor resources than initially allocated.

After discovering an extension system and allocating resources, the endpoint is ready to launch an extension that will execute at the extension system with a specified resource allocation. The extension execution model supports code mobility based on a single-hop, move-and-execute model. The endpoint specifies to the extension system an extension type and a set of URLs pointing to the extension's code. The extension system retrieves the code, creates an

[1] In this paper, it is assumed that the endpoint has some way of determining the amount of processor resources it needs. This may be by repeated refinement over multiple executions, or by some analytical method of predetermination, etc.

instance of the extension, and spawns a new thread of execution to begin the extension. The extension executes autonomously within an isolated execution context possessing a share of the extension system's processor resources.

The code for an extension must conform to a minimal interface that contains a single method, named `run`. This method, implemented by the extension's developer, defines the extension's entry point much like the `main` function of a traditional program written in C or Java. The extension's execution begins when the extension system invokes the extension's `run` method in a thread of execution bound to the resource allocation for the extension. Execution continues until the extension voluntarily returns from the `run` method. The `run` method accepts a single argument that is a set of resource objects supplied by the extension system. These resource objects can provide the extension with access to specialized software (e.g., database) or hardware (e.g., display) resources. At minimum, the set contains an object that implements message-passing between the extension and the endpoint application. Aside from requiring a pre-defined entry point, the extension model places no design constraints on an extension's code.

Recognizing that no single communication mechanism will likely meet the needs of a variety of existing and future distributed applications, the extension model does not mandate the manner in which extensions and endpoints communicate. On the other hand, to support uses of extensions that require specialized communication with an endpoint, the extension system provides a basic message-passing communication mechanism supporting delivery of arbitrarily-formatted messages to and from the extension, with message delivery following in-order and at-most-once semantics.

Message passing works as follows. Upon loading an extension, the extension system returns to the endpoint an object through which the endpoint and extension can exchange messages. Each message is an arbitrary sequence of bytes; it is up to the extension's developer to choose a suitable format. The extension accesses the mechanism through a corresponding object passed as a parameter to its `run` method when execution begins. The message-passing communication interface consists of two operations. The `send` method queues a message for the recipient. The `receive` method dequeues a message, or blocks until a message is available. Both the extension and the endpoint use this same interface to communicate, as shown in Fig. 4. The communication object provided to an extension by the extension system maintains two message queues per extension, one for messages sent to the extension and one for messages sent to the extension's handle.

A common use of this simple message-passing mechanism is to bootstrap application-specific communication channels. For example, if an endpoint needs to send data over a UDP channel between itself and an extension, the exten-

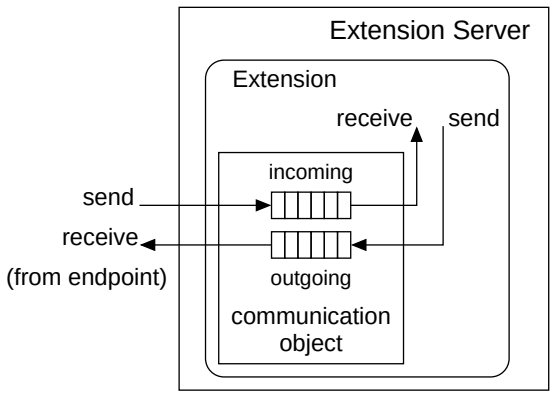


Fig. 4. An extension server provides a message-passing mechanism that enables an endpoint and an extension to exchange arbitrarily-formatted messages.

sion is programmed to create a UDP socket and bind the socket to an available port at the extension system. The extension informs the endpoint of the location to address UDP datagrams by sending a message (using the extension system's default message passing mechanism) that contains the IP address and the UDP port to which it was able to bind the socket. In cases where communication performance is non-critical, the message-passing mechanism may be suitable for all communication between endpoint and extension.

## 3 Extension System Architecture

An extension system exposes two primary components with which an endpoint interacts to deploy an extension. Dividing the architecture into two primary components supports scalability of hardware resources and promotes a separation of resource-sharing policy and performance-isolated execution. Execution duties are performed by an *extension server*, which provides a share of an extension system's processor resources in the form of an execution environment for one or more of an endpoint's extensions. An extension system's *manager* component enforces a locally-defined policy for resource sharing and lease renewal. An endpoint submits resource requests to the manager, which grants requests by creating an extension server for use by the endpoint. The manager then schedules resources among allocated extension servers to fulfill resource sharing agreements that it makes. The public interface to which endpoints are programmed consists of operations on the manager and the extension server (Fig. 5). Managers and extension servers are described in more detail below, but first we describe how an endpoint specifies a resource request.

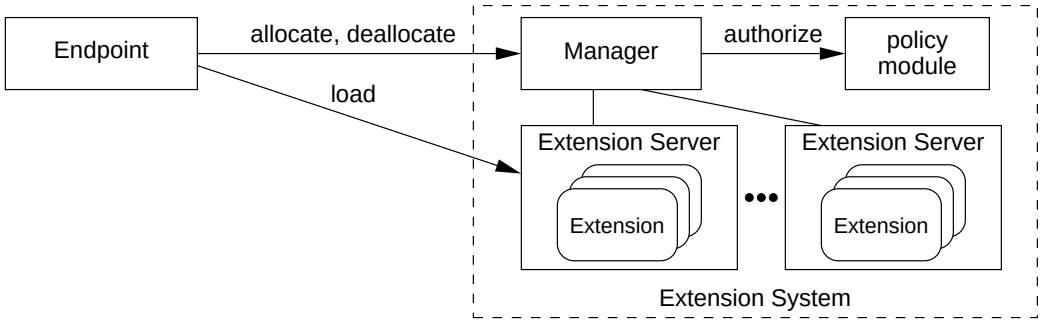


Fig. 5. An endpoint interacts with an extension system by invoking operations on a manager and an extension server. A manager fields resource allocation requests by consulting a locally-defined resource policy module. An extension server implements an allocated resource share for the execution of one or more extensions.

### *3.1 Quality of Service*

To provide quality of service, an extension system employs a user-level scheduler (described in Section 4.2) to enforce shares of processor resources. The user-level approach promotes ease of deployment, at the cost of coarser control and softer guarantees than what a kernel-level proportional-share scheduler could provide. The resource model is simple: an endpoint will receive a given share of the processor over a given period of time. So that applications can request an appropriate level of service, we make an endpoint responsible for determining the period over which resource guarantees are to be met.

An extension system's manager shares processor resources in multiples of a fixed-length time unit, or *quantum*. Each extension system locally defines the length of its quantum, and advertises this value (along with other information, such as the type of machine and its speed) to potential endpoints as part of the discovery scheme. An endpoint requests processor resources in the form of quanta per period of time ($\langle quanta, period \rangle$). This tuple expresses both a share of the processor resources and a periodic deadline when the share will be satisfied. If an endpoint requires 20% of the processor resources, then the endpoint may request $\langle 2, 10 \rangle$ or $\langle 20, 100 \rangle$, depending on the demands of the application. The first specifies a higher degree of control and higher level of service quality (which may naturally require higher payment in a pay-for-use

extension system). [2]

A manager grants a share of resources to an endpoint for a finite duration of time, in the form of a lease [4]. Leases benefit both endpoints and extension systems. To an endpoint, a lease guarantees availability of resources for the duration of the lease. For an extension system, a lease enables macro-scale resource-scheduling decisions and reclamation of resources reserved by failed endpoints. When a lease expires, the manager deallocates the associated extension server, terminating any extensions executing within it. If an endpoint needs to use the extension server's resources for longer than the initial lease duration, the endpoint can request a renewal of the lease granted on those resources. Lease maintenance can be performed by the endpoint, the extension, or even a third-party (see Section 4.1).

### *3.2 Extension Server*

An extension server is an isolated execution environment possessing a share of an extension system's computational resources. Within an extension server, an extension executes with full Java runtime semantics, including the ability to spawn threads or open network sockets (though, individual site administrators may impose security policies with fine-grained network or file permissions). An extension system schedules resources using the extension server as the resource principal, with all extensions executing in a particular extension server contributing toward that extension server's resource usage.

The interface to an extension server contains a single operation to load an extension. The `load` operation accepts the class name of the extension, initialization parameters, and the location where the extension's code resides. The operation returns a handle by which the endpoint may communicate with the extension using the message-passing interface. Consequently, the extension server retrieves the extension code, creates an instance of the extension, establishes message queues for communication, and begins execution of the extension by invoking the object's `run` method.

The execution environment provided by an extension server is an instance

[2] It is the responsibility of the endpoint to determine how many shares it needs to achieve a required level of performance. Assuming the latter is in terms of some absolute measure (e.g., MIPS), the endpoint can roughly determine the required shares by knowing the quantum length and details of the machine's performance (e.g., its speed in MIPS, scaled to account for executing Java on that type of machine). A more refined determination can result from trial-and-error, i.e., executing, seeing if the performance is adequate, and adjusting the shares. If the endpoint cannot afford to initially underperform, it can over-allocate shares appropriately.

of the Java Runtime Environment (JRE) [1,2]. When allocating an extension server, a manager launches a new instance of a Java Virtual Machine (JVM) in which the extension server creates a thread of execution for each extension that an endpoint loads. The JRE provides the following well-known benefits. First, the JRE operates at user-level and is available for a wide variety of platforms. Second, it provides a homogeneous execution environment across nodes that have heterogeneous hardware and system software. Extensions always run within a JVM, which abstracts the underlying hardware and operating system. This enables providers of computational resources to evolve their low-level execution platform without impacting the manner in which endpoints use the resources. Finally, the JRE provides security mechanisms to protect the host system and to isolate extensions from one another. Isolation exists because each extension server is a distinct JVM with an isolated class namespace and its own address space with respect to the underlying operating system. To control the processor resources consumed by an extension server, a user-level scheduler is used to schedule the JVMs (see Section 4.2).

### *3.3 Manager*

A manager supports two operations. The `allocate` operation requests a new extension server with a desired share of processor resources. The parameters to the operation are $\langle quanta, period, duration \rangle$, where quanta and period define the resource share and duration specifies the desired lease duration. If the request is granted, the operation returns a tuple containing a handle to an extension server and a handle to a lease for the extension server. A handle is a local identifier for a remote entity. The extension server handle is used when loading an extension, to identify the resource allocation with which the extension will execute. The lease handle is used when renewing the lease associated with an extension server's resource allocation. The `deallocate` operation releases an endpoint's interest in an extension server. When an extension server is deallocated, the manager destroys all extensions executing in the extension server and reclaims any resources reserved for the extension server.

Each extension system has a locally-defined *resource policy module* that governs decisions about resource allocation and lease renewal. Upon receiving a request to allocate an extension server or to renew a lease, a manager consults the local resource policy module before taking action to grant or deny the request. To define the policy, an extension system's administrator defines a Java class that implements an `authorize` method. The `authorize` method inputs a resource request tuple ($\langle quanta, period, duration \rangle$), and returns a boolean value whether or not to permit the request. The manager possesses an instance of this class on which it invokes the `authorize` method upon each resource

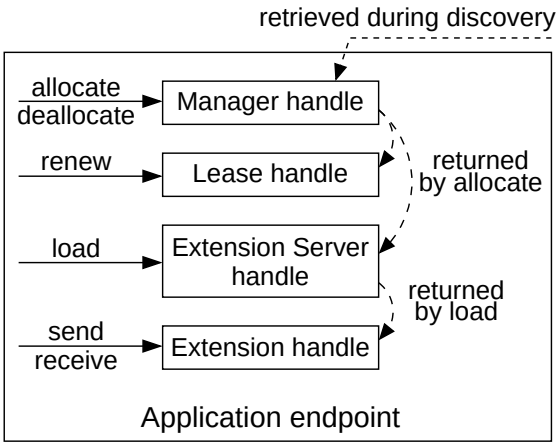


Fig. 6. An application endpoint interacts with system components by invoking methods on local handle objects that manage communication with remote counterparts.

request or lease renewal.

## 4 Implementation

Our implementation consists of several Java classes that implement the components of the extension system architecture, and a user-level scheduler implemented in C. In this section, we describe how we use Java objects to abstract communication and promote the scalable design of the extension system architecture. We also explain the basic operation of the user-level scheduler and how it integrates with the Java Active Extensions system.

### *4.1 Scalability*

Endpoints perform operations on extension system components by invoking methods on local objects, or *handles*, that represent the remote entity. The local handle objects manage communication between the endpoint and the extension system. In this way, endpoint applications interact with a consistent API, while the underlying communication mechanisms can evolve with the system implementation. The implementation of the handle object is loaded dynamically from the network as the endpoint application interacts with an extension system (Fig. 6). In our current implementation, we use Java RMI (remote method invocation) as the communication substrate. Each handle object contains sufficient information and functionality (implemented by the object's Java class) to contact its corresponding extension system component directly. Thus, the objects can be transferred from one node to another without loss of functionality.

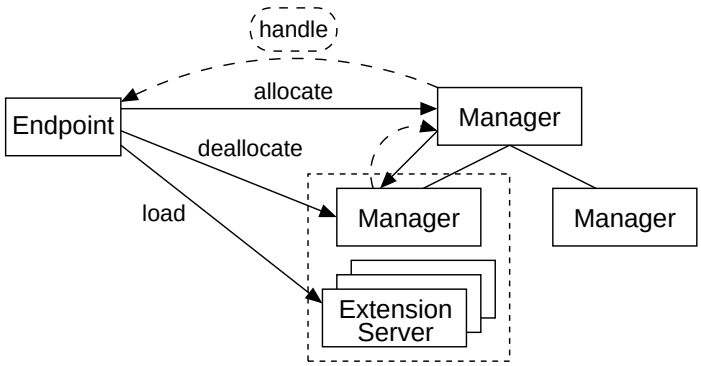


Fig. 7. System architecture supports direct communication between endpoint and extension system components to minimize bottlenecks when accessing distributed resources.

By decoupling allocation policy from execution mechanism and supporting the use of transferable handles, the extension system architecture promotes scalability of distributed hardware resources. This is achieved by supporting a hierarchical arrangement of managers without maintaining a chain of managers in the critical path of extension loading and execution. At the top of the hierarchy, the "root" manager of an extension system provides the public interface by which endpoints make resource allocation requests. Below the root manager, a "node" manager runs on each physical machine that will provide computational resources to extensions. When an endpoint requests resources, the root manager routes the request to a node manager. The node manager creates an extension server and returns the extension server's handle to the root manager, which in turn returns the handle to the endpoint. Now, the endpoint can load extensions by invoking the `load` operation on the extension server handle. The handle communicates a load request directly to the node on which its corresponding extension server runs, bypassing the hierarchy of managers (Fig. 7).

A hierarchical arrangement of managers also supports incremental growth of hardware resources. An extension system's administrator can increase hardware resources by starting a node manager on a new machine. The user-level implementation of Java Active Extensions means that starting a node manager is the equivalent of simply launching an application. The node manager will contact the root manager to register the availability of the new machine's processor resources. Much like the discovery process for an endpoint using an extension system, the node manager can dynamically locate the root manager using Jini, or the location of the root manager can be specified as a configuration parameter.

This design also benefits other operations, such as lease maintenance. A lease handle contains contact information for the manager component that granted the lease, as well as an identifier of the extension server for which the lease was granted. The holder of the lease handle may perform a lease renewal operation

independent of possession of the manager handle or extension server handle. This supports the possibility of a lease renewal service that maintains leases on behalf of endpoints, but without requiring access to an endpoint's extension server. A device with weak connectivity can use such a lease renewal service to ensure that an extension server remains allocated even while the device is disconnected. Alternatively, an extension running in the extension server can maintain the lease.

### *4.2 Enforcing Processor Shares*

A key challenge in the user-level implementation of Java Active Extensions is how to efficiently enforce processor resource-sharing guarantees. For this purpose, we have developed a user-level scheduling framework [5] that assumes the availability of Unix mechanisms (which are generally available in some form or another in most modern operating systems). The user-level scheduler operates in tandem with the system's underlying kernel scheduler by making coarse-grained scheduling decisions about which *group* of processes is eligible to run, and relying on the kernel scheduler to perform fine-grained scheduling among the processes in the group. This group scheduling technique is an effort to reduce overhead by taking the user-level scheduler out of the critical path of each scheduling decision. In addition, the user-level scheduler does not have access to information such as when a running process blocks, and thus cannot make a responsive decision to run a different process. Instead, the user-level scheduler uses coarse-grained (relative to the kernel) timers to periodically determine which processes have not exceeded their share of the processor, and allows the kernel to schedule processor time amongst the processes in that group.

To monitor the progress of processes, the scheduler must determine the amount of processor time consumed. We implement this using Unix's `/proc` file system facility. To promote and demote processes to the "running" group, the scheduler uses the `SIGCONT` and `SIGSTOP` signals, respectively. Sending a `SIGSTOP` to a process ensures that a process is not on the kernel's ready queue, thus preventing the kernel from scheduling the process. Subsequently, sending a `SIGCONT` to the process returns the process to its previous state, either on the ready queue or waiting on an event, so that the kernel will once again consider it for scheduling.

Consequently, to maintain isolation and to control resources assigned to individual extension servers, a manager launches a distinct JVM for each extension server that it allocates. Separate JVMs provide isolation in that an extension of one extension server cannot manipulate the environment of an extension executing in a different extension server. For scheduling purposes,

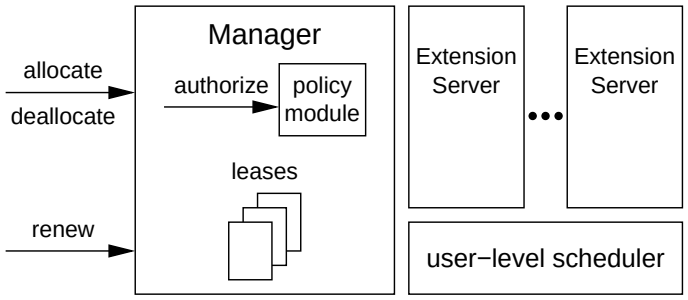


Fig. 8. Manager enforces resource-sharing policy using a user-level scheduler.

the JVM serves as the resource principal scheduled by the user-level scheduler. All threads executing in a JVM, started either in response to loading an extension or spawned by a running extension, contribute to the processor time consumed by the JVM process. When the manager creates a new extension server, it notifies the user-level scheduler of the new JVM's process identifier and the allotted processor share. The user-level scheduler executes external to all JVMs, including the manager, to monitor and control each extension server's processor consumption (Fig. 8).

Mechanisms for creating isolated, resource-controlled environments *within* a JVM are currently being developed [6,7]. When available as part of the standard Java platform, we can use such mechanisms to implement the extension server without requiring a separate JVM to maintain isolation. Our extension system architecture and the interface by which applications request resources and launch extensions would remain the same.

## 5 Usage Scenario

To illustrate the use of Java Active Extensions, we present an example of how a mobile device might use an extension system to generate captions for the audio portion of a video clip. The video encoding does not already contain captions, so the mobile device's application must generate them using real-time speech recognition. Because the mobile device does not have enough computational power to perform speech recognition locally, the application offloads this computation to an extension (Fig. 9). In Listing 1 and Listing 2, we show an example of the code that the endpoint and the extension use to interface with an extension system.

To begin, the application endpoint first locates an extension system. If supported by the chosen discovery scheme, the endpoint prefers the extension system to be "close" to the device to reduce the round trip latency for sending audio data and receiving the captions. For this example, we assume the details of discovery are encapsulated within a method named `locateManager()` (List-

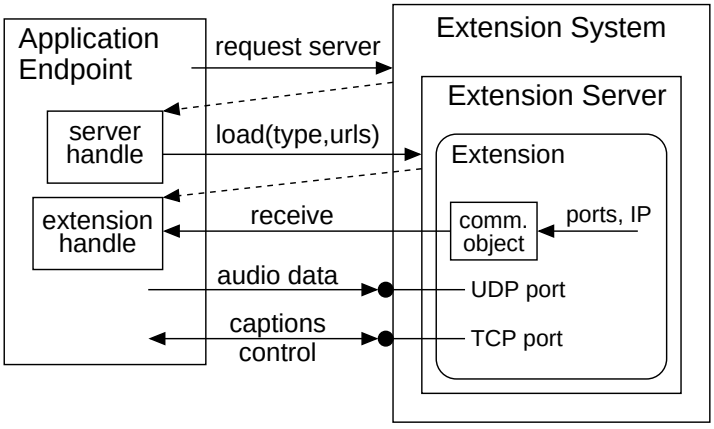


Fig. 9. An application endpoint uses an extension to generate captions for the audio portion of a video stream.

ing 1, line 2). The `locateManager()` method returns a handle to an extension system's manager.

The application endpoint uses the manager handle to allocate an extension server with enough processing resources to support the task of speech recognition. The endpoint requests 5 quanta out of every 20 quanta for a duration of 10 minutes. The `allocate()` method of the `Manager` object contacts the extension system to make the request. If the request is granted, the `allocate()` method returns a `ServerAllocation` object that contains a handle to the allocated extension server and a lease on the resources assigned to the extension server. We ignore leasing in this example. If the allocation request is denied, then the `allocate()` method throws an exception (not shown).

After allocating an extension server, the endpoint is ready to load an extension. It passes the following to the extension server handle's `load()` method: a class name, initialization parameters (none in this case), and an array of `URL` objects pointing to the locations of the extension's code. The `load()` method sends the parameters across the network to the extension server. The extension server loads the code from the network and creates an instance of the specified extension. After the extension has been instantiated, the extension server begins execution of the extension and returns to the endpoint a handle to the loaded extension. The `ExtensionHandle` interface enables the endpoint to pass messages between it and the extension. In this example, the endpoint calls `receive()` to accept messages containing the address and port for UDP and TCP communication channels.

The code for the `MyExtension` class is shown in Listing 2. A class that is loaded as an extension must implement the `Extension` interface. This interface requires that the class implement a `run()` method. The `run()` method is the entry point at which an extension system begins an extension's execution. The parameter to the method is a collection of objects that enable access to

**Listing 1** Endpoint code to load an extension

```
 1: // retrieve extension system's manager
 2: Manager m = locateManager();
 3:
 4: // allocate resources
 5: int q = 5;      // quanta requested
 6: int p = 20;     // period
 7: int d = 10;     // 10 minutes
 8: ServerAllocation sa = m.allocate(q,p,d);
 9: ExtensionServer es = sa.getServer();
10:
11: // launch the extension
12: URL[] urls = { new URL("http://myserver.com/mycode.jar") };
13: ExtensionHandle handle = es.load("MyExtension", null, urls);
14:
15: // receive address and ports opened by extension
16: InetAddress host = (InetAddress) handle.receive();
17: Integer tcpPort = (Integer) handle.receive();
18: Integer udpPort = (Integer) handle.receive();
19:
20: // send audio datagrams and display captions
21: doDisplayCaptions(host, tcpPort, udpPort);
22:
23: // release extension server resources when done
24: m.deallocate(es);
```

local resources provided by the extension server. For example, the collection always contains an object of type `ExtensionCommunicator` that implements the message-passing communication channel.

Upon starting, the extension opens two network ports, a UDP port to which the endpoint will send audio data, and a TCP socket over which captions and control messages will be exchanged. The extension uses the message-passing mechanism provided by the extension server to send to the endpoint messages containing the IP address of the machine on which it is executing and the ports the extension was able to open (Listing 2, lines 13–15). This message-passing mechanism is provided by the extension server for exactly this purpose of bootstrapping application-specific communication.

Using the extension handle returned by the extension server, the endpoint receives the messages containing the address and port numbers for the communication channels setup by the extension. The endpoint begins sending audio data to the extension using UDP datagrams. The extension converts the audio to text that it sends back to the endpoint using the TCP connection. When the application finishes playing the video clip, the endpoint informs the ex-

**Listing 2** Extension code

```
 1: class MyExtension
 2:   implements Extension
 3: {
 4:     public void run (Resources r)
 5:     {
 6:         ExtensionCommunicator comm = (ExtensionCommunicator)
 7:             r.getByType(ExtensionCommunicator.class);
 8:
 9:         ServerSocket tcp = new ServerSocket(0);
10:         DatagramSocket udp = new DatagramSocket();
11:
12:         // send port and address to application endpoint
13:         comm.send(InetAddress.getLocalHost());
14:         comm.send(new Integer(tcp.getPort()));
15:         comm.send(new Integer(udp.getLocalPort()));
16:
17:         // perform processing loop
18:         doGenerateCaptions(tcp, udp);
19:     }
20: }
```

tension by sending a control message over the TCP connection. The extension closes the ports it had opened, and exits. The endpoint also informs the extension system that it can reclaim the resources assigned to the application's extension server.

To conserve space, we have excluded from the code listings any error handling. All the methods of our API that contact the extension system throw an exception in the event that there is a communication failure.

## 6 Evaluation

The benefits that can be gained by using the extension system depend largely on the manner in which endpoints use extensions. Potential benefits include increased performance, increased reliability, and reduced power consumption. The cost of using an extension lies in the overhead of interacting with the system. In this section, we present the costs associated with using an extension system. These costs include the basic operations necessary for launching an extension, time to send and receive messages, and the overhead of enforcing resource control.

### *6.1 Launching an Extension*

Launching an extension requires locating an extension system's manager, allocating an extension server, and loading the extension. We performed each of these operations 1000 times with a 1 second delay between each iteration. The extension system and endpoint ran on separate machines, each configured with a 2.2 GHz Pentium 4 processor, 512 MB of memory, and the FreeBSD 4.8 operating system. The Jini service directory ran on a machine with dual 600 MHz Pentium III processors and 1 GB of memory, running the Solaris 8 operating system. The machines communicate over a switched 100 Mbit/s Ethernet. Each iteration of the endpoint code ran in a new JVM, subjecting each iteration to class loading overhead. *Typically, much of this class loading will not exist, as many classes will have already been loaded from previous instantiations, and so these times reflect worst-case scenarios.*

To locate a manager, we tested both Jini and a direct approach using sockets. For the Jini test, the extension system registers a manager object with a Jini service directory running on the local network. The test measures the time for an endpoint to contact the directory, perform a lookup of the extension system service, and retrieve the manager handle object. We do not test the multicast discovery of the Jini directory, but instead contact it directly. Therefore, the time represents primarily the lookup and retrieval costs when using Jini. The second approach directly contacts the extension system using a socket. The endpoint opens a connection to the extension system and downloads the manager handle. The primary cost in this test is the time to retrieve and load the manager handle object into the endpoint's JVM.

In the allocation test, the endpoint first retrieves a handle to a manager. Measurement begins when the endpoint makes an allocation request, and ends when the `allocate` method returns the extension server handle and lease handle. The result includes the time to transfer the handle objects from the manager to the endpoint and the time for the manager to create a new extension server. The extension server allocation cost is dominated by the time to fork a new JVM. To reduce latency, a manager may pre-allocate extension servers, though we have not yet experimented with this technique.

The load test measures the time to load a "null" extension. The extension contains no data and its `run()` method contains zero statements. However, the extension server must still retrieve and load the minimal extension code from the network. In this test, the code resides on a Web server in the local network. Therefore, this time represents the minimum load time of our implementation.

For each test, Table 1 lists the mean operation time with a 99% confidence interval. The total time to launch an extension is approximately 1 second (or less

Table 1
Time to launch an extension (ms)

| | Mean |
|---|---|
| Discover manager using Jini | $629.5 \pm 1.2$ |
| Discover manager using socket | $307.0 \pm 0.3$ |
| Allocate extension server | $317.1 \pm 0.7$ |
| Load extension | $97.0 \pm 0.2$ |

if using socket-based discovery) for an endpoint that has not yet discovered an extension system nor allocated an extension server. Discovery, resource allocation, and loading an extension are one-time costs incurred by an endpoint at the beginning of a session that uses an extension. The cost of these operations is acceptable with respect to the typical session times of the applications that can benefit from using an extension (e.g., adapting Web content, positioning network services, automated speech recognition).

### *6.2 Sending Messages*

Though our design does not prescribe performance guarantees for the message-passing communication mechanism, we show that even a simple implementation of the mechanism has acceptably low overhead. Like the rest of the implementation, the message-passing mechanism uses Java RMI for communication duties. We test both the one-way and round-trip time to send messages from an endpoint to an extension. The message contains a byte array filled with randomly generated data. In the round-trip test, the return message is identical to the message sent from the endpoint. Each result is the mean over 1000 iterations. Fig. 10 shows the times for message sizes of 1, 1000, 10000, and 100000 bytes. The one-way message cost is less than the time to retrieve a file of equal size from a local Web server. Thus, the communication mechanism exhibits acceptable cost, especially considering it is intended for bootstrapping application-specific communication.

### *6.3 Resource Control*

At present, our user-level scheduler implementation enforces shares using a 20-millisecond quantum and a fixed-length period containing 50 quanta (e.g., an endpoint may request an extension server that receives $N$ quanta per every 50 quanta). We are currently developing a rate-based scheduling algorithm that allows endpoints to request the period over which guarantees are made, and developing a policy module that performs admission control. Here, we report

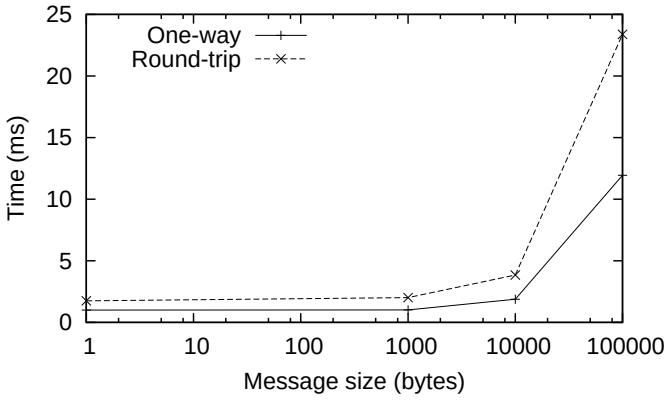


Fig. 10. Message delivery times using the Java Active Extension message-passing mechanism.

on the accuracy and overhead of the user-level scheduler to demonstrate the feasibility of our approach. We scheduled 4 compute-bound processes with fractional processor rates of 10%, 20%, 30%, and 40%. Fig. 11 displays the processor share and relative error at the end of each period. To summarize the accuracy, the root mean square relative error calculated over all periods of the experiment is 0.54%. The overhead of the scheduler is less than 3% for up to 32 processes. To test workloads that involve both processing and I/O, we tested the ability of the scheduler to control shares of 3 Apache Web servers configured to use 50 processes each. The scheduler was capable of enforcing proportional share throughput within 1% relative error using only 3.2% of the total processing time. More detailed results are available in [5].

## 7 Related Work

The benefits of placing code at a strategic point in the network and having it hosted in a protected execution environment have been recognized by many others. Our contributions focus on the design of an easily deployable, user-level system that provides an execution environment with guaranteed shares of resources for remotely launched code. We divide our discussion of related work into systems for intermediate processing and solutions for hosting code in isolated execution environments.

### *7.1 Placing Code "in" the Network*

Many prior works have explored using mobile code and remote execution to perform processing in the network. Typically, they have concentrated on how to dynamically interconnect service components with one another, how to

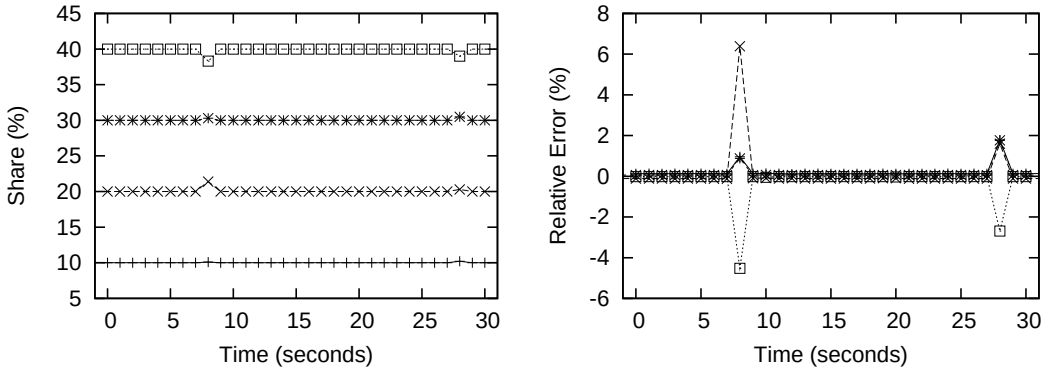


Fig. 11. Processor share and relative error for compute-bound processes measured over 1-second periods.

seamlessly integrate the components with existing client and server frameworks, and how to dynamically distribute services for scalability. Java Active Extensions are complementary to works in this area, as we have focused in detail on the mechanisms for loading code to a remote host and providing an isolated resource share with which to execute code. A common mechanism can support the needs of many higher-level frameworks.

In the simplest form, a proxy is a static system that performs processing on data as it passes through the network. Dynamic proxy architectures [8] and edge services [9] can load functionality on demand to support adaptation for network load and device heterogeneity. Our design for a general remote execution service enables edge services to be deployed dynamically. Hence, rather than statically deploy a multitude of specific services at edge locations, a general execution system can be deployed once to support dynamic deployment of higher-level services.

The Web& system [10], Chroma [11], server-directed transcoding [12], and MARCH [13] split application functionality between an endpoint and a network node. Beyond these capabilities, we focus on providing a resource request model and implementing performance isolation in the execution environment.

Dahlin et. al. describe how "mobile server extensions" can improve access to dynamic content and present a framework to seamlessly integrate the functionality into HTTP requests [14]. The mobile server extension can execute at the client, server, or a proxy node. The Java Active Extensions system provides a general execution environment that shares resources with mobile code. An extension system serves as a platform on which mobile server extensions can be deployed at intermediate network nodes.

CANS provides an infrastructure for data adaptation that supports the insertion of application-specific components along the data path [15]. The execution environment they propose for intermediate nodes is tailored to their compos-

able, component-based model. The Java Active Extensions system provides a general execution environment on top of which individual components of a composable system can execute directly, or on which a support framework for higher-level system frameworks can operate with an isolated share of system resources.

Remote evaluation (REV) [16] focused on remote execution as an optimization for RPC. REV's execution model is based on a procedure call, and the implementation is tightly integrated with the language and compiler. We have designed a system with a more general execution model and implemented it on top of the widely available Java Runtime Environment. We can simulate REV's procedure-call semantics by launching an extension, sending parameters in a message, and waiting for a message containing the return value.

### *7.2 Isolated Execution Environments*

Whereas our extension system is a user-level approach to hosting endpoint-supplied code, others have taken a low-level approach using a virtual machine monitor (VMM) to create strongly isolated execution environments. Denali [17] is an isolation kernel designed to support thousands of virtual machines on a single physical host, with each virtual machine providing an execution environment for an untrusted network service. If isolation kernels such as Denali become widespread, our extension system interface can be implemented using their virtual machines to provide execution environments.

Research in grid architectures examines issues relating to distributed resource allocation and remote execution of applications. The Open Grid Services Architecture [18] specifies how interactions take place between services (e.g., naming, communication), but does not prescribe the execution environment a node provides. The Globus Architecture for Reservation and Allocation [19] enables an application to reserve collections of resources for end-to-end QoS. The OSGi service platform [20] executes on a device to provide service providers and developers with an open platform to which services can be deployed. The service platform enables remote loading, updating, starting, and stopping of services to a device, and also supports interoperation between services. The service platform does not specify a resource model by which services can request a share of computational resources at the device. We focus on providing the computational resources to execute endpoint-supplied functionality at a particular node. We define a general resource sharing and execution model that is suitable for the execution of endpoint-supplied code.

Finally, a Ninja [21] "base" provides an execution environment that automatically scales service functionality for both performance and fault tolerance. The

base requires services to be programmed to a specialized event-based model, and distributes services across a cluster of workstations. Another component of Ninja, "active proxies," provides adaptation by dynamically inserting "operators" that transform data along a communication path. The Java Active Extensions system provides a general, resource-controlled execution environment that can support the deployment of higher-level execution models.

## 8 Conclusions

The ability to execute application-specific functionality at intermediate points between a client and server can enhance distributed applications targeted at users of wireless networks. Such applications require a service that will execute extensions of their functionality with predictable performance.

Java Active Extensions offer a scalable and practical approach to remote execution with processor quality of service. The system's interface is simple and minimal in order to support a variety of client and server applications without constraining their design. Scalability results from a highly decentralized architecture that supports incremental growth of hardware resources. Deployment is practical in that all of our mechanisms operate at user level. The system provides processor quality of service using a rate-based reservation scheme that allows applications to express quality of service needs while supporting flexible, efficient scheduling at user-level. The system's user-level scheduler enforces processor shares using a two-level "group scheduling" technique that operates with low overhead.

For future work, we are extending our design to include a security framework that will operate in tandem with the site-specified resource policy module to enable site administrators to define a secure authorization scheme. To assist endpoints in selecting an extension system, we plan to develop a directory service by which an endpoint can find extension systems that satisfy its quality of service requirements. We are also investigating mechanisms for resource control of additional resources, such as network bandwidth and storage.

## References

[1] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java™ Language Specification, 2nd Edition, Addison-Wesley, Boston, MA, 2000.

[2] T. Lindholm, F. Yellin, The Java™ Virtual Machine Specification, 2nd Edition, Addison-Wesley, Boston, MA, 1999.

[3] Sun Microsystems, Jini Network Technology, http://www.sun.com/software/jini/ (2004).

[4] C. G. Gray, D. R. Cheriton, Leases: an efficient fault-tolerant mechanism for distributed file cache consistency, in: Proceedings of the 12th ACM Symposium on Operating Systems Principles, ACM Press, 1989, pp. 202–210.

[5] T. Newhouse, J. Pasquale, A user-level scheduling framework for processor resource sharing, in: Proceedings of the 2004 IEEE International Conference on Services Computing, 2004.

[6] G. Czajkowski, Application isolation in the Java$^{TM}$virtual machine, in: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press, 2000, pp. 354–366.

[7] G. Czajkowski, S. Hahn, G. Skinner, P. Soper, C. Bryce, A resource management interface for the Java$^{TM}$platform, Tech. Rep. TR-2003-124, Sun Labs (May 2003).

[8] B. Zenel, D. Duchamp, General purpose proxies: solved and unsolved problems, in: Proceedings of the Sixth Workshop on Hot Topics in Operating Systems, 1997, pp. 87–92.

[9] L. Gao, M. Dahlin, A. Nayate, J. Zheng, A. Iyengar, Application specific data replication for edge services, in: Proceedings of the Twelfth International Conference on World Wide Web, ACM Press, 2003, pp. 449–460.

[10] S. H. Phatak, V. Esakki, B. R. Badrinath, L. Iftode, Web&: An architecture for non-interactive web, in: Proceedings of the Second IEEE Workshop on Internet Applications, 2001, pp. 104–113.

[11] R. K. Balan, M. Satyanarayanan, S. Park, T. Okoshi, Tactics-based remote execution for mobile computing, in: Proceedings of the First USENIX International Conference on Mobile Systems, Applications, and Services, 2003.

[12] B. Knutsson, H. Lu, J. Mogul, B. Hopkins, Architecture and performance of server-directed transcoding, ACM Trans. Inter. Tech. 3 (4) (2003) 392–424.

[13] S. Ardon, P. Gunningberg, B. Landfeldt, Y. Ismailov, M. Portmann, A. Seneviratne, MARCH: A distributed content adaptation architecture, International Journal of Communication Systems, special issue on Wireless Access to the Global Internet: Mobile Radio Networks and Satellite Systems 16 (1) (2003) 97–115.

[14] M. Dahlin, B. Chandra, L. Gao, A.-A. Khoja, A. Nayate, A. Razzaq, A. Sewani, Using mobile extensions to support disconnected services, Tech. Rep. TR-2000-20, Department of Computer Sciences, University of Texas at Austin (2000).

[15] X. Fu, W. Shi, A. Akkerman, V. Karamcheti, CANS: Composable, adaptive network services infrastructure, in: Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS 2001), 2001.

[16] J. W. Stamos, D. K. Gifford, Remote evaluation, ACM Transactions on Programming Languages and Systems 12 (4) (1990) 537–564.

[17] A. Whitaker, M. Shaw, S. D. Gribble, Scale and performance in the Denali isolation kernel, in: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002), Boston, MA, 2002.

[18] I. Foster, C. Kesselman, J. M. Nick, S. Tuecke, Grid services for distributed system integration, Computer 35 (6).

[19] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, A. Roy, A distributed resource management architecture that supports advance reservations and co-allocation, in: Proceedings of the International Workshop on Quality of Service (IWQoS), 1999.

[20] OSGi$^{TM}$ Alliance, About the OSGi service platform, http://www.osgi.org/documents/osgi_technology/osgi-sp-overview.pdf (2004).

[21] S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, B. Zhao, The Ninja architecture for robust Internet-scale systems and services, Journal of Computer Networks 35 (4).