Resource-Controlled Remote Execution to Enhance Wireless Network Applications

Travis Newhouse and Joseph Pasquale
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093–0114
{newhouse,pasquale}@cs.ucsd.edu

***Abstract*—**
We present the design and implementation of a system that provides resource-controlled execution environments for client and server application functionality. The system supports a remote execution model that "extends" a client or server endpoint by allowing either to insert functionality at a point along the communication path between the two endpoints. This is especially useful for wireless clients, as resource limited clients can take advantage of nearby processing power, and for clients that access the Internet via a fast wireless LAN, servers can move services close to the base station to reduce latency and improve bandwidth. The system operates entirely at user-level and so is easily deployable, and it supports quality of service in the form of rate-based resource reservations.

I. 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 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. Further, 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 insert functionality into the network. Consider 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.

Both examples benefit from the ability to select a node between two endpoints and to 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 effects of the Internet's best-effort design. To support improved quality of service, a node thats hosts functionality must provide guarantees on resource availability.

In this paper, we present the design and implementation of a system that provides a resource-controlled execution environment for client and server application functionality. Because many networked systems conform to the client/server model, in which a client sends a request to a server which then computes and sends back a response, our system design supports a remote execution model that "extends" the client or server endpoint by allowing either to insert functionality at a point along the communication path between the two endpoints. For maximum flexibility, the extended endpoint chooses the code to execute at the intermediate node, and the intermediate node dynamically loads the code at runtime (Figure 1).

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

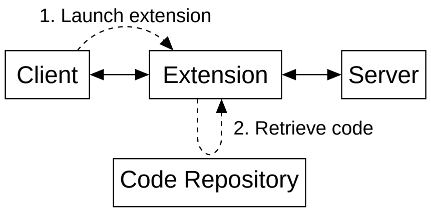

Fig. 1. The Extension Model

spawned by an endpoint acts directly as a server rather than as an intermediary, and operates in tight coordination with its creating endpoint to provide additional or other resources than those locally available. For example, a mobile device can reduce power consumption by offloading computation to a private server.

The rest of the paper is organized as follows. In Section 2, we present the extension execution model. In Section 3, we describe the system architecture, followed by descriptions of its implementation in Section 4 and a performance evaluation in Section 5. In Section 6, we discuss related work, and present conclusions in Section 7.

II. Extension Execution Model

For the remainder of this paper, we use the term *endpoint* to refer to either a client or a server that makes use of the extension model for remote execution. An *extension* is a unit of code that implements the functionality that an endpoint executes at a remote node. We define the term *extension system* to be a remote service that executes code on behalf of an endpoint. An extension system provides isolated execution environments in the form of *extension servers*. An extension server represents a share of an extension system's processor resources dedicated to executing functionality for an endpoint.

A. Execution Model

The execution environment provided by an extension server is homogeneous across network nodes, even though it may be provided by nodes with heterogeneous hardware and system software. This design decision enables providers of computational resources to evolve their execution environment (e.g. host operating system) without impacting how endpoints gain access to resources. We achieve homogeneity by expressing the functionality of an extension using a intermediate runtime language [1], [2], and implementing extension servers using the language's execution environment. Our particular implementation is based on Java (Section IV).

The type of code mobility provided by our extension model is a single-hop, move-and-execute mechanism. An endpoint may load one or more extensions to an extension server that it has allocated. The extension server retrieves the code from a network location specified by the endpoint (code need not come from the endpoint) and creates a thread of execution for the extension. An extension's execution begins at a pre-defined entry point in the code (similar to the `main()` function of a traditional application) and continues until the extension voluntarily returns from the entry function or the extension server's resources are reclaimed (described below). Aside from requiring a pre-defined entry point, our system design imposes no further constraints on an extension's code.

B. Resource Model

An extension system 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 to potential endpoints. To acquire an extension server, an endpoint makes a request that specifies its desired share of the extension system's processor resources. The request for processor resources is made in the form of quanta per period of time ($\langle quanta, period \rangle$). This tuple simultaneously expresses a rate of execution and a periodic deadline by which the rate will be satisfied. If a request is granted, the extension system ensures that the extension server does not consume processor resources at a rate greater than the requested rate. All extensions loaded to an extension server contribute to the resource consumption of that extension server.

When granting a resource request, an extension system provides the endpoint a lease [3] on the extension server. Leases benefit both endpoints and extension systems. To an endpoint, a lease guarantees availability of an extension server and its associated resources for the duration of the lease. For an extension system, a lease enables macro-scheduling of resources and reclamation of resources reserved by failed endpoints.

Our design does not specify policies for resource allocation, quantum length, or lease renewal. Instead, each extension system has a site-specific policy that governs such decisions. Upon receiving a request for an extension server or a lease renewal, an extension system consults the local policy before taking action to grant or deny the request.

C. Communication Model

Because we recognize that no single communication mechanism will likely meet the needs of a variety of existing and future distributed applications, our design does not mandate the manner in which extensions communicate with endpoints. On the other hand, so that extensions and endpoints can bootstrap application-specific communication channels, an extension server supplies the endpoint with a simple message-passing mechanism supporting delivery of arbitrarily-formatted messages to and from the extension (Figure 2). Message delivery follows in-order and at-most-once semantics, with message queues maintained by the extension server. Our design does not specify any performance requirements for the mechanism.

D. Usage Scenario

To illustrate the features of our design, we present a simple example of how an endpoint might use an extension system. In this example, the endpoint is a mobile device that wishes to view captions for the audio portion of a video clip. The video encoding does not already contain captions, so the application

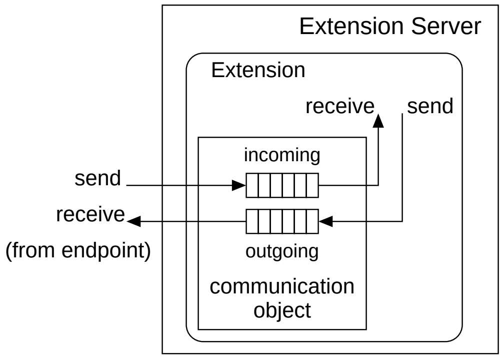

Fig. 2. Message-passing mechanism provided by the extension server

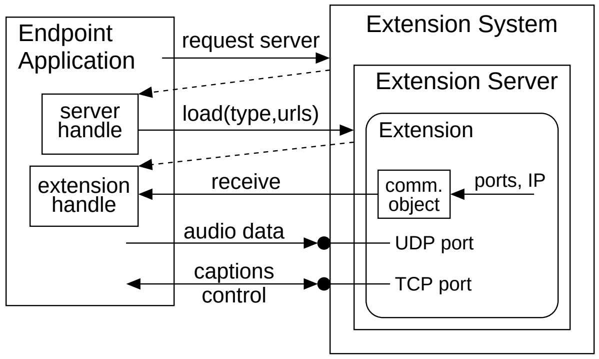

Fig. 3. Using an extension to generate captions

needs to 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 (Figure 3).

To begin, the application first locates a nearby extension system. The application wants the extension system to be "close" to the device to reduce the round trip latency for sending audio data and receiving the captions. Once an extension system has been located, the application requests an extension server with enough processing resources to support the task of speech recognition. In response, the extension system returns a handle that the application can use to load an extension to the extension server.

The application launches an extension by specifying to its allocated extension server the type of extension to create and a location, in the form of a set of URLs, where the extension server can retrieve the extension's code. The extension server loads the code from the network and begins execution of an instance of the specified extension type. The extension server returns to the application a handle to the loaded extension so that the application can establish necessary communication channels.

Upon starting, the extension opens two network ports, a UDP port to which the application 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 application a message containing the ports the extension was able to open and the IP address of the machine on which it is executing. 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 application receives the message containing the extension's port numbers and address. The application begins streaming audio data to the extension using UDP datagrams. The extension converts the audio to text that it sends back to the application using the TCP connection. When the application finishes playing the video clip, the application informs the extension by sending a control message over the TCP connection. The extension closes the ports it had opened, and exits. The application also informs the extension system that it can reclaim the resources assigned to the application's extension server.

III. Extension System Architecture

An extension system exposes two primary components with which endpoints interact. Dividing the architecture into two primary components supports scalability of hardware resources and maintains a separation between policy and execution mechanism. An *extension server* is responsible for executing one or more extensions. The extension server is also the principal for resource scheduling. All extensions executing within an extension server contribute to that extension server's resource usage. An extension system's *manager* component enforces a locally-defined policy for resource sharing and lease renewal. An endpoint requests resources from the manager, which grants requests by creating an extension server for use by the endpoint. The manager schedules resources among allocated extension servers to fulfill the resource sharing agreements it makes. The public interface to an extension system consists of operations invoked via remote procedure calls on the manager and the extension server. Figure 4 depicts the primary components of an extension system.

A manager supports two operations. The `allocate` operation requests a new extension server with a desired share of processor resources. Processor resources are requested with the tuple $\langle quanta, period, duration \rangle$, where quanta and period are as before, 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. The lease handle is used when renewing the lease associated with an extension server. The `deallocate` operation releases an endpoint's interest in the extension server referenced by the extension server handle passed as a parameter. When the extension server is deallocated, all extensions executing in the extension server are destroyed, and any resources reserved for the extension server are reclaimed.

The interface to an extension server contains a single operation to load an extension. The `load` operation accepts the

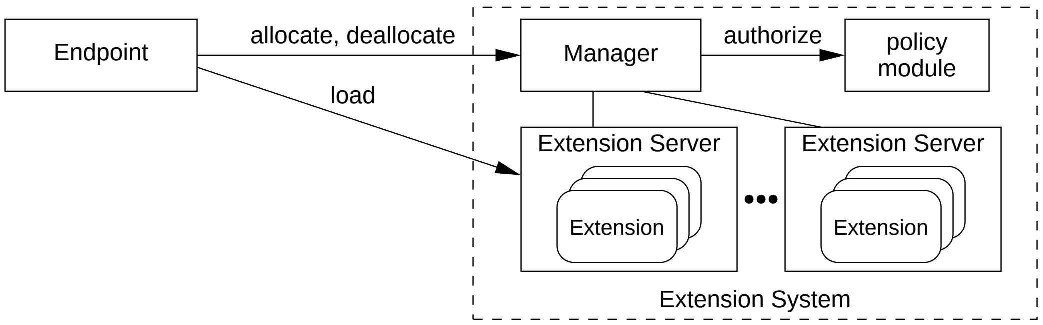

Fig. 4. Extension system architecture

type of an extension, initialization parameters, and the location where the extension's code resides. The extension server retrieves the code, creates an instance of the extension, and returns a handle by which the endpoint can interact with the extension. The handle enables the endpoint to communicate with the extension using the message-passing interface.

A third component, hidden from the perspective of an endpoint, is the locally defined *policy module*. A manager consults the policy module before granting resources or renewing leases. The site administrator must provide a policy module that implements an `authorize` operation. The `authorize` operation inputs a resource request tuple, and outputs a boolean value as to whether the request is permitted.

The architecture supports scalable and distributed hardware resources because it enables a hierarchical arrangement of managers for policy resolution without maintaining a chain of managers in the critical path of extension loading and execution. For example, a provider of distributed execution resources can create a hierarchy of managers in which a top-level "brokerage" manager makes requests to bottom-level "provider" managers running on the nodes possessing available resources. After an endpoint obtains a resource reservation in the form of an extension server, the endpoint interacts directly with the extension server to load extensions, bypassing any chain of managers through which its initial resource request was routed (Figure 5).

IV. Implementation

We have implemented our design using Java [1]. The Java Virtual Machine (JVM) provides a homogeneous execution environment that operates on a variety of platforms, and is the foundation of the execution environment provided by an extension server. The compiled format of extension code is that of machine-independent Java bytecodes. The JVM's high-level execution environment and the portability of Java bytecodes ensure that an extension can execute at any extension system, regardless of the underlying hardware. In this section, we describe some of the details of our implementation.

A. Component Handles

Interactions between an endpoint and an extension server do not use a pure remote procedure call mechanism. Instead, we take a middleware approach that hides the underlying communication mechanisms. Endpoints perform operations on extension system components by making method invocations on local objects, or *component handles*, that represent a remote entity (Figure 6). The local handle objects manage the communication duties between the endpoint and the extension system. In this way, endpoint applications interact with a consistent API, while the underlying communication mechanism can evolve with the implementation. In our current implementation, we use Java RMI as the communication substrate.

Each component handle contains sufficient information and functionality to contact its corresponding extension system component directly. Furthermore, an endpoint can transfer the handle to another node without loss of functionality. This provides endpoints with flexibility in how they use an extension system.

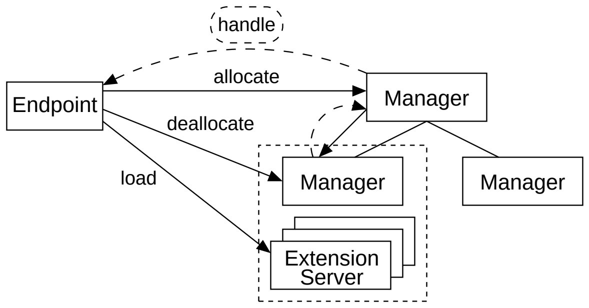

Fig. 5. Hierarchy of managers

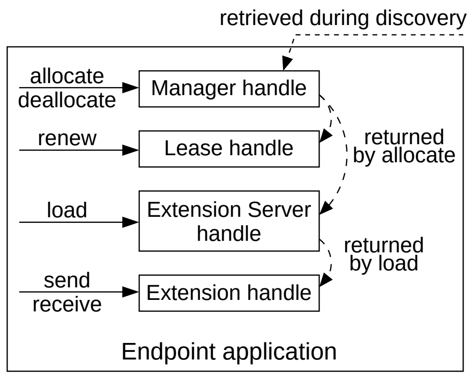

Fig. 6. Interface provided by local component handles

For example, 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 an endpoint to share permission to execute on its 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.

With respect to an extension server handle, one node may allocate an extension server, and yet transfer the handle to another node. This supports the creation of transitive resource sharing schemes that allow an endpoint to use an extension server for execution even if the endpoint does not itself have sufficient permission to allocate an extension server. For instance, managers may be arranged hierarchically, with each manager invoking the `allocate` operation on the manager at the level below it (Figure 5). When a bottom-level manager is reached, it returns an extension server handle that gets passed up the tree until it reaches the endpoint. The endpoint receiving the extension server interacts with the extension server directly. We are currently investigating such ideas for use in a security and authorization model for the extension system.

B. Resource Control

To maintain isolation and to control resources assigned to individual extension servers, our manager implementation launches a distinct JVM for each extension server 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. In particular, classloaders and static class variables are protected, and class-level monitors cannot become points of contention. For processor resource control purposes, the JVM serves as the resource principal scheduled by a user-level scheduler we have developed. The user-level scheduler uses mechanisms commonly supported by UNIX-like operating systems.

When the manager creates a new extension server, it registers with the user-level scheduler a rate of execution and the process identifier of the new JVM. The user-level scheduler executes external to all JVMs, including the manager, and monitors each extension server's processor consumption (Figure 7). Using UNIX signals, the scheduler starts and stops a JVM's execution to ensure that it does not exceed its share of the processor.

Mechanisms for creating isolated, resource-controlled environments *within* a JVM are currently being developed [4], [5]. If and when such mechanisms are made available, we can incorporate them into our implementation without modification to our extension system interface.

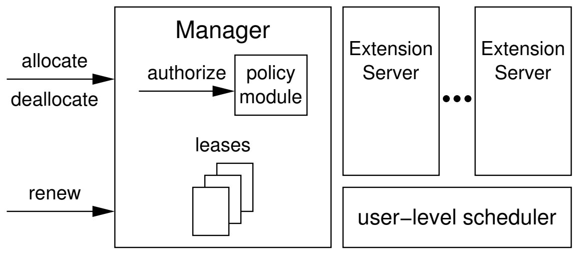

Fig. 7. Manager enforces resource sharing policy using a user-level scheduler

V. Programming With Extensions

To execute functionality at an extension system, an endpoint must perform three steps:

- discover an extension system,
- request an extension server with desired resources,
- load an extension to the extension server for execution.

Once an extension is loaded, the endpoint may communicate with the extension using the simple message-passing mechanism provided by the extension server. If an endpoint wishes to use an extension server for longer than the duration granted in the original resource request, the endpoint must periodically renew the lease associated with the extension server.

Code loaded as an extension must conform to a minimal interface that contains a single method named `run`. This method defines the entry point of an extension and is implemented by the extension's developer. To begin execution of a loaded extension, an extension server calls the extension's `run` method, passing a set of resource objects as a parameter. These resource objects 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 handle returned to the endpoint upon loading the extension.

The message-passing communication interface consists of two operations. A message is an arbitrary sequence of bytes; it is up to the extension's developer to choose a suitable format. 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 Figure 2. The communication object provided to an extension by the extension server maintains two message queues per extension, one for messages sent to the extension and one for messages sent to the extension's handle.

Our design does not currently define how an endpoint initially locates an extension system. We make this design decision to maintain flexibility for future uses of the system. We have, however, experimented with two discovery techniques. The simplest uses sockets and an out-of-band information publishing scheme, such as a Web page that lists the address and port number of the extension system. A more decentralized approach that we have explored uses Jini Network Technology [6]. Jini provides a multicast discovery protocol to locate directories of registered services. The extension system reg-

```
// retrieve extension system's manager
ManagerLocator loc =
  ManagerLocator.create(host, port);
Manager m = loc.getManager();

// allocate resources
int q = 5;      // quanta requested
int p = 20;     // period
int d = 10;     // 10 minutes
ServerAllocation a = m.allocate(q,p,d);
ExtensionServer es = a.getServer();

//launch the extension
URL[] urls = { new URL(
  "http://myserver.com/mycode.jar") };
ExtensionHandle handle =
  es.load("MyExtension", null, urls);

// print message from the extension
System.out.println(handle.receive());
```

Fig. 8. Endpoint code to load an extension

isters itself with any "nearby" directories. Endpoints wishing to use an extension system use the same multicast protocol to locate nearby directories that can be searched for a registered extension system service.

A. Programming Example

To illustrate the simplicity of the extension system interface, we present an example program that loads an extension. The example illustrates how an endpoint interacts with an extension system. The endpoint code is only a few lines long, as shown in Figure 8.

The first step is discovery, in which the endpoint acquires a handle to the manager of the extension server. The manager handle is an object on which the endpoint can make local method invocations to allocate or deallocate an extension server. In this example, the endpoint uses a `getManager()` method to directly contact the extension system at the specified hostname and port. The details of this method are specific to the discovery technique.

Next, the endpoint uses the manager handle to allocate an extension server. 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. For now, we ignore leasing. If the allocation request is denied, then the `allocate()` method throws an exception (not shown).

With a handle to an allocated extension server, the endpoint is ready to load an extension. It passes to the extension server handle's `load()` method: a class name, initialization parameters (none in this case), and an array of `URL`s pointing to the locations of the extension's code. The `load()` method sends the parameters across the network to the extension server. After the extension has been instantiated, the extension server returns 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 a message from the extension.

```
class MyExtension
  implements Extension, Serializable
{
  public void run (Resources r)
  {
    ExtensionCommunicator comm =
      (ExtensionCommunicator)
      r.getByType(
        ExtensionCommunicator.class);

    // send a message back to owner
    comm.send("Hello world.");
  }
}
```

Fig. 9. Code for a simple extension

The code for the `MyExtension` class is listed in Figure 9. A class that is to be 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 the collection of resource objects from which the extension obtains an object providing access to the message-passing communication channel. The sample extension sends a `String` to the holder of its extension handle.

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.

VI. 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 the following: improved reliability; improved performance, in terms of bandwidth, latency, execution time; 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.

TABLE I

TIME TO LAUNCH AN EXTENSION (MS)

	Mean
Locate manager (Jini)	770 ± 1
Locate manager (socket)	542 ± 1
Allocate extension server	764 ± 4
Load extension	315 ± 3

A. 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 tests ran on machines 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. Thus, the results represent worst case times.

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. Our 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 might 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 I lists the mean operation time with a 99% confidence interval. The total time to launch an extension is less than 2 seconds 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).

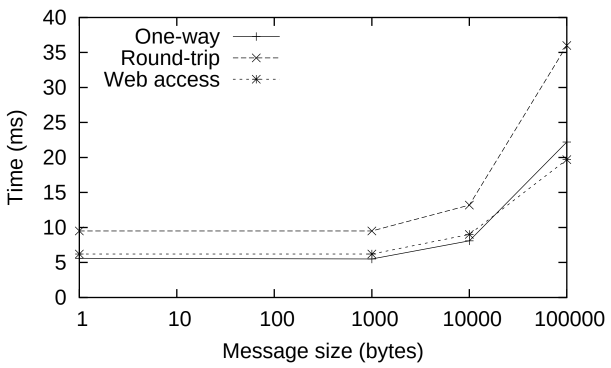

Fig. 10. Message passing times compared to local Web requests

B. Sending Messages

While our design does not prescribe performance guarantees for the message-passing communication mechanism, we show that even a simple implementation of the mechanism has acceptable overhead. Like the rest of our 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. Figure 10 shows the times for message sizes of 1, 1000, 10000, and 100000 bytes. As a point of reference, we include on the graph a plot of the time to retrieve files of equivalent size from a local Web server. The one-way message cost is comparable to the Web request time. Thus, the communication mechanism exhibits acceptable cost, especially considering it is intended for bootstrapping application-specific communication.

C. 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 on the accuracy and overhead of our user-level scheduler to demonstrate the feasibility of our approach. For synthetic compute-bound workloads, we are capable of enforcing shares with less than 2% relative error and with not more than 3% runtime overhead for up to 32 processes. To test workloads that involve both processing and I/O, we tested the ability of our scheduler to control shares of 3 Apache Web servers configured to use 50 processes each. Our scheduler was capable of enforcing proportional share throughput within 1% relative error using only 3.2% of the total processing time.

VII. 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 are 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 untrusted code.

A. 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 seamlessly integrate the components with existing client and server frameworks, and how to dynamically distribute services for scalability. Our work is 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 [7] and edge services [8] 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.

Both the Web& system [9] and Chroma [10] split application functionality between an endpoint and a network node. Beyond these capabilities, we focus on providing protection and 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 [11]. The mobile server extension can execute at the client, server, or a proxy node. Our extension 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 [12]. The execution environment they propose for intermediate nodes is tailored to their composable, component-based model. Our extension 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) [13] 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.

B. Hosting Untrusted Code

While our extension system is a user-level approach to hosting untrusted code, others have taken a low-level approach using a virtual machine monitor (VMM) to create strongly isolated execution environments. Denali [14] 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 untrusted network services. 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 untrusted code. The Open Grid Services Architecture [15] 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 [16] enables an application to reserve collections of resources for end-to-end QoS. We focus on providing the computational resources to execute endpoint-supplied functionality at a particular node. We define general resource sharing and execution model that is suitable for the execution of endpoint-supplied code.

Finally, a Ninja [17] "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 only across a cluster of workstations. Another component of Ninja, "active proxies," provides adaptation by dynamically inserting "operators" that transform data along a communication path. Our extension system provides a general, resource-controlled execution environment that can support the deployment of higher-level execution models.

VIII. 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. We have designed and implemented a remote execution system that loads remote code, and executes the code with a pre-allocated, rate-based share of the processor. The system's interface is simple and minimal in order to support a variety of client and server applications without constraining their design. The system is implemented entirely at user-level to support ease of deployment, and to abstract variations in the underlying systems of computational resource providers.

For future work, we are currently extending our design to include a security framework that will operate in tandem with the site-specified policy module to enable site administrators to define a secure authorization scheme. We are also investigating user-level mechanisms for resource control of additional resources, such as network bandwidth and storage.

References

[1] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java^{TM} Language Specification*, 2nd ed. Boston, MA: Addison-Wesley, 2000.

[2] E. Meijer and J. Gough, "A technical overview of the common language infrastructure," http://research.microsoft.com/∼emeijer/Papers/CLR.pdf, 2001.

[3] C. G. Gray and 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.

[4] 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.

[5] G. Czajkowski, S. Hahn, G. Skinner, P. Soper, and C. Bryce, "Resource management interface for the Java^{TM}platform," Sun Labs, Tech. Rep. TR-2003-124, May 2003.

[6] Sun Microsystems, "Jini Network Technology," http://www.sun.com/software/jini/, 2004.

[7] B. Zenel and 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.

[8] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and 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.

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

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

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

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

[13] J. W. Stamos and D. K. Gifford, "Remote evaluation," *ACM Transactions on Programming Languages and Systems*, vol. 12, no. 4, pp. 537–564, 1990.

[14] A. Whitaker, M. Shaw, and 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, Dec 2002.

[15] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke, "Grid services for distributed system integration," *Computer*, vol. 35, no. 6, 2002.

[16] I. Foster, C. Kesselman, C. Lee, B. Lindell, K. Nahrstedt, and 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.

[17] 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, and B. Zhao, "The Ninja architecture for robust Internet-scale systems and services," *Journal of Computer Networks*, vol. 35, no. 4, March 2001.