Technical Report 98-1715. Department of Computer Science, Cornell University, November 1998.

Design and Evaluation of an Extensible Web & Telephony Server based on the J-Kernel

Daniel Spoonhower, Grzegorz Czajkowski, Chris Hawblitzel,
Chi-Chao Chang, Deyu Hu, and Thorsten von Eicken
Department of Computer Science, Cornell University

Abstract

This paper describes the design and performance of the J-Server, an integrated web and telephony server that allows untrusted Java servlets to be dynamically uploaded to extend the server's functionality. The J-Kernel provides for protection and communication between J-Server servlets, and ensures that servlets can be cleanly terminated. A resource monitor called JRes is used to account for servlet resource usage. Two sample applications show that the overhead of J-Kernel task boundary crossings is small compared to the applications' overall running time. Experience developing applications for the J-Server demonstrates the benefits of extensible systems based on safe language protection, and the flexibility of the servlet model.

1 Introduction

The ongoing merger of networking and telephony is bringing a new wave of commodity telephony equipment designed to be controlled by a standard workstation. Examples are telephone interface cards with two to 24 analog telephone lines and the DSPs to handle real-time tasks, and full telephone switches (PBXs) that can be controlled over a LAN. On the software side, standard telephony APIs (e.g., Windows' TAPI or Java's JTAPI) allow programs to control the equipment in a relatively portable way.

The upshot of these developments is that today a single commodity PC can provide integrated communication services that use both the Internet and the telephone system. This is of great benefit to businesses, many of which already use the web, fax servers, and telephone lines almost interchangeably to communicate with customers. Integrating the different communication media into a single server not only reduces duplication of effort and data; it also enables integrated applications. For example, technical support messages sent via email, web forms, 800 numbers, and faxes can all be handled by the same management system. In addition, communication can cross media, such that a voice message received over the telephone can be retrieved through a web interface or via email. An integrated server can also connect media in real-time, allowing a conference call or a technical support call to use the internet for documents and images and the telephone for voice.

Telephony technology and applications are developing rapidly, and the demands of users differ significantly. System integrators need to be able to extend products to add value, system administrators need to add local features, and end users should be able to personalize their view of the system, perhaps by purchasing third-party software. It appears that at this point, an integrated server needs to be highly flexible accommodate varying and unexpected types of applications.

This paper describes a prototype integrated web and telephony server called the J-Server, which uses safe language technology to support the extensibility necessary in this setting. The J-Server core manages a hierarchical namespace of resources where each resource can correspond either to a URL or to a telephone number. Users of the system can upload programs (called servlets) to the J-Server and attach each servlet to one or more resources. The servlets can then handle all HTTP requests and phone calls that are sent to the resources. Furthermore, servlets may communicate with one another, so that one servlet may provide a service to another servlet. For example, a servlet that uses speech recognition to determine the nature of a call might need to dispatch the call to another servlet once it is clear who the call is for.

Several existing web servers allow extensions to be loaded dynamically. For instance, Jigsaw [W3C] allows extensions written in Java to be loaded dynamically to customize the server. While this extensibility is very useful, when using Jigsaw we found that Jigsaw does not have a strong enough security model. For instance, any servlet can stop the entire server by calling `System.exit()`. More subtly, it is impossible to cleanly unload a broken servlet, making servlet development painful at best. Sun's Java Web Server [Jav-a], which introduced the servlet API, has a stronger notion of security, but it lacks a clear model for inter-servlet communication. If one servlet wants to use classes belonging to another servlet, the second servlet's classes typically must be implemented as *system classes*, which cannot be unloaded without shutting down the entire server. In general, our experience with using these servers has shown that they place high demands on the extension mechanism:

- Protection is important because multiple users create their own independent webs.
- Failure isolation is important because webs are continuously expanded and new features must not disrupt old ones.
- Clean servlet termination is essential.
- Extensions must be able to call other extensions, and these cross-domain calls should be cheap so that they can occur frequently.
- Resource management and accounting are necessary.

These demands motivated the development of the J-Kernel [HCC+98], which is an operating systems layer for Java that underlies the J-Server, and JRes [CvE98], which is a resource accounting system for Java. The J-Kernel provides well-defined protection domains (called *tasks*) with a clean task creation and task shutdown model, and a clear inter-task communication model based on capabilities. In contrast to the Java Web Server, servlets do not have to sacrifice dynamic loading and unloading to communicate with one another.

The rest of the paper is structured as follows. Section 2 provides more background on the J-Kernel and JRes, and on the telephony equipment used by the J-Server. Section 3 demonstrates how the J-Kernel fits in with Sun's servlet API, and walks through the processing of an HTTP request as it passes through various tasks. Section 4 shows how the J-Server extends the servlet model to handle telephony, and develops a broader picture of an extensible integrated server's architecture. Measurements presented in Section 5 show how often cross-task calls occur, what the size of the arguments is, and how the J-Server performs overall. Section 6 discusses related work, and is followed by conclusions.

2 Background

The background information on our prototype integrated web and telephony server falls into two categories: an introduction to the telephony-related hardware and software used in the implementation, and introduction to relevant issues concerning Java and the J-Kernel.

2.1 Telephony

A rapidly increasing variety of telephony hardware is available to connect to PCs. In our J-Server setup (see Figure 1), we use two off-the-shelf products: Lucent Technologies DEFINITY® Enterprise Communications Server and the Dialogic Dialog/4 voice interface. At the core, the DEFINITY server is a Private Branch Exchange (PBX), i.e. a telephone switch, augmented by an Ethernet-based network controller. Traditional telephone switches map telephone addresses to physical terminals and appropriately route calls using this mapping. The additional network controller in this PBX allows the routing to be monitored and controlled from a PC over the LAN. The PC is a Windows NT Server running an NT service provided by Lucent that communicates with the PBX and that implements the Windows Telephony Services API (TSAPI). Lucent also provides a Java to TSAPI bridge so that Java applications can control the PBX using the Java Telephony API (JTAPI) [Jav-b].

The JTAPI interface exposes resources in the telephony system in the form of objects corresponding, for example, to telephone numbers (logical addresses) and physical endpoints (terminals). Java applications can invoke methods on these objects to configure the associated resources and to initiate actions such as starting a call. Incoming events, such as a telephone call, a line being picked-up, etc., are signaled through Java's Observer/Observable model: an application registers a callback with the JTAPI object and that callback is invoked with parameters describing the event. Based on this, a Java application can monitor incoming calls, decide how they should be handled, and route them appropriately.

The Dialog/4 Voice Processing Board, from Dialogic Corporation, provides four telephone interfaces on a half-size ISA card. Each board includes a digital signal processor (DSP) to play and record audio, and to detect and transmit DTMF (dual-tone multi-frequency) signals. Two Dialog/4 boards give our server access to eight telephone lines. The programming interface for the Dialog/4 consists of a proprietary API accessible from C. To interface to Java, a wrapper library was implemented in C, utilizing Microsoft's JDirect native interface to call the native library functions from Java. With the Dialog/4 boards a Java application can initiate and receive calls, play and record sound, and send and receive DTMF digits.

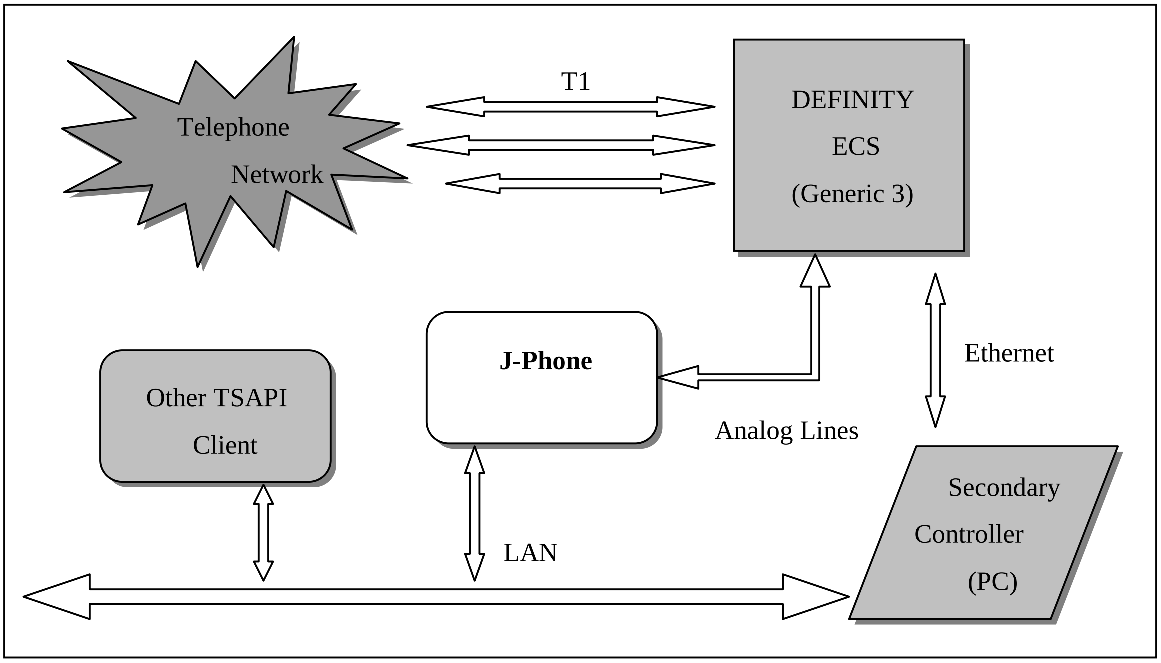


Figure 1. J-Phone hardware setup.

2.2 The J-Kernel

Safe languages such as Java provide several simple and flexible protection mechanisms: (i) references (pointers) to objects cannot be forged, and can therefore be used as capabilities, (ii) access modifiers like `private` and `public` restrict which actions certain code can perform on objects, and (iii) dynamic loading and linking mechanisms control which classes are visible to a particular piece of code.

Nevertheless, safe languages lack features found in traditional operating systems. None of the protection mechanisms listed above provide support for revocation; for example, once someone has a reference to an object, that reference cannot be revoked. Traditional operating systems provide a notion of a task or process, which encapsulates all of the memory and processor time used by a program. Java lacks such a concept; in Java, all programs' objects go into a single heap, with no clearly defined boundaries between one program's objects and another's. This makes it difficult to hold a program accountable for the memory it uses, or to free up a program's memory when the program must be shut down.

2.2.1 Tasks, Capabilities and Cross-Task Calls

The J-Kernel adds *tasks* to Java, and makes a strong distinction between objects that can be shared between tasks, and objects that are confined to a single task. The only objects that can be shared between tasks are *capabilities*, which are stub objects generated by the J-Kernel. Capabilities are essentially wrappers around ordinary objects. Internally, a capability holds a private pointer to an ordinary object, and all method invocations on the capability are forwarded to this internal object. Capabilities can always be revoked simply by nullifying the internal pointer inside the capability. When a task is shut down, all the capabilities that it allocated are revoked, so that all of the task's memory becomes inaccessible.

Since only capabilities are shared between tasks, capabilities form the communication channels through which cross-task communication occurs. Capabilities form clear boundaries between tasks: a method invocation on a capability may switch to another task (for this reason we refer to invocations on capabilities as *cross-task calls*), but an invocation on an ordinary object does not switch to another task. Knowing exactly where cross-task calls occur allows the run-time system to insert extra code into a cross-task call (such as code to manage threads or perform resource accounting), and makes it easier for the programmer to reason about the security of the system as a whole.

To enforce the property that only capability objects can be shared between tasks, method invocations on capabilities use a special calling convention. While capabilities can be passed by reference as arguments to a cross-task call, ordinary objects are passed by copy through such an invocation (deep copy). This is very similar to the convention used in remote method invocation (RMI), and we borrowed elements of Sun's RMI specification for the J-Kernel's capability interface. To create a capability, a program must first write a class that implements one or more *remote interfaces*, where a remote interface is any Java interface that extends the class `Remote`. A capability is created as a wrapper around an object that implements one or more remote interfaces; this capability then implements all of the methods declared in the remote interfaces implemented by the object that the capability wraps. Examples of how this works are described in sections 3 and 4.

2.2.2 Resource Management

The term *resource management* is a convenient way of talking about several related issues: resource accounting, resource usage constraining, guaranteeing resource availability, providing resource consumption information, and resource scheduling. Providing adequate levels of all of them is a requirement in an extensible system.

Accounting for resources is necessary for billing users/organizations and for diagnostics purposes. Failing to provide means of constraining resources available to extensions leads to problems with malicious extensions that monopolize use of particular resources. Guaranteeing resource availability enables time-critical services. Resource consumption information is useful for extensions that can trade one resource for another in order to maintain a certain level of quality of service in the presence of resource load fluctuations [CCH+98]. For instance, extensions transferring large amounts of data across the network may choose to either compress before sending or send the data verbatim, switching the load between CPU and network.

Despite the need for providing resource management facility, general-purpose safe languages fail to address this issue. This is rather surprising in view of the fact that runtime environments of safe languages such as Java aspire to provide operating-system environments. The lack of resource management prevents a safe language runtime environment from subsuming the role of a traditional operating system.

We have addressed this problem by designing and implementing JRes - a Java interface for accounting for heap memory, CPU time, and the number of bytes sent and received over the network on a per thread and per task basis. JRes provides mechanisms to set limits on the resources available to particular threads and tasks and to associate *overuse callbacks* that are invoked whenever any such resource limit is exceeded [CvE98]. Although the interface is simple, it is flexible enough to support a variety of resource consumption enforcement policies. For instance, a policy which ensures that no thread gets more than 100 milliseconds of CPU time out of every second is easily expressible. Similarly, a few lines of code suffice to create a policy in which no task can send more than 2MB of data, or a policy in which the combined size of all live objects allocated by a task does not exceed 1MB.

3 An Extensible Web Server Architecture

The architecture of the extensible web server component of the J-Server grew out of our experience in using the Jigsaw server (version 1.0) developed by the World Wide Web Consortium and Sun's Java Web Server (JWS, version 1.01). Both servers are extensible but offer very different degrees of flexibility: Jigsaw's extension interface is highly flexible, but difficult to use, while Java Web Server's is the relatively simple Servlet API. The main problems we faced with both servers were the lack of protection, of failure isolation, and of clean extension termination. When running a production server these features are essential: restarting the server to fix a problem with one extension is rarely possible.

The J-Server divides the components of the web server into many tasks. The server core, called J-Web, runs in a privileged task that listens to connections on the HTTP port, receives requests, dispatches them to appropriate servlets, and returns the generated responses. Each servlet runs as a separate task to isolate it from the J-Web core as well as from other servlets. Requests are passed from J-Web to a servlet using a cross-task call which returns the response from the servlet. This setup also allows new servlets to be introduced into the system or crashed ones to be reloaded without disturbing the server operation. It also allows J-Web to track the resource consumption of each servlet.

3.1 The Servlet API

The API for the J-Web servlets is based on Sun's Servlet API. The pivotal class in the servlet API is an interface called `Servlet`, which declares methods for initializing and cleaning up a servlet, processing a request to a servlet, and configuring the servlet:

```
public interface Servlet {
    void init(ServletConfig config) throws ServletException;
    void destroy();
    void service(ServletRequest req, ServletResponse res)
        throws ServletException, IOException;
    ...
}
```

When a request comes to the servlet, the service method is invoked and provided with *request* and *response* parameters. These parameters encapsulate the data sent by the client, providing access to parameters and allowing servlets to report status including errors. Servlets can obtain an input stream to read data from the request and an output stream to send data back in the response.

Servlets may handle many types of requests (e.g. FTP, HTTP, etc.). For HTTP requests, the API defines a class called `HttpServlet` that implements `Servlet`, and forwards HTTP requests to a more specific service method that deals with `HttpServletRequests` and `HttpServletResponses`:

```
public abstract class HttpServlet implements Servlet {
    public void service(ServletRequest req, ServletResponse res)
        throws ServletException, IOException {...}
    public void service(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {...}
    ...
}
```

User-defined servlets may then extend `HttpServlet` and override the more specific service method to process HTTP requests.

3.2 Adapting the Servlet API to the J-Kernel

Adapting Sun's original servlet API to the J-Kernel required clearly identifying the boundary between the server and the servlet, in order to establish the necessary remote interfaces. Luckily, the `Servlet` interface already marks this boundary, so the simplest approach is to turn `Servlet` into a remote interface. The only issue is that the `service` method takes the `ServletResponse` as an argument, and the servlet communicates back to the server through side effects on the `ServletResponse` argument. If `ServletRequests` and `ServletResponses` are passed by copy, this won't work — side effects won't be propagated back to the server. Therefore the J-Server uses a modified `Servlet` interface which returns the `ServletResponse` as a return value:

```
public interface Servlet extends Remote {
    void init(ServletConfig config) throws RemoteException;
    void destroy() throws RemoteException;
    ServletResponse serviceInternal(ServletRequest req, ServletResponse rep)
        throws RemoteException;
    ...
}
```

The implementation of `serviceInternal` simply forwards calls to the standard `service` method.

Once a servlet is written, it can be dynamically loaded into the J-Server and attached to a location in the server's URL space. The interface to the dynamic loader is provided by a privileged servlet called the *servlet loader*. The servlet loader is started by the J-Web core and holds a capability that allows it to register new servlets in the URL tree. The servlet loader's `service` method returns a simple HTML form into which a user can enter the information required to load a new servlet. The servlet loader launches a task for the servlet to execute in. In the new task, the loader creates a capability for the servlet based on the `Servlet` remote interface implemented by the servlet class. This capability forms the channel for communication between the server and the servlet.

3.3 A Voice Mail Retrieval Servlet

The following example shows a simple voice mail retrieval servlet that runs on the J-Server. The servlet extends `HttpServlet` and overrides the `service` method to play back voice mail messages.

```
public class VoiceMailServlet extends HttpServlet {
    private WaveFile greeting;
    private String messageDirectory;
    private Hashtable currentMgs;

    public void service(HttpServletRequest req, HttpServletResponse res)
            throws ServletException, IOException {

        String action = req.getParameter( "action");

        if( action == null) {
            // Generate a list of stored voice-mail messages
            ...
        } else if( action.equals( "play")) {
            String filename = req.getParameter( "file");
            File file = new File( messageDirectory + filename);
            WaveFile message = new WaveFile( file);

            res.setContentType( "audio/wav");
            OutputStream out = res.getOutputStream();
            message.write( out);
            out.close();
        } else if( action.equals( "delete")) {...}
    }
    ...
}
```

An HTTP request sent to the voice mail servlet is processed by the J-Web task, which then dispatches the request to the servlet task. The details of this depend on how the J-Server is configured: the J-Server can be run both as a standalone web server, based on Java sockets, and as an extension to Microsoft’s Web Server (IIS). In the former case, the J-Server creates threads which block waiting for incoming connections to the sockets. In the latter case, the J-Server runs within the same process as IIS (as an in-proc ISAPI extension) where it receives requests and transmits responses through the Internet Server API (ISAPI). In this case, IIS makes a function call into the J-Server each time a request is received.

Once the J-Web task has a request to process, it parses the request's URL path to decide which servlet should handle the request. J-Web fetches the capability for the appropriate servlet from a table, and constructs an `HttpServletRequest` to pass to this capability's `service` method. The `HttpServletRequest` is a data structure containing the results of parsing the request. When control passes into the servlet's task, the servlet's `serviceInternal` method forwards the copied `HttpServletRequest` object to the user-defined `service` method. In the voice mail servlet shown above, the servlet examines the request and either plays a voice message, deletes a voice message, or returns a list of available messages.

3.4 Servlet-to-Servlet Communication

Servlets often need to contact other tasks while processing requests. For example, the voice mail servlet above communicates with the task that implements the file system in order to fetch audio data from wave files (internally, the `File` and `WaveFile` classes shown in the voice mail servlet use capabilities to talk to the file system task). This section shows an example of inter-servlet communication in more detail.

Suppose we would like to extend the voice-mail servlet to handle authentication. It makes sense to keep a database of authentication information in a separate task, since this allows the information to be shared among multiple servlets. The communication channel to the authentication task is defined by a remote interface; a sample interface based on user names and passwords, and an implementation of the interface, are shown below.

```
public interface AuthInterface extends Remote {
    public boolean verify(String user, String password) throws RemoteException;
}
```

```
public class AuthServlet extends HttpServlet implements AuthInterface {
    private Hashtable passwords;
    ...
    public boolean verify(String user, String passwd) {
        String storedPasswd = (String) passwords.get(user);
        return storedPasswd == null ? false : storedPasswd.equals(passwd);
    }
}
```

Once the interface is established, the servlet loader loads the authentication servlet. The authentication servlet then creates capabilities for accessing the `verify` method of the authentication service, and give these capabilities to other servlets, such as the voice mail servlet. The voice mail servlet calls the `verify` method on the capability to check a password:

```
String user = req.getParameter( "user");
String passwd = req.getParameter( "passwd");
boolean okay = auth.verify( user, passwd);
```

4 An Extensible Telephony Server Architecture

One challenge in designing the telephony component of the J-Server, called J-Phone, and in integrating telephony into the J-Server is to find a programming model that is appropriate for both. The Java Telephony API (JTAPI) uses a rather complex event model based on the Observer/Observable paradigm. In the context of the J-Server, many of the JTAPI and Dialogic events and actions are too hardware dependent to be useful. Instead, we wanted a higher-level API and decided to expose telephone call events to servlets in a manner similar to HTTP requests.

4.1 The PhoneServlet API

The key to using this interface for telephony lies in extending the definition of `ServletRequest` and `ServletResponse`. The servlet model was designed primarily to create HTTP extensions and thus has several limitations. First, an HTTP servlet has only one form of response: an array of bytes, usually an HTML page, and a set of well-defined headers (which allow small variations for error responses and redirection). The action associated with this response is always sending data over the network. For telephony, the actions associated with a single servlet response can vary much more widely. To accommodate this, J-Phone defines a PhoneServlet API, which broadens the servlet model by extending the request and response classes. The second limitation is that the servlet API assumes that the servlets make only one response for each request. In a telephony application, a servlet may want to initiate several actions in response to a single request. The PhoneServlet API loosens this restriction as well.

To allow for a broad spectrum of telephony activity, the PhoneServlet API defines a specific request and response type for each telephony event and action. For example, a telephony servlet may receive a `RingingRequest`, signaling that a call is pending in the system. The servlet can respond by setting the next response object to either a `SetOffHookResponse` or a `RedirectCallResponse` depending on the application.

```
if( req instanceof RingingRequest) { res.setNext( new SetOffHookResponse()); }
```

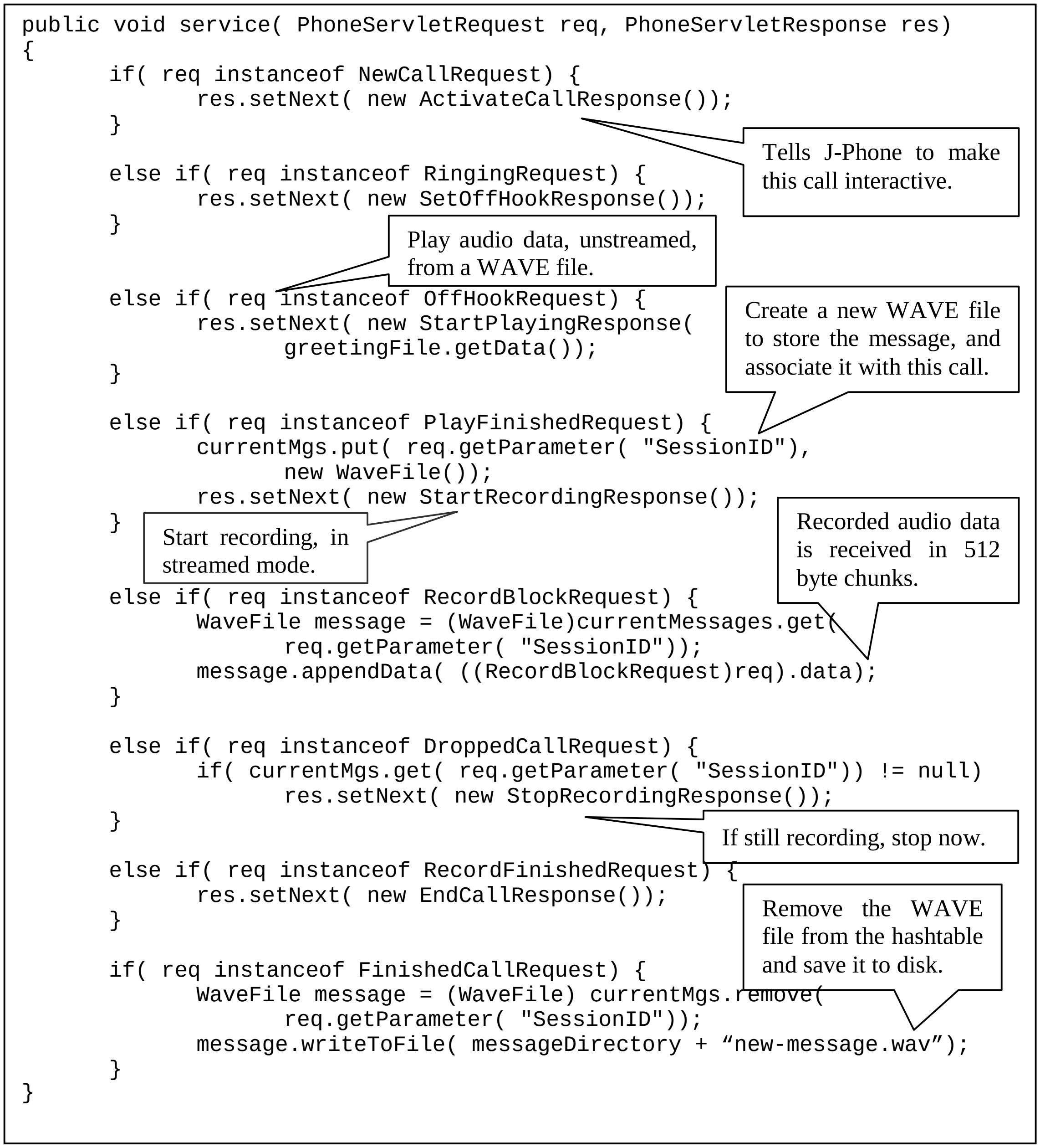
Just as a single HTTP session may encompass several invocations of the `service` method (i.e. a user may make several requests to a single servlet), a single telephone call typically results in many method calls into the telephony servlet. In this example, after the servlet tells the system to take the telephone off hook, the action is carried out and then `service` is invoked once again to notify the servlet that the action has been completed. Most servlet interaction falls into this request/response model. The PhoneServlet API also allows servlets to chain responses together, using the `setNext` method. In this fashion, a servlet may specify several responses to a single servlet request.

4.2 The Voice Mail Servlet, Revisited

A new method, given in Figure 2, is added to the `VoiceMailServlet` class from section 3.3 to complete the voice mail servlet. This method defines the behavior of the application as follows:

- All new calls are accepted.
- Once a call is connected, a greeting is played.

- Next, a message is recorded.
- When the call is dropped, the message is saved to disk.



```
public void service( PhoneServletRequest req, PhoneServletResponse res)
{
        if( req instanceof NewCallRequest) {
                res.setNext( new ActivateCallResponse());
        }

        else if( req instanceof RingingRequest) {
                res.setNext( new SetOffHookResponse());
        }

        else if( req instanceof OffHookRequest) {
                res.setNext( new StartPlayingResponse(
                        greetingFile.getData());
        }

        else if( req instanceof PlayFinishedRequest) {
                currentMgs.put( req.getParameter( "SessionID"),
                        new WaveFile());
                res.setNext( new StartRecordingResponse());
        }

        else if( req instanceof RecordBlockRequest) {
                WaveFile message = (WaveFile)currentMessages.get(
                        req.getParameter( "SessionID"));
                message.appendData( ((RecordBlockRequest)req).data);
        }

        else if( req instanceof DroppedCallRequest) {
                if( currentMgs.get( req.getParameter( "SessionID")) != null)
                        res.setNext( new StopRecordingResponse());
        }

        else if( req instanceof RecordFinishedRequest) {
                res.setNext( new EndCallResponse());
        }

        if( req instanceof FinishedCallRequest) {
                WaveFile message = (WaveFile) currentMgs.remove(
                        req.getParameter( "SessionID"));
                message.writeToFile( messageDirectory + “new-message.wav”);
        }
}
```

Tells J-Phone to make this call interactive.

Play audio data, unstreamed, from a WAVE file.

Create a new WAVE file to store the message, and associate it with this call.

Start recording, in streamed mode.

Recorded audio data is received in 512 byte chunks.

If still recording, stop now.

Remove the WAVE file from the hashtable and save it to disk.

Figure 2. Implementation of the `service` method of `VoiceMailServlet`.

4.3 Implementation

In order to keep the PhoneServlet API small and simple, much of the complexity of JTAPI and the Dialogic driver is hidden from servlets by the components of J-Phone. The details hidden underneath the interface are discussed below.

4.3.1 Privileged Tasks

J-Phone contains three privileged tasks: the PbxSubsystem, monitoring and controlling the PBX, the VoiceSubsystem, which handles the Dialog/4 interfaces, and the DispatcherSubsystem, which is responsible for queuing and dispatching telephony events (see Figure 3).

The PbxSubsystem loads Lucent's JTAPI implementation classes: all objects pertaining to JTAPI are contained within this task and cannot be accessed directly from any other task. This task exports capabilities which give other tasks indirect access to JTAPI. In this implementation, the right to monitor and control a telephone addresses can be revoked at any time by the PbxSubsystem. Because JTAPI uses the Observer/Observable model, objects within the PbxSubsystem task register themselves as observers. When the JTAPI implementation makes upcalls, these objects forward events to the DispatcherSubsystem.

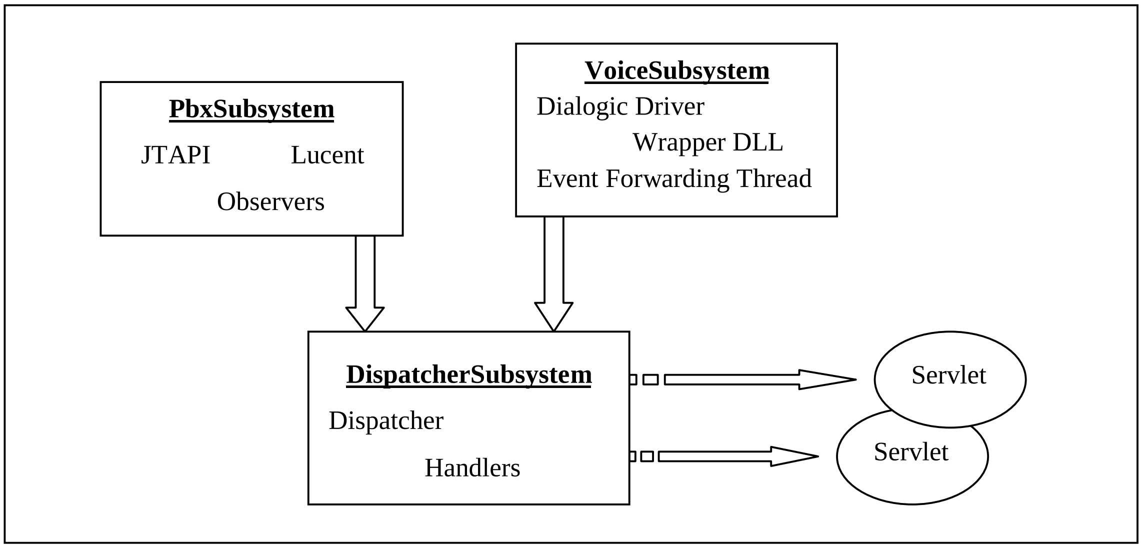


Figure 3. J-Phone tasks.

The VoiceSubsystem provides an interface to the Dialogic boards. It loads the C libraries that provide the interface to the device driver and, like the PbxSubsystem, grants capabilities that provide control of telephony hardware. In the VoiceSubsystem, these capabilities are called voice channels, corresponding to half-duplex analog telephone lines. A half-duplex channel allows an application to either play or record audio over the telephone line, but not both simultaneously. Full-duplex connections can be created by assigning two channels to a single telephone call. Voice channel capabilities permit servlets to record and play standard μ-law PCM encoded audio data in either streamed or unstreamed modes. These capabilities also signal incoming DTMF digits and allow servlets to generate these digits as well.

The third privileged task in J-Phone is the DispatcherSubsystem. This task receives events from the other two privileged tasks and routes them to servlets. The DispatcherSubsystem maintains record of the mapping between servlets and telephone addresses and keeps track of all active calls in the system.

4.3.2 Event Handling

In order to move information from one task to another, J-Phone uses a specialized event model. This model was developed with several stipulations in mind. First, associating a single event consumer with each telephone call greatly simplifies problems such as preserving telephone call states and redirecting calls from one servlet to another. Secondly, the vast majority of events can pass directly from source to destination, and so this path should be as short as possible. Finally, each event should be sent to exactly one consumer. The Java Observer/Observable model fails to meet this last condition; the J-Phone system fulfills all three requirements.

As stated above, events are generated by the PbxSubsystem and the VoiceSubsystem. In keeping with the Observer model found in Java and in JTAPI, events are forwarded by upcalls into the DispatcherSubsystem task. Within this task are several event handlers, one per active telephone call. Each handler is responsible for receiving events related to a single telephone call and invoking the `service` method of the appropriate servlet. To achieve this, each handler also maintains its own queue into which events are placed by the PbxSubsystem and the VoiceSubsystem. The PbxSubsystem relies on threads created by JTAPI to make this cross-task call; the VoiceSubsystem task contains its own event forwarding thread. This thread makes calls down into native code, pulls an event from the Dialogic driver event queue and then forwards it to the appropriate handler. Handlers also contain their own threads, in order to process events concurrently. In processing an event, the handler removes the next event from the queue, calls the `service` method of a servlet, and then carries out the action dictated by the servlet's response before retrieving the next event.

4.4 Call Walk-Through

A closer look at the execution of the voice mail servlet reveals the details of this event model in action. As seen in section 4.2, the voice mail servlet monitors a telephone address and answers all calls made to this address, both playing and recording audio in the process.

When a call is placed to the voice mail telephone address, an event is produced by JTAPI and the PbxSubsystem requests a handler for this address from the DispatcherSubsystem. Since this is a new call, the dispatcher creates a new handler and returns it to the PbxSubsystem. Now that the PbxSubsystem has a handler associated with this call, it will send events for this address directly to the handler (that is, without querying the dispatcher) until the call is finished or redirected and the handler becomes invalid for this address.

A new call event is passed from the PbxSubsystem to the handler and on to the servlet, which responds with an `ActivateCallResponse`. This tells J-Phone that the servlet will respond to the call interactively, by playing or recording audio, or sending and receiving DTMF signals. To accommodate this, J-Phone reroutes the call to one of the telephone lines connected to a Dialogic interface. This is all accomplished before the telephone starts to ring, and now that the call is connected to a physical terminal, the VoiceSubsystem begins to generate events. Just as in the PbxSubsystem, it requests a handler from the DispatcherSubsystem, and then sends events directly to the handler, starting with a "ringing" event.

The servlet proceeds to take the telephone off hook, and to start playing audio data. In this application, the data is passed from the servlet task to the handler and then on to the VoiceSubsystem (two cross-task calls altogether) in one large chunk. Playing begins asynchronously and a `PlayingStartedRequest` is passed back to, and ignored, by the servlet. When the voice channel has finished playing the audio sample, it alerts the servlet, which, in turn, responds by creating a new `StartRecordingResponse`. This response object constructor takes no arguments, such as an empty buffer, because audio data is returned to the servlet in a series of request objects. Each `RecordBlockRequest` holds 512 bytes, or 1/16 of a second, of raw data. Receiving raw audio data allows servlets an enormous amount of flexibility in what happens next: data could be passed through a noise filer, to a PC sound card, or even to a speech recognition system. In the voice mail servlet, this data is simply stored in a `WaveFile` object until the call is completed.

When the calling party hangs up, JTAPI alerts the PbxSubsystem that the call has been dropped and an event is placed in the handler queue. The servlet instructs the voice channel to stop recording, and when all recorded data has been received by the servlet, the call is ended. The servlet saves the recorded sample as a Microsoft WAVE file for later playback over the web, and finally, the voice channel is freed and the handler destroyed.

4.5 Resource Management

The performance/cost tradeoffs cause telephony systems to be designed so that their total capacity is almost always below potential peak demand (exceptions include some military systems, where availability is extremely important). However, in a typical phone system there is no way of prioritizing users so that they can have guaranteed service (except for the 911 number and isolated leased lines). This issue can be addressed in our system, through the following resource management policy: new connections are admitted only if CPU load is below 90%. Only privileged parties and individuals can connect when CPU load is above this threshold. It is important to stress that since our system currently provides neither fault tolerance nor real-time guarantees, this resource management policy will not provide guaranteed availability.

Billing is an integral part of most telephone systems. At about 20% overhead of total execution time [CvE98] JRes can account for CPU time, network resources and heap memory on very fine-grain basis (e.g. for every live object allocated by every thread). Accounting for CPU time only results in negligible overheads. We have used JRes in the later part of the paper to account for CPU time consumed by various threads participating in performance gauging experiments. The same code can be used to provide detailed resource usage statistics for either billing or for diagnostic purposes.

5 Performance

In our previous work [HCC+98] a detailed study of J-Kernel microbenchmarks was presented. While it is important to know the costs of basic operations, such as cross-task calls, microbenchmarks provide only limited insight into how real applications will perform. In order to understand the behavior of the J-Kernel when running real code, this section presents two experiments using J-Phone, and a simple calculation concerning scalability of J-Phone.

The experiments were carried out on a 300MHz Pentium II, running Windows NT 4.0. The Java virtual machine used was Microsoft's VM version 1.1, with SDK 2.01.

5.1 Overheads of Cross-Task Calls

The objective of the two experiments is to measure the costs of cross-task calls and the frequency of such calls in a non-trivial application. The first test, *Half-Duplex Conversation*, mimics a 40 second phone conversation between two people, each of whom is simulated by a servlet. The call initiator "speaks" for two seconds, then "listens" for two seconds, then "speaks" again, while the receiver is doing the opposite. "Speaking" involves sending uncompressed audio data in one kilobyte segments (which is equivalent to 1/8s of conversation) to the voice subsystem, where they are presented to the Dialogic driver, which plays the data over the phone line. "Listening" consists of recording data (in 512B segments), collecting the data to accumulate two seconds worth of audio (16kB) and writing it out to the disk in a standard, uncompressed *wav* file format.

The second test, *VoiceBox Listing*, measures the performance of checking for voicemail. A user with an account on J-Phone types a URL at her browser. The corresponding HTTP request is sent to the server, which in turn creates a thread which lists the user's voice box directory and formats the reply as an HTML page. The simple authentication servlet from section 3.4 was used.

Table 1 summarizes the results. The *Initiator* and *Responder* columns correspond to two parties involved in *HalfDuplex Conversation*, while the *VoiceServlet* column corresponds to the servlet fetching information about voice messages. The first five rows present the actual measurement values; the remaining rows contain derived information.

In all cases the overheads of crossing tasks (which include the cost of copying arguments and then return values) are below 1% of the total consumed CPU time. On the average, crossing tasks costs between 3-4.6μs and an average call transfers between 312 (*Initiator*) to 450 (*VoiceServlet*) bytes. The cost of copying data accounts for 27-31% of an average cross-task call. This suggests that overheads of crossing tasks in real applications when no data transfer is involved are between 2.1-3.3μs. The value can be contrasted with the cost of a null cross-task call, measured in an isolated tight loop: 1.35μs.

The numbers discussed above should be viewed in the perspective of the fastest way to invoke a method in another NT process we know of: a null interface invocation to a COM component located in a separate process on the same machine. On the hardware used for these experiments, this cost is 75μs (measured in a tight loop). If average overheads of crossing task boundaries in the J-Kernel were equal to 75μs, the percentage of time spent in crossing tasks would rise from a fraction of one percent to between 3% (*VoiceServlet*) and 16% (*Receiver*).

The achieved bandwidth of copying data across tasks is quite satisfactory. About 85% of the data transferred across tasks in the two experiments is stored in byte arrays of 0.5kB and 1kB sizes; the remaining data are either references to capabilities, primitive data types or data inside objects of the type `java.lang.String`. The average copying bandwidth of 330-380 B/μs can be contrasted with a peak data copying bandwidth of 630 B/μs achieved in Java with a tight loop of calls to `System.arraycopy()`.

While analyzing the performance numbers it is important to note that Java introduces non-obvious overheads. For instance, in the *VoiceServlet* experiment, a large part of the 18.5ms of CPU time can be accounted for as follows. About seven milliseconds are spent in the network and file system Java classes and native protocol stacks. Roughly eight milliseconds are spent performing string concatenations in one particular part of the voice mail servlet (Java string processing is very slow under MS JVM). These overheads, introduced by Java and not always avoidable, dwarf the relative costs of crossing task boundaries and lead to conclusions that may be not true if the JVM performance improves dramatically. One conclusion, based on the current performance numbers, is that in large applications similar to J-Phone, the task boundaries crossing overheads are very small relative to the total execution time and optimizing them will bring barely noticeable performance improvements. Second, in applications where crossing tasks is frequent and optimizing cross-task calls actually matters, on the average 30% of task crossing overheads can be removed by wrapping data structures and arrays in capabilities. This avoids copying data between tasks but comes at the expense of increased programming effort. It is important to stress that the conclusions are drawn from observing several experiments based on a single, although complex and realistic, application. More study is needed to fully understand application behavior on top of Java in general and the J-Kernel in particular.

	Initiator	Responder	VoiceServlet
Total CPU time [ms]	1503	1062	18.5
Total number of cross-task calls	3085	2671	8
Total cross-task data transfer [B]	962304	984068	3596
Total cross-task call overhead [ms]	9.37	8.71	0.037
Total cross-task data copy time [ms]	2.91	2.58	0.01
Average CPU time/cross-task call [ms]	0.0030	0.0033	0.0046
Average bytes/call	312	368	450
Average copy overhead/call	31%	30%	27%
Average copy bandwidth [bytes/µs]	331	381	360
Relative cross-task call overhead	0.62%	0.82%	0.20%

Table 1. Selected performance measurements of *HalfDuplex Conversation* and *VoiceServlet.*

5.2 J-Phone Scalability

Our current hardware configuration supports at most eight simultaneous half-duplex connections. Every connection results in an additional 1.7-2.3% increase in the CPU load. From this we estimate that, given enough physical lines, our system could support about 44-59 simultaneous half-duplex connections before reaching 100% CPU utilization. Since J-Phone demands modest amounts of physical memory, permanent storage and network resources, CPU is the bottleneck from the perspective of scaling the system, so the range of 44-59 simultaneous half-duplex connections is very likely to be achievable in practice.

6 Related Work

Work relevant to constructing J-Phone and J-Web on top of the J-Kernel falls into several categories: design of other telephony-related systems, Java-related safety and protection issues, and various aspects of research on extensible systems. In this section we summarize the most important work from these areas influencing our research.

Commercial incentives have caused several vendors to design and implement computer-telephony integration systems. Sixth Sense, from AnswerSoft [Ans], automates customer service call centers, by routing customer calls and prompting service representatives with useful information. Through rules-based intelligence and a custom scripting language, Sixth Sense allows managers to customize call routing and product promotions. A more general example of commercial CTI solution is ActiveVoice's TeLANophy [Act]**.** TeLANophy uses either TSAPI or TAPI to integrate a local telephone network and the LAN and allows users to manage telephone calls from a PC, transferring them between extensions and a voice mail system. TeLANophy also unifies voice mail, email and fax into a single messaging system. A non-commercial server, WebGALAXY [LFP+97], implements a combined web and telephony interface and uses natural language processing, on both spoken and written text, to allow users to retrieve information from online sources. All three of these systems offer computer telephony integration and combine telephony with other types of media, such as email or the web, but in each case they provide services for only a specific application. Sixth Sense also gives providers a mechanism for dynamically configuring services offered to customers. However, none of these systems allows providers, or end-users, to add features, nor do they provide a means to trace resource consumption in the system.

Several commercial web servers support some form of extensibility, but face a difficult trade-off between protection and performance. Microsoft's ISAPI interface allows extensions to run in the same process as the server, but does not protect the server from the extensions. Servers supporting CGI and FastCGI run extensions in separate processes for protection, but this requires an expensive switch between address spaces to handle an incoming request.

The J-Kernel enforces a structure that is similar to traditional capability systems [Lev84, WLH81]. Both the J-Kernel and traditional capability systems are founded on the notion of unforgeable capabilities. In both, capabilities name objects in a context-independent manner, so that capabilities can be passed from one task to another. The main difference is that traditional capability systems used virtual memory or specialized hardware support to implement capabilities, while the J-Kernel uses language safety. The use of virtual memory or specialized hardware led either to slow cross-task calls, to high hardware costs, or to portability limitations.

Wallach et al [WBD+97] discuss several approaches to security in Java, mostly aimed at protecting users from potentially hostile applets, and several vendors [GS98, Net, Mic] implement variations on one or more of these approaches. However, the techniques don't directly address applet-to-applet communication (or, in the context of the J-Server, servlet-to-servlet communication), resource management, and task termination.

Several extensible operating systems [BSP+95, EKO95, SES+96] aim to expose operating systems components to applications or user-defined extensions, using a combination of virtual memory and language-based protection. In particular, SPIN allows extensions written in Modula-3 to be downloaded into the operating system kernel. However, SPIN and Modula-3 do not have an equivalent of the J-Kernel's tasks at the language level.

A very recent specialized programming language PLAN [HKM+98] aims at providing an infrastructure for programming Active Networks. The programs replace packet headers (which are viewed as ‘dumb’ programs) and are executed on Internet hosts and routers. In order to protect network availability PLAN programs must use a bounded amount of space and time on active routers and bandwidth in the network. This is enforced by the language runtime system. PLAN’s approach to resource management (e.g. providing it at the language level) is an exception rather than the rule among safe languages. On the other hand, enforcing resource limits has long been a responsibility of operating systems. For instance, many UNIX shells export the `limit` command, which sets resource limitations for the current shell and its child processes. Among others, available CPU time and maximum sizes of data segment, stack segment, and virtual memory can be set. Enforcing resource limits in traditional operating systems is coarse-grained in that the unit of control is an entire process.

7 Conclusions

Extending the Servlet API leads to a simple and easy-to-use environment for developing telephony applications. The Servlet API is useful for applications that respond to some sort of user stimulus, either when someone clicks a link on a web page or picks up a telephone and dials. This type of behavior is easily implemented in the request/response model. However, it’s not clear how to integrate this model with applications that independently initiate actions. For example, imagine an application called `WakeUpCallServlet`. Such a servlet would place a telephone call at a specific time of day. This action, to initiate a telephone call, is not a response to any PhoneServletRequest, so this servlet must momentarily step outside of the PhoneServlet API and make a cross-task call to the J-Phone. The best way of meeting the requirements of such a servlet still needs to be addressed.

Using the J-Kernel provides several advantages. The clear-cut protection domains allow the server to dynamically load untrusted extensions. The fault isolation permits changes to be made to individual components, which can consequently be reloaded without restarting the remainder of the system. From the performance analysis point of view, overheads currently introduced by Java (e.g. inefficient string processing, slow network and file classes) make costs of crossing task boundaries in J-Phone look negligible. In absolute values, costs of crossing task boundaries are an order of magnitude smaller than the costs of interprocess communication in the underlying operating system. At the same time, tasks are protected from each other and programming of inter-task communication is much easier than typical operating system IPC.

Overall our experiences with building J-Phone on top of the J-Kernel are encouraging. Safety and protection guarantees, short development time, portability, straightforward extensibility and simple interaction with the Web make the environment of the J-Kernel an attractive platform for both development of realistic applications and for research and experimentation with extensible systems based on safe languages.

8 References

[Act] ActiveVoice. *TeLANophy*. http://www.activevoice.com.

[Ans] AnswerSoft. *Sixth Sense*. http://www.answersoft.com.

[BSP+95] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. *Extensibility, Safety and Performance in the SPIN Operating System*. 15th ACM Symposium on Operating Systems Principles, p.267-284, Copper Mountain, CO, December 1995.

[CCH+98] G. Czajkowski, C. C. Chang, C. Hawblitzel, D. Hu, and T. von Eicken. *Resource Management for Extensible Internet Servers*. 8th ACM SIGOPS European Workshop, Sintra, Portugal, September 1998.

[CvE98] G. Czajkowski, and T. von Eicken. *JRes: A Resource Accounting Interface for Java*. To appear, OOPSLA'98, Vancouver, BC, October 1998.

[EKO95] R. Engler, M. Kaashoek, and J. O'Toole. *Exokernel: An Operating System Architecture for Application-Level Resource Management*. 15th ACM Symposium on Operating Systems Principles, p. 251–266, Copper Mountain, CO, December 1995.

[GS98] L. Gong, and R. Schemers. *Implementing Protection Domains in the Java Development Kit 1.2*. Internet Society Symposium on Network and Distributed System Security, San Diego, CA, March 1998.

[HCC+98] C. Hawblitzel, C. C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. *Implementing Multiple Protection Domains in Java*. 1998 USENIX Annual Technical Conference, p. 259-270, New Orleans, LA, June 1998.

[HKM+98] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. *PLAN: A Packet Language for Active Networks.* To appear in the International Conference on Functional Programming (ICFP) '98.

[Jav-a] JavaSoft. *JavaServer Documentation*. http://java.sun.com/products/java-server.

[Jav-b] JavaSoft. *Java Telephony API*. http://java.sun.com/products/jtapi/index.html.

[Lev84] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, Massachusetts, 1984.

[LFP+97] R. Lau, G. Flammia, C. Pao and V. Zue. *WebGALAXY:Beyond Point and Click - A Conversational Interface to a Browser.* Sixth International World Wide Web Conference, Santa Clara, CA, April 1997.

[Mic] Microsoft Corporation. *Microsoft Security Management Architecture White Paper*. http://www.microsoft.com/ie/security.

[Net] Netscape Corporation. *Java Capabilities API*. Available at http://www.netscape.com.

[SES+96] M. Seltzer, Y. Endo, C. Small, and K. Smith. *Dealing with Disaster: Surviving Misbehaved Kernel Extensions*. 2nd Symposium on Operating Systems Design and Implementation (OSDI), p. 213-227, Seattle, WA, October, 1996.

[WBD+97] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. *Extensible Security Architectures for Java.* 16th ACM Symposium on Operating Systems Principles, p. 116–128, Saint-Malo, France, October 1997.

[WLH81] W. A. Wulf, R. Levin, and S.P. Harbison, *Hydra/C. mmp: An Experimental Computer System*, McGraw-Hill, New York, NY, 1981.

[W3C] World Wide Web Consortium, *Jigsaw 1.0 Alpha 3*, http://www.w3.org/Jigsaw.