Extending Applications to the Network

by

David Marwood

B.Sc., University of Calgary, 1994

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE STUDIES

(Department of Computer Science)

we accept this thesis as conforming
to the required standard

The University of British Columbia

August 1998

Also published as Technical Report TR-98-09, Department of Computer Science,

University of British Columbia, Vancouver, British Columbia, Canada.

Abstract

Network applications are applications capable of selecting, at run-time, portions of their code to execute at remote network locations. By executing remote code in a restricted environment and providing convenient communication mechanisms within the application, network applications enable the implementation of tasks that cannot be implemented using traditional techniques. Even existing applications can realize significant performance improvements and reduced resource consumption when redesigned as network applications.

By examining several application domains, we expose specific desirable capabilities of a software infrastructure to support network applications. These capabilities entail a variety of interacting software development challenges for which we recommend solutions.

The solutions are applied in the design and implementation of a network application infrastructure, Jay, based on the Java language. Jay meets most of the desired capabilities, particularly demonstrating a cohesive and expressive communication framework and an integrated yet simple security model.

In all, network applications combine the best qualities of intelligent networks, active networks, and mobile agents into a single framework to provide a unique and effective development environment.

Contents

Acknowledgements

I owe thanks to many people for an enjoyable and educational time at UBC. First, I thank my supervisor, Dr. Norm Hutchinson, for his always-valuable advice and for his patience, clarity, and encouragement. He allowed me the freedom to explore my ideas and learn from my mistakes. I thank Dr. Mike Feeley, not only for reviewing this thesis, but for improving my skills in writing, reviewing, presenting, and otherwise participating in the wider research community.

Thanks goes out to many of my fellow students at UBC, but especially to those with whom I shared the lab: Brad Duska, Margaret Petrus, and Alistair Veitch. Their day-to-day help in the lab and good times outside the lab will be remembered.

Most of all, I thank my wife, Sonja Struben. She has offered understanding and support throughout. Our time in Vancouver has been a pleasure.

David Marwood

The University of British Columbia
August 1998

To my parents, Everett and Linda Marwood, who have always offered encouragement regardless of my choices.

Chapter 1

Introduction

In its brief history, the Internet has seen apparently inexhaustible exponential growth, driven partly by the ongoing supply of network technologies and partly by the ongoing demand for richer applications. These applications have become increasingly oriented towards user needs. Early applications such as Telnet and FTP were designed as tools to accomplish another task rather than as an end in themselves. More task-oriented applications of that time included Internet mail and Usenet news for individual or group communication, but both are batch-oriented and intentionally hide network activities from users; mail and news simply periodically appears in a file accessible through a user-oriented application.

As network technologies advanced, applications increasingly exposed users to the network. Online multi-user games, interactive discussion forums, and the World-Wide Web demonstrated the Internet's vagaries and limitations, resulting in complaints about the "World-Wide Wait". Future applications promise further direct use of networks for user tasks, such as video phones and electronic commerce. If the current rapidly increasing size of the Internet and increasing disparity of bandwidth is an indication of future patterns, then users can expect further problems.

Solutions to these problems are often well understood by system developers and network operators, but require modifications to protocols entrenched by de jure standards. Sometimes a problem can be resolved by changing only a high-level protocol used by a class of applications. This has been the case for HTTP/1.1 proxy enhancements to enable sharing a single cache among multiple users [FGM+97], the FTP REST command to resume transfers interrupted by system failure [PR85] and S/MIME encryption to address network insecurity [DHR+98]. Even in this relatively simple case, modifications can take years to complete and often become de facto standards well before they are accepted by the Internet Engineering Task Force. Where the modifications are demanded by only a small portion of the community, the cost in time and organizational effort can be prohibitive.

In more complex cases, a solution can require modified behavior within the network or changes to low-level protocols, as is the case for quality of service guarantees, bandwidth reservation, multicast and mobility. Such changes consistently span many years from the initial need to deployment and can require widespread changes to network hardware.

This is not to disparage the value of standards, only to show that efforts to reflect user and application needs in the network are slowed by the demands of the standards process. What is desired is the ability for applications to include, in the network, application-specific functionality in the form of application-specified code. Such a system would allow applications to customize network behavior to user needs; a network that "understood" the user's task could more efficiently respond, to the benefit of both the network and the user. Using this same architecture, applications could perform tasks otherwise not possible, such as accessing resources only available at a remote host. Applications capable of starting portions of their code at remote hosts, or *network applications*, are the topic of this thesis.

1.1 Network Applications

A network application functions as a traditional application, but portions of its data and code exist and execute on remote hosts. These application extensions may modify data, contain threads, and otherwise act as an application would, subject to the security restrictions imposed by the remote environment.

Network applications exceed the abilities of traditional distributed systems in a variety of respects because they do not require *a priori* arrangements to accomodate individual needs. Because network behavior can be modified for individual applications, users with particular needs are easily accommodated without large-scale adaptation of the system. Similarly, applications requiring unique behavior of the network or service can implement that behavior immediately. Finally, systems requiring particular behavior embedded in the network can do so without modifying low-level network protocols. In each case, network applications allow an immediate need to be filled while allowing modifications to standards to continue or to be delayed indefinitely.

The additional functionality of network applications can be placed in three categories:

- **Data transformations:** By using application-specific information, network applications can filter, compress, cache, pre-fetch, and otherwise transform a data stream at appropriate network locations, reducing total network consumption and decreasing latency.
- **Remote resource access:** Efficient or effective access to system services and resources is sometimes available only locally. Examples include processing, memory, network capacity, local disks, and specialized system services. An application extension executing near the resource can act locally to accomplish application-specific tasks.
- **Network modification:** Where an application is restricted by existing network behavior, properly located application extensions can alleviate the restrictions. If effect, applications can modify low-level network behavior.

Demand for such increased flexibility over traditional distributed systems comes from a variety of fronts. Wireless, mobile, and personal (hand-held) computers are often distant from their home resources. Optimal use of often slow and intermittently available network links requires a high degree of network efficiency; transferring many kilobytes of data to present only a small portion of it to the user is unacceptable. Additionally, strains on limited processing, memory, and battery capacity can be alleviated by offloading resource-intensive tasks to a more powerful fixed host.

Users of well-connected and resource-rich hosts can also benefit. When these hosts are distant from the services they wish to access, as is common for Web or other remote database access, network limitations between the user and the service may be a bottleneck. Similarly, network applications may gain performance benefits by starting extensions on nearby resource-rich hosts. For widely distributed systems, lower limits on latency imposed by the speed of light can be overcome by performing filtering and processing data remotely.

1.2 Overview

The goal of this thesis is to determine the effectiveness of a software infrastructure to support network applications. We will show that use of a network application infrastructure increases the efficiency of certain applications and allows the implementation of applications that would otherwise be impossible. Through examples, we will see that network applications can be implemented with a minimum of inconvenience to the developer and without sacrificing security considerations.

Chapter 2 describes specific uses and requirements of a network application infrastructure. In Chapter 3, we look at Jay, a prototype infrastructure implementation, along with some sample Jay applications. Chapter 4 discusses systems related to network applications and to Jay, and Chapter 5 concludes.

Chapter 2

Network Applications

To this point we have described network applications loosely as applications capable of loading, starting and communicating with application modules on remote hosts. The need for network applications derives from need for applications to modify network behavior, either at the end-points of communication or within the network. The goals of network applications, then, is to fulfill that need.

In this chapter, we will first adopt terms with which to speak of the components of a network application infrastructure. Next, we will examine groups of applications suitable for implementation as network applications, and specific sample applications falling within those groups. Lastly, Section 2.3 will turn to the specific requirements and capabilities of network applications, including the trade-offs and recommended choices.

2.1 Terms

A system supporting network applications can be easily divided into several components, as illustrated in Figure 2.1. The remote portions of a network application form one or more *extensions*. Each extension exists within a single *extension environment*, a service running on a host capable of receiving and loading extensions.

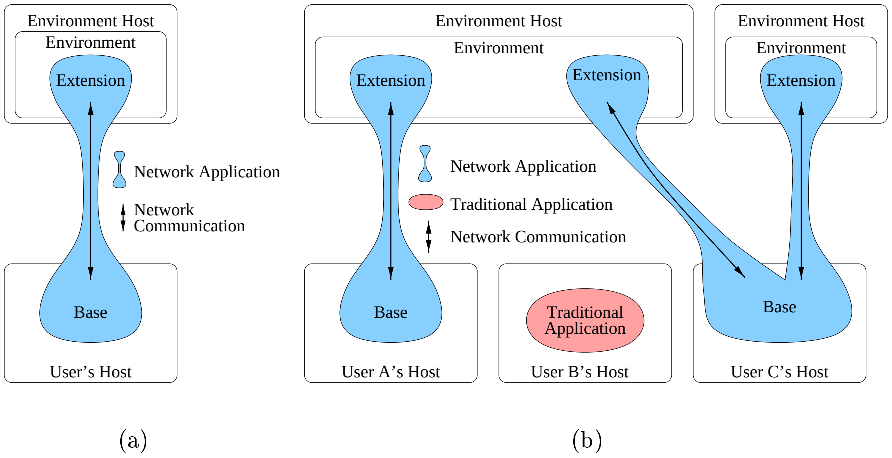

Figure 2.1: Components of a network application infrastructure. Figure 2.1(a) shows a single network application composed of a base and one extension. Figure 2.1(b) shows two network applications, one with two extensions, and a traditional application. Traditional applications are restricted to execute on a single host. These diagrams each illustrate a simple infrastructure; both could be augmented by a variety of useful capabilities.

Each environment may run multiple extensions, even from a single application, and prevents extensions from maliciously or accidentally interfering with one another. Typically, a single host supports one environment. A *network application* is composed of a *base* running on a user's host and any extensions started by the base. A *network application infrastructure* is a software system designed to support the goals of network applications.

Within the framework of Figure 2.1, network applications and extension environments have considerable flexibility as mechanisms and policies are largely unspecified. An infrastructure is free to specify the technique for communication within applications and between extensions, restrictions on the lifetime of an extension (particularly in the face of planned or unexpected disconnection from the base), and the security and language issues surrounding remotely executed code. Many of these choices are difficult and dependent on application demands. The next section will look at several classes of applications suited to implementation using extensions and their specific demands on the infrastructure.

2.2 Applications

Section 1.1 briefly named three categories of tasks well suited to implementation as network applications. This section expands on each of the three categories to demonstrate that network applications are useful and necessary to solve many problems, and to determine what demands applications place on the infrastructure.

2.2.1 Data Transformation

Applications typically transform data retrieved from services for display or processing in an application-specific manner. The transformation often removes or summarizes retrieved data, thereby reducing the information size, sometimes by many factors. Performing the transformations near the information source, or simply closer to the information source than any network bottlenecks, decreases the bandwidth requirements proportionally. These application-specific transformations are well-suited to network applications, as it is inappropriate and impractical to modify a service to account for all application needs. Similarly, a transformation that would be widely useful simply may not be available from the service.

Suppose a wireless palm-top computer user wants to monitor dozens of stock prices. Instead of downloading all the stock data in real time[1], an application extension placed in the fixed network could monitor the data, sending only notifications of important events across the slow, unreliable, intermittently available wireless link. If only one in one-thousand data items collected from the server are of interest to the user, bandwidth savings are of a similar magnitude.

"Distilling" Web images near the server reduces the size of the data by using lossy compression, reducing the colors to a number the client can display, or similar techniques. Distilled image data can be a factor of seven times smaller than

[1] This technique is used by Marimba Incorporated's Castanet software's Wall Street Web channel.

the original with minimal loss of quality [FGC$^+$97]. A host with a low-resolution monochrome display may reduce data size by many factors without any apparent loss in quality.

Existing protocols intended for use on a LAN are often inefficient when used in situations of high latency, low bandwidth, or poor connectivity. A network application extension located near a remote server could communicate with the server using a standard protocol and transmit the responses to the base using an optimized custom protocol. For example, text data can often be significantly compressed using simple compression schemes. An extension could compress data near a distant NNRP News server to reduce bandwidth and latency. Combining compression with batching and pre-fetching would result in further savings.

2.2.2 Remote Resource Access

The various hosts of a large network naturally offer a diversity of services and resources. General-purpose access to such services, however, is sometimes inefficient or impossible for code executing remotely. Application code executed on the host offering the service avoids the need for general-purpose remote access, allowing for access to services that could not be exported effectively as a traditional general-purpose services on a WAN.

For example, an extension developed to respond to a remote condition can be notified and respond more quickly by executing at or near the monitored condition. To extend the stock monitoring example above, the wireless palm-top computer user could react to important changes in stock price immediately, avoiding any network delays. Although the stock server could be modified to perform such a task, recognizing the variety of conditions applications may wish to respond to would require considerable application-specific modifications.

Consider an intelligent agent that searches a database for records satisfying a complex query. The agent may transfer megabytes of data in several transactions as the query is refined, a potentially very slow process on a WAN. By executing near the database host, an extension could interact at LAN speeds and return only the result to the base, greatly shortening the interaction time.

Resources other than distributed services may be naturally accessible only locally. Memory, processing power, and special-purpose hardware fall into this category. A hand-held host may be able to save battery power or improve response time by running compute- or memory-intensive tasks on a more powerful host. Particularly resource-intensive programs could present little more than a user interface on the hand-held host.

2.2.3 Network Modification

Sometimes applications or distributed systems are limited by existing peculiarities of network hardware or protocols. Extensions run at key network locations can aid in overcoming such peculiarities.

For example, exchanges between two hosts can be inefficient or impossible because they use a different underlying network. An internal corporate network using only IPX may connect a single host to the Internet to act as a bridge. A network application extension started on that host could communicate to its base using IPX and to the Internet using TCP. A similar technique could be used to pass outside a corporate firewall.

A computer connected via a wireless link may become occasionally disconnected from the fixed network. In an infrastructure tolerant of disconnection, an extension running on a wired host could preserve and continue a long-running computation. For example, a long-running simulation could continue after an application base intentionally or unintentionally disconnected.

Network protocols typically provide each host a view of the network tailored to that host. An application with extensions running on several hosts can gain a more global perspective and can aid in network monitoring and diagnosis. For example, a monitoring application could start extensions on important network nodes. On detecting a failed link or a network partition, the extensions could act to restore connectivity from both sides of the failure.

As seen, a variety of tasks can be effectively implemented as network applications. Each application uses extensions differently and each exploits different capabilities of the infrastructure. The next section will look at these capabilities, as well as recommended approaches and trade-offs in an implementation.

2.3 Network Application Infrastructure Capabilities

A network application infrastructure must offer considerable functionality to meet the goals of network applications. We can separate this functionality into a set of capabilities. Some capabilities are vital:

- module transmission,
- module reception, and
- communication.

Many applications demand other capabilities of the infrastructure:

- heterogeneity,
- remote resource access,
- performance,
- versioning,
- security,
- fault tolerance,
- concurrency control, and

- development tools.

For each of these capabilities we look at alternatives for implementation and potential conflicts in meeting the goals of network applications.

2.3.1 Module Transmission

Transmitting code is a crucial support function at the network application base. A *module* is the semantic unit of code that can be transmitted alone, typically a function, object-oriented class, or semantically related set of classes. An extension may be composed of several modules. Module transmission includes obtaining the code from the system, marshaling it, and transmitting it. While the mechanisms of the steps are largely implementation-dependent, there is a more general question: when should modules be selected and transmitted?

One technique for module selection is to require the application developer indicate modules to transmit at compile-time. Although simple, a run-time error will result if a necessary module is accidentally omitted.

The selection could be automated to include the transitive closure of modules referenced by an initial module. However, even as the application starts, correctly selecting the minimal set of required modules may be impossible. Where reference to a module is conditional on run-time state, the module may be necessarily selected but never referenced. Conversely, modules could be omitted in languages that permit the name of referenced modules to depend on run-time state. For example, in a language that references modules using a URL represented as a string, predicting the value of the string as the application starts is generally impossible. A conservative (but highly inefficient) approach could automatically transfer all modules with names, but if modules can be named by URLs and located throughout the network then a list of all module names may be unavailable and transmitting all of them would be impractical. While automated selection may supplement the previous technique in an effort to aid programmers, it is not reliable.

Finally, modules may be requested from the base on demand at run-time using a call-back mechanism. Unlike the two previous solutions, modules are transmitted only if they are referenced, minimizing bandwidth use. Unfortunately, a naive implementation incurs an extra round-trip delay on the critical execution path for each module, a considerable penalty if latency is high and modules are numerous. Further, if a module is required while the extension is disconnected from the base, execution must be suspended.

2.3.2 Module Reception

The corresponding and equally important requirement of the extension environment is module reception. This includes unmarshaling, loading, linking, starting and eventually unloading modules. These steps are conceptually straight-forward and widely implemented in existing single-host application programming languages. Extending these principles to the distributed network application setting introduces several concerns.

Unmarshaling, loading, and linking present concerns to both heterogeneity and security. Received code must be executable on a platform potentially different from that hosting the network application base. From a security perspective, it is not sufficient to allow loaded code the same permissions as enjoyed by the base, as is done for modules dynamically loaded into traditional applications. Instead, security must be enforced according to the policies of the remote host, not the policies of the base. Heterogeneity and security concerns are examined in more detail in Sections 2.3.4 and 2.3.8, respectively.

Unloading modules is necessary to prevent monotonically increasing demands on limited environment memory and other resources. Problems and solutions in this area are largely comparable to those in traditional operating systems: resources allocated to an extension are recorded and forcibly released by the operating system when the extension completes. Of note is the need for a policy describing when to unload a module, given that extensions can naturally outlive their base and that the status of the base may be unknown during a communication failure.

2.3.3 Communication

Communication in network application infrastructures is needed not only to transmit modules, but to communicate between a base and its extensions. System-level interprocess communication facilities are typically not adequate. Describing application semantics in an asynchronous byte-oriented medium is too cumbersome to allow the expressive application-level interaction expected between a base and extension. Communication should fit comfortably with existing application semantics and allow for both asynchronous and synchronous communication.

2.3.4 Heterogeneity

Network applications exist within a heterogeneous distributed environment. Baring very special needs, a homogeneous network application infrastructure would be strongly limited in its ability to meet the goals of network applications.

In a heterogeneous network, modules must be transmitted in a platform-independent form to preserve byte order, byte alignment, and other characteristics of code and embedded data. The code must be in a representation that can be executed by the remote environment. Responses to the need for platform-independent executable code have included interpreting source code as in Tcl [Out94] and the Bash command shell [NR98]; intermediate byte-code representations as in Java and a variety of early Pascal compilers including the University of California, San Diego P-Code [Nel79]; and "fat" or multi-architecture binaries containing native executables for multiple platforms as used by MacOS [Sta].

2.3.5 Remote Resource Access

Access to resources and services at remote environments is a common task of network applications. Desirable resources and services range from the common to the rare: processing, memory, disks, and networks; special-purpose hardware such as a video camera or robot; or location-specific software such as licensed libraries or multi-user applications. The infrastructure should provide facilities for discovering and accessing such resources.

At compile time, network applications should be deemed semantically valid even if the resource does not exist locally. Strong type checking would require that a description of the resource interface be available at compile-time, and that the interface be verified when an extension is dynamically linked.

Access and use of resources should be moderated either by the infrastructure or the operating system. Resource allocation must be managed between extensions and, where appropriate, between extensions and traditional applications competing for the resources. Similarly, resources must be reclaimed by the infrastructure or operating system even when an extension exits without explicitly releasing the resources. If extensions execute as threads of a single long-lived environment process then resource reclamation cannot simply be deferred until the environment exits. The environment must reclaim resources as they are no longer required so they may be allotted to other extensions.

2.3.6 Performance

In conflict with many of the requirements of this section is the need for performance. Network applications designed to improve performance over a similar traditional application must recognize there is a run-time overhead in starting the network application. That overhead must not exceed the performance gain. Network applications designed for tasks that are otherwise impossible must be sufficiently efficient to run on potentially resource-poor clients.

Some performance loss will be necessary for communication, heterogeneity, resource management and security in a distributed environment. The infrastructure must ensure the loss is minimal and tolerable.

2.3.7 Versioning

A single network application can execute code from a variety of sources: locally within the base and in within the infrastructure, remotely within the extension and within the environment, and possibly from other network sources. The varied sources of code invites a clash between the versions of code used for development (at compile-time) and the versions used to execute (at run-time).

Versioning allows such version variations to be detected or tolerated. In its simplest form, this implies compatibility of newer versions of the infrastructure with older applications (backward compatibility). In addition, executing a base and extension on different versions of the infrastructure should be transparent. Ideally, execution of an application on an infrastructure older than the version the application was developed with should succeed so long as functionality introduced between the two versions is not called on.

2.3.8 Security

The goal of security in traditional applications is to prevent accidental or malicious damage to the system and to user information. Damage may come in the form of unauthorized use or, where limited use is permitted, excessive use. Limits are placed on resource use according to the verified identity of the user or principal.

These same properties are desirable for network applications. A user of a network application must not be able to access unauthorized resources or exceed authorized resource consumption. Given the sample applications in Section 2.2, it is inadequate for a network application environment to simply deny access to users without local accounts. An environment located on the same host as a Web server may wish to allow known search engines to start extensions for the purpose of indexing the Web server. Direct access to the file hierarchy may be allowed to facilitate thorough, efficient indexing. Similarly, a multi-user game may wish to allow any players, without authentication.

In many cases, it would be desirable to allow access to users selected by another party. For example, a set of small wireless network access providers may agree to allow all their subscribers access to any of their network application environments. Similarly, an anonymous FTP site wanting to allow access to anyone identifying themselves may instead run a network application environment configured to allow controlled file access to any identified user of any host.

Conceivably, an environment may wish to restrict access to users running specific extensions known to the environment operators. To some extent, this defeats a considerable advantage of network applications: that the code is specified by the user. Even without this advantage, however, extensions may provide better communication facilities and better security than traditional special-purpose services.

Tailoring access restrictions to users requires user authentication. The environment should not require passwords or other secret information be transmitted in the clear across an insecure network. Since the environment may not be trusted by the user, a still better system would avoid the exchange of confidential information during authentication.

The unique properties of network applications can require that confidential information used within an application be transmitted between the base and extension over the network. This presents a second, distinct border that must be secured, as shown in Figure 2.2. Since the need for secure communication is common to many network applications (as well as many traditional applications), including it in the infrastructure or at lower layers is advisable. Implementors should take care to avoid making communication within the application more cumbersome or reducing

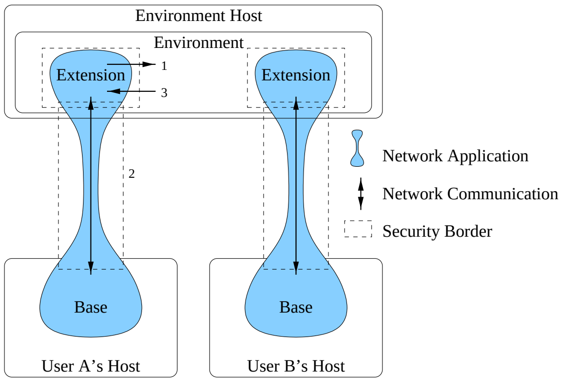

Figure 2.2: Three borders showing security concerns. (1) The environment should be protected from interference by the extension. (2) Network communication should be secure from eavesdroppers. (3) The extension should be protected from interference by the environment.

the performance of the communication medium.

Finally, network applications necessitate that confidential information stored in an extension should not be accessible to the environment or to another extension in the same environment. This implies a third security border between the application and the infrastructure, as shown in Figure 2.2. Further, the application should be able to ensure the environment is correctly executing the extension code. Unfortunately, both these capabilities present considerable challenges. Bennet Yee suggests a separate verifiably secure co-processor [Yee94]. [MvRSS96] recommends running the computation on different hosts. The results are compared and a voting algorithm determines the correct output. Frank Hohl [Hoh98] considers refusing the results of a computation at a remote host after the application has been exposed to the host for a pre-determined amount of wall-clock time, with the thinking that an attack against the computation must always exceed that time. A final alternative would be to have the application transform encrypted data in its encrypted form. Since no clear representation ever exists at the extension, the application runs no risk of revealing the data. Further, if the encrypted result contained redundant information, the base could detect incorrect operation of the environment by detecting an invalid result. Unfortunately, performing an encrypted transformation without revealing the contents is obviously complex.

2.3.9 Fault Tolerance

The distributed nature of network applications provides opportunity to loosen the coupling of the components. Neither failure nor disconnection of an extension need necessarily result in failure of the application. For a network monitoring tool that reports when a host has failed, failure of an extension would be a normal condition. For wireless and low-power hosts, unexpected temporary disconnection may be a normal aspect of the network. The same distributed nature, however, increases the likelihood some portion of the application will fail, emphasizing the need for robust fault tolerance.

Some failures may be handled transparently to applications. More commonly, as in the two examples mentioned, the failure prevents the infrastructure from offering a service, or forces the infrastructure to modify or withdraw a services already in use. In such cases, the application should be notified so that it may respond accordingly. Where possible, the infrastructure should provide a default failure policy suitable to a wide range of applications.

2.3.10 Concurrency Control

Network applications, as exemplified in Section 2.2, often require concurrency control due to their naturally multi-threaded implementations. Concurrency control allows thread synchronization and protection of shared data structures. Because concurrency control at a single host is a common feature of multi-threaded languages and operating systems, implementation may be left to the underlying system or the application developer. However, the infrastructure may extend the concurrency control offered by the language or operating system, or provide concurrency control between a base and its extensions as a form of implicit, structured communication.

2.3.11 Development Tools

The difficulty of developing distributed software systems increases the need for debugging capabilities. At compile-time, a strong type system enhances error detection. Interpreted languages can perform syntax and semantic checking similar to a compiled language. At run-time, the debugger can support analysis of logical errors causing system failure, livelock, deadlock, or other failure modes at either the base or extension.

2.4 Summary

The goal of a network application infrastructure is to enable loading, starting, and communicating with application modules at remote locations. These goals and the definitions of Section 2.1 leave considerable lee-way for the implementation of an infrastructure. We saw, however, that the target applications place numerous additional demands on the infrastructure. The capabilities an infrastructure must provide to meet these demands are also numerous and sometimes conflicting. Section 2.3 discussed the most important of these capabilities, exposed issues and conflicts, and recommended solutions.

Having explored the needs of a network application infrastructure, we turn to a prototype implementation, the Jay system. Jay incorporates all the components and most of the capabilities discussed, highlighting expressive communication facilities and security for the environment. Jay reveals practical issues in developing an infrastructure, and provides a detailed basis to which we can compare similar work.

Chapter 3

The Jay System

3.1 Overview

The Jay system is a prototype implementation of a network application infrastructure. Jay demonstrates the practical effectiveness of network applications and reveals implementation complexities that may challenge developers. It incorporates all the components of the network application infrastructure, as well as most of the capabilities of Section 2.3. It includes a complete Application Programmer's Interface (API) and libraries. Jay and the examples presented here are written in the Java language, version 1.2 [CW96][1].

This chapter first describes Jay as seen by the developer, then looks at the internal operation of Jay. The next section evaluates Jay in terms of the capabilities of network applications. Finally, we examine two sample applications using the Jay infrastructure to accomplish realistic tasks.

[1]The current, Second Edition of this text discusses the Java Development Kit version 1.1. Future editions are expected to cover version 1.2. While differences between the versions (especially in the `java.security` package) are important to the implementation of Jay, a developer using Jay can largely ignore these differences.

3.2 Architecture

Jay is closely modeled after the network application infrastructure described in the previous chapter and supports bases, extensions, and extension environments. Here we look at each of these components and details of their implementation in the Jay system. The components are demonstrated using a simple network application, PrintDate. PrintDate starts an extension at a remote environment to retrieve the current time, then prints that time on the local console. The application contains two classes, the base class, `PrintDateBase`, listed in Figure 3.1 and the extension, `PrintDateExt`, in Figure 3.2. For clarity, all error checking has been removed from the PrintDate example.

3.2.1 The Jay Base

As shown in Figure 3.1, Jay bases are normal Java applications with a `main()` function entry point. Jay bases first create an `ExtensionSender` object, as in the PrintDate example. The `ExtensionSender` contains various parameters and settings required before sending an extension to the environment. PrintDate uses most of the default values and so interacts minimally with the `ExtensionSender`. A Jay base then specifies the class of the initial object with a call to `ExtensionSender.setInitClass()` and sends that class and any it may reference to the environment. The example sends only one class by calling `addLocalClass()`.

Once initialized, the extension is started with a call to `ExtensionSender.startExtension()`. Jay creates the extension and returns a `BaseServices` object. This object primarily facilitates communication with the extension through an asynchronous communication stream (a Java `InputStream` and `OutputStream` pair). If the ReturnRemote setting of `ExtensionSender` is set, the `BaseServices` also includes a Java Remote Method Invocation (RMI) reference to allow synchronous, type-safe remote procedure calls to the extension.

Finally, the extension begins executing. The PrintDate example reads a `Date`

```java
import jay.base.*;
import java.io.ObjectInputStream;
import java.util.Date;


/**
 * Starts an extension at a host and prints the date at that host.
 * The first command-line argument is the remote host name.
 */
public class PrintDateBase
{
  public static void main(String argv[])
  {
    // The first command-line arg is the remote host name.
    String environHost = argv[0];

    // The ExtensionSender sends the classes to the remote.
    ExtensionSender sender = new ExtensionSender(environHost);

    // The init class contains the entry point, start().
    sender.setInitClass("PrintDateExt");
    sender.addLocalClass("PrintDateExt");

    // We don't need a reference to the init class; we'll use the socket.
    sender.setReturnRemote(false);

    // Start the init class.
    BaseServices services = sender.startExtension();

    // Expect a Date object from the socket.
    ObjectInputStream in =
      new ObjectInputStream(services.getInputStream());

    // Read and print the Date.
    Date remoteDate = (Date)in.readObject();
    System.out.println("The date and time at " + environHost +
                       " is " + remoteDate.toString());

    // Close the extension.
    services.close();
  }
}
```

Figure 3.1: The `PrintDateBase` class of the PrintDate application.

```
import jay.env.*;
import java.io.ObjectOutputStream;
import java.util.Date;


public class PrintDateExt implements Extension
{
  public void start(ExtensionServices services)
  {
    ObjectOutputStream out =
      new ObjectOutputStream(services.getOutputStream());
    out.writeObject(new Date());
  }
}
```

Figure 3.2: The `PrintDateExt` class of the PrintDate application

object from the communication stream and prints its value to the console. Once complete, the extension is closed with a call to `BaseServices.close()`, causing resources allocated by the extension to be released, including the communication resources and any threads active in the extension.

3.2.2 The Jay Extension Environment

The Jay extension environment accepts TCP socket connections for the purpose of receiving and starting extensions. When a base connects, the environment accepts commands which can be grouped into five categories:

Add classes The base can transfer classes on the connection or can specify a URL from which classes can be retrieved. Either technique allows the transfer of the byte-code of a single class or a complete Java Archive (JAR) [Fla97][Sun96] file containing multiple classes. Commands in this category may be used repeatedly.

Select the initial class Each extension has an initial class, selected by the base, which will be instantiated by Jay.

Identify principals The base can identify any principals from whom the extension should derive permissions. After verifying the identify of the principal, permissions are assigned according to the Java security policy database. Environment security is described in detail in section 3.4.

Set connection parameters The Jay environment accepts two parameters for an extension. First, the base can request a reference to the extension's initial class. If it does (and if the initial class extends `java.rmi.UnicastRemoteObject`, as required by the RMI libraries) then Jay creates an RMI reference to the initial object and transmits it to the base after the extension is created. Second, the base may request any network connections be "disconnectable". Disconnectable network connections can be temporarily disconnected during expected network outages.

Start the extension Starting the extension must be the last command issued by a base. It creates the initial object by instantiating the initial class.

On starting the extension, if the initial class implements the `java.env.Extension` interface, as does `PrintDateExt`, Jay creates a new thread and calls the initial object's `start()` method. The single parameter, an `ExtensionServices`, is similar to the `BaseServices` class. It offers access to an asynchronous stream connection to the base, and an environment name space shared by all the extensions of an environment to facilitate communication between extensions.

When the base closes the extension, Jay allows the extension five seconds to complete[2] before stopping its initial thread and any other threads it created. At that time, Jay's references to the initial object are removed to allow the extension to be garbage collected.

[2]Five seconds is a compromise between allowing the extension a chance to complete and promptly reclaiming resources. An environment creating and destroying many extensions per second would likely prefer less time, while an environment running on a slow or overloaded machine may prefer more time.

3.2.3 The Jay Extension

An extension is created when the base calls `BaseServices.startExtension()` and its `start()` method is invoked if it implements the `java.env.Extension` interface. In the `PrintDateExt` class, the `start()` method immediately sends the current date as an object through the stream connected to the base. On receiving the date, the base closes the extension.

It is likely `PrintDateExt.start()` will complete execution before Jay destroys its thread. However, because `PrintDateExt` executes no statements after sending the date, completion is not necessary. In general, extensions should avoid executing statements after notifying the base to exit to avoid the case where remaining statements may or may not execute.

3.3 Implementation

The implementation of Jay described here is largely complete. It is composed of four Java packages, of which two contain utility classes used by Jay, but not required to write a Jay application. The remaining two packages, `jay.base` and `jay.env`, contain a total of only four public classes and interfaces.

Internally, Jay is written entirely in Java, without any platform-specific native libraries. Java was chosen for its inherent heterogeneity, its natural use of dynamic linking, its ability to construct classes at run-time, its integrated and granular security API, its built-in Remote Method Invocation support, and several other standard Java APIs such as reflection, data collections, and weak references. Most of these features do not exist in Java versions prior to version 1.2.

Although providing considerable support for Jay, Java's APIs are, at times, lacking. Security, for example, allows untrusted code (in this case, extensions) to consume all the memory and processing resources available to the environment. Granting a class permission to write to a single file allows that class to consume unlimited disk space. Although an administrator may prevent any permanent damage to the system such as lost files or unauthorized access to confidential data, several denial-of-service attacks are available to malicious code. The RMI libraries contain similar short-comings. While simple applications using RMI are straight-forward, more advanced developers of Java or Jay applications will find an understanding of distributed garbage collection and Java's RMI stub compiler useful.

The bulk of Jay's code is concerned with moving and managing untrusted code while providing integrated and expressive communication between a Jay base and extension. The entirety of Jay is about 7500 lines of commented Java code. Of that, roughly 1000 lines are specific to the `jay.base` package, roughly 1600 lines are in the `jay.env` package, and the remainder are in the two packages of utility classes. While portions of Jay exhibit considerable conceptual complexity, the most difficult part of implementing Jay was managing what can become a highly distributed system. A single environment must respond quickly and predictably to the actions of potentially hundreds of extensions connected to as many hosts while presenting a simple and consistent interface to each application developer. It is the integration of the concepts used by Jay that make it a unique and interesting implementation.

3.4 The Capabilities of Jay

As an implementation of a network application infrastructure, Jay can be judged in terms of the capabilities described in Section 2.3. This section will revisit each capability, examining the technique and extent of Jay's implementation. Many of the features of Jay leverage facilities of the Java language, and these will be noted as they become relevant. Where Jay's implementation does not incorporate a capability of network applications we will look at suitable additions to Jay and their complications. As appropriate, we will describe the API methods through which Jay offers each capability. A complete list of methods discussed in this chapter appears

Class or interface	Methods	
`ExtensionSender`	`setInitClass()`	`addLocalClass()`
	`addUrlCodebase()`	`addLocalJar()`
	`setReturnRemote()`	`addClassAtUrl()`
	`startExtension()`	`addJarAtUrl()`
`BaseServices`	`connect()`	`getInputStream()`
	`disconnect()`	`manageDisconnectableSocket()`
`ExtensionServices`	`addExport()`	`getOutputStream()`
	`getExport()`	`makeDisconnectable()`
	`removeExport()`	
`Extension`	`start()`	

Figure 3.3: Methods needed by a Jay application developer. `ExtensionSender`, `BaseServices`, and `ExtensionServices` are classes while `Extension` is an interface. Only the methods mentioned in this chapter are listed.

in Figure 3.3. The reader may wish to review the sample applications of Section 2.3 to consider how they would be implemented in Jay.

3.4.1 Module Transmission

Module transmission in Jay combines two of the techniques of Section 2.3.1. Specific extension classes can be pre-loaded before the extension starts or classes can be loaded on-demand as the extension executes. Pre-loaded classes can be transfered directly from the base with a call to `ExtensionSender.addLocalClass()`, or from a URL with a call to `ExtensionSender.addClassAtUrl()`. Alternatively, pre-loaded classes can be transfered as groups, compressed in a JAR file, using the `ExtensionSender.addLocalJar()` or `ExtensionSender.addJarAtUrl()` methods. Classes cannot be transfered after the extension starts, except those loaded on-demand.

Classes loaded on-demand are loaded as late as possible in execution. The base specifies a URL prefix to the environment with a call to `ExtensionSender.addUrlCodebase()`. For each class required by an extension that is neither a system class nor pre-loaded, Jay checks the existence of a URL composed of the URL prefix concatenated with the usual class file name. If the class file exists, it is loaded.

As detailed in Section 2.3.1, loading classes on-demand minimizes bandwidth use but can delay execution. The technique used by Jay includes the advantages of on-demand loading while avoiding some of the disadvantages. Because on-demand classes are not loaded from the application base, there is no risk of stalling or failing when the base is disconnected and the URL prefix can refer to a host connected by a faster network than the base. If a developer desires, classes can be loaded from the base host using the HTTP (or other) protocol, at the risk of the class not being available if the base becomes disconnected. If any class is not available at load-time, Jay's class loader will throw a `ClassNotFoundException`.

3.4.2 Module Reception

Module reception involves unmarshaling, loading, linking, starting, and unloading extensions. Java's platform-independent byte-code representation of classes provides a convenient mechanism for managing these steps. Classes transfered individually are received directly in byte-code form. Classes contained in a JAR file are first extracted and decompressed to derive their byte-code form. In either case, byte-codes are stored until a class is needed, then are loaded and linked using Java's `SecureClassLoader`. The `SecureClassLoader` is responsible for performing byte-code verification [GM96], a necessary first step in preventing loaded classes from performing insecure operations, and for assigning security information to new classes.

For extensions that implement the `jay.env.Extension` interface, a thread is created and used to call the `start()` method of the initial object. The thread is stored by the environment and eventually destroyed (along with any threads it created) once the connection to the base is closed. Releasing memory and other resources held by the extension is left to the usual Java mechanisms: memory is automatically garbage collected once the environment no longer refers to it, and resources held by objects using that memory are released when the object is destroyed.

3.4.3 Communication

Communication within and between modules of a traditional application typically takes the form of method calls (synchronous) or message passing (asynchronous). These mechanisms are specifically designed to be expressive and convenient to the developer: explicit marshaling, unmarshaling, and narrowing type conversions are not necessary. Network applications should emulate these features, even for communication between the base and extension.

Jay does not hinder existing communication facilities within the application base or extension. Further, it facilitates expressive synchronous and asynchronous communication between the base and extensions. Jay automatically supplies a TCP socket as an `InputStream` and `OutputStream` pair connecting the base and each extension for asynchronous messaging. These may be used to transmit ASCII strings, binary data, or entire object graphs using Java's object serialization facilities. In addition, bases of extensions that subclass Java's `UnicastRemoteObject` class are automatically supplied an RMI reference to the extension. This permits remote procedure calls using the same syntax and comparable semantics as local procedure calls (in some cases, pass-by-value is substituted for pass-by-reference [Far98, p.74-75]). Type-safety is maintained whether RMI or object serialization is chosen.

Communication during Disconnection

Although Jay extends the facilities of traditional applications to network applications, the fact that the communication medium of network applications is much less tightly coupled imposes additional communication demands. In particular, mobile wireless hosts, an excellent application domain for network applications, can become temporarily disconnected from the fixed network as they move among buildings or exhaust battery power. Jay permits bases to disconnect from their extensions temporarily by calling `BaseServices.disconnect()`, passing the maximum duration of the disconnection as the only parameter. The TCP sockets used for asynchronous communication and for the RMI reference are closed until the base calls `BaseServices.connect()`. Applications can cause any `DisconnectableSocket` instance to be disconnected and reconnected by registering it with a call to `BaseServices.manageDisconnectableSocket()`. Sockets used by RMI references are made disconnectable and registered with a call to `ExtensionServices.makeDisconnectable()`. If the base requests an RMI reference to the initial object and it requests to be disconnectable, the RMI reference is automatically made disconnectable and registered.

Although an application can best manage the necessary semantic changes due to disconnection, Jay attempts to minimize the disruption in the case the application does not. Asynchronous messages are buffered and delayed until reconnection, allowing the base to continue execution. RMI calls may still be made, but will not return until after reconnection.[3] A base that performs no communication with a disconnected extension is otherwise unaffected.

Jay bases must reconnect with a disconnected extension within the time specified on disconnection or the environment will assume the base has failed and destroy the extension. Unexpected disconnection results in the extension being closed and an exception being thrown to the base when the communication channel is next used.

Communication between Extensions

Communication between extensions in an environment is enabled through a name space supported by the environment for that purpose. Extensions can

[3]Even RMI calls that return `void` cannot return immediately. RMI semantics dictate that the call has completed successfully without throwing an exception and that any side effects of the call have occurred before the call returns.

export to the name space an object reference and its name via a call to `ExtensionServices.addExport()`. Removal is accomplished through the `ExtensionServices.removeExport()` function. Extensions in the same environment can obtain the exported reference by calling `ExtensionServices.getExport()`, passing the reference name as the only parameter. A reference already obtained by `getExport()` is not revoked by `removeExport()`.

By exporting a reference, an extension allows other extensions to call the methods of the referenced object. Using this mechanism, extensions can effectively "meet" in an environment for high-bandwidth, expressive communication. Since the environment name space is accessible to all extensions in the environment, exported objects will typically implement access restriction to allow access to only certain users. Access can be controlled using the policy database of the environment, but since those policies are likely not controlled by the exported extension, alternative arrangements would be advisable.

Jay does not provide such alternatives, but extending the existing Java security mechanisms to use the policy database on the base host would be effective. That task is not trivial. The permissions available at a point of execution depends on the execution stack, which is not transfered during an RMI method call. A normal RMI method call to the base to query for permissions would not take into account the execution stack at the extension. The extension would need to additionally transfer the execution stack or comparable identifying information to the base, or the base would need to transfer the security policy to the extension. Both of these solutions would be able to use the standard Java security mechanisms only peripherally.

Communication and Security

To obtain references to exported objects, their class and the class of any object to which they refer must be dynamically linked to the extension. Providing the extension's class loader with the byte-codes used by the exporter is not sufficient; the class must be loaded by the exporter's class loader for Java's run-time type checker to consider the classes equivalent. When an extension retrieves an exported object, Jay automatically delegates the loading of any otherwise unknown classes to the class loader of the exporter. A single extension may, by this mechanism, delegate to multiple other class loaders, which may, in turn, delegate to still others. An extension developer should ensure that byte-codes for classes of objects retrieved through the environment are not added to the extension at load-time so they do not supersede classes that would otherwise be loaded from a delegate. Similarly, classes of objects created by the extension must be added when the extension is created to prevent them from being satisfied by a delegate class loader.

Vitek et al. [VST97] argue that communication between Jay extensions is insecure since the extensions' object graphs necessarily intersect and "once object graphs cease to be disjoint, security is a lost battle." Once an attacker gains access to an object of another object graph, they argue, the remainder of the graph is highly vulnerable to breaches of secrecy and integrity, masquerading, and denial of service. In particular, "the problem is that method invocation knows nothing about protection domains." They use Java as an example of an insecure language. However, in Java version 1.2, permissions available to method calls across protection domains is clear: briefly, a thread may use the intersection of permissions available to each of the classes appearing on the execution stack.

Still, Vitek et al. show that shared object graphs do present opportunities for an attacker. Application developers unfamiliar with object-oriented programming can accidentally expose confidential information. Developers uncertain of the security of their objects should consider exchanging only objects of built-in types, objects that are immutable and declared final (i.e. cannot be subclassed), deep copies of mutable objects, and objects that do not refer to other portions of the extension.

3.4.4 Heterogeneity

Jay draws its heterogeneity capabilities directly from Java. The transmitted class byte-codes are in a platform-independent intermediate representation and can execute on any platform supporting the Java Virtual Machine (JVM). This currently includes many Unix variants, Windows NT, Windows 95, and MacOS[4]. Jay environments allow the JVM to optionally transform the byte-codes into a native form to improve execution speed.

Jay may be viewed as homogeneous with respect to the language applications may use. This limitation is partly true: Jay extensions must be implemented in Java to achieve the heterogeneity and security requirements. However, Jay bases could be implemented in any language if the necessary client libraries were written, losing only the ability to perform RMI calls. The protocol between the Jay base and extension is intentionally straight-forward and language-independent.

Jay does not permit the transmission of Java native libraries, libraries of native code accessible directly from Java. Since such libraries are platform-specific, transmission would defeat Jay's goal of heterogeneity. Further, native libraries execute with fewer security controls than Java methods, introducing a considerable security breach. Native libraries made accessible to the environment by the environment host are accessible to extensions.

3.4.5 Remote Resource Access

Jay provides access to the network, file, and printer resources of the environment through the Java API, subject to the security policy.

Unusual resources not available through the standard Java API can also be made available to extensions using either of two distinct techniques. The first technique requires the system operator to create an extension capable of controlling

[4] Although Java version 1.2, required to use Jay, is available only for Solaris and Windows NT at the time of writing, ports to other platforms are expected shortly.

the resource, load it into the environment with sufficient security privilege to access the resource, and export an interface to the resource to the shared extensions name space. Other extensions can import the interface to access the resource. The second technique requires the system operator to add classes capable of controlling the resources to the system classes available to all extensions. Both techniques allow access to the resource to be controlled through the security policy. As well, both require the classes or an interface to the classes be available at compile time to allow strong type-checking. The resource itself need not be available. The first technique integrates more smoothly with Jay, allowing the resource interface to be loaded and unloaded while the environment continues to run. The second technique is more familiar to long-time Java users. Using the second technique, the resource availability can be changed only be restarting the environment, but any Java native libraries needed to support the classes can also be installed at that time.

Resource allocation in Jay uses the normal procedures of the underlying operating system, reflecting any resource contention back to the extension. Resource reclamation is largely managed by the object offering access to the resource (e.g. the `Socket` object deletes unused sockets), with the exception of processor and memory resources. Consumption of processing resources is stopped when the base exits by stopping any threads belonging to the extension, and preventing any threads from being created. Memory is reclaimed by Java's garbage collector when there are no more local or remote references to it. Objects exported to the shared environment name space should take particular care, since reference to the exported object can outlive the exporting extension, allowing the service they offer to be accessed after the base exits. Although the environment returns strong references to exported objects, it maintains only a weak reference internally; objects are collected if all remaining references are weak references.

3.4.6 Performance

Jay uses several techniques to minimize the amount of data and number of round trips required to start an extension. A minimum of one and one half round trips are required to start an extension after a socket connection is established with the environment. First, the environment sends an identifier and a nonce [CDK94, p.494] to the base for the purpose of authentication. Next, the base sends a collection of messages about the extensions, including any necessary class byte-codes. Concurrently, the environment responds to these messages. The final response indicates the extension has started.

The only large message transmitted to the environment is the class byte-codes. It is for this reason that Jay allows the byte-codes to be retrieved from a URL located on a faster network than the base, and allows byte-codes to be compressed in JAR files for transmission[5].

The largest message transmitted to the base is the remote reference. For this reason, a base not requiring a remote reference can suppress its transmission. Further, the remote reference is small compared to typical classes and compared to the object it references. The implementation byte-codes for the remote reference are not transmitted, as they are known to exist at the base.

The performance of Jay is not directly comparable to any existing systems since its functionality is somewhat different than any other. For micro-benchmarking, it is most comparable to mobile agent systems and distributed systems that can include functionality at various network locations. Application benchmarks can be used to compare the performance of traditional implementations with a network application implementation, recognizing the implementations will be quite different.

[5] JAR files are created and compressed before the application starts, so the cost of compression is not incurred at run-time. The full cost of decompression is incurred at the environment, but decompression of JAR files is less costly than compression and it is common for processing resources to be more powerful at environments as compared to bases.

We do not include performance measurements of other systems here. Instead, we measure the time and bandwidth consumption of simple Jay operations to assist future performance comparisons and to assist in understanding Jay's performance. We look at three network applications:

- The minimal application, listed in Figures 3.4 and 3.5. It performs no function, does not gain a thread from the environment, and does not return a remote reference to the base.
- The threaded application. It is like the minimal extension, but implements the `jay.env.Extension` interface and the `start()` method it requires.
- The remote reference application. It is like the minimal extension, but the base requests a remote reference to the extension. As required, the base sends stub and skeleton classes to the environment, along with an interface extending `java.rmi.Remote`, but implementing no methods. The extension implements the interface and subclasses `java.rmi.server.UnicastRemoteObject`.

For each application, we start and close one thousand extensions from a single base, measuring the size of the extension, the average number of bytes transmitted to the environment, the average number of bytes transmitted to the base, and the average amount of elapsed wall-clock time per extension.

All tests were performed with the base on a 166 MHz Intel Pentium CPU and the environment on a 200 MHz Intel Pentium Pro. Both machines ran Solaris 5.5.1 and the Java Development Kit version 1.2beta3 distributed by Sun Microsystems. The two machines were connected by a 10 megabit per second (shared, but largely idle) Ethernet network. To improve the reproducibility of the results, optimizations were not used, including the optimizing compiler, native threads, or the just-in-time compiler.

The results are shown in Figure 3.6. We see that implementing `jay.env.Extension`, and thereby gaining a thread, has a minimal impact on perfor-

```
import jay.base.*;

public class MinimalBase
{
  static long totalTime;

  // argv[0] is the host of the environment.
  public static void main(String argv[])
    throws Exception
  {
    for (int i = 0; i < 1000; i++)
    {
      long startTime = System.currentTimeMillis();
      ExtensionSender sender = new ExtensionSender(argv[0]);
      sender.addLocalClass("MinimalExt");
      sender.setInitClass("MinimalExt");
      sender.setReturnRemote(false);
      sender.setDisconnectable(false);
      BaseServices services = sender.startExtension();
      services.close();
      totalTime += System.currentTimeMillis()-startTime;
    }
    System.err.println("Average time to start and close an extension is " +
                       (totalTime/1000) + " ms");
    System.exit(0);
  }
}
```

Figure 3.4: The minimal network application base.

```
public class MinimalExt
{
}
```

Figure 3.5: The minimal network application extension.

	Minimal Application	Threaded Application	Remote Reference Application
Extension size (bytes)	240	345	3060
Bytes transmitted to the environment	453	558	4123
Bytes transmitted to the base	106	106	696
Wall-clock time (ms)	80	81	309

Figure 3.6: Performance tests of Jay. All results are the average of one thousand iterations. Byte counts include bytes transmitted on sockets created for RMI references. The size of the remote reference extension includes the transfered skeleton, stub, and interface.

mance. Requesting a reference to the extension, however, has considerable impact. The larger extension size includes various classes required by the RMI libraries: a remote procedure call stub and skeleton and an interface exported by the extension. The addition bytes transfered and most of the additional wall-clock time is consumed by the RMI libraries in creating and transmitting the remote reference, and in creating a socket connection between the reference at the base and the referenced extension. Still, for long-running extensions, the wall-clock time may be negligible for all configurations. Developers should take care in selecting these features.

We also tested the amount of memory consumed by the infrastructure for each extension by starting up to one thousand minimal extensions from a single base in a single environment and monitoring the change in the size of the base and environment processes. For both the base and the environment, the process size increased roughly linearly with the number of extensions at a rate of 18960 bytes and 73788 bytes per extension, respectively.

3.4.7 Versioning

Versioning allows a base, its extensions, and the network application infrastructure used by each to be of different versions. Jay relies on Java's dynamic linker to implement versioning in the environment. In Java, classes may be successfully dynamically linked against any class that implements the necessary methods. More specifically, any class that contains the called methods with the same parameters and return value (the *method signature*) is sufficient. Methods may be added or removed from a class between versions, and method implementations may change; the dynamic linker requires only that the referenced method signatures exist.

This allows extensions to run on versions of the environment other than the version they were compiled against, so long as the methods called by the extension exist. The environment need only avoid changing existing method signatures to ensure backward compatibility. The disadvantage to this approach is that changes in method behavior are not automatically detected by an application.

Similarly, the extension may be composed of classes different than those it was compiled against. If an extension is compiled against an old version of a library, and loads the newer version of the library from a URL location at link-time, the extension need not fail. Unfortunately, an extension linked against a known version of Jay will run on a modified environment without warning, whether that modification be beneficent or malicious.

3.4.8 Security

Section 2.3 describes security on three fronts: protecting the environment and other extensions from damage by an extension, protecting transmitted data from eavesdroppers, and protecting the application from interference from the infrastructure. The second and third of these are beyond the intended scope of Jay. Protecting transmitted data is a general, well-recognized problem not specific to network applications. Protecting the applications presents enough challenges to justify a new topic of research.

The first of these, protecting the system and users from an extension, is of interest here. In Jay, extensions and their classes can be associated with principals. Classes gain permissions according to these associations, the network location from which their byte-codes originated, and the environment's security policy. This scheme can be examined from two points of view: that of the network application and that of the environment.

Security for the Application

On starting an extension, Jay allows any number of principals to identify and authenticate themselves. All the classes of the extension gain the union of permissions the environment would assign to the principals individually, or the default permission set if no principals are authenticated. In addition, classes transfered in JAR files may be digitally signed by one or more principals. Verified signatures grant the class the permissions available to those principals.

Authentication in Jay is performed using a public key [DH76] challenge/response system to avoid transferring clear-text passwords over an insecure network or to an untrusted environment. At the base, encrypted private keys are stored in a database managed using Java's `keytool` application [Sun98]. To associate a principal with an extension, an application calls `ExtensionSender.addAuth()`, passing two parameters: the principal's alias in the database and the password required to retrieve the private key. Jay encrypts a nonce selected by the environment using the private key, and sends the result to the environment. If the environment is able to decrypt the result using the principal's public key to recover the nonce, the principal is authenticated. Neither the private key password nor the private key are transmitted during this exchange.

Security for the Environment

On receiving an authentication request for a principal, Jay retrieves the corresponding public key from the `keytool` database on the environment's host. If the retrieved key correctly decrypts the nonce, all classes of the extension, whether previously re-

```
grant signedBy "VanGogh",
      codeBase "http://jay.abc.com/HostMonitor.class"
{
        permission java.net.SocketPermission "*.abc.com:161",
          "connect";
};
```

Figure 3.7: A sample Java security policy entry.

ceived or not yet transmitted, are treated as if signed by that principal. Specifically, the classes gain any permissions assigned to that principal by the security policy.

The security policy is specified using standard Java mechanisms [GMPS97]; typically, it is stored on the environment host in a file shared by all other Java applications. The policy assigns permissions (instances of Java's `Permissions` class) to classes based on both the principals associated with the class and the source of the class byte-codes, or code source. In Jay, the code source for classes loaded from the network is the URL of the byte-codes or the JAR files containing the byte-codes. Byte-codes transfered from the base are said to have a code source of "`http://jay-transfer/`"[6], regardless of whether they are contained in a JAR file. As we will see shortly, this allows distinct permissions to be assigned to classes loaded from a URL and classes transfered from the base.

Figure 3.7 shows a sample entry in the security policy database. The entry grants permission to the class (in byte-code form) at `http://jay.abc.com/HostMonitor.class` to create socket connections to port 161 (a port reserved on the Internet for the Simple Network Management Protocol, SNMP [CFSD90]) when loaded by "VanGogh". We can imagine this would allow user VanGogh, the network manager of the abc.com domain, to start an extension that monitors the hosts in the domain, reporting problems back to VanGogh's wireless palmtop computer. Either the principal or code source can be

[6]Using a protocol of "jay" would be more sensible, but Java `URL` instances cannot be constructed using an unknown protocol.

omitted from a policy database entry, in which case the permissions are granted regardless of the principal or code source, respectively. Where multiple policy database entries match the code source and set of principals for a particular class, the class gains the union of permissions granted by the matching entries. Java security policy file syntax is discussed in [Sun98].

Although assigning permissions to a particular class intended to be loaded as an extension is permitted by Jay, it is akin to providing a particular service rather than providing a platform for user applications. In Jay, it is more common to assign permissions to a user and leave the selection of application code to that user. In the example of Figure 3.7, it would have been more appropriate to allow VanGogh to connect to SNMP ports regardless of the extension code source. VanGogh could then use a host monitoring application of his choice without compromising security.

Principals need not have accounts on the environment host, they need only be listed in the public key database. Currently, this implies the environment host must know, *a priori*, of all principals and their public keys. The Java security framework, however, is specifically designed to allow a more advanced public key infrastructure, including widely trusted certificate authorities, to be dropped in place of the current scheme. Jay would adopt the new scheme (or any other scheme used by a host) without need for recompiling and would automatically incorporate keys available from the certificate authorities.

Permissions listed in the policy database can grant granular access to specific files, directories, network end-points, and many other resources; or can broadly allow access to all files or complete network access. New permissions can be easily created, allowing resources or other services offered by other extensions to be protected using this same mechanism.

Notably absent from Java's security measures is the ability to limit processor or memory resource consumption. Extensions are able to consume all the processing and memory resources available to the environment. This same lack of control applies equally to all Java applications. Since the JVM provides little information on processor or memory usage, and that only for the entire environment, restricting these resources is not possible in Jay.

Also absent is the ability to authenticate principals listed in public key databases maintained at other sites. This ability is required to implement the sample security policies of Section 2.3.8, to wit, a group of wireless network access providers wanting to allow all their subscribers use of any of their network application environments, and an anonymous FTP site wanting to allow access to any user authenticating in any domain. Adding this ability is a straight-forward extension to Jay. Along with providing a signed nonce, users could optionally provide their public key contained in a certificate signed by a widely trusted certificate authority [Bra97]. The public key would be used to authenticate the user and may be added to the public key database.

3.4.9 Fault Tolerance

Fault tolerance in Jay is targeted less towards failure recovery than towards avoiding and detecting failures. To avoid communication failure, bases may disconnect the communication channel before an anticipated failure of the communication medium, as described in Section 3.4.3. Detection of failures either in the communication medium or the environment results in an exception being thrown in the base. The Java compiler forces the base to handle the exception. A failure of the base results in its extensions being closed.

No attempts are made in Jay to recover from a communication failure, although such a feature would not be unreasonably difficult to implement. A failure of the communication medium could result in automatic disconnection of the extension. Jay would need to detect data lost during socket failure and retransmit the data after reconnection. The application would need to manage unexpected disconnection instead of managing unexpected failure.

Similarly, sockets could be automatically reconnected when buffered data needed to be transmitted and the communication medium was restored. Implementing both automatic disconnection and reconnection would allow applications to ignore unexpected disconnection entirely; an application anticipating data from an extension would apparently pause during periods of disconnection. For many applications, unexpected delays of arbitrarily length are not acceptable, particularly during user interaction or real-time monitoring. In that case, the infrastructure would need to revert to the case without automatic disconnection: the application should be notified of the disconnection and manage it appropriately.

3.4.10 Concurrency Control

Java's concurrency control model attaches a lock for thread synchronization to objects. Both the object's methods and methods of other objects can use the lock for synchronization. Because there is no built-in notion of distributed objects in Java, synchronization within an object can always be performed locally; such lock access need not be extended to work between the base and extension. Acquiring a lock on another object, however, can involve objects at both the base and extension. Ideally, Java's usual mechanisms would be extended by Jay, but since Java's synchronization primitives cannot be overridden by an application, Jay cannot do so. Instead, synchronizing on a reference to a remote object acquires a lock on the reference stub, not the referenced object.

One can imagine extending Jay to include concurrency control for the purpose of consistency management between a base and extension. Objects could be replicated and a transaction model, a locking protocol, or another form of synchronization could ensure a desired level of consistency. Given the existing communication capabilities of Jay and the thread synchronization available in Java, implementing these techniques in Jay would be a lengthy but straight-forward exercise.

3.4.11 Development Tools

The Java language is designed to enhance development and reduce errors by ensuring compile- and run-time type-checking. Jay preserves these properties in RMI method calls and object serialization between the base and extension.

Most notably lacking is a debugger for extensions. Such a debugger would require support from the JVM that is not currently provided. Although there are JVM facilities for debugging applications remotely, they are tailored towards debugging all the threads of a JVM rather than a subset.

3.5 Sample Network Applications

As a final test of the effectiveness and usability of network applications and the Jay system, we will look at the implementation of two sample applications, Multi-Host Ping and Lightcycles. Each is a simple application that demonstrates the effective use of Jay to perform tasks that are not possible with traditional applications. In addition, they stand as a testament to the completeness and extent of Jay's features.

3.5.1 Multi-Host Ping

The Multi-Host Ping application is intended to monitor network activity. It starts extensions on a collection of hosts and measures network round-trip (ping) times between all pairs of hosts. Ping times are reported back to the base where they are displayed on a grid and recorded for display over time. The base also records for display a history of ping times for all pairs of hosts.

In its steady state, the environment of each host displayed by the base runs an extension. Each extension knows of all the other extensions and transmits ping packets, UDP/IP packets containing two bytes of data, to them. Extensions listen on a UDP port for incoming ping packets and return a two-byte UDP ping reply to the sender. Ping packets are sent no more frequently than once each second and there

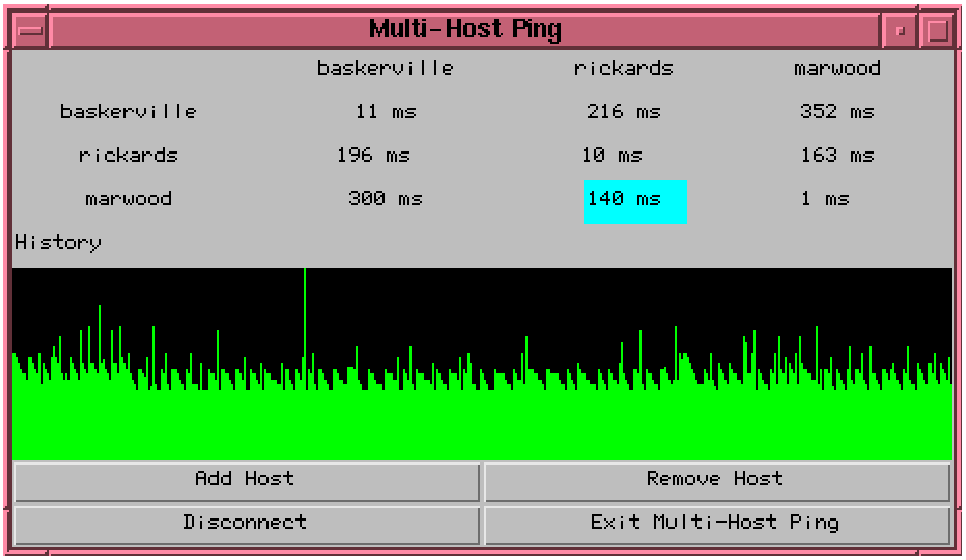

Figure 3.8: The Multi-Host Ping application user interface. Network round-trip (ping) times are shown in milliseconds from the host listed on the left to the host listed on the top and back. The History windows shows past ping times for the selected grid entry.

is never more than one outstanding packet to any host. If a ping reply is received from a host within a five-second time-out period, the elapsed time between sending and receiving is reported to the base. Ping replies arriving after the time-out period or not at all are reported to the base as lost. Ping times are sent to the base using Jay's asynchronous communication facility to avoid delays in transmitting further packets. Sending and receiving UDP packets requires the following two permissions, the second of which is granted by the default Java security policy:

- `permission java.net.SocketPermission "*:1024-", "connect";`
- `permission java.net.SocketPermission "localhost:1024-", "listen";`

Figure 3.8 shows a screen image of Multi-Host Ping. The three hosts used in this example, Baskerville, Rickards, and Marwood, are displayed on a grid. A fourth host executes the base of the application. Each grid entry is the ping time in milliseconds between the host listed on the left to the host listed across the top. Ping times are updated as they are received. For communication lines with symmetric bandwidth and delay, we would expect ping times for each direction between two hosts to be comparable. On the bottom of the screen is displayed a history of ping times between the selected pair of hosts, Marwood and Rickards.

Reception can be temporarily interrupted if the user disconnects the base from its extensions using the “Disconnect” button. During disconnection, the base and extensions remain functional, but all reports are buffered at the extensions. A “Connect” button replaces the “Disconnect” button during disconnection and may be used to reconnect, at which time buffered reports are immediately flushed to the base and the display is updated accordingly. Multi-Host Ping disconnects for a maximum of sixty seconds. If the user fails to reconnect during this time then each environment assumes the base has failed and destroys its extension. On reconnection, the base will immediately detect the failure of all the extensions and remove them from the display.

A user may use the “Add Host” button to start a new extension. The new extension is created and returns to the base a UDP port number on which it will reflect ping packets. All other extensions are informed of the new host and its UDP port. Jay’s synchronous communication facility, the RMI reference, is used when starting an extension to ensure the extension has fully started before other extensions start sending ping packets to it.

The “Remove Host” button stops an existing extension. Other extensions are informed of the impending removal so they may stop transmitting to that host, and the extension is closed. Jay automatically stops the extension threads, thereby allowing it to be garbage collected.

When an extension fails, either due to a failure in the network communication or a hardware failure of the environment host, attempts to read ping reports at the base throw an exception. The base recognizes this exception and removes the extensions first from the display and then from the destination lists of the other extensions. If the environment survives the failure, either because the failure occurred in the network or because of a software failure in the extension, the environment will destroy the extension. Thus, unexpected failure always results in remote resources being reclaimed.

The implementation of Multi-Host Ping offers a real example of the amount of code required to manage extensions. In the base, the `addHost()` function takes a host name as its only parameter, starts an extension on that host, updates internal data structures, and notifies other extensions of the new extension. The function is 84 lines in length. 24 lines of that function create the extension, of which roughly half are detailed error checking and nine are calls to `ExtensionSender.addLocalClass()` (these nine lines would combine into a single method call if the extension classes were placed in a JAR file). The compiled extension is contained in nine Java class files, including the RMI stub, skeleton, and remote interface for a total of 17936 bytes of class byte-codes. Contained in a JAR file, the byte-codes occupy 11662 bytes, but the current implementation inefficiently transmits the classes individually for simplicity during development. In the extension, only four lines specifically exist to allow the extension to run within an environment. The application as a whole is 1446 lines, including the extension.

3.5.2 Lightcycles

Lightcycles is a game for four players at different locations in the Internet. In the game, each player controls the direction of their lightcycle, a constantly moving point of light, leaving a trail in their wake. Players who guide their lightcycle across a trail are removed from the game, and the last player in the game wins.

The Game Server

The game server is a network application that exports a game interface into a remote environment name space. Players join the game by starting a client extension in the same environment, and retrieving the exported game server. Once four players have joined the game, it begins.

During the game, the game server controls forward movement of the lightcycles. The movement events of all players' lightcycles are queued. Client extensions can request the next event in the queue, typically transmitting the events to their base for display. They may also request the game server immediately turn their own lightcycle to a new direction. The environment prevents client extensions from interfering with the game server in any other way.

The Game Client

Each Lightcycles player starts a client that presents a square, white board with four players of different colors located in the center of each side. The client starts an extension in the same environment as the game server and retrieves the exported game. When the game starts, each player's lightcycle starts moving away from the side. The player is then able to change the direction of travel using the keyboard direction keys: up, down, left and right. The "simple" version of the client immediately transmits the direction through the client extension directly to the game server. Baring network latency, the lightcycle will immediately turn to the selected direction. Figure 3.9 shows a game in progress.

Since the game server is located remotely from each client, there is potential for network latency and jitter to interfere with the real-time activities. This can make the game considerably more challenging for users with lower quality network connections. To reveal the effects of delay in a controlled situation, users are able to artificially introduce latency and jitter on their network link. Incoming packets, mostly player position updates, are delayed accordingly, although the client thread

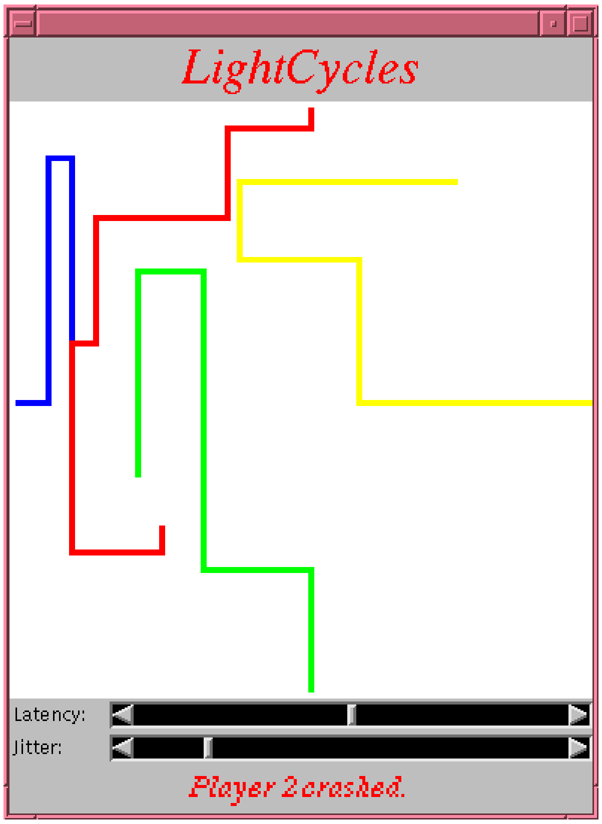

Figure 3.9: The Lightcycles game client as seen by one of the players. The other three players see similar displays.

does not block. Not surprisingly, increased delays worsen player performance.

To demonstrate the advantage of network application extensions, the Lightcycles game can be used in "real-time" mode. To counteract latency, players using real-time mode can pre-select a direction by holding the shift key while choosing a direction. The pre-selected direction has no immediate impact, but is transmitted to the client extension. The client extension continues to pass movement events to the base. However, if the extension detects that the lightcycle will soon crash into a trail, the extension immediately turns the lightcycle to the pre-selected direction. There is no network latency in this process as all communication occurs within a single environment.

Lightcycles as a Network Application

A traditional implementation of the Lightcycles game server in the C language would accept socket connections directly from clients. Clients would transmit their direction selections to the server and the server would transmit player movements to the client. By this means, clients could access the Lightcycles game, a software resource, remotely.

The simple network application implementation of the Lightcycles client is fundamentally comparable to the traditional implementation, and demonstrates how a network application gains remote access to a resource, the Lightcycles game server. The simple implementation has a few advantages over the traditional implementation. First, the interface to the game server has built-in type safety. Second, an incorrectly implemented game server could not breach desired security constraints. In the traditional implementation, an implementation error could allow unintended access to system resources, as has been demonstrated by a variety of services implemented using traditional techniques [Sta89]. In contrast, an implementation error in the network application game server is still subject to the system security policy. Comparable errors would appear in the security policy database, a much simpler description language than C; or in the Java security architecture, a much better tested system than the Lightcycles game server. Unfortunately, these two advantages alone do not justify the conceptual complexity of a network application infrastructure. The simple client stands only as an example of using network applications to gain remote access to a local resource, not as an effective use of network applications.

It is the real-time mode of the Lightcycles client that effectively demonstrates the network application infrastructure. By making decisions within the environment, the game client can react much more quickly to real-time changes than the simple version. It is the ability of the user to extend the application to the environment that allows user-programmable decisions to be made there, an advantage the traditional implementation cannot offer. Further, the intelligence of the extension is entirely tailored by the user. It is easy to imagine game clients making much better decisions than the client described here.

It is important to recognize that the simple and real-time network application implementations do, in fact, extend the client application to the environment. The division between client and server is not at the network, as with the traditional implementation, but within the environment. It is the location of this division that enables network applications to offer what traditional applications cannot.

3.6 Summary

This chapter described the implementation of Jay, a network application infrastructure closely modeled after that described in Chapter 2. Jay provides a straightforward interface for creating, starting, and communicating with network application extensions in remote environments. Jay implements most of the capabilities of network applications, notably excepting development tools. Where network applications present complications above those of traditional applications, we have seen how to address them with Jay. The sample applications of Section 3.5 stand as evidence that the network application capabilities, and in particular the subset implemented by Jay, allow the implementation of applications that perform tasks that are difficult or impossible to implement as traditional distributed applications.

Chapter 4

Related Work

The ideas behind network applications draw on several similar areas. This chapter will specifically discuss software agents, mobile code, active networks, mobile computing, and intelligent networks. We will look at important projects in each area, comparing them to the approach taken by network applications.

4.1 Mobile Software Agents

From a technical perspective, mobile software agents share many capabilities with network applications. Both store data and run code remotely to incorporate functionality into specific network locations. Both face issues of module transmission and reception, heterogeneity, performance, and security. The solutions used in these systems are often applicable to network applications.

The differences are primarily conceptual. Mobile agents do not have a base or other "home"; they are independent entities responsible for their own existence. Mobile agents are able to change their location while executing[1]. Suggested applications typically target fixed networks, although the capabilities of mobile agents

[1] To be precise, network applications do not forbid movement of extensions, however Jay does not support it. Also, most mobile agent systems and other mobile code systems do not allow a thread to move with the agent, but instead start a new thread at a known entry point after the agent moves. Exceptions include Agent Tcl [Gra96] and the Emerald [JLHB88] mobile object system.

make them almost as suitable as network applications in a wireless scenario. In contrast, extensions are a part of a larger application, rooted by a base. Communication between base and extensions is cohesive and expressive, trust is implicit, and threads may move easily between an extension and base. These conceptual differences affect important design decisions in communication and other areas.

Knabe [Kna95] describes a mobile agent system implemented in Facile [TLP+93], a variant of the ML [Pau96] language. Knabe's language and infrastructure, collectively called MSA, implements many of the capabilities listed in Section 2.3 as well as others suitable to mobile agents. Above the features provided by ML, the system includes:

- module selection and transmission. Referenced ML functions are determined when the agent is compiled and transmitted entirely before the agent starts.
- four distinct code representations to facilitate heterogeneity: a high-level interpreted Lambda language, a platform-independent "middle CPS"[2] byte-code form, a "final CPS" form with a small amount of platform-specific structure, and an optimized machine code form for transmission along with a platform-independent representation.
- dynamic linking using proxy structures not unlike Java method signatures.

Among the network application capabilities one would expect in a mobile agent system, only security and development tools are missing.

The Aglet Workbench (AWB) project at IBM's Tokyo Research Laboratory [KZ97] is a software agent infrastructure implemented in Java. Like Jay, AWB draws on Java to implement several capabilities, including module transmission and reception, communication, heterogeneity, remote resource access, performance, versioning, and security.

[2]The Continuation-Passing Style, or CPS, is an intermediate code representation described in detail in [App91].

The current Alpha 5c release of AWB is different from Jay most notably in module transmission and communication. Module transmission in AWB is performed on-demand. Classes are transfered from the agent's or "aglet"'s starting host when they are first referenced during execution. Aglet execution is necessarily delayed when a new class is referenced, and an aglet must never be disconnected from its starting host. AWB communication is performed using synchronous or asynchronous messages queued by priority and order of receipt. Messages cannot be type-checked at compile-time, and must be of a class known to both the sender and receiver.

Perhaps the more striking differences between AWB and Jay are in the areas of security and resource control. AWB security offers only two levels of trust, trusted and untrusted, based entirely on whether the aglet was started locally or remotely. Although aglets are associated with an owner, no attempt is made to authenticate the owner and no permissions can be gained through this property. File, network, property and other permissions can be set for both trusted and untrusted aglets. Resource reclamation in AWB is relatively limited. Aglet resources are never forcibly reclaimed by the AWB, even if the user and host starting the aglet permanently leaves the network. Further, aglets may travel to an unlimited number of hosts during their lifetime. Fortunately, future releases of the AWB promise improved security and resource control. [KLO97] describes a security model for aglets including several principals responsible for an aglet, a highly flexible security context, and resource limitation and reclamation on a local and network-wide scale.

The third mobile agent system we will examine here, AgentTcl [Gra96], is similar in capabilities to the AWB. Its implementation of those capabilities is distinct, largely because of the use of the Tcl [Out94] language and the SafeTcl extensions [LO95]. As an interpreted scripting language, Tcl is an excellent choice for developing mobile agents. Since source code can be transmitted directly to remote hosts for execution, module transmission and reception is simple and heterogeneity and versioning are inherent. For communication purposes, agents exist within a network-wide hierarchical name space. Messages are passed using the agent's name as the destination. Remote resource access is similar to Jay: "indirect" resources are accessed through other agents that may enforce security controls of their choosing. Security of "builtin" resources, categorized as wall-clock and CPU time, screen, network, file system, and external programs, is controlled by diverting insecure requests through a resource manager. On starting, agents authenticate using a public key system. Access is granted by the resource manager only if the authenticated principal appears on the resource category access list.

The most unique aspects of AgentTcl are derived from the underlying Tcl language. Tcl stores all variables as strings and data type is implied only by usage. Semantic checking occurs at run-time and at the remote host. The lack of compile-time checking increases the need for effective remote debugging tools, as is recognized by the authors of AGDB [HK97], a debugger for AgentTcl. Finally, performance suffers in AgentTcl compared to the other code representations considered here. While advocates have asserted that the tasks for which scripting languages are best suited are rarely performance-oriented tasks, the authors of AgentTcl highlight efficient access to remote resources and information retrieval as suitable tasks for agent systems.

The mobile agent systems above are a broad but small sampling of available systems. The once-popular commercial Telescript [Whi96] system by General Magic, Inc. included a powerful security model and features for electronic commerce. Tacoma [JvRSS95] offers fault tolerance in Tcl through the Horus [RBM96] toolkit. More recent efforts have made use of Java. Odyssey is a Java implementation by General Magic incorporating concepts from Telescript. Mole [BHR97] is an early and evolving Java implementation used to research security, performance, communication and other capabilities.

As exemplified above, mobile agent systems include many of the capabilities of network applications. As such, agent implementations offer guidance in the implementation of a network application infrastructure. Network applications are distinct in both the distributed state and the integrated, expressive communication between the base and extension. In Jay, this manifests as the ease with which threads may move between the base and extensions and the degree to which structured data can be shared.

4.2 Mobile Code

The terms "mobile code" and "mobile object systems" have been used to characterize a variety of platforms. Here, mobile code refers to systems that enable a single application to execute components at multiple network locations, as do network application infrastructures. These systems emphasis the cohesion of an application while allowing for distribution. As examples, we will look at the Emerald system and the Rover toolkit.

The Emerald language and run-time system [JLHB88] enables general-purpose object mobility. A collection of Emerald run-time environments act as peers. Emerald applications start in one environment, but application objects may move freely to other environments. Emerald objects are compiled to a byte-code form, allowing for heterogeneity and module transmission and reception while maintaining a degree of performance. Communication is by synchronous remote procedure calls, syntactically and semantically indistinguishable from local procedure calls. Emerald offers no security among peered environments; objects may move freely among peers to execute in any environment without restriction. Network data is sent as clear text and, like the other systems in this chapter, the application has no protection from interference by the environment. The failure of application components can be detected by applications, and immutable objects are automatically replicated. No development tools exist for either local or remote objects.

The Rover toolkit [JTK97] specifically targets wireless mobile hosts. The mobile host may start a Rover application, written in Tcl, on the fixed network and import replicas of selected application objects called RDOs. The replica presents a user interface and communicates user operations to the application using queued remote procedure calls (QRPCs). QRPCs are asynchronous, returning a promise [LS88] to the application. Each sample application presented in [JTK97] requires C or C++ code to be installed at the client before the application can be imported. While this limits general-purpose object mobility, Rover's target platform is strictly mobile hosts.

Emerald, Rover, and Jay are conceptually similar as mobile code systems even though a variety of their implementation details are distinct. Emerald offers a general-purpose platform in which environments are peers. Rover models the application executing on the fixed network, while the mobile host communicates through object replicas. Jay starts the application first on the local or mobile host and allows extensions to start remotely. All three emphasize a high degree of cohesion within a distributed application. A key difference is that Jay addresses security issues in executing untrusted application code at remote locations.

4.3 Active Networks

Active networks, as characterized by Tennenhouse et al. [TSS+96], are networks in which "the routers or switches of the network perform customized computations on the messages flowing through them." Computations are customized by the application to make "intelligent" choices at routers. Using active networks, applications can extend existing router behavior to implement multicast packet delivery, reduce jitter of multimedia data streams, or other special-purpose network functionality.

Each active network capsule is a network message containing a typically small amount of application-specific code to be executed at some or all of the routers the capsule passes through. The code normally executes without authentication in a transient environment, although it may alter non-transient router state. System access is largely limited to network functions: testing packet and network state and generating and forwarding packets. Other than the payload data, capsules carry very little state, typically not more than a few parameters. Numerous active network implementations exist; here we will examine two: Active IP and PLAN.

The Active IP system [WT96] incorporates an IP header option containing Tcl code to be interpreted by selected specialized routers. Remote resource management, performance, and security are not major goals of the prototype. It allows arbitrary code to be executed in a restricted Tcl interpreter without authentication, although code size is limited by the maximum size of the IP option. It supplies a simple library to access payload and router information and to generate new messages. In addition to the multicasting and jitter reduction examples recommended at the start of this section, the authors of Active IP consider active networks a viable replacement for mobile proxies, video gateways, and TCP snooping at wireless base stations, as well as higher-level services such as Web proxies, file caching, and transcoding of images.

The Programming Language for Active Networks (PLAN) [HKM+98] is similar in concept to Active IP, but strives for security and resource management. PLAN programs run without authentication but are restricted from accessing system services. Programs can be proven to terminate within a predictable number of steps and are bounded in the number of routers they may visit. PLAN is designed primarily for low-level network functions such as resource discovery and network diagnosis.

Active IP and PLAN are typical active networks and allow easy comparison to network applications. They share several capabilities with network applications for the purpose of involving the network in a computation: module transmission and reception, heterogeneity, remote resource management, performance, versioning and, to a limited degree, security. However, there are more differences than similarities. Network applications use a (probably passive) network to install extensions at remote hosts. Extensions perform user tasks at the application level, sharing considerable application state with the base. Active networks function at the network level: their capabilities emphasize brief decisions at routers to enhance low-level network protocols. Code is associated with a packet rather than a connection or stream and runs without authentication or access to general-purpose operating system services.

4.4 Mobile Computing

The field of mobile computing looks at issues specific to mobile, resource-poor hosts. While there is no distribution of code inherent in mobile computing, networked mobile hosts make an excellent target platform for network applications because their mobility necessarily increases their degree of network awareness. Looking at problems and solutions in mobile computing gives insight into issues faced by network applications on that platform. How well network applications address these issues determines their utility on mobile hosts.

Satyanarayanan [Sat96] explains that mobile computing is not simply a special case of low-bandwidth distributed computing, nor are the limitations on mobile computing specific to current technology. He lists four constraints on mobility, each of which is ameliorated by network applications.

- "Mobile elements are resource-poor compared to static elements." Using the mobile host as a base, network application extensions allow mobile hosts to draw on the resources of more powerful fixed (static) hosts.

- "Mobility is inherently hazardous." Mobile hosts are more easily destroyed by being dropped or lost. Although network applications cannot prevent the destruction of a mobile host, they encourage data to be safely stored remotely by making access to remote storage possible.

- "Mobile connectivity is highly variable in performance and reliability." Mobile code placed at key network locations can use pre-fetching and caching to reduce perceived variability. Infrastructures supporting disconnected operation allow processing to continue even during network outages.

- "Mobile elements rely on a finite energy source." By moving computation off a mobile host, power consumption is reduced.

Voelker and Bershad [VB94] explore mobility and the importance of location to applications on mobile hosts. Under this topic, others have considered discovery of resources as the mobile host moves to new network locations and sensitivity to surrounding network characteristics. In general, network applications are not expected to facilitate mobility *per se*, although a recognition of location and environment would be useful at network extensions. In fact, network applications can be seen to increase the complexity of mobility by introducing a cleavage plane [Dea98], a place where one entity, the base, is mobile with respect to another, each extension.

In all, while network applications can aid mobile computing, there is still work to be done to enable applications to react to changes in location and environment.

4.5 Intelligent Networks

Intelligent network software or "middleware" attempts to improve network performance by placing network services at carefully selected network locations. Middleware services can reduce network load, speed computation, and otherwise achieve results not possible using end-to-end protocols.

The techniques used in intelligent networks and those used by network applications are fundamentally the same. Both place "intelligent" code oriented to user needs at key network locations. In intelligent networks, unlike network applications, that code is selected by network operators and designed to efficiently serve large groups of users with similar needs. Network applications take a contrasting approach: code is selected by the user to serve that user alone.

The Squid Internet Object Cache [fANR] and CERN httpd [LNBL] cache Web documents near clients to reduce latency and network traffic. Correctly configured and serving an appropriate user base, these intelligent network services can satisfy 55% of requests without consulting the Web [DMF97].

The TranSend system [FGBA96] translates between encodings of Web data. Clients pass all requests through the TranSend server. The server ensures the data returned to the client is of a minimum size and of a format suitable for the client, typically passing it through lossy compression and conversion filters tailored to the client's hardware capabilities. Bandwidth (and consequent latency) savings of four to ten times and higher are reported, depending on the filtering performed.

The Large-Scale Internet Middleware [TOHO97] project at the Information Sciences Institute proposes several intelligent network services. Two of these are relevant here, both directed at reducing Web latency. The first, Lowlat, pre-fetches Web documents referenced in a requested document, storing them in a nearby Web cache. The second, the Multicast Web Tuner, causes Web servers to push documents requested by one user to a set of known Web caches. Assuming documents requested by one client will be requested by others, this will reduce latency.

In all the systems described above, the location of the service is crucial. A Web cache serving too few users will achieve a very low hit rate, while a Web cache distant from its users will offer little latency reduction. Further, the most desirable location is highly dependent on the client's particular situation. Consider a TranSend server located distant from a client. If the server is used to reduce image size and bandwidth requirements, there will be no saving in latency on requests to Web servers near the client. Alternatively, if the server is used to convert images from a format that cannot be displayed to a format that can, latency may be of secondary importance to the client. The pre-fetching implemented by Lowlat can definitely reduce latency. A client connected by a slow but largely idle wireless network link will likely pre-fetch documents across the wireless link to achieve the greatest latency reduction. However, a client that is charged per byte crossing the link may prefer to pre-fetch only to the remote end of the link to circumvent any latency in the fixed network. In general, the needs of a client are dependent on the client's situation and that situation is not easily accommodated by intelligent network systems.

Intelligent networks, like network applications, enable code executing in the network to enhance application performance. However, because intelligent networks perform tasks selected by network operators, they are best suited to serving large numbers of users. Users with specific or unique needs may find the offered services to be non-optimal or desired services non-existent. Intelligent networks and network applications, then, are not competing solutions but complementary solutions to a single problem.

4.6 Summary

We have seen how network application infrastructures draw on concepts also used in other areas to create a new architecture. Like intelligent networks, code executed within the network, between the participants of a distributed system, can perform tasks that cannot be accomplished at the edges of the system. Like mobile agents and active networks, they allow applications to select code to be run remotely in a secure environment. By combining code mobility, security, and several other concepts exhibited by the systems described here, network applications offer more than the individual systems offer collectively.

Chapter 5

Conclusions

Traditional distributed applications typically followed a client/server model: clients and servers run on different machines communicating through an opaque network "cloud" using a standardized protocol. Sometimes this simple model is inadequate for an application, either because the does not satisfy the needs of a particular user or application instance, or because the server does not offer a particular service desired by the application.

A network application infrastructure aims to meet the needs of such applications by extending application functionality into the network. Specifically, network applications can start portions of their code at remote network locations as extensions. Extensions operate as part of the application, but execute remotely.

5.1 Network Applications and Jay

By starting code at remote network locations, applications can perform certain tasks more efficiently, and certain applications can be implemented that would otherwise be impossible using traditional distributed system techniques. Using network applications, efficiency can be improved to the benefit of both the network and the user by monitoring data remotely and transmitting only important events to the base; by transforming an existing protocol into one more suited to the application by compression, distillation, or other filtering; or a variety of other techniques. In addition to merely optimizing existing interactions, network applications enable otherwise impossible tasks such as high-bandwidth, real-time interaction with a remote resource; access to services available only locally on a remote host; protocol conversion to an encrypted form or between network-level protocols at gateways; or network monitoring and diagnosis from remote network locations.

These sample applications demand a variety of capabilities from the infrastructure. Besides the vital tasks of module transmission, module reception, and communications a network application must allow for heterogeneity, remote resource access, performance, versioning, security, fault tolerance, concurrency control, and software development tools. Since a network application infrastructure must integrate smoothly with existing facilities to allow a base and extension to communicate easily, the implementation details of these capabilities depend highly on the existing facilities. However, a variety of concerns are common to network application infrastructures and we have examined options and tradeoffs of their implementation.

The Jay system demonstrates a network application infrastructure incorporating most of the capabilities, with an emphasis on communication and security. It is intended to reveal implementation complexities and, through sample applications, demonstrate the effectiveness of network applications. The Multi-Host Ping and Lightcycles applications both demonstrate the use of a network application infrastructure to accomplish tasks that cannot be implemented as traditional applications. Both run portions of their code remotely in a secure environment using highly cohesive communication between the base and extensions to tightly integrate application modules.

5.2 Conclusions

Our examination of network applications and the prototype implementation, Jay, shows that network applications meet the goals initially set forth. They are an effective technique for extending application functionality into the network in a secure and cohesive manner. Jay shows that, in additional to making network applications possible, an infrastructure can be designed to do so with minimal additional code and minimal modifications to familiar design patterns. Jay applications can be written in a fairly natural and intuitive form with only a few lines of additional code to start an extension. Only time and experience can show whether the advantages of the techniques presented here are sufficient to encourage their use, but the ideas behind network applications offer a new way to think about application design.

5.3 Future Work

This thesis presents the ideas of network applications and a prototype infrastructure. Although the results are promising, there is room for further research. That research should focus on revealing the advantages and shortcomings of network applications through experimental implementations. First, Jay could be extended to improve the functionality of certain capabilities. Communication between extensions currently allows careless extension developers to introduce security vulnerabilities by unintentionally exposing references to confidential information. Vitek [VST97] suggests a more secure alternative. Security could be integrated with a more complete public key infrastructure [Bra97] to allow new principals to identify themselves. Standard fault tolerance techniques could aid failure recovery, although such techniques are often more easily and more efficiently implemented by the application. Finally, a debugger modeled after AgentTcl's AGDB would be an obvious addition to the development tools. The resulting more complete infrastructure would allow more complete testing of network applications.

A second step in exploring the advantages of network applications would be to implement more significant applications, possibly on a large and widely distributed network of environments. Information monitoring and knowledge gathering applications would fit naturally into the infrastructure, as would existing applications optimized to take advantage of the ability to change network behavior. Such applications would undoubtedly reveal additional desirable capabilities.

Third, subjecting a well developed network application infrastructure to rigorous tests of speed and sensitivity to network properties would allow for further optimization of the infrastructure. Micro-benchmarks measuring specific infrastructure operations would aid in fine-tuning and help application developers to determine *a priori* the cost and benefits of using the infrastructure. Application benchmarks would offer an overview of the performance of common sequences of operations. For example, an application to synchronize directories on distinct hosts could be written as a traditional application using an existing FTP server, as a network application using the same FTP server but running an extension near the FTP server, and as a network application with direct access to the file system through the system API. The speed, reliability and improved functionality of the network application would be a valuable result.

In all, network applications successfully combine ideas from a collection of related fields to create a novel approach to distributed application development. At the same time, the existing research leaves room for both refinement and innovation.

Bibliography

[App91] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1991.

[BHR97] Joachim Baumann, Fritz Hohl, and Kurt Rothermel. Mole – concepts of a mobile agent system. Technical Report TR-1997-15, Universität Stuttgart, 1997.

[Bra97] Marc Branchaud. A survey of public-key infrastructures. Master's thesis, McGill University, 1997.

[CDK94] George Colouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems: Concept and Design.* Addison-Wesley, 1994.

[CFSD90] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (SNMP). Internet Engineering Task Force RFC 1157, May 1990.

[CW96] Mary Campione and Kathy Walrath. *The Java Tutorial : Object-Oriented Programming for the Internet, Second Edition.* Addison-Wesley, 1996.

[Dea98] Alan Dearle. Toward ubiquitous environments for mobile users. *IEEE Internet Computing*, pages 22–32, January-February 1998.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[DHR+98] S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, and L. Repka. S/MIME version 2 message specification. Internet Engineering Task Force RFC 2311, March 1998.

[DMF97] Bradley M. Duska, David Marwood, and Michael J. Feeley. The measured access characteristics of World-Wide-Web client proxy caches. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 23–35, Dec 1997.

[fANR] National Laboratory for Applied Network Research. Squid internet object cache. URL: `http://squid.nlanr.net`.

[Far98] Jim Farley. *Java Distributed Computing.* O'Reilly & Associates, Inc., 1998.

[FGBA96] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Oct 1996.

[FGC+97] Armando Fox, Steven Gribble, Yatin Chawathe, Eric Brewer, and Paul Gauthier. Extensible cluster-based scalable network services. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 207–218, Copper Mountain, CO, Oct 1997. ACM Press.

[FGM+97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Internet Engineering Task Force RFC 2068, January 1997.

[Fla97] David Flanagan. *Java in a Nutshell*. O'Reilly and Associates, 2nd edition, 1997.

[GM96] James Gosling and Henry McGilton. The Java language environment. Technical report, Sun Microsystems, Inc., May 1996.

[GMPS97] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems Proceedings*, pages 103–112. USENIX Association, December 1997.

[Gra96] Robert S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Usenix Tcl/Tk Workshop*, pages 9–23, 1996.

[HK97] Melissa Hirschl and David Kotz. AGDB: A Debugger for Agent Tcl. Technical Report PCS-TR97-306, Dartmouth College, Computer Science, Hanover, NH, February 1997.

[HKM+98] Michael Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A programming language for active networks. In *Proceedings of the 1998 International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, September 1998.

[Hoh98] Fritz Hohl. Time limited blackbox security: Protecting mobile agents from malicious hosts. In Giovanni Vigna, editor, *Mobile Agents and Security*. Springer-Verlag, 1998.

[JLHB88] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[JTK97] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers: Sepcial issue on Mobile Computing*, 46(3), March 1997.

[JvRSS95] Dag Johansen, Robbert van Renesse, Fred B. Schneider, and Scott D. Stoller. Operating system support for mobile agents. In *Proceedings of the Fifth IEEE Workshop on Hot Topics in Operating Systems*, May 1995.

[KLO97] Gunter Karjoth, Danny B. Lange, and Mitsuru Oshima. A security model for Aglets. *IEEE Internet Computing*, 1(4):68–77, July-August 1997.

[Kna95] Frederick Colville Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie-Mellon University, Dec 1995.

[KZ97] Joseph Kiniry and Daniel Zimmerman. A hands-on look at Java mobile agents. *IEEE Internet Computing*, 1(4):21–30, July-August 1997.

[LNBL] Ari Luotonen, Henrik Frystyk Nielsen, and Tim Berners-Lee. CERN httpd. URL: `http://www.w3.org/Daemon/`.

[LO95] Jacob Y. Levy and John K. Ousterhout. Safe Tcl toolkit for electronic meeting places. In *Proceedings of the First USENIX Workshop on Electronic Commerce*, pages 133–135, July 1995.

[LS88] B. Liskov and L. Sharira. Promises: Linguistic support for efficient asynchronous procedure calls. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 260–267, June 1988.

[MvRSS96] Y. Minsky, R. van Renesse, F. B. Schneider, and S. D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Proceedings of the 1996 SIGOPS European Workshop*, September 1996.

[Nel79] Philip A. Nelson. A comparison of PASCAL intermediate languages. In *SIGPLAN Notices*, pages 208–213, Aug 1979.

[NR98] Cameron Newham and Bill Rosenblatt. *Learning the Bash shell.* O'Reilly Associates, January 1998.

[Out94] John Outsterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[Pau96] L. C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1996.

[PR85] J. Postel and J. Reynolds. File transfer protocol (FTP). Internet Engineering Task Force RFC 959, October 1985.

[RBM96] Robbert Van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus, a flexible group communication system. *Communications of the ACM*, April 1996.

[Sat96] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Fifteenth ACM Symposium on Principles of Distributed Computing*, May 1996.

[Sta] Apple Computer Staff. *Mac OS Runtime Architectures.* Addison Wesley.

[Sta89] James W. Stamos. The Internet worm: Crisis and aftermath. *Communications of the ACM*, 32(6):678–687, June 1989.

[Sun96] Sun Microsystems, Inc. JAR - Java archive, 1996. URL: `http://java.sun.com/products/jdk/1.1/docs/guide/jar/`.

[Sun98] Sun Microsystems, Inc. Security, 1998. URL: `http://www.javasoft.com/products/jdk/1.2/docs/guide/security/`.

[TLP+93] Bent Thomsen, Lone Leth, Sanjiva Prasad, Tsung-Min Kuo, Andre Kramer, Fritz C. Knabe, and Alessandro Giacalone. Facile antigua release programming guide. Technical Report ECRC-93-20, European

Computer Industry Research Centre, Munich, Germany, December 1993.

[TOHO97] J. Touch, K. Obraczka, A. Hughes, and A. Oswal. Proactive web caches, June 1997. Presented at the NLANR Web Cache Workshop '97. URL: `http://www.isi.edu/lsam/publications/web-cache97/`.

[TSS$^+$96] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *Computer Communication Review*, 26(2), April 1996.

[VB94] Geoffrey M. Voelker and Brian N. Bershad. Mobisaic: An information system for a mobile wireless computing environment. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, December 1994.

[VST97] Jan Vitek, Manuel Serrano, and Dimitri Thanos. Security and communication in mobile object systems. In *Mobile Object Systems: Towards the Programmable Internet*, pages 177–199. Springer, 1997.

[Whi96] James E. White. Telescript technology: Mobile agents. In Jeffrey Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, Menlo Park, California, USA, 1996.

[WT96] David J. Wetherall and David L. Tennenhouse. The active IP option. In *Proceedings of the Seventh ACM SIGOPS European Workshop*, Sept 1996.

[Yee94] Bennet Yee. *Using Secure Coprocessors*. PhD thesis, Carnegie Mellon University, 1994. CMU technical report number CMU-CS-94-149.