

An architecture for secure, client-driven
deployment of application-specific proxies

by

David John Kennedy

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2000

©David John Kennedy 2000

I hereby declare that I am the sole author of this thesis.

I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize the University of Waterloo to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Abstract

Wireless devices are becoming a popular tool for accessing the Internet. Unfortunately, these devices are often unable to access services that are tailored for more powerful desktop client devices with faster network connections. In client-server applications, the client alone cannot address the limitations of these devices or their networks, and server modifications will only be beneficial until another device is released with different characteristics.

One practical solution is to insert a proxy between the client and server. Traditional proxies face a trade-off between specialization and wide deployment, and a highly specialized proxy is required to tailor specific services for a particular client device. Because of the relatively small user base of these devices, such a specialized proxy is unlikely to be deployed widely.

This thesis explores an architecture that allows application-specific proxies to be deployed automatically and securely at the request of a client. A general proxy provider is deployed, allowing the automatic deployment of new proxies tailored for a specific application to meet the needs of a particular client device. This architecture contains security features to protect the client, proxy provider, and proxy, while still giving proxies the flexibility to assist clients.

This architecture, its implementation, and case studies of its application are all presented in detail.

Acknowledgements

I would like to thank everyone who has helped and encouraged me during the completion of this thesis.

First, I would like to thank my advisor, Jay Black, for providing me with a great deal of freedom to explore the field, for being patient while I waded through an ocean of research in search of a topic, and for his encouragement and good advice when I finally found a direction in which to travel.

I should also thank the members of the Shoshin research group at the University of Waterloo. The distilled contents of some of our heated discussions are found in this thesis. Many others go undocumented but have contributed to my overall understanding.

This research has been funded in part by an National Science and Engineering Research Council Scholarship, allowing me to live, eat, and have a warm place to sleep throughout my studies.

Finally, I would like to thank my family and friends for bearing with me during this process, especially when it began to take longer than anticipated.

Trademarks

Palm is a trademark of Palm, Inc.

Palm OS is a registered trademark of Palm, Inc.

Cryptix is a trademark of The Cryptix Foundation Ltd.

The SSLeay software is copyright ©1997 Eric Young.

WAP is a trademark of the Wireless Application Protocol Forum Ltd.

Contents

1	Introduction	1
1.1	The Problem	2
1.1.1	Limitations of Wireless Devices	4
1.1.2	Approaches	6
1.1.3	Security	7
1.2	Contribution	7
1.3	Chapter Organization	8
2	Related Work	9
2.1	Transparent Proxies	9
2.2	Proxy-Aware Clients	11
2.3	Proxy-Aware Servers and Services	12
2.4	Mobile Code Architectures	13
2.5	Opportunities	15
3	Architecture	16
3.1	Overview	16

3.2	Cryptographic Terms	17
3.3	Relationships	20
3.4	Authentication	22
3.5	The Code Provider	25
3.6	Application Program Interfaces	28
	3.6.1 The Proxy API	29
	3.6.2 The Client API	32
	3.6.3 Examples	35
3.7	Secure Sockets	39
3.8	Secure Socket Example	44
3.9	Comparison to Related Work	46
3.10	Summary	47
4	Implementation and Case Studies	49
4.1	Implementation	49
4.2	Verifying the Integrity of the Proxy Provider	51
	4.2.1 Secure Sockets	52
	4.2.2 Proxy Authentication	52
	4.2.3 Proxy Execution	54
4.3	Case Studies	56
	4.3.1 World-Wide Web Browser	56
	4.3.2 Streaming Media Proxy	57
	4.3.3 Proxied Server	62
	4.3.4 Summary	67

5	Summary and Future Work	68
5.1	Summary	68
5.2	Future Work	69
5.2.1	Persistent Storage	69
5.2.2	Proxy Mobility	70
5.2.3	Resource Limitation	71
5.2.4	Integration with Other Proxy Architectures	71
5.2.5	Verifying the Integrity of the Proxy Provider	72
5.3	Conclusion	73
	Bibliography	74

List of Tables

1.1	Comparing typical consumer desktop computers and typical consumer wireless devices	5
3.1	Client error codes	34
4.1	Greyscale MPEG results	60

List of Figures

3.1	Architectural Overview	17
3.2	Public-Key Cryptography	18
3.3	Secret-Key Cryptography	18
3.4	Digital Signatures	19
3.5	Certificates	20
3.6	Overview of Code Downloading	26
3.7	Structure of a Proxy Descriptor	26
3.8	The Abstract Class <code>shoshin.proxy.Proxy</code>	29
3.9	The <code>shoshin.proxy.ProxySocket</code> Interface	30
3.10	The <code>shoshin.proxy.ProxyServerSocket</code> Interface	30
3.11	The <code>shoshin.proxy.SecureProxySocket</code> Interface	30
3.12	The Client API	32
3.13	The Preferences API	33
3.14	Example Execution Timeline	36
3.15	The <code>ProxySSLClient</code> Structure	40
3.16	The Client SSL Interface	41
3.17	A Secure Proxy Socket Accessible to the Client and Proxy	41

3.18	An Ordinary Proxy Socket Accessible to the Client and Proxy . . .	42
3.19	A Secure Proxy Socket Accessible to Only the Client	43
3.20	Possible Socket Configurations	43
4.1	MPEG Streaming Media Proxy	59
4.2	Existing Comma Execution Environment Monitor	64
4.3	Proxied Comma Execution Environment Monitor	65

Chapter 1

Introduction

A large number of services are available in the Internet, including the World-Wide Web, electronic mail, and multimedia services. Often, these services are being tailored for increasingly powerful client devices with faster network connections. With ubiquitous wireless networking just around the corner, there will be an increase in the popularity of smaller, less powerful client devices accessing the Internet through lower-bandwidth wireless networks. While the constraints of such devices are rapidly being relaxed, they will always fall behind the desktop computer in terms of processor speed, memory, persistent storage, input device characteristics, and display size, resolution and colour.

In order to provide such devices with access to the many useful services on the Internet, proxies can be used to decrease the network and processor requirements on the client device, moving them to a proxy server somewhere within the network. For example, such proxies have been used to “distill” images, reducing their resolution and depth to suit small, hand-held devices. Traditional proxy environments have

faced a difficult trade-off between wide deployment and specialization. Proxies that are widely deployed, such as HTTP proxies, are general purpose and difficult to apply to different applications or to customize for a particular client device. More specialized proxies are not widely deployed; clients are forced to access proxies far away in the Internet, increasing latency and network usage.

This motivates the need for a generalized proxy provider environment that can be widely deployed in the Internet near client devices that might require its services. Such a proxy provider must be able to load application-specific proxy code from another server in the Internet. The execution of this proxy code must be secure, preventing the proxy from tampering with other proxies or the proxy provider environment. The proxy must be allowed to access secure persistent storage to maintain configuration and state information. Finally, the proxy must be given the flexibility to perform complex tasks on behalf of the client, and must not compromise the security of the client or the services that it uses.

1.1 The Problem

The task of providing sophisticated services to users of devices on wireless networks is more complicated than simply developing and deploying a client and a server. The client and the protocols used between the client and server must be designed with knowledge of the inherent limitations of the target environment. In this context, protocols include transport-layer protocols, transfer and security protocols, and multimedia encodings. Unfortunately, there are many useful services that are already deployed widely using standardized protocols designed without these limi-

tations in mind. A client cannot address this problem itself, and modifying widely deployed servers to support new protocols is rarely an option. This problem can be solved by inserting a proxy on a more powerful computing platform in the fixed network between the client and server, to handle some of the complex processing or protocol support on behalf of the client. This eases support of widely deployed protocols, but also benefits developers who would like to introduce new services to wireless devices.

Security is a concern in any networked application. Security is of greater concern to users of devices on wireless networks who might be roaming onto foreign networks with which they are not familiar. When an application is divided between a client and a proxy, a new interface is exposed between the two layers, along with any sensitive information that may pass through this interface. Here, sensitive information may include passwords or credit card numbers, but also any information that, if a third party were able to tamper with it, would affect the behaviour of the application in some undesirable manner. Hence, security must be considered carefully when dividing an application into multiple parts located at different points in the network.

The remainder of this section details the limitations of wireless devices, describes why the ability to deploy application-specific proxies is useful in addressing these limitations, and emphasizes the importance of security to mobile users.

1.1.1 Limitations of Wireless Devices

Wireless devices are generally designed with two related goals: portability, and low power consumption. The need for portability generally results in small, battery-powered units with limited input and output capability. The use of batteries necessitates low power consumption, resulting in slower processing speeds, less memory, and less sophisticated displays compared to computers on fixed networks. Over time, better battery technology and improved low-power devices such as memory and displays will relax these restrictions. However, the cost of miniaturization and low-power design will always leave wireless devices noticeably less powerful than traditional computers on fixed networks. In addition, due to their compact size, the types of input methods used on wireless devices are often quite limited. See Table 1.1 for a comparison between current typical consumer desktop computers and typical consumer wireless devices.

These limitations have a noticeable impact on software. Network speeds are slower, processing takes longer, and applications must be less memory hungry than in traditional computing environments. Limited display technologies mean that graphics must be converted to have smaller dimensions or fewer colours. Processing of complex structured documents leads to a noticeable increase in latency, as does transmitting or receiving large amounts of data of which only a portion is important to the user. Unless an application developer takes these limitations into account when designing an application and the protocols that it uses, the result will be slow or unusable, and use excessive amounts of the device's limited memory and battery power.

Device	Intel PC	Palm TM III	HP Jornada 430se
CPU Speed	450 MHz	16 MHz	133 MHz
Dynamic Memory	64 MB	96 KB	8 MB
Stable Storage	10 GB	2-8 MB	8 MB
Wired Network	100 Mbps	57.6 Kbps	57.6 Kbps - 10 Mbps
Wireless Network	11 Mbps	1.2-14.4 Kbps	1.2 Kbps - 11 Mbps
Display	17" monitor, 1280×1024 pixels and 2 ³² colours	3.25" LCD, 160×160 pixels and 4 greyscales	4" LCD, 240×320 pixels and 65536 colours
Input	104 key keyboard, two-button mouse	6 buttons, stylus for hand- printed input	4 buttons, stylus for hand- printed input

Table 1.1: Comparing typical consumer desktop computers and typical consumer wireless devices

1.1.2 Approaches

The easiest approach to addressing these limitations in client-server applications is to modify the client and the server. However, modifying widely-deployed servers is a time-consuming and often impossible task. Even for servers that are not widely deployed, such modifications would be effective only until the introduction of a new wireless client device that has different characteristics. In the past, multimedia servers have existed that supported devices on high- and low-bandwidth network connections differently. However, many such servers do not yet take into account the limitations of wireless devices. The benefit to a small number of users of client devices rarely justifies the cost of supporting a wide variety of client devices at the server.

Modifying the server is a temporary solution at best, is costly, and poses difficult problems of deployment in most cases. Furthermore, it is clear that modification of only the client cannot account for problems posed by protocols used by the server, and so the remaining option is to insert a new entity into the client-server model of computation. The client-proxy-server model of communication has been used quite successfully in the past to support a variety of applications. However, to keep up with the changing characteristics of wireless devices and the desire to support bigger applications on smaller devices, the problem shifts from whether or how proxies are to be used, to where and how proxies are to be deployed.

1.1.3 Security

The security of many wireless networking technologies is limited or non-existent. While some wireless networks do a very good job of either hiding or encrypting data, this property cannot be relied upon in a general-purpose architecture; it may be possible for a third party to intercept private information including passwords and credit card numbers transmitted over any network, especially over wireless networks. This possibility can be reduced dramatically by encrypting this sensitive data, or more generally by encrypting all data sent over the wireless network.

Inserting a proxy between a client and a server exacerbates this situation; a proxy is a third party that has explicitly been given access to any data sent between the client and the server. An untrusted party in this position could tamper with this data, affecting the behaviour of the client or server. Thus, it is very important to know where the proxy came from and what actions it will perform on the data passing through it. Since the proxy must perform some processing on this data, the proxy will likely need access to much of it. However, there is often information that is private to the client and server that the proxy should not be able to access. Thus, a mechanism is required to allow such information to pass through the proxy securely.

1.2 Contribution

This thesis presents an architecture in which proxies can be deployed dynamically and securely to perform a useful task, specifically, to benefit users of wireless de-

vices in client-server applications. As will be seen, several architectures allow for the deployment and execution of code throughout a network. However, these architectures face various security challenges. This thesis restricts the domain of applications that can be supported to client-proxy-server applications, resulting in security considerations that are well-defined and tractable. The resulting architecture can support traditional proxies as well as experimental and special-purpose proxies for wireless devices.

1.3 Chapter Organization

Chapter 2 discusses the larger class of work on mobile code, proxies, and architectures to support various types of applications in restricted environments. Chapter 3 describes the architecture in detail. Chapter 4 describes the implementation of this architecture, and examines a number of applications and how they can be designed to take advantage of this architecture. The results are summarized in Chapter 5 along with suggested directions for future work.

Chapter 2

Related Work

Because this thesis is focused on the secure deployment of proxies, this investigation will examine work that supports the deployment of executable code throughout a network to support some computation in the larger context of a client-server application. These works can be divided into four broad categories: those that are transparent to the client and server, those where only the client is aware of the architecture, those where the server may also be aware of the architecture, and those where code mobility is an intimate part of the interaction between client and server.

2.1 Transparent Proxies

Transparent proxies are useful in applications where it is not possible or desirable to modify either the client or the server. A transparent proxy can be inserted between the client and the server when they begin communicating by intercepting

and processing individual packets at a gateway or base station on the path between the client and server. Such a proxy must be careful to preserve the semantics of the data flowing through it so that the client and server can continue to function as if it were not present; for example, although the client may see lower resolution images than were sent by the server, it can process them in the same manner.

The Comma Service Proxy [12] implements exactly this mechanism. When deployed on a base station in a wireless network or at the foreign agent in a Mobile IP [15] environment, a service proxy can intercept data sent between a client and server. In this architecture, a service proxy can manipulate individual packets of data, or it can interact with a TCP stream, modifying the data on the stream without breaking the end-to-end TCP semantics; the flow of data and acknowledgements continues as if the service proxy were not present, preventing an application from believing that data was delivered when it was not. Service proxies are applied to TCP streams when they are initiated, based on patterns that can be configured at run-time. These patterns allow connections to different services to be handled by different service proxies. For example, this architecture can be used to throttle a bulk file-transfer stream to improve the responsiveness of an interactive application over the same low-bandwidth network connection. While this architecture allows service proxies to be applied to streams automatically, and allows users to modify the patterns dynamically, the service proxies must be deployed manually; there is no support for the automatic deployment of new service proxies to support new services or the changing needs of users.

2.2 Proxy-Aware Clients

Environments in which the client is aware of the proxy are generally more flexible than transparent proxy environments. As a simple extension to transparent proxies, a client and proxy that are aware of each other can cooperate to support each other. In particular, where a transparent proxy must not change the semantics of the information flowing between the server and the client, a proxy-aware client can accommodate changes in semantics where they are beneficial. For example, in such an environment a proxy may choose to compress data that was not previously compressed; if the client were unaware of the proxy's actions, it would not know that it must begin decompressing the data that it receives.

Traditional proxies and special-purpose proxies also fit into this category. While a transparent proxy must be placed at a gateway where it can intercept all traffic between the client and a server, a proxy-aware client can be configured manually or automatically to access a proxy at a fixed location within the network. Widely-deployed proxies such as caching proxies for the World-Wide Web are located close to clients; the time required to access popular documents is reduced by caching these documents, avoiding the high latencies involved when accessing a distant or popular server in the Internet. Special-purpose proxies are less widely deployed, often forcing users to access them at a distant location in the Internet. The latency in accessing such proxies is compounded by the proxy's need to access the data from a server that is located elsewhere within the Internet. The route that data must take from client to proxy to server and back will always be longer than the route between the client and the server. However, special-purpose proxies can still benefit

users by adding value to the information obtained from the server. An example of such a value-added service is the TranSend proxy [7], developed at the University of California at Berkeley. This proxy makes the World-Wide Web accessible to users of compact devices by pre-parsing structured documents and reducing the size and colour of images to make them more appropriate for the small, limited displays on these devices. While the server is unaware that such a proxy is being used, the client must know where to find this proxy in the Internet, and how to interpret its results.

These architectures face deployment challenges. Because of their wide deployment, traditional proxies are difficult to update to support new protocols or clients. However, their benefit is derived from their proximity to the client. On the other hand, special-purpose proxies would provide greater benefit to users by being moved closer to the client, but their limited user base restricts the number of administrators willing to expend the effort required to deploy them. These architectures would clearly benefit from the ability to deploy new proxies where and when they are needed, whether to support a number of small, diverse groups of users or a large group of users with common, changing needs.

2.3 Proxy-Aware Servers and Services

Proxy-aware servers and services can aid in the deployment and functionality of new proxies. For example, in the World-Wide Web, features have been added to the transfer protocols to give hints about the type of caching that can be performed on a document [6]. Using these hints, a caching proxy is able to make

more informed decisions about whether or how to cache a particular document. A long-lived, unchanging document could be cached, while little value is gained by caching a short-lived or very dynamic document. Caching documents containing private information poses a security threat to a user.

More recently, the World-Wide Web consortium has created the Web Accessibility Initiative [3, 10] to address the needs of users of compact and wireless devices, as well as users for whom graphics, audio, or text are not accessible. Two documents provide guidelines for accessible content, and the design of accessible client applications. These guidelines are intended to help ensure that a client can produce a complete and comprehensible view of a document even though the client does not support some features used in it, or makes use of a user interface for which the document was not originally intended. This effort can greatly benefit special-purpose proxies deployed for users of wireless client devices; taking hints from the content-provider, a proxy can reduce the size and complexity of a document by eliminating graphics, tables, and other elements that are too large to transmit over a low-bandwidth connection or too complex to process on a constrained client device.

2.4 Mobile Code Architectures

Mobile code architectures take the deployment of code in client-server applications to the extreme. Mobile agent architectures [4, 9, 13, 14] allow clients to push executable code into the network, to access and process information on the server where it is located, possibly interacting with other agents accessing similar information.

Agents are useful where a client requires access to a large amount of information, but the results of its computation are much smaller. Agents can reduce the network bandwidth requirements in database and information retrieval applications at the expense of computational resources at the server.

Similarly, active networks [1, 17] allow clients or servers to push executable code into routers in the network to implement new protocols and services that require specific information about the network or its topology. Code in active networks may access routing tables and other resources at each active node in the network. The goal of active networks is to ease the deployment of new protocols that require processing throughout the network, rather than just at the endpoints. Such protocols could dynamically decide to perform retransmission over unreliable links or compression over low-bandwidth links, or could perform multicasting in a network that lacks multicast support.

As with any architecture where unfamiliar executable code is deployed into arbitrary nodes in a network, security is a big concern in mobile code architectures. A great deal of care must be taken to ensure that code does not access restricted resources or adversely affect the behaviour of the network. For agent architectures, the required security measures will depend on the specific application for which they are deployed. For active networks, making them useful requires a trade-off between security, flexibility, and the speed at which active packets can be processed. Controlling access to services on an active node requires additional processing for each active packet. Such access controls must be simple and scalable. However, when deploying proxies, the security problems are well defined and easily solved.

The authenticity of the proxy code must be verified, but proxies are not necessarily accessing resources at the proxy provider.

2.5 Opportunities

The work done with proxies has demonstrated that they are a powerful tool for delivering services and information to wireless devices. Mobile code architectures have shown that new applications are made possible when executable code is given the ability to move among nodes within a network. Where these fields intersect, there is an opportunity to bring greater benefit to users of wireless devices. By applying mobile code techniques to the deployment of proxies with well-defined security requirement and procedures, users of wireless devices can quickly be given access to new and existing services.

Chapter 3

Architecture

3.1 Overview

The client-proxy-server architecture shown in Figure 3.1(b) extends the traditional client-server architecture shown in Figure 3.1(a) by inserting a new entity between the client and server: a proxy. This proxy provides some benefit to the client, the server, both, or the network as a whole. This benefit often takes the form of reduced network utilization, lower end-to-end latency, or increased security. Three additional entities must be inserted into the system to allow secure client-driven proxy deployment: the client library and the proxy provider cooperate to perform the various forms of authentication required in this system, and the code provider stores the executable code for the proxy. This code is downloaded across the network by the proxy provider at the request of the client, to form the running proxy. See Figure 3.1(c) for an overview of this system. The remainder of this chapter will expand on the components of this system and their respective roles.

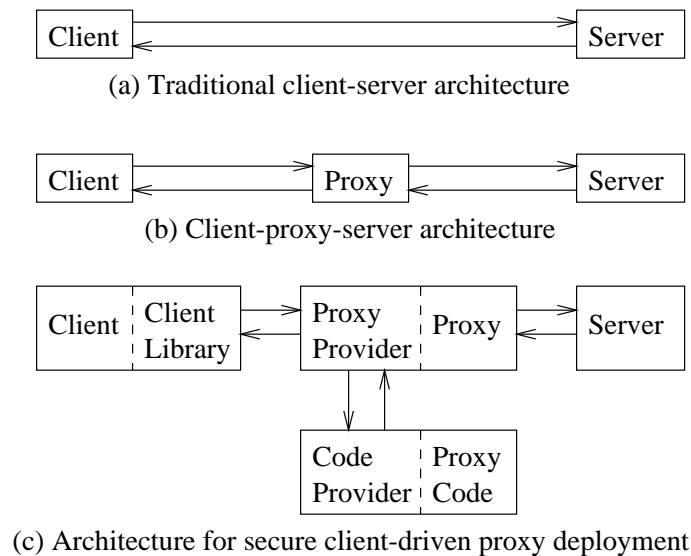


Figure 3.1: Architectural Overview

3.2 Cryptographic Terms

The concepts of public- and secret-key cryptography, digital signatures and certificates are used in the remainder of this chapter. Readers familiar with these concepts should not hesitate to skip to Section 3.3.

Public-key cryptography refers to cryptosystems in which a pair of keys is used: a public key and a private key. Messages can be encrypted and decrypted with either of these keys, but messages encrypted with the public key can only be decrypted correctly with the corresponding private key and vice versa, as shown in Figure 3.2. In this way, the public key can be advertised to anyone who would like to send messages to its owner. Since the private key cannot be obtained easily from the public key, any message encrypted with this key can only be decrypted by the owner of the corresponding private key. The most popular public-key cryptosystem in use

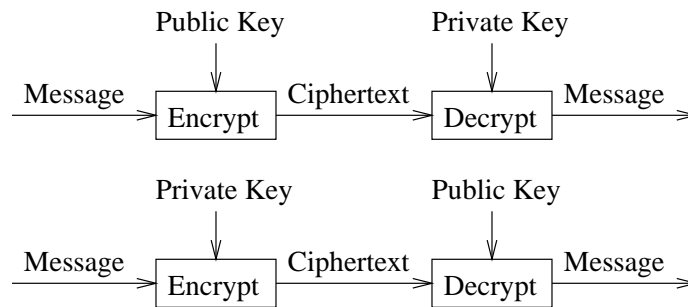


Figure 3.2: Public-Key Cryptography

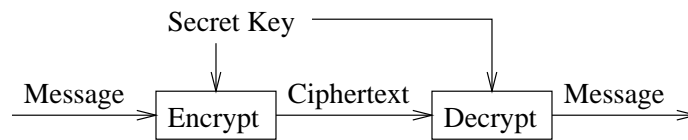


Figure 3.3: Secret-Key Cryptography

today is RSA [16], which forms the basis of many popular security protocols. All current public-key cryptosystems are very computationally expensive, and hence quite slow.

On the other hand, secret-key cryptosystems are computationally cheap to execute. Unlike public-key cryptosystems, secret-key cryptosystems make use of a single, shared key. Messages encrypted with this secret key can be decrypted correctly by anyone possessing this same secret key, as in Figure 3.3. Transmitting the secret key between communicating parties is a task accomplished by public-key cryptographic methods. Since this key exchange only needs to be done once, slower public-key-based key-exchange algorithms perform this job well and have become common in modern security protocols.

Digital signatures are based on public-key cryptography and cryptographic

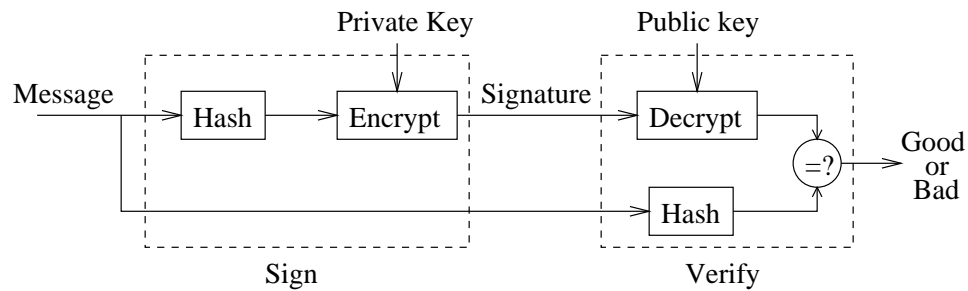


Figure 3.4: Digital Signatures

hashes. A cryptographic hash is a small value obtained from a larger message by some convoluted algorithm. It is extremely difficult to reverse a cryptographic hash, or to generate two messages with the same cryptographic hash value. A digital signature of a message is created by encrypting the hash of the message using the private key of the sender of that message, as shown in Figure 3.4. Since it is difficult to produce another message with the same hash value, and only the sender has access to the private key, digital signatures can be used to verify the authenticity of a message, that is, that the message really came from the sender. Any party can access the well-advertised public key of the sender to decrypt the signature and verify that the result matches the true hash of the message. Note that digital signatures cannot determine the validity of the contents of a message, only that their origin is authentic. Validity of the contents must be gauged by how well the originator is trusted.

Digital signatures can be used alone, or as the basis for certificates in the Public Key Infrastructure [11]. As shown in Figure 3.5, a certificate contains a public key, the identity of its owner, called the subject of the certificate, and the identity of the

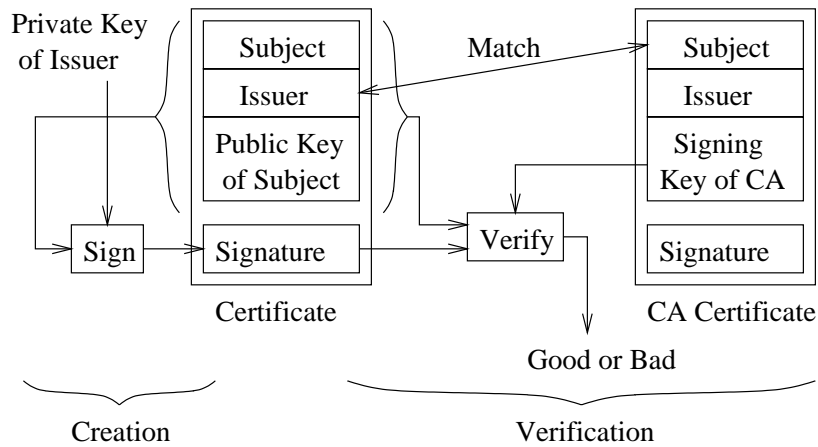


Figure 3.5: Certificates

issuer of the certificate. This information is signed by the issuer of the certificate. The issuer is vouching for the fact that the subject of the certificate really owns the public key contained in the certificate and the corresponding private key which remains concealed. If this issuer is a certificate authority who is trusted to sign only legitimate certificates, any party who trusts this certificate authority can be confident of the ownership of the public key contained in the certificate. The Public Key Infrastructure (PKI) is based on trees of certificates, each rooted at a certificate authority. Trust of a certificate authority implies trust of any certificate contained in its tree.

3.3 Relationships

A number of relationships must exist outside of the technological realm. These relationships allow for the exchange of authentication information and help to enforce

rules that cannot be enforced through technology alone.

The first and most important such relationship is an agreement between the user of the client device and the operator of the proxy provider. This agreement serves two purposes. First, it gives the user of the client device the right to access the proxy provider using an agreed-upon user ID and password. Second, it provides the user with some guarantee that the proxy provider will perform its duties correctly. As will be seen in Section 4.2, it is difficult and time-consuming to verify that the proxy provider is behaving according to its specification. While such a verification should be performed periodically by the cautious user, a legal agreement outside the realm of technology is the most effective method of ensuring proper operation.

The second relationship that must exist is a standard relationship from the Public Key Infrastructure. The proxy provider must be issued with a cryptographic key pair and certificate from a trusted certificate authority. The client library must have access to the signing certificate of any certificate authority that will be used in this manner. As with secure World-Wide Web sites and electronic mail systems, these signing certificates can be used to authenticate the party at the other end of a secure connection. Section 3.4 explains why this is important.

The final relationship that must exist is between the developer of the proxy and the developer of the client application. In many cases, these will be the same person. If not, such a relationship must already exist in order for the client and proxy to share a common interface and communication protocol. In addition, the developer of the proxy must have a cryptographic key pair used for signing the proxy code. A certificate authenticating this key pair must be issued and made

available to the client application. This certificate will be used to authenticate the origin of the proxy code when it is downloaded by the proxy provider. The use of this certificate and signature are described in detail in Section 3.4.

3.4 Authentication

Authentication is employed in three different ways to ensure that the client and proxy provider are protected from abuse. First, the proxy provider must be correctly identified to the client library. Second, the user of the client must identify him- or herself to the proxy provider. Finally, and central to secure proxy deployment, the proxy code that is downloaded from the network must be authenticated to the client.

Authentication of the proxy provider to the client library is accomplished using certificates and the Secure Socket Layer (SSL) protocol [8]. The SSL protocol establishes a channel between a client and a server that provides security from eavesdropping and tampering. During the negotiation of this channel, the client can require that the server provide a certificate signed by a trusted certificate authority that includes the name of the host on which the server is running. This certificate and the corresponding private key known only to the proxy provider ensure that no other party can pretend to be the proxy provider expected by a client library. This is important because the client library can only expect correct operation from the expected proxy provider. Any party that could masquerade as that proxy provider might undermine the security features of the architecture. In addition to authenticating the proxy provider to the client library, the SSL protocol

simplifies the next authentication steps.

Once a secure channel has been established between the client library and proxy provider, the user must be authenticated to the proxy provider. This authentication determines whether the proxy provider should provide the client with access to its services. Since the same user may use multiple client devices and thus multiple client libraries to access the same proxy provider, this authentication must consist of a small piece of information that the user can easily carry between devices. Because a secure channel exists, a user ID and password provide sufficient security and are very easy to administer. In addition, a certificate-based scheme would require additional overhead in terms of public-key cryptographic operations on a client device that is assumed to have restricted computational resources. Two such operations are performed during SSL negotiation, but these are a justifiable expense given the benefits that it provides. Additional security is obtained during this authentication step by restricting access to the proxy provider to specific networks or interfaces. A proxy provider configured to serve a specific wireless network can easily refuse connections from clients outside that network. This would reduce the possibility of malicious access without affecting legitimate clients.

Finally, once the user has been given access to the proxy provider, code will be downloaded across the network at the request of the client. Because the code providers storing this data may be distant, and are not likely under control of the user of the client or the administrator of the proxy provider, tampering with this code is a strong possibility. The code could simply be replaced at the code provider, another server could masquerade as the code provider, or the data could be

intercepted and modified in transit through the network. To detect such tampering, proxy code is signed by its developer, and this signature must be delivered with the proxy code by the code provider. The developer must have a signing key pair and a certificate for the public key in this pair. This certificate should be signed by a trusted certificate authority so that the developer of the client application can be sure of its authenticity when it is obtained from the proxy developer. This certificate is compiled into the client application and delivered to the proxy provider along with the request for the proxy code. The signature on the code is verified with the public key contained therein. Notice that by contractual agreement, this verification step can be performed by the proxy provider, saving execution time on the client device. Thanks to the secure connection between the client library and proxy provider, there is no possibility of an outside party tampering with the certificate in transit between the client and proxy provider. In addition to preventing a malicious party from introducing proxy code that is not expected by the client, code signing allows the proxy code to be updated by the proxy developer to add new features or correct bugs without the need to notify all users of the client. The proxy code can be changed at any time provided that it is properly signed by the proxy developer.

These authentication steps address the threats that might be directed towards this system. The first threat is that a malicious proxy provider may masquerade as another proxy provider, fooling clients into using its services while it performs some unwanted actions. This threat is addressed by authenticating the proxy provider. The second threat is that a client may make unauthorized use of a proxy provider.

This is prevented through authentication of the client. A third threat is that some third party may eavesdrop on or modify the data sent on the private interface between the client and proxy. This threat is addressed through the use of a secure socket for communication between the client library and the proxy provider. Another threat is that someone might present malicious code in place of a useful proxy. This is avoided by authenticating the proxy code that is downloaded from the code provider. A final threat is that even a carefully written proxy may accidentally or intentionally attempt to perform some malicious action on the proxy provider. Section 3.6.1 discusses how the run-time security environment of the proxy provider addresses this threat.

3.5 The Code Provider

The executable code for a proxy consists of Java classes that follow certain conventions described in Section 3.6.1. These classes are stored in a Java archive (JAR) file. Java was chosen because of its platform-independence, its solid networking features, and its support for the necessary security features to maintain control over an executing proxy. Other languages that support embeddable interpreters, including Perl, Safe Tcl and Lisp, were considered, but Java had advantages in terms of a larger user and development community, and the fact that many client-server applications are already written using the Java programming language.

Proxy code may be distributed using File Transfer Protocol (FTP) or World-Wide Web Hypertext Transfer Protocol (HTTP) servers which are very widely deployed. The use of these standard and very popular protocols means that no new

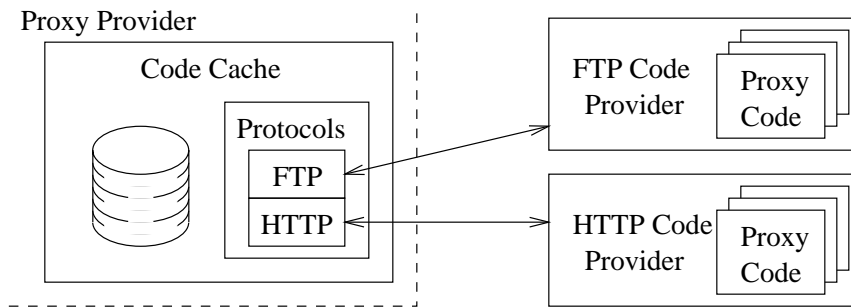


Figure 3.6: Overview of Code Downloading

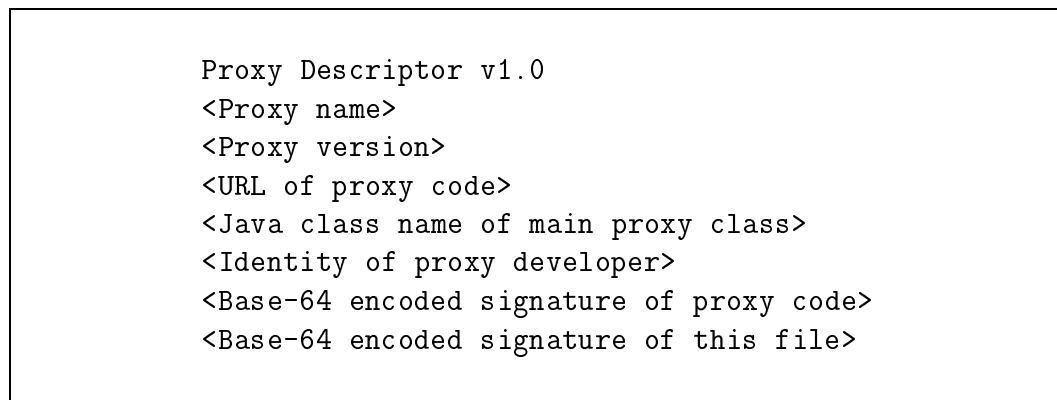


Figure 3.7: Structure of a Proxy Descriptor

file servers need to be deployed to support this architecture. As shown in Figure 3.6, the downloading of proxy code is accomplished by the code cache within the proxy provider. The code cache consists of a cache of downloaded code in persistent storage, and protocol handlers to download code from various types of servers.

Proxy code is stored on code providers in two separate files. The first of these is the proxy descriptor. Figure 3.7 details the structure of this file. When requesting proxy code, the client specifies the Uniform Resource Locator (URL) of this file, the proxy name, a minimum acceptable proxy version, and the certificate of the

proxy developer. The proxy descriptor is downloaded first, and the proxy name, version, and signature are verified. The proxy descriptor file is signed to prevent a malicious party from substituting a different proxy from the same developer or the same proxy with an incorrect version. Once the proxy descriptor has been verified, the JAR file containing the proxy code is downloaded from the URL specified in the proxy descriptor. The signature of this JAR file is verified in the same manner as the signature of the proxy descriptor, and if it is valid, the proxy code is made available for execution. Both the proxy descriptor and the JAR file containing the proxy code are stored in a cache in persistent storage to reduce the time required to start this proxy on subsequent requests. If a proxy is requested that is already found in the cache, the proxy descriptor is reloaded first and, if it is valid and the version number has increased, a new version of the proxy code is loaded. If the version number has not increased, the cached code is verified and used directly. This means that new proxies distributed at the same location, it must be backward-compatible with older versions of that proxy. Version numbers will increase when the proxy is updated to fix bugs or change behaviour. If the interface between the client and proxy is changed the client must be updated and the proxy code will be distributed at a different URL. Because the proxy descriptor is small, this algorithm does not impose a large overhead if the proxy has not been updated. It is also more reliable than systems that rely on the server to provide the time when the file was last updated, a feature that is not available on all servers.

3.6 Application Program Interfaces

Once the proxy code has been loaded, it can be executed. To protect the proxy provider, the proxy code is provided with access to a restricted application program interface (API) that provides useful network functions while restricting access to the local filesystem and other running proxies. The proxy makes use of the proxy API provided by the proxy provider to communicate with the client and with servers. The client makes use of a client API at the client library to initiate communication with the proxy provider and to communicate subsequently with the running proxy.

The API selected for communication between the client and proxy is based on streams, rather than on remote procedure calls. Streams are more light-weight and general than remote procedure calls, and clients are easily given complete access to any streams that the proxy has open. The protocol that the client and proxy use to communicate is defined by their developers, and a simple remote procedure call mechanism could be built over this stream interface for applications that require it. In addition to network sockets, the client and proxy APIs provide access to secure sockets that can be layered over any stream connection. These secure sockets are compatible with common secure servers in the Internet. The API allows the endpoint of a secure connection to be passed securely between the client and the proxy, giving the client the ability to secure information that the proxy should not see while allowing the proxy to process less sensitive information. This will be discussed in Section 3.7.

```
public abstract class Proxy {
    public ProxySocket mClient;
    public ProxyServerSocket bind( int port );
    public ProxySocket connect( String host, int port );
    public SecureProxySocket secureSocket( ProxySocket s,
        String serverName, SSLState oldState );
    public abstract void main();
};
```

Figure 3.8: The Abstract Class `shoshin.proxy.Proxy`

3.6.1 The Proxy API

The code for a proxy consists of one or more Java classes, exactly one of which extends the abstract class `Proxy`. All non-standard Java classes discussed here are contained in the `shoshin.proxy` package. The `Proxy` class provides the interface shown in Figure 3.8. The function `main()` is to be overridden by the proxy and serves as the starting point for the proxy. This function is executed in a separate thread created for each instance of a proxy. The proxy inherits the remaining members of the `Proxy` class. They provide access to communication functions through the `ProxySocket` interface shown in Figure 3.9, the `ProxyServerSocket` interface shown in Figure 3.10, and the `SecureProxySocket` interface shown in Figure 3.11.

The `mClient ProxySocket` is the communication channel that is set up between the client and proxy before the proxy begins execution. New communication channels to other hosts in the network can be created using the `bind()` and `connect()` methods. The `bind()` method creates a listening TCP socket, or server socket, to receive incoming connections from other hosts. New connections are then received

```
public interface ProxySocket {
    public int getRemotePort();
    public String getRemoteHost();
    public int getLocalPort();
    public String getLocalHost();
    public int getChannel();
    public int write( byte[] data, int off, int len )
        throws IOException;
    public int read( byte[] data, int off, int len )
        throws IOException;
    public int clientAvailable() throws IOException;
    public void close();
};
```

Figure 3.9: The `shoshin.proxy.ProxySocket` Interface

```
public interface ProxyServerSocket {
    public int getLocalPort();
    public ProxySocket accept();
};
```

Figure 3.10: The `shoshin.proxy.ProxyServerSocket` Interface

```
public interface SecureProxySocket extends ProxySocket {
    public ProxySocket getUnderlyingSocket();
    public SSLState getState();
};
```

Figure 3.11: The `shoshin.proxy.SecureProxySocket` Interface

using the `ProxyServerSocket.accept()` method. The `connect()` method makes an outgoing TCP connection to a server. Finally, an SSL connection can be layered on top of any other connection using the `secureSocket()` method of the `Proxy` interface. Secure sockets and the `SecureProxySocket` interface are discussed in greater detail in Section 3.7. Because TCP and SSL are used by the vast majority of services in the Internet, this API allows proxies to be integrated into most existing client-server applications.

Associated with any instance of `ProxySocket` is a channel number. Each `ProxySocket` has a unique channel number for a given proxy, and the `ProxySocket` connected directly to the client always has channel number 0. As will be seen in Section 3.6.2, these channel numbers are a requirement for communication between the client and proxy as all client communication functions require that a channel number be specified to identify the appropriate connection.

A proxy has free access to much of the remaining standard Java API. The Java protection mechanisms are used to disallow access to files, system properties, virtual machine features, other threads, graphical user interface elements, and network functions other than through the proxy API. This means that a proxy has access to a very rich set of APIs and classes, including abstract data types, input and output filters, mathematical functions, and text manipulation functions. However, proxies cannot adversely affect the proxy provider or other proxies.

```
PrxErr Proxy_open();
PrxErr Proxy_load( char *proxyURL, char *proxyName,
                  int minversion, Byte *signerCert, int certlen );
PrxErr Proxy_close();
int Proxy_read( int channel, char *data, int length );
int Proxy_write( int channel, char *data, int length );
int Proxy_available( int channel );
ProxySSLClient *Proxy_secureChannel( int channel,
                                     char *serverName, SSLState *state );
void Proxy_destroySecureChannel( ProxySSLClient *client );
char *Proxy_getError( PrxErr err );
```

Figure 3.12: The Client API

3.6.2 The Client API

Client applications for this architecture are written as native “C” programs and are given access to a library of functions to access the proxy provider and proxies. The “C” programming language was chosen for its speed of execution and easy application to a number of different wireless devices. Other languages could be used where they are available, although interpreted languages would require some native support as the client library performs a number of computationally expensive encryption operations.

The client library provides the interface shown in Figure 3.12. The client library supports communication with only one proxy provider and proxy. This simplifies the interface but is flexible enough to handle most applications. The connection between the client library and proxy provider is initialized with a call to `Proxy_open()`. The host and port number of the proxy provider and the user ID and password of

```
void Proxy_setLogin( char *userid, char *password );
void Proxy_setProxyHost( char *host, Short port );
char *Proxy_getUserid();
char *Proxy_getPassword();
char *Proxy_getProxyHost();
Short Proxy_getProxyPort();
```

Figure 3.13: The Preferences API

the user must have been previously configured using the preferences API shown in Figure 3.13, normally from a separate configuration program. `Proxy_open()` and other client library functions return an error code of type `PrxErr`. Possible error codes are listed in Table 3.1. An error code can be converted into a string using the `Proxy_getError()` function. Once a connection to the proxy provider has been created successfully, the client must request that a proxy be loaded using the `Proxy_load()` function. As described in Section 3.5, the client provides to the proxy provider the URL specifying the location of the proxy descriptor, the name of the proxy, the minimum acceptable version of that proxy, and the certificate of the developer of that proxy. Again, there is a possibility that this function call could fail with one of the possible error codes. If this function call succeeds, then the requested proxy has been loaded and started, and the client can now communicate directly with the proxy. The connection to the proxy provider and the proxy is terminated with a call to `Proxy_close()`.

The client communicates with the proxy using the functions `Proxy_read()`, `Proxy_write()`, and `Proxy_available()`. `Proxy_read()` and `Proxy_write()` re-

Error Code	Description
PrxOK	Success
PrxBadLogin	The user ID and/or password were incorrect
PrxNotLoggedIn	Attempted an operation before a successful call to <code>Proxy_open()</code>
PrxNoProxy	Attempted to communicate with proxy before a successful call to <code>Proxy_load()</code>
PrxProxyNotFound	No proxy code was found at the URL specified
PrxBadProxy	The proxy code that was downloaded was not signed correctly
PrxProtocolFailed	Internal protocol failure (should not happen)
PrxNoSuchChannel	A <code>Proxy_read()</code> or <code>Proxy_write()</code> call was made to a non-existent channel
PrxNotConfigured	Host, port, user ID or password not configured
PrxAlreadyConnected	Called <code>Proxy_open()</code> while already connected
PrxNoCertificateDatabase	Certificate authority database needed by SSL not installed
PrxNoNetLibrary	Failed to initialize the network library
PrxConnectFailed	Failed to connect to the proxy provider
PrxAuthFailed	SSL authentication of the proxy provider failed

Table 3.1: Client error codes

spectively read from and write to a specific channel. `Proxy_available()` returns the number of bytes available to be read from a specific channel. Each of these functions takes a channel number as a parameter. Channel number 0 is used to communicate directly with the proxy. All other channels may be used to read from or write to `ProxySockets` created by the proxy. If a client wishes to communicate on one of these `ProxySockets`, it must first obtain the channel number of the `ProxySocket` from the proxy over channel number 0. It is then the responsibility of the client and proxy to ensure that communication on a `ProxySocket` shared in this manner is coordinated properly. Channels beyond channel number 0 are created only by the proxy, including those that are to be used by the client. While it would be possible to implement a mechanism to explicitly pass the end-points of sockets between the client and proxy and to keep track of where each socket is terminated, the method chosen is simpler, more flexible, and imposes less overhead.

An SSL connection can be layered on top of any existing channel using the `Proxy_secureChannel()` function. Secure sockets and the `ProxySSLClient` type are discussed in greater detail in Section 3.7

3.6.3 Examples

Figure 3.14 shows the sequence of interactions between the client, client library, proxy provider, code provider, and proxy when a client initiates a session with the proxy provider, loads a proxy, and closes the session with the proxy provider assuming that all actions complete successfully.

The following pseudo-code example shows the use of the application program

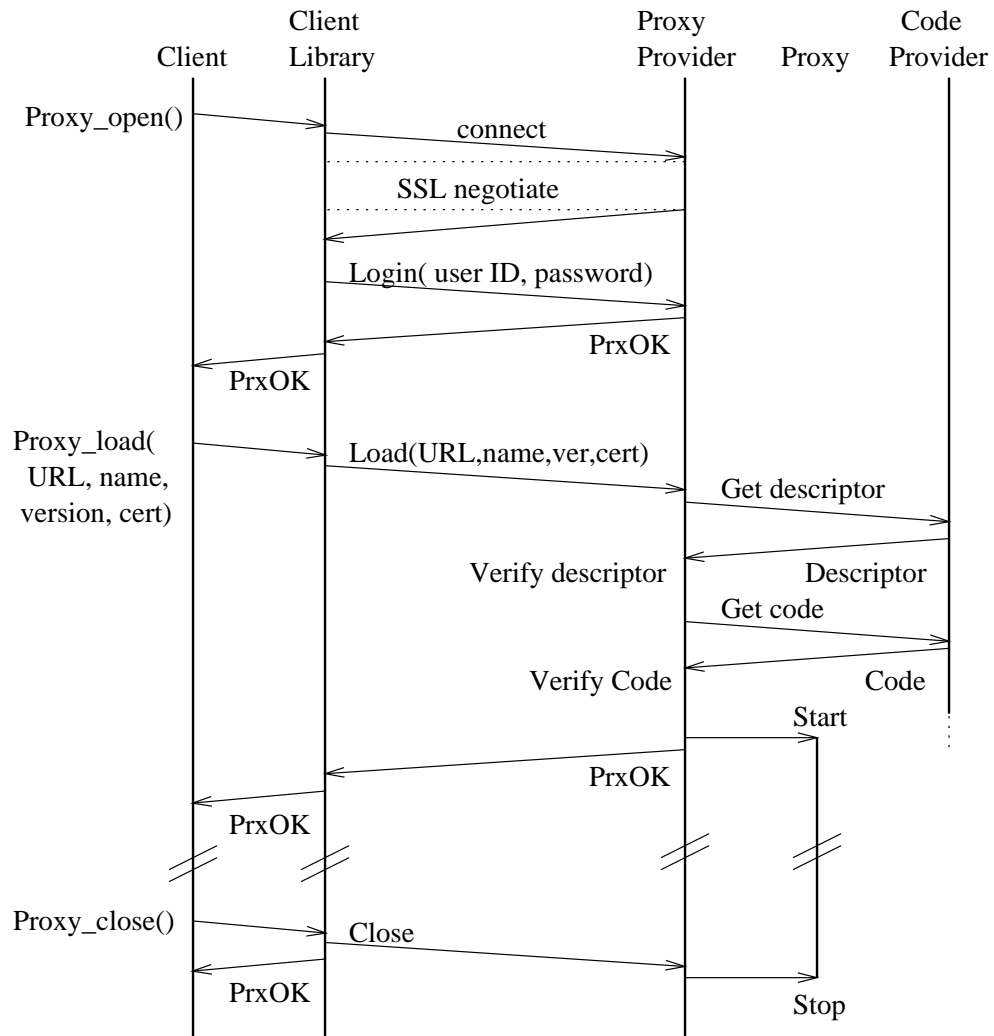


Figure 3.14: Example Execution Timeline

interfaces to download an HTML page through the HTTP protocol using a proxy.

Client**Proxy**

```

/* Load the proxy */
PrxErr err = Proxy_open();
if (err != PrxOK) // display error message and quit
err = Proxy_load( ProxyURL, ProxyName, ProxyVersion,
    ProxyCertificate, ProxyCertificateLength );
if (err != PrxOK) // display error message and quit

```

```

/* Begins execution */

```

```

/* Request the URL */
Char *URL = "http://www.yahoo.com";
Byte length = StrLen( URL );
Proxy_write( 0, &length, 1 );
Proxy_write( 0, URL, length );

```

```

/* Get the requested URL */
byte[] len = new byte[1];
mClient.read( len, 0, 1 );
byte[] urldata = new byte[len[0]];
mClient.read( urldata, 0, len[0] );

/* Parse the URL using Java's built-in
   URL parsing functions */
URL u = new URL( new String( urldata ) );
byte[] success = new byte[1];
success[0] = 0;
if (!u.getProtocol().equals("http")) {
    mClient.write( success, 0, 1 );
    return;
}

```

Client**Proxy**

```

int port = u.getPort();
if (port == -1) port = 80;
/* Connect to the HTTP server */
ProxySocket ps = connect( u.getHost(), port );
if (ps == null) {
    mClient.write( success, 0, 1 );
    return;
}
/* Request the file */
String request = "GET " + u.getFile() +
    " HTTP/1.0\n\n";
success[0] = 1;
mClient.write( success, 0, 1 );

/* Determine whether proxy was successful */
Byte success;
Proxy_read( 0, &success, 1 );
if (success == 0) {
    // display error message and quit
}

/* use ps.read() to read HTTP response. Parse
HTML and send resulting data to client. */

/* use Proxy_read() to get parsed response from proxy */

/* Shut down the proxy */
Proxy_close();
/* Display the pre-parsed web page */

```

3.7 Secure Sockets

Secure sockets are often used in client-server applications when the information being transmitted between the client and server is of a confidential nature. They are used every day to send passwords, credit card numbers, banking transactions, and other information across the global Internet. The secure socket layer (SSL) protocol can be applied on top of any reliable stream-based protocol, usually the TCP protocol in the Internet. SSL over TCP provides security in the Secure Hypertext Transfer Protocol for World-Wide Web access, and secure variants of the Post-Office Protocol (POP) and the Interactive Mail Access Protocol (IMAP) for electronic mail. Since many client-server applications make use of SSL, it is important that our proxy architecture support them in an elegant manner.

The proxy and client APIs allow SSL to be layered on top of any socket or channel created by the proxy. When an SSL connection is created, two pieces of information must be provided in addition to the underlying socket to be used. The first is the hostname of the server at the other end of the connection. As described in Section 3.4, this name is used to authenticate the server by comparing it with the name contained in the certificate provided by the server during negotiation. The second piece of information is an optional `SSLState` structure. This structure contains information about the cryptographic protocols, keys, and other information representing the cryptographic state of the SSL connection. When an SSL connection is initially created, this structure is omitted, and most applications will not need to work with it. However, the proxy's `SecureProxySocket` interface shown in Figure 3.11, the client's `ProxySSLClient` structure shown in Figure 3.15, and the

```
typedef struct ProxySSLClient {
    SSLClient *mSSLClient;
    int channel;
} ProxySSLClient;
```

Figure 3.15: The ProxySSLClient Structure

client's SSL interface shown in Figure 3.16 give access to this structure, facilitate the passing of the SSL state between the client and proxy. The details of this are described below.

Normally, a secure socket is used in this architecture exactly like a normal socket. A proxy calls `Proxy.secureSocket()` to create an SSL connection on top of an existing `ProxySocket`. The result is a `SecureProxySocket` that behaves exactly like a `ProxySocket` and includes its own channel number. The `SecureProxySocket` interface extends the `ProxySocket` to give access to the underlying `ProxySocket` and the `SSLState`. Once created, the proxy can read from or write to this socket as it would any other socket. In addition, by passing the channel number to the client, the client can read from or write to this socket with the aid of the proxy provider. A protocol stack diagram for this situation is shown in Figure 3.17. Contrast this with a similar diagram depicting a non-secure `ProxySocket` shown in Figure 3.18.

Occasionally, information sent through secure sockets is sufficiently confidential that neither the proxy nor the proxy provider should have access to it. In this case, the client can create a secure socket independent of the proxy using the `Proxy_secureChannel()` function, which takes the same parameters as the

```

void SSLClient_renegotiate( SSLClient *client );
Boolean SSLClient_read( SSLClient *client,
    SSLFragment *fragment );
Boolean SSLClient_write1( SSLClient *client,
    Byte *data, int len );
Boolean SSLClient_write( SSLClient *client,
    Byte **data, int *lens, int num );

SSLState *SSLClient_getState( SSLClient *client );
int SSLState_getEncodedSize( SSLState *s );
void SSLState_encode( SSLState *s, Byte *buffer );
SSLState *SSLState_decode( Byte *buffer, int length );

```

Figure 3.16: The Client SSL Interface

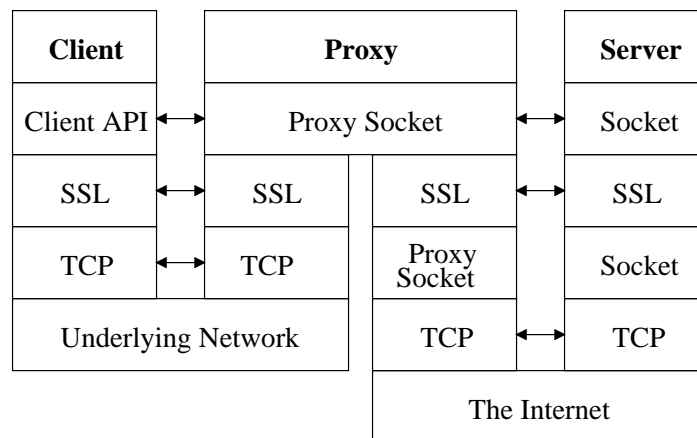


Figure 3.17: A Secure Proxy Socket Accessible to the Client and Proxy

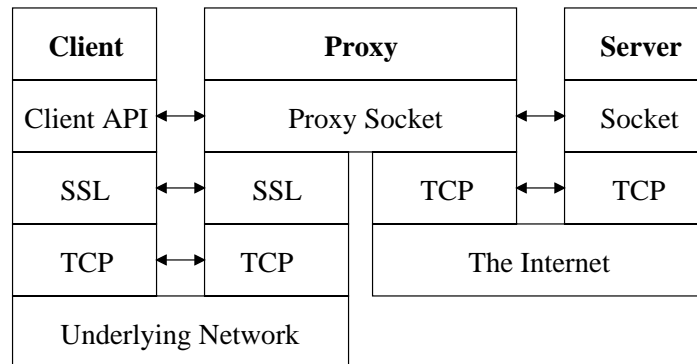


Figure 3.18: An Ordinary Proxy Socket Accessible to the Client and Proxy

`Proxy.secureSocket()` call. The client must then use the client SSL interface from Figure 3.16 to interact with this new secure socket. A protocol stack diagram for this situation is shown in Figure 3.19. Although the encrypted data of SSL passes through the proxy provider and can be accessed by the proxy, this information cannot be decrypted without the SSL state which is kept secret by the client and server.

Unfortunately, having a secure socket that passes securely through the proxy does not allow the proxy to manipulate the data passing through it. Since many client-server interactions involve first sending a password, then manipulating data of a less sensitive nature, mechanisms exist to pass the state of an SSL connection between the client and the proxy, effectively passing the end-point of the connection between these two parties in a manner that is transparent to the server. The SSL protocol supports the renegotiation of a connection, changing the cryptographic parameters used for subsequent data. Section 3.8 contains a detailed example and description of the actions that are taken to ensure the secure and successful hand-off

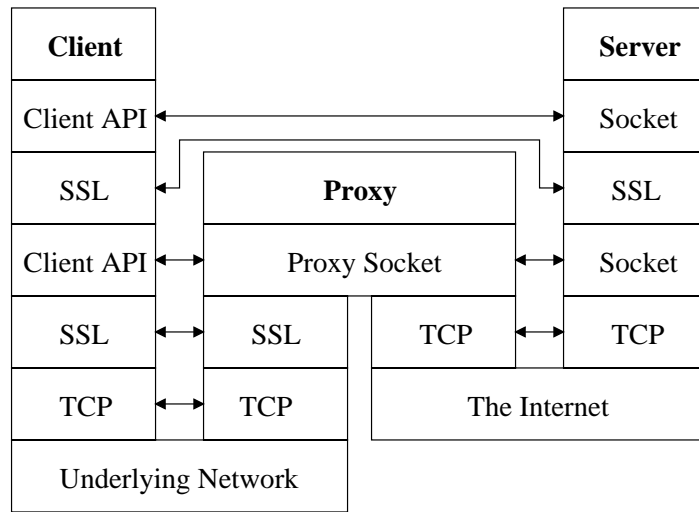


Figure 3.19: A Secure Proxy Socket Accessible to Only the Client

of an SSL end-point.

Figure 3.20 shows the possible configurations for sockets in this architecture. There is a single secure connection between the client library and the proxy provider over which all other connections are multiplexed. The socket with channel number 0 is used exclusively for communication between the client and proxy. The next

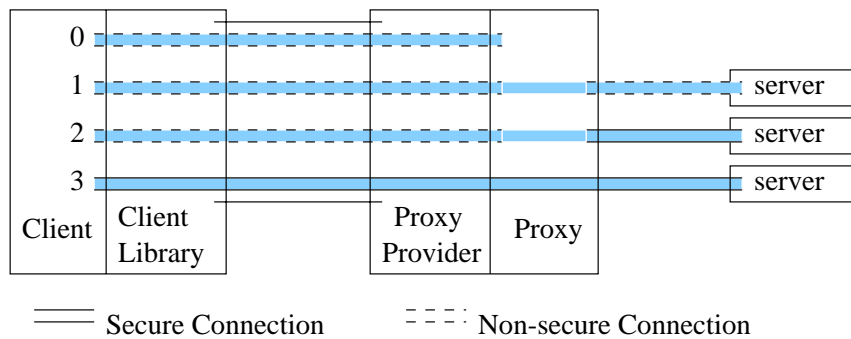


Figure 3.20: Possible Socket Configurations

socket is a non-secure socket that has been created by the proxy. The proxy can communicate with the server on this socket, and the client can also communicate with the server using channel number 1. A secure socket created by the proxy is shown next. This is created by the proxy to communicate with secure servers, but can also be used by the client from channel number 2. However, the proxy provider is able to access any data on this channel. The final socket is a secure socket created by the client. Using channel number 3, the client can communicate with a server without the possibility of eavesdropping from the proxy provider.

3.8 Secure Socket Example

The following example demonstrates the creation of a `ProxySocket`, the creation of an SSL connection over this socket at the client, and the passing of this SSL connection to the proxy. This example demonstrates the use of SSL state passing to access an electronic mail account without disclosing the account's password to the proxy or proxy provider.

Client**Proxy**

```

/* Load proxy */
Proxy_open();
Proxy_load(...);

```

```

/* Connect to the mail server */
ProxySocket ps =
    connect( servername, port );
/* Send channel number to client */
byte[] data = new byte[1];
data[0] = (byte)ps.getChannel();
mClient.write( data, 0, 1 );

```

```

/* Read channel number */
Byte buffer[1];
int channel;
Proxy_read( 0, buffer, 1 );
channel = buffer[0];

```

```

/* Negotiate SSL channel */
ProxySSLClient *client = Proxy_secureChannel(
    channel, servername, NULL );

```

```

/* send the password */
SSLClient_write( client->mSSLClient, ... );

```

```

/* renegotiate the connection. This ensures that even if the proxy recorded
   the encrypted data, it can no longer decrypt it when the state is passed. */
SSLClient_renegotiate( client->mSSLClient );
/* Extract the SSL state */
SSLState *state = SSLClient_getState( client->mSSLClient );
int size = SSLState_getEncodedSize( state );

```

Client**Proxy**

```

Byte stateData[size];
SSLState_encode( state, stateData );
Proxy_destroySecureChannel( client );
/* Pass the state to the proxy */
Proxy_write( 0, &size, sizeof(size) );
Proxy_write( 0, stateData, size );

/* Read the SSL state */
data = new byte[2];
mClient.read( data, 0, 2 );
int size = ((data[0] & 0xFF) << 8)
          | (data[1] & 0xFF);

/* Decode the SSL state and restart
   the secure session */
SSLState state = new SSLState();
state.decode( data );
SecureproxySocket sps =
    secureSocket( ps, servername, state );

/* Interact with proxy */      /* Interact with mail server and client */
:                               :

```

3.9 Comparison to Related Work

This architecture has a number of benefits over other proxy architectures and architectures that facilitate the deployment of executable code. As will be seen in Chapter 4, this architecture improves on traditional proxy architectures by delivering greater flexibility, more security features, and the ability to pull the proxy closer to the client, decreasing the network latency experienced by the user. Prox-

ies can be updated, and new client devices can be supported in a manner that is transparent to the end-user and the administrator of the proxy provider. Proxies for new applications can quickly become widely deployed, even before the user base becomes large, but the proxy will only ever be deployed where it is actually used.

This architecture builds on mobile code architectures. General mobile code architectures have security problems that cannot be solved in a general manner. When mobile code is to be given access to local resources such as local storage, and databases at the nodes that the code visits, this code must be authenticated. Who is to be given access to these resources is application-specific. In the case of active networks, the computation required to perform this authentication must be limited to reduce the overhead that it imposes on the processing of packets. However, by restricting the range of applications that are supported to include only proxies, the security problems become well defined and manageable. While mobile code architectures are faced with a trade-off between speed, security, and flexibility, proxies can be deployed in a manner that provides all three of these features. Finally, mobile agents and active networks must be widely deployed in order to be useful. On the other hand, proxy providers can initially be deployed in a few locations where they would benefit many users, with additional proxy providers deployed as they are required.

3.10 Summary

This chapter has described an architecture that allows new proxies to be deployed in a secure manner at the request of a client. This architecture easily allows clients

and proxies to communicate with each other and servers elsewhere in the network. Clients and proxies are given access to the popular Secure Socket Layer protocol to secure data transmitted to or from secure servers. The next chapter will discuss the implementation of this architecture and study its use in a number of applications.

Chapter 4

Implementation and Case Studies

This chapter discusses the implementation of the architecture, and goes on to discuss the steps required to verify that a particular proxy provider is executing correctly through the use of specially designed clients and proxies. Finally, a number of applications are studied to demonstrate how they are integrated into this architecture.

4.1 Implementation

The implementation of this architecture has been completed with the proxy provider implemented in the Java programming language using the Java Development Kit version 1.2.2 under the Linux operating system, and the client library implemented in the “C” programming language for the PalmTM line of portable devices running Palm OS[®] version 3.0 or higher. For cryptographic functions, version 3.1.1 of the free Cryptix libraries are used in Java. Under Palm OS[®], cryptographic functions

are provided by version 2.01 of the free pilotSSLey library, based on version 0.8.1b of the free SSLey libraries. The SSL protocol has been implemented from scratch in Java and in “C” to support the state transfer functions required by this architecture.

The proxy provider has been used variously on a 366 MHz Intel Pentium II notebook computer with 128 MB of RAM, and a dual 450 MHz Intel Pentium III desktop computer with 256 MB of RAM, both running RedHat Linux version 6.1. The former is currently equivalent to a typical consumer desktop computer, whereas the latter is a reasonable multi-user server system. Even on the consumer desktop computer, multiple simultaneous proxies can be supported performing complex tasks. By giving each proxy at least one thread for computation, the proxy provider is able to take advantage of multi-processor systems. Like many PalmTM devices, the PalmTM III device used for testing contains a 16 MHz Motorola 680x0 series processor with 2 MB of RAM.

Networking between the Palm and the rest of the network is provided by a PPP connection at a speed ranging from 9600 to 57600 bits per second. In this environment, with a 57600 bps PPP connection, the negotiation of a secure connection between the client library and the proxy provider takes approximately three seconds when using certificates signed with 512-bit keys. Because two RSA encryption operations are performed on the Palm device during negotiation, this time will increase for larger keys. However, 512-bit keys provide sufficient security in most applications. Over a slower 9600 bps PPP connection, this time increases by approximately two seconds because of the time taken to send the negotiation messages. However, the transmission time is dominated by the time required by RSA

operations. This negotiation time is acceptable given that the negotiation of the PPP connection also takes a number of seconds. As well, in most applications this negotiation need only be done once; all subsequent communication is multiplexed across this initial connection.

During operation, an SSL stream in this environment, over a 57600 bps PPP connection, can achieve throughput of up to 80% of the throughput achieved by a TCP connection in the same environment. This ratio depends on the size of the blocks of data being sent, with 80% encountered in experiments with 256-byte blocks of data. The SSL protocol encrypts each block of data and adds up to 8 bytes of padding, 20 bytes of message authentication code, and 5 bytes of protocol data. The additional data accounts for some of the difference in throughput: a 25-byte overhead on 256-byte blocks reduces the throughput by 9%. The remaining difference is due to the delay required to encrypt the data. As the speed of the PPP connection is reduced, the impact of this additional data remains constant while the impact of the encryption time decreases, so these numbers will actually improve for slower network connections.

4.2 Verifying the Integrity of the Proxy Provider

As previously mentioned, some degree of trust must be placed in the proxy provider and its administrator that the proxy provider will correctly perform its share of the security steps outlined in Chapter 3. This is because verifying that a system is behaving correctly from outside of that system is difficult. While a legal agreement can help to back up some claims, there are a number of tests that can be performed

periodically to determine whether a proxy provider is executing correctly. This section discusses the advantages of using secure sockets to secure the connection between the client and proxy, and examines tests that can be performed to verify the integrity of other aspects of the system.

4.2.1 Secure Sockets

It is worthwhile to examine which secure systems are known to work correctly without additional verification, and where problems may be encountered. Any portions of the system involving SSL are considered to be secure. The Secure Socket Layer protocol has been scrutinized by security experts for many years, and has not been shown to be vulnerable to the attacks that we are concerned with: server masquerading, eavesdropping and tampering. The SSL protocol depends on keeping a small amount of information secret, specifically the server's private key, but in practice this has been shown to be a manageable task. Even the passing of SSL end-points is secure; although the proxy has access to all of the data passing across the SSL connection, the protocol has been designed to allow a secure session to be negotiated under exactly these circumstances.

4.2.2 Proxy Authentication

Problems can arise when authenticating the proxy code. As long as the proxy provider is behaving correctly, the algorithms described prevent a party from substituting malicious code in place of the correct code before it is downloaded to the proxy provider. However, it cannot guarantee that the proxy provider even per-

forms the check, or that after performing the check the proxy provider does not simply substitute some malicious code in place of the correct code. Thus, the user should periodically verify that the authentication of code is performed correctly. This is a two-step process, first ensuring that the authentication procedure is followed, then ensuring that the code that is authenticated is, in fact, the code that is executed.

Verification of the authentication procedure requires a client, a proxy, and seven proxy descriptors. All of the proxy descriptors point to the same proxy code, but only one is valid. The remainder have some error; one has an incorrect proxy code signature, one an incorrect proxy descriptor signature, one has both incorrect, and the remaining three have correct signatures but an incorrect proxy name, proxy version, and identity of the proxy developer, respectively. These errors can be used to determine whether each of the signatures is verified, and whether each of the other pieces of information in the proxy descriptor is checked. Which proxy descriptor is correct should be known to a central server, and they should be rotated regularly to prevent a malicious party from learning the pattern and building a proxy provider that reproduces it. The client should first establish a secure connection with this central server to determine which proxy descriptor is correct. Next, a session with the proxy provider is established and each proxy descriptor is loaded in turn. Only when the correct proxy descriptor is specified should the proxy be loaded successfully. If the proxy is loaded successfully when any other proxy descriptor is specified, the user of the client can be confident that the proxy provider is not performing the proxy authentication procedure correctly.

The client should also verify that the authentication procedure is followed when the proxy code has been cached. When a proxy is loaded that has previously been placed in the cache, the authentication procedures must be repeated. Otherwise, a malicious party could “pre-cache” a malicious proxy, replacing it at the server and using a fake client that provides a valid certificate for this fake proxy. If the proxy descriptor and proxy code are not re-authenticated before they are used again, the client cannot determine that this has occurred. To detect this case, a client should first load a valid proxy using a correct certificate. Then, the client can attempt to load that same proxy with a certificate containing the same identity but a different public key. If this attempt succeeds, then the proxy provider has not correctly re-authenticated the proxy code.

4.2.3 Proxy Execution

The final step is to verify that the code that has been authenticated is, in fact, the code that is executed. This is done using a client, a proxy, and a special server that is established in the Internet. This server must maintain two key pairs. The first key pair is used only for signatures. The second key pair is used for encryption and is regenerated on an hourly basis. When this second key pair is regenerated, its public key is signed using the signing key pair and published in a public forum like a Web page. The client must be distributed with knowledge of the public half of the signing key pair, in addition to the certificate used to authenticate the proxy.

The client must first obtain the public half of the current encryption key pair from the public forum. This public key is verified using the signature and the

public half of the signing key to detect tampering. The client then connects to the proxy provider and requests that the proxy be loaded. The client generates a block of random data and sends it to the proxy. The proxy performs some arbitrary action on this data, such as a cryptographic hash, and then connects to the server and passes this processed data on. The server encrypts the data that it receives using the private encryption key and returns it to the proxy where it is passed to the client. The client verifies that the proxy code was executed and that the connection to the server was carried out by repeating the same arbitrary action on the original data, decrypting the result from the server using the public description key and comparing the results.

This test is made more robust by regularly varying the type of arbitrary action that is performed, distributing a new client and proxy on a regular basis. If the number of actions that can be performed is sufficiently large, it will be very difficult to produce a proxy provider that can correctly produce a fake response for all possible combinations. By following these steps when using a new proxy provider and on a regular basis thereafter, a user can be confident that the proxy code is being downloaded, authenticated, and executed correctly.

These tests can be used to verify that the proxy provider is performing at least the actions that the client and proxy expect of it. However, there is no guarantee that the proxy provider will not perform actions beyond those described here. In particular, there is no way to prevent the proxy provider from logging any messages that it sees, or from tampering with information that is not sent over a secure connection. On the other hand, in the absence of this architecture these

problems would still be present. Where the client does not use secure sockets, this architecture can provide additional security against tampering and eavesdropping for information exchanged between the client and proxy. When secure sockets are used, the client can obtain a secure connection through to the server, protected from tampering and eavesdropping, for extremely sensitive information, yet still benefit from the proxy's ability to process less sensitive information delivered over the same connection. This architecture gives the client and proxy greater control over end-to-end security.

4.3 Case Studies

This section examines how the proxy provider architecture can be used in a number of applications to deploy proxies when and where they are needed. The first application that will be studied is a proxied World-Wide Web browser that can interact securely with secure web servers. The second, and simplest application is a streaming media proxy. The final application is a proxied server, where the wireless device is supporting a server with the assistance of a proxy.

4.3.1 World-Wide Web Browser

The World-Wide Web has become the most popular application in the global Internet. World-Wide Web content is currently available to compact and wireless devices. Sometimes a full-featured browser is available, but more often the browser is limited to a text display, or a proxy is used to assist with page layout and image decoding. Proxies like the TranSend proxy [7] bring graphical web browsing to the

Palm^[TM] device, but centralize the processing of this proxy to one or more systems located at possibly distant locations in the Internet. Because such a specialized proxy is useful to a limited population of users, wide deployment is unlikely, so users face higher latencies than they would if they could access a proxy closer to their own location in the network. Secure, client-driven deployment of this proxy would allow the proxy to be moved close to the users that need it.

The TranSend proxy does allow access to secure web servers. However, all information delivered to that secure web server is visible to the proxy. For some applications, this might be a source of concern. On the other hand, the information that is received from a secure web server is often less sensitive than the information that is sent to it, including passwords and credit card numbers. A World-Wide Web proxy can take advantage of the security features of the proxy-provider architecture to ensure that information sent to the web server is encrypted from the client, through the proxy to the server, while still allowing the proxy to process the results that are returned. This makes the system more secure by eliminating the need to trust the developer of the proxy.

4.3.2 Streaming Media Proxy

Motivation

A proxy that processes streaming media on behalf of a wireless device can provide great benefit in this architecture. Consider an MPEG video stream. Often, such streams contain full-colour motion video to be rendered at resolutions of 320 by 200 pixels or higher, at 15 frames per second. The MPEG video standard achieves high

video compression rates using both standard loss-less data-compression techniques, and lossy compression techniques based on the computationally expensive discrete Fourier transform function. However, there are circumstances where one might like to view an MPEG video stream on the 160 by 160 pixel, four-bit greyscale display of a Palm^[TM] device with a reasonable frame rate. It is unreasonable to expect the Palm^[TM] device to perform the complex processing required to decode the video stream, and wasteful to send the entire video stream across a low-bandwidth connection when the resulting images will be reduced in scale and have much of their colour information removed. On the other hand, a proxy tuned to the specific characteristics of the Palm^[TM] devices can decode the video stream on a more powerful computing platform with a faster network connection, before sending a smaller video stream to the Palm^[TM] device.

Description

The MPEG proxy is based on the Java MPEG viewer developed by J. Anders at TU-Chemnitz [2]. Once loaded, the client provides a URL from which an MPEG video stream should be loaded. As it is loaded, it is decoded into individual frames; these frames are each reduced to four shades of grey and the smaller resolution required by the Palm^[TM] device. Finally, the frames are run-length encoded and delivered to the Palm^[TM] device in this compressed format. Run-length encoding (RLE) was chosen to provide reasonable compression while being computationally inexpensive to decode. The system is shown in Figure 4.1.

Seven sample MPEG streams were used to test this system. These video streams had different initial dimensions, but all were viewed at the largest dimension that

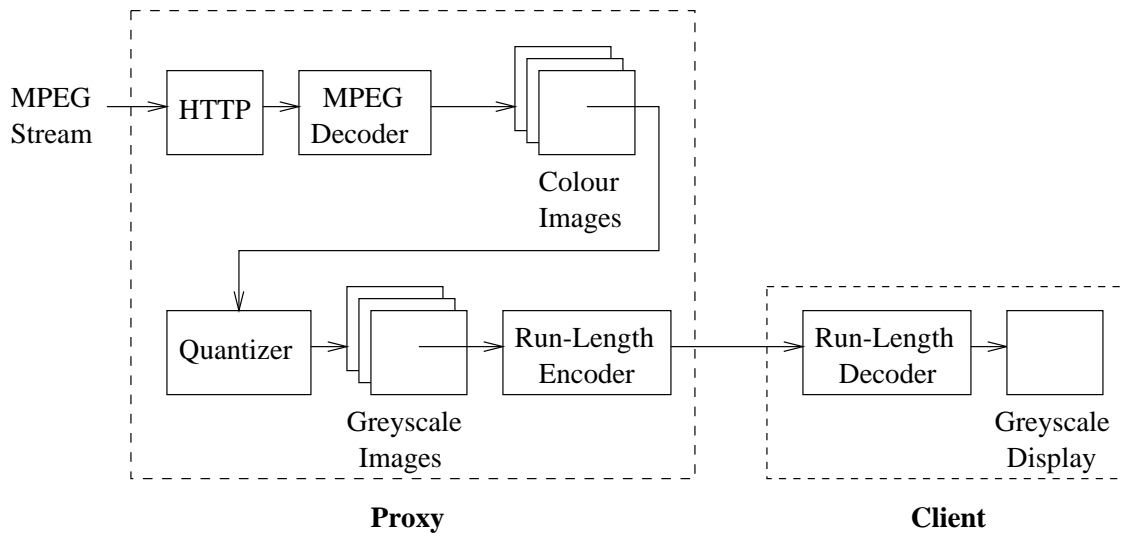


Figure 4.1: MPEG Streaming Media Proxy

would fit within the 160 by 120 pixel viewer. In cases where the MPEG stream contained an audio track, these were discarded during decoding. Details of these tests can be found in Table 4.1. For each stream, this table lists its total length in bytes, the dimensions of its frames in pixels, the total number of frames, and whether it contains an audio track. This table also lists statistics about the data after it has been processed by the proxy, including the minimum, maximum, and mean run-length encoded frame lengths in bytes, and the total size of the run-length encoded data sent to the client.

Benefit

This particular application benefits from this architecture in two ways, first through the use of a proxy, and second through the ability to securely deploy the proxy when and where it is needed.

Stream	dreh.mpg	random.mpg	Ext.mpg	ibookad.mpg
Total Length	50 456	39 769	5 256 892	2 211 840
Dimensions	256×192	320×240	240×160	208×160
Number of Frames	15	27	903	799
Audio Track?	No	No	Yes	Yes
RLE Frame Lengths:				
Minimum	1613	209	35	3
Maximum	2369	390	4737	1621
Average	2131	325	2003	786
Total Transmitted Data Length	31 961	8799	1 809 462	627 921

Stream	starwars.mpg	topgun3d.mpg	soxmax.mpg
Total Length	1 751 044	1 092 280	17 881 523
Dimensions	160×112	160×128	240×160
# of Frames	2918	1015	9376
Audio Track?	Yes	Yes	Yes
RLE Frame Lengths:			
Minimum	3	371	3
Maximum	5303	3182	2965
Average	2045	1253	1530
Total Transmitted Data Length	5 966 714	1 271 511	14 342 707

(lengths are in bytes, dimensions are in pixels)

Table 4.1: Greyscale MPEG results

This application would not be feasible without the use of a proxy. The Palm^[TM] devices simply do not have the processing speed required to decode MPEG video at a reasonable rate. Based on tests of the type of integer arithmetic operations performed during MPEG decoding, the Palm^[TM] devices are approximately 175 times slower than the consumer desktop computer used for testing, so whereas the desktop computer can decode approximately 5-7 frames per second for streams used, the Palm^[TM] device would decode one frame every 25 to 35 seconds. Since much of the information is lost from the decoded video images when displayed on the Palm^[TM] device's limited screen, the size in bytes of the run-length encoded images from the MPEG proxy is almost always smaller than the highly compressed MPEG stream, as shown in Table 4.1. Tests have also shown that the MPEG proxy can decode frames faster than they can be transmitted to the Palm^[TM] device over a 57600 bps connection, so the network connection becomes the bottleneck when the proxy is used. The benefit of the proxy is clear since the original MPEG video stream would usually take longer to transfer to the Palm^[TM] device, and still longer to decode without the aid of the proxy. In the few cases where the decoded images are larger than the original MPEG stream, there would be a trade-off between the extra time taken to download the frames and the time saved by not decoding the frames on the Palm^[TM] device. Where the difference is small, the proxy still provides a benefit. In the test environment, the Palm^[TM] device can display an MPEG stream at a frame-rate of approximately one frame per second. The proxy has been further extended to drop frames to achieve the frame rate specified in the video stream or some multiple of that frame rate. These frames never pass between

the proxy and client, so this scheme does not adversely affect the rate at which the remaining frames are displayed. Where the size of the decoded stream is greater than the size of the original stream, dropping frames now results in a smaller stream passing over the network link.

Such a proxy could be deployed at one or more locations in the global Internet to assist users of PalmTM devices in viewing MPEG video streams. However, the ability to deploy such proxies when and where they are needed has its advantages. First, this would distribute the computational requirements to multiple nodes in the network. Users of this proxy would be sharing resources with other users of their proxy provider, rather than with users around the world. New instances of the proxy would be deployed when users needed them, and the proxy provider's resources could be reclaimed and used by other applications when the proxy was not in use. In addition, because the network connection between the proxy and the client is the bottleneck in this application, it is desirable that this connection be used effectively. Compared to a proxy deployed close to the client device, a proxy located at some distant point in the global Internet would see higher latencies and be more likely to experience network congestion.

4.3.3 Proxied Server

Motivation

Occasionally, one would like to operate a server on a wireless or portable device. An example of such a server is the Comma execution environment monitor (EEM), developed at the University of Waterloo [12]. The Comma EEM allows applications

to determine details about an execution environment such as the available memory or the speed of its network connection. This information can be used to support adaptation to accommodate changes in the device or network environment.

Unfortunately, when one attempts to deploy such a server on a compact, wireless device one encounters several limitations. Often such devices have a limited computation model that restricts the creation of multiple threads, or a limited network implementation that severely restricts the number of simultaneous network connections that can be created. For example, on Palm^[TM] devices background threads must be used very sparingly, and where they are used they must limit their use of the available memory because they share the extremely limited memory with other applications that are not normally expected to compete with background threads. The TCP/IP networking stack can maintain at most four open TCP connections, restricting the number of simultaneous clients that can be supported, especially when another client-server application is being used at the same time. These limitations affect the server's ability to accommodate multiple simultaneous clients. As well, if one or more clients of the EEM are located across a wireless link in the network, disseminating this information to the multiple clients will consume more of the already limited network bandwidth. Finally, because it is important to restrict the consumption of the limited memory available on these devices, it becomes a challenge to maintain state information for multiple clients: what EEM variables they are interested in and how often they would like updates.

This section investigates the creation of a proxied Comma EEM server for use on Palm^[TM] devices.

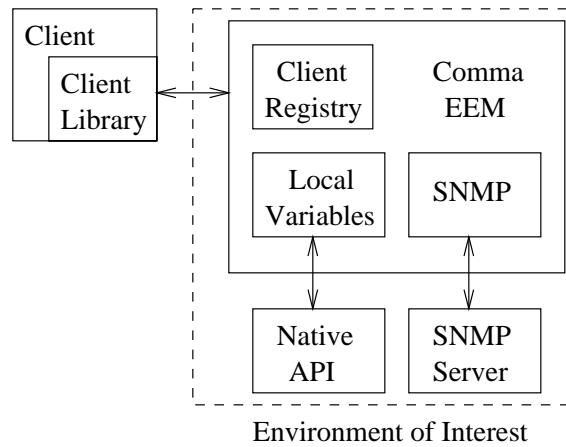


Figure 4.2: Existing Comma Execution Environment Monitor

Description

The structure of the existing Comma EEM is shown in Figure 4.2. The client library handles all interaction with the Comma EEM. Multiple clients can connect and have access to variables whose values are obtained through the native API of the system on which the EEM runs as well as variables obtained from an SNMP (Simple Network Management Protocol) server. The current Comma EEM obtains the bulk of its information through SNMP with only a few additional local variables. In particular, SNMP provides information and statistics about network interfaces, and the IP, TCP, and UDP protocols, as well as the system name and other descriptive information. These variables are accessed in a standard manner and can be reported by any environment that supports an SNMP server. On the other hand, local variables such as the amount free memory are collected in a system-dependent manner and separate handlers must be written for each platform on which they are available.

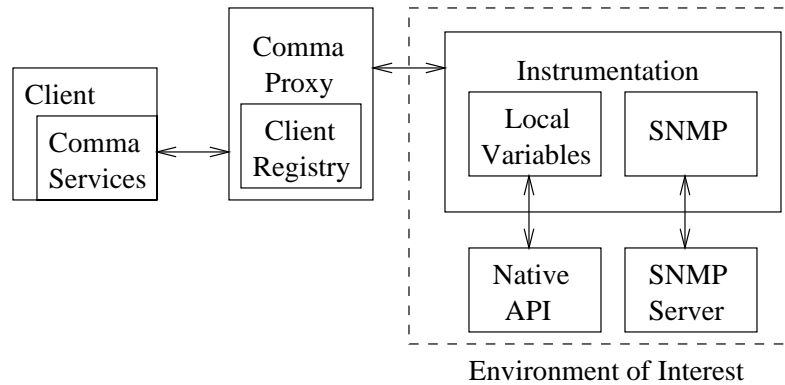


Figure 4.3: Proxied Comma Execution Environment Monitor

The structure of the new, proxied Comma EEM is shown in Figure 4.3. The original Comma EEM has been divided into a Comma Proxy and the Instrumentation. The Comma Proxy is responsible for handling requests from multiple clients, maintaining client state, keeping track of time for periodic variable updates, and handling the complex protocol between the client and proxy. When the Instrumentation initially connects to the Comma Proxy, it provides a table of variables for which it can give values. For simplicity, the Instrumentation does not need to parse this table, simply generate it; normally, it would be compiled into the Instrumentation. Each variable is assigned a name and a number based on its location in this table. The protocol between the Comma Proxy and the Instrumentation is simplified greatly by using only these numeric values, reducing the bandwidth used by this protocol. The Instrumentation does not handle any client interaction, nor does it deal with periodic updates. Instead, it simply provides a value for a variable when requested by the Comma Proxy.

This design is more general and simplifies the implementation of the Comma

Proxy and the Instrumentation. Because they are separate, the Comma Proxy does not need to know how variables are obtained, and the Instrumentation does not need to know how to interact with clients or perform periodic updates. To the Comma Proxy, all variables are treated the same; for example, on a Palm^[TM] device all variables that would normally be accessed via SNMP are available, but must be accessed through the native API on the device since an SNMP server does not exist. Moreover, because the Comma Proxy does not require any platform-specific handling, it is more portable. When porting to a new platform, most of the changes will be required in the smaller, simpler Instrumentation.

Benefit

Automatic deployment of proxies makes it possible to use the Comma EEM to monitor environments on small, portable, wireless devices. The use of a proxy reduces the resource requirements of the Comma EEM on the device being monitored by simplifying its processing requirements and eliminating the need to interact with multiple clients. In particular, the Instrumentation deployed on the Palm^[TM] device contains less than 10 kilobytes of code and leaves enough memory available to run any of a number of popular World-Wide Web browsers and other network applications. Automatic deployment means that the Comma Proxy will appear where it is needed, when it is needed, to support wireless devices that are only intermittently connected to the network and that may connect from multiple different locations.

4.3.4 Summary

These three case studies demonstrate that a diverse set of applications can benefit from the ability to use proxies that are deployed automatically and securely. There is one final benefit that can be realized from these case studies: that these varied applications can be supported on the same platform, developed by different developers, and deployed automatically and securely without needing the intervention of a system administrator.

Chapter 5

Summary and Future Work

5.1 Summary

With a growing population of users accessing services in the Internet through wireless networks using small, portable devices, proxies have become a useful tool to make these services more convenient. Wireless devices often have limited display and input devices, less memory and persistent storage, and slower processing speed than do other devices used to access the Internet, and this is a trend that is unlikely to change. Proxies can pre-process information into a form that is more manageable for these restrictive devices, reducing wasted network bandwidth and time. Unfortunately, traditional proxies have faced a trade-off between wide deployment and specialization, a trade-off that is addressed by this architecture.

The architecture that has been presented allows proxies to be deployed automatically and securely to perform a task that is useful, in particular for the benefit of wireless devices in client-server applications. Proxies are deployed when and where

they are needed, providing wide deployment of specialized proxies to benefit small and large user communities alike. The executable code for proxies is distributed using existing infrastructure in the Internet, simplifying wide deployment of this architecture. All executable code is authenticated, protecting the proxy's client from malicious intent. Proxies are executed in a secure environment, but are still provided with a rich API, including secure networking functions, giving them the flexibility to perform a variety of tasks to bring Internet services to the wireless user.

5.2 Future Work

Like any architecture, there is room for extension and improvement. This section describes a number of avenues for future investigation.

5.2.1 Persistent Storage

While many proxies can be written to simply process the data that passes through them, there are times when the ability to store some state information between proxy sessions would be useful. Such state information might include configuration parameters or cached data. In many cases, this state information should only be accessible by the client and proxy that stored it. However, in some cases, like caching Web proxies, it is beneficial to have this state information shared among all instances of a particular proxy. As well, future versions of a particular proxy will likely want access to state information from previous versions, and some state information might be useful to a number of similar proxies used by the same user.

It is possible to encrypt any data stored in persistent storage, but if this data were to be accessible to only the client, this scheme would require that this data be transmitted to the client for encryption. In this case, the data may be better stored by the client in the first place. It may be more efficient to attach one or more certificates to this data, requiring an appropriate challenge and response from the client or the proxy, depending on what access restrictions are in place. Investigation is required into what specific combinations of security are useful to proxy developers, and how this security can be accomplished efficiently.

5.2.2 Proxy Mobility

Depending on the nature of the proxy, its proximity to the wireless device may be very important. Since many wireless devices are portable, the gateway or network that a device uses to access the Internet may change, moving the device further from a particular proxy provider. Moving the proxy to a new proxy provider each time the device moves will impose a significant cost on that movement, but the benefits might outweigh this cost in some circumstances. Proxy mobility may be accomplished transparently, bundling the data and the code automatically when the proxy should move, or some of these steps might be better performed manually by the proxy when mobility is required. The latter option allows the proxy to control the amount of data transferred by eliminating temporary data, and can give the proxy more control over its mobility. Areas of investigation include when proxy mobility might be useful, how it should be accomplished smoothly and efficiently, and how to reduce the costs associated with proxy mobility to make it more effective

where it is used.

5.2.3 Resource Limitation

The secure environment in which proxies are executed currently gives those proxies a great deal of range in terms of the amount of memory, and computational resources they consume. This was done to ease implementation and to give proxies more flexibility. In a production environment, limits should be imposed to prevent a rogue proxy from consuming large amounts of resources, affecting the smooth operation of the proxy provider. Imposing such limitations on a proxy would require modification of the proxy provider and the Java virtual machine to monitor the proxy's resource consumptions and taking appropriate actions when these limits are exceeded[5]. Further investigation is required to determine what level of resource limitation would leave proxies with enough flexibility to perform a broad range of tasks.

5.2.4 Integration with Other Proxy Architectures

A final area of investigation is integration with other proxy architectures. This architecture has been designed on its own, but the authentication features could be applied to other proxy architectures to add client-driven deployment.

One such architecture is the Comma Service Proxy architecture designed at the University of Waterloo. Here, network packets passing through a gateway are transparently passed through filters to transform the data on stream connections without the client or server's intervention. Currently, filters are written in native

code, but a fast interpreted language could be employed to give filters the flexibility to transform the packets in an arbitrary manner without being given access to the native API. Filters must be configured by a separate configuration program which could be modified to instruct the service proxy to download, authenticate, and apply new filters. To maintain a secure environment, such a client should only be allowed to apply filters to packets destined for itself. The primary problem here is in selecting or designing the language and API to provide good filter performance and flexibility while maintaining system security. Because the delivery of packets is a time-sensitive task, many interpreted languages will not be able to process packets at a rate that does not adversely affect the end-to-end performance of the application being filtered.

Another architecture in which proxies are employed is the Wireless Application Protocol (WAP^[TM]) suite of protocols and technologies used to bring Internet services to portable devices including cellular phones and pagers. Currently, proxies must be installed by system administrators, but there is no reason that the process could not be client-driven, giving the user the ability to drive the deployment of new services.

5.2.5 Verifying the Integrity of the Proxy Provider

Section 4.2 describes steps that can be taken to determine whether the proxy provider is doing its job correctly. Work is still required to develop a client, proxy, and server to implement these steps. An implementation can be used to determine how long these tests take, indicating the feasibility of performing these tests on a

regular basis. Additional work must also be done to modify the proxy provider to fail these tests in some way to ensure that these tests function as required.

5.3 Conclusion

This thesis presents an architecture that allows application-specific proxies to be deployed automatically and securely to perform tasks that benefit wireless devices in client-server applications. This architecture affords wireless devices greater access to the abundant services currently offered in the Internet, and allows new services to be extended to wireless devices with ease. The existing implementation provides a starting point, opening the door to future work that will make this architecture even more powerful and flexible.

Bibliography

- [1] D. Scott Alexander, William A. Arbaugh, Michael W. Hicks, Pankaj Kakkar, Angelos D. Keromytis, Jonathan T. Moore, Carl A. Gunter, Scott M. Nettles, and Jonathan M. Smith. The SwitchWare active network architecture. *IEEE Network*, 12(3):29–36, May/June 1998.
<http://www.cis.upenn.edu/switchware/papers/switchware.ps>.
- [2] J. Anders. MPEG-1-Player.
http://rnvs.informatik.tu-chemnitz.de/ja/MPEG/MPEG_Play.html.
- [3] Wendy Chisholm, Gregg Vanderheiden, and Ian Jacobs. *Web Content Accessibility Guidelines 1.0*. World Wide Web Consortium, May 1999.
<http://www.w3.org/TR/1999/WAI-WEBCONTENT-19990505/>.
- [4] P. R. Cohen and A. Cheyer. An open agent architecture. In O. Etzioni, editor, *Software Agents — Papers from the 1994 Spring Symposium (Technical Report SS-94-03)*, pages 1–8. AAAI Press, March 1994.
- [5] Grzegorz Czajkowski, Tobias Mayr, Praveen Seshadri, and Thorsten von Eicken. Resource control for Java database extensions. In *Proceedings of the*

- Fifth USENIX Conference on Object-Oriented Technologies and Systems*, pages 85–97. The USENIX Association, 1999.
- [6] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1, RFC 2068*. IETF Network Working Group, January 1997.
- [7] Armando Fox, Ian Goldberg, Steven D. Gribble, David C. Lee, Anthony Polito, and Eric A. Brewer. Experience with Top Gun Wingman, a proxy-based graphical web browser for the USR palmpilot. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Lake District, U.K., September 1999.
<http://gunpowder.Stanford.EDU/fox/PAPERS/wingman.ps.gz>.
- [8] Alan O. Freier, Philip Karlton, and Paul C. Kocher. *The SSL Protocol Version 3.0, Internet Draft*. IETF Transport Layer Security Working Group, November 1996.
- [9] Robert Gray, David Kotz, Saurab Nog, Daniela Rus, and George Cybenko. Mobile agents for mobile computing. Technical Report PCS-TR96-285, Dept. of Computer Science, Dartmouth College, May 1996.
- [10] Jon Gunderson and Ian Jacobs. *User Agent Accessibility Guidelines 1.0*. World Wide Web Consortium, March 2000.
<http://www.w3.org/TR/2000/PR-UAAG10-20000310/>.
- [11] R. Housley, W. Ford, W. Polk, and D. Solo. *Internet X.509 Public Key*

- Infrastructure: Certificate and CRL Profile, RFC 2459*. IETF Network Working Group, January 1999.
- [12] David Kidston, J. P. Black, Thomas Kunz, Michael E. Nidd, Marcello Lioy, Brent Elphick, and Michal Ostrowski. Comma, a communication manager for mobile applications. In *The Eleventh International Conference on Wireless Communications*, pages 103–116, Calgary, Alberta, Canada, July 1998.
<http://www.shoshin.uwaterloo.ca/pub/papers/Ps/conf24.ps.Z>.
- [13] David Kotz and Robert S. Gray. Mobile code: The future of the Internet. In *Proceedings of the Workshop “Mobile Agents in the Context of Competition and Cooperation (MAC3)” at Autonomous Agents ’99*, pages 6–12, Seattle, Washington, USA, May 1999.
- [14] Mitsuru Oshima, Guenter Karjoth, and Kouichi Ono. *Aglets Specification 1.1 Draft*. IBM, 0.65 edition, September 1998.
<http://www.trl.ibm.co.jp/aglets/spec11.html>.
- [15] C. Perkins. *IP Mobility Support, RFC 2002*. IETF Network Working Group, October 1996.
- [16] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 2(21):120–126, 1978.
- [17] David Wetherall, John V. Guttag, and David L. Tennenhouse. ANTS: A toolkit for building dynamically deploying network protocols. In *IEEE*

OPENARCH '98, San Fransisco, USA, April 1998.

<http://www.tns.lcs.mit.edu/publications/openarch98.html>.