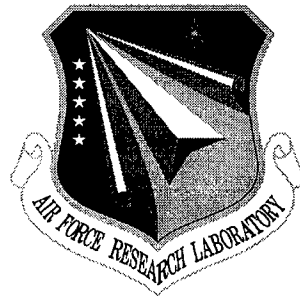AFRL-IF-RS-TR-1999-223
Final Technical Report
October 1999

ADAPTABLE DEPENDABLE WRAPPERS

Key Software, Inc.

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. E288

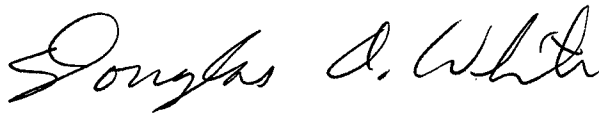*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

19991220 036

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1999-223 has been reviewed and is approved for publication.

APPROVED:
DOUGLAS A. WHITE
Project Engineer

FOR THE DIRECTOR:
NORTHRUP FOWLER, III, Technical Advisor
Information Technology Division
Information Directorate

ADAPTABLE DEPENDABLE WRAPPERS

Franklin Webber

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED	
	October 1999	Final	Sep 96 - Jun 99

4. TITLE AND SUBTITLE

ADAPTABLE DEPENDABLE WRAPPERS

6. AUTHOR(S)

Franklin Webber

5. FUNDING NUMBERS

C   - F30602-96-C-0355
PE - 62301E
PR - E017
TA - 04
WU - 03

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Key Software, Inc.
131 Hopkins Road
Ithaca NY 14850

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFTD
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1999-223

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Douglas White/IFTD/(315) 330-2129

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The objective of the Adaptable Dependable Wrappers effort was to design and implement tools that facilitate the creation of specialized wrappers for software components of a distributed system. The affordable upgrading of systems requires better technology than currently available to support evolutionary adaptation through incremental change. Support is needed for distributed systems so that incremental changes may be applied routinely and inexpensively. This support must allow: incorporation of new code into running systems and diffusion of new code throughout distributed subsystems; automatic analysis of new code to identify its worst-case behavior and interference with existing code; fine-grained control over privilege given to new code; protocols to coordinate the diffusion of new code; replacement of protocol layers without disruption to processing in other protocol layers; and protocols to tolerate and adapt to subsystem failures.

Much of the technology needed for evolutionary computing can be localized in software wrappers. Software wrappers are often used to glue existing subsystems into a larger system with new properties and functions. A wrapper is a software layer used to change the interface of a component or to give new properties, such as fault tolerance or security to the interaction between components. By changing the wrappers, component interfaces can be changed to allow new connections between components. Properties can be changed to satisfy new requirements. These changes permit much of the flexibility that evolutionary computer systems need. This report describes an investigation into tools which support the creations of such wrappers.

14. SUBJECT TERMS

wrappers, fault tolerance, adaptability, evolutionary computing, non-stop computing, distributed computing

15. NUMBER OF PAGES

84

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

4 Conclusion   **66**

Chapter 1

Introduction

1.1 Report Identification

The Adaptable Dependable Wrappers research project was performed by Key Software, Inc., of Ithaca, New York, over a 33 month period ending June 1999. This project was sponsored by the US Defense Advanced Research Project Agency (DARPA). The USAF contract number for this work is F30602-96-C-0355. Its DARPA Order Number is E288.

This document is the final report of technical results from the project. It is a complete and self-contained description of those results. It incorporates most of the text of previous project reports, and therefore supersedes them.

We assume that readers of this report are familiar with general terminology and concepts underlying computer systems and computer programming, such as compilation, execution, processors, processes, networking, and operating systems. We do not assume any prior knowledge of specific system properties, such as computer security, fault tolerance, or real time, or of specific programming languages, network protocols, or operating systems. The report will introduce terminology and summarize relevant concepts from these specific areas when they are needed, and will provide references to the literature for more detailed study.

1.2 Project Goals

The goal of the Adaptable Dependable Wrappers project was to design and implement tools that make it easy to create specialized wrappers for software components of a

distributed system. These wrappers allow components to be glued together into larger systems. The wrappers must be dependable for the system to be dependable, and they must be adaptable to allow for a changing environment.

Adaptable, dependable wrappers are a step toward building evolutionary computer systems, which are defined in the following section.

1.2.1 Evolutionary Computing

In the future, an increasing number of systems will require evolutionary computing. An *evolutionary computer system* is one that must continue processing

- while adapting to a changing environment,

- while permitting software and hardware upgrades, and

- in spite of failures of some of its subsystems.

Evolutionary computer systems continue to operate over long periods of time and must not be rebooted during those periods, either because reboot is impractical or because a reboot would take too long, preventing the system from carrying out its real-world mission.

An evolutionary computer system must be flexible enough to adapt and to be modified while it is in use. Over a long period of time, change is inevitable and flexibility is needed to cope with that change.

The largest evolutionary computer system in use today is the Internet. In principle, the Internet could be rebooted: it does not have strict performance requirements and therefore is not a real time system. Rebooting the Internet, however, is impractical because too many people are involved in its administration. Rebooting the Internet to upgrade it would also be expensive because it means replacing specialized hardware. So the currently planned upgrade of the Internet, to replace the Internet Protocol (IP), will be a gradual replacement of IP version 4 with IP version 6[BM95]. The two versions of IP will coexist in the Internet for a long time, and at no time will the Internet be rebooted to make the upgrade.

Evolutionary computing systems will become more common as the embedded systems now used for real time control of many processes become increasingly linked into larger distributed systems. Currently, most embedded systems are isolated: in cars, VCRs, machine tools, etc. Soon, however, many of these embedded controllers will be part of the Internet and will be interconnected into systems of systems. The larger such

5

a system of systems grows, the harder it will be to arrange to reboot it. Eventually reboot will be unacceptable and a new evolutionary computing system will be born.

Evolutionary computer systems are typically distributed. A system is *distributed* if it has components that run on separate computers, called *hosts*, that share no physical memory. The hosts of a distributed system are linked by a network of communication paths. Components of the distributed system communicate using *protocols*, which are well-defined patterns of interaction involving two or more components.

Better technology is needed to support evolutionary computing. The upgrade of the Internet Protocol will be an expensive change and so is not ideal as an example of how to upgrade an evolutionary system. Better would be support for incremental changes that could be applied as routinely and as inexpensively as single-host operating systems are now upgraded and patched. This support should allow:

- incorporation of new code into running systems and diffusion of new code throughout distributed subsystems;

- automatic analysis of new code to identify its worst-case behavior and identify interference with previously running code;

- fine-grained control over privilege given to new code;

- protocols to coordinate the diffusion of new code;

- replacement of protocol layers without disruption to processing in other protocol layers;

- protocols to tolerate and adapt to subsystem failures.

1.2.2   Wrappers

Much of the technology needed for evolutionary computing can be localized in software wrappers. A *wrapper* is a software layer used to change the interface of a component or to give new properties, such as fault tolerance or security, to the interaction between components. By changing the wrappers, component interfaces can be changed to allow new connections between components. Properties can be changed to satisfy new requirements. These changes permit much of the flexibility that evolutionary computer systems need.

Software wrappers are often used to glue existing subsystems into a larger system with new properties and functions. The wrappers know the protocols needed to make the subsystems work together, even if they were not originally designed for a common

6

purpose. When a system is reconfigured, either to replace a subsystem or to change the quality of service offered on a communication link between subsystems, it is the software wrappers that enable the reconfiguration by replacing or augmenting the protocols they use.

1.3 Project Overview

The software developed on this project supports the construction of wrappers that can run a variety of protocols. The wrappers can use protocols that enhance system dependability. The wrappers can adapt by changing the set of protocols they use.

The project built upon previous work. Wrappers for dependable systems and wrappers with variable sets of protocols have both been built before. The contribution of this project, however, was to focus on wrappers that can adapt by downloading code for new protocols from other wrappers and running it, while maintaining critical properties of the system in which they are embedded. This focus on adaptability and critical properties to support evolutionary computing distinguishes this project from other work on wrappers.

1.3.1 Background: Dependable Wrappers

A wrapper that is *dependable* imparts critical properties to each component that it wraps. In this report "dependability" includes both fault tolerance and security. Protocols exist to support both these critical properties.

A system is *fault tolerant* if it will continue correct operation in spite of failures of some of its subsystems. A conceptually simple way to increase the fault tolerance of a software component is to replicate it, distribute it, and coordinate the replicas. If not too many of the distributed component replicas fail, then the component can recover from the failures and continue to operate correctly.

Replica coordination can be implemented by a component wrapper. This means that a component need not be changed when replica coordination is added to it: the wrapper carries out the algorithms needed for replica coordination and offers the component the same interface it had before wrapping. Wrapping separates the design of functionality from the design of fault tolerance. This has several advantages:

- To protect against a different kind of component failure one need only change the wrapper, not the component.

7

- In principle any kind of component may be wrapped, including legacy systems.

- Wrappers may be used to protect against design flaws as well as hardware failure if independent designs are used for each of the replicated components[AK84].

One well-known approach to building fault tolerant wrappers is Schneider and Lamport's "State Machine Approach"[Sch90]. Many different protocols can be used within the State Machine Approach, for example [Cri85][LSP82][BSS91][Rei96].

On a previous project, Key Software implemented the State Machine Approach within Rome Laboratory's Knowledge-Based Software Assistant (KBSA)[Ben94]. Our implementation constructed fault tolerant wrappers from a specification of the number and kind of failures to be tolerated[Key95]. The wrapper functionality is described in the KBSA component specification language, which is object-oriented and similar to C++.

Building some kinds of security into a system can also be done with wrappers. For example, data sent over a public network can be kept confidential by encrypting in the sending wrapper and decrypting in the receiving wrapper. Key distribution and authentication protocols can also be implemented in the wrapper.

Security wrappers of the kind just described are often implemented in standalone hardware. The Network Encryption System (NES), sold commercially by Motorola, is a recent example[Fra94]. Implementing the wrapper in hardware avoids the threat of circumventing security by tampering with underlying operating system software.

Security wrappers can be implemented in software if the operating system support for that software is also trusted. For example, current plans for the Next Generation Internet Protocol (IPv6) allow software to use the network layer protocol to carry application specific security and authentication data possibly generated in software[BM95]. Another example is Netscape's Secure Sockets Layer[Net96]. This protocol wraps whatever lies above the transport layer with authentication and encryption in a transparent way.

A more ambitious example can be found in the THETA secure distributed operating system[ORA92]. THETA is designed to support distributed multilevel secure services. These services are typically created by embedding single-level code in a multilevel wrapper. Security in THETA depends on running the software wrappers in a trusted operating system such as Synergy[S$^+$93], Trusted Solaris, or HP-BLS. THETA wrappers can also be configured to tolerate network partition failures.

Fault tolerance and security are not independent. In particular, the fault tolerance of a distributed system that uses replica coordination must depend on authentication. To see this, suppose that a component replica cannot tell the source of any message it

receives. Then a malicious attacker could masquerade as any of the replica's peers and could thwart any protocol for coordinating the replicas. Thwarting their coordination would destroy their ability to act as a single fault tolerant component.

In many fault tolerant systems of practical interest it is not necessary to protect against malice, and the source of messages can be known with high confidence. In these systems fault tolerance need not involve security. But when malicious attacks are possible a distributed authentication protocol such as Kerberos[NT94] must be used.

To build a system that can survive in the presence of malicious failures is a difficult problem, but one that has received much attention. Protocols exist to tolerate *Byzantine* failures (i.e., arbitrary and possibly malicious component behavior). These include protocols for reaching consensus among a group of component replicas, some of which may have failed[BMD93], and protocols for masking failures with voting and efficiently transmitting the voted result from one component to another[Ech86].

1.3.2   Approach: Adaptable Wrappers

A wrapper that is *adaptable* can be changed to fit well into its environment. Adaptability has two parts:

1. A wrapper is *configured* at compile-time to integrate it into a system and to optimize its performance. Configuration will typically be done by human designers, offline.

2. A wrapper is *reconfigured* at runtime to support evolutionary computing. Reconfiguration will typically be automatic, an interaction between running wrappers in which code is exchanged.

The approach taken by this project addresses both configuration and reconfiguration in a uniform way: both were treated as instances of protocol addition. A wrapper is configured with an initial set of protocols for interacting with its environment and can reconfigure itself by learning new protocols from other wrappers.

An essential part of the project has been to design a wrapper framework that accommodates protocol addition and replacement. A wrapper framework is a chassis with slots into which protocol peers can be plugged. A *protocol peer* is an instance of an executing protocol. The protocol peer contains protocol code for executing one *role* in the protocol, e.g., in a client-server protocol the client and server roles may execute different code. The protocol peer also contains data encoding the state of the protocol peer during execution.

9
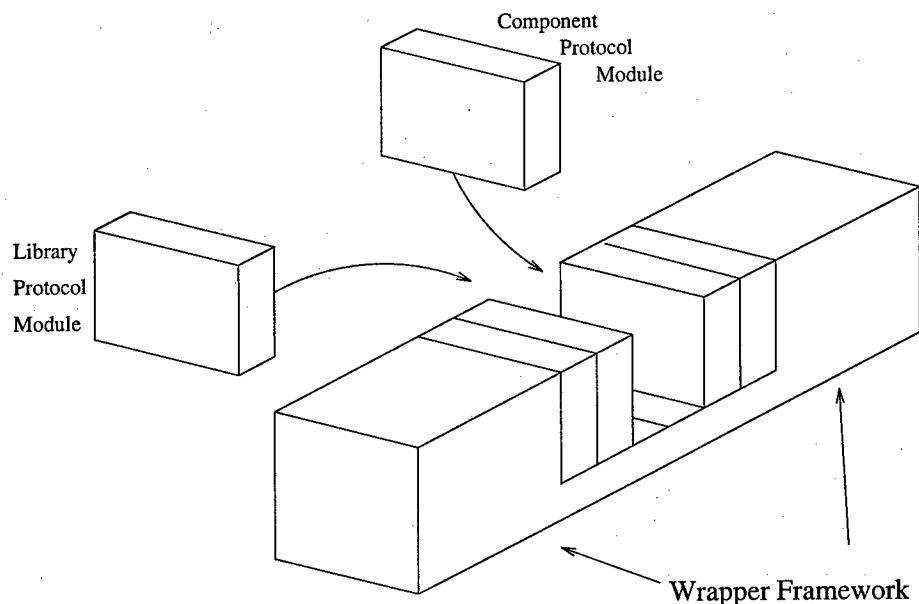
Figure 1.1: inserting protocols into the wrapper framework, both at compile-time from a library and at runtime from another component

Figure 1.1 shows the wrapper framework conceptually. Protocol peers can be inserted into slots either at compile-time or at runtime, as shown in the figure.

The framework design addresses several key problems. First, the framework must offer a way to connect a component to its environment via protocols that are run within the wrapper. The form of these connections will depend on various factors, including details of the component interface and of the local operating system. Second, the framework must allow the possibility that different protocol peers will need to interact. The most common interaction between peers is sharing data that is global within the wrapper. Such sharing is not only efficient, it supports continuous operation when protocols are upgraded. Third, the framework must protect protocol peers and the shared data because some peers may be untrustworthy. Fourth, the framework must allow the shared data to be extended as new protocols are added. These issues are discussed at length in this report.

1.4 Report Outline

The rest of this report is structured as follows:

- Chapter 2 discusses the wrapper framework, including its requirements and high-level design. The discussion of requirements shows how the wrapper is expected to adapt, and what this implies for the design. The key design issues are the protection and extension of wrappers, and both issues are analyzed.

  A key part of the design is a mechanism, called *dynamic extension*, which allows data structures to be modified while they are being used, and to maintain protection during and after modification. Dynamic extension allows adaptable, dependable wrappers to be built.

- Chapter 3 lists the applications we made of the wrapper framework during this project. Most of the applications are protocols for use in the wrapper framework. One application was a wrapper for an off-the-shelf expert system shell. Some of the applications make use of dynamic extension. All use the protection facilities of the wrapper framework.

- Chapter 4 concludes with a summary of the accomplishments of this project and some comments about ways in which the work could have been extended.

Chapter 2

Wrapper Framework

A *wrapper* is a layer of design that changes the interface to, or the properties of, a part of a system. The wrapper forms a boundary for that system part. We will refer to the part of a system that is wrapped as a *component* and the rest of the system as the component's *environment*.

A design may include a wrapper for any of several reasons:

- to translate data passed between the component and its environment from one form to another;

- to change the syntax of the calls recognized by a component or to modify a communication protocol used by the component;

- to extend the component's functionality;

- to add properties such as fault tolerance or security to the interactions of the component with its environment.

These reasons often arise when a component must be embedded into a larger system.

Although a wrapper is a boundary layer, it need not prohibit direct access to the component it wraps. For example, many wrappers have been written for the MS-DOS operating system[Mic91]. These wrappers typically provide some new set of operating system calls with new syntax and semantics but still allow applications to use most or all of the original MS-DOS interface. One might argue that such extensions are not wrappers at all because they do not form a complete boundary for the component. We view such extensions, however, as a complete design layer that simply leaves some of the component's original interface unchanged; where the interface is unchanged the boundary is simply very thin.

For some wrappers it may be unclear which side of the wrapper boundary is the component and which the environment. These cases are rare. Usually the design of the component is largely fixed before the wrapper is built, whereas the environment is unbounded. In all ambiguous cases it is left to the system's designers to decide, if necessary, which part of the system is the component being wrapped.

A *wrapper framework*[1] is a simple wrapper that can be extended into more complicated wrappers and that contains general mechanisms needed by many such wrappers. Wrapper frameworks are important in a changing environment. When the environment changes, the wrapper framework allows modifications and extensions to the wrapper so that the wrapper can continue to function in the new environment.

Wrappers are usually, but not always, implemented in software. We will confine our attention in this report to software wrappers. Software is more easily modified than hardware and therefore a software-implemented wrapper framework will more easily support the goals of wrapper modification, extension, and evolutionary computing.

This chapter presents our design and rationale for a wrapper framework. Section 2.1 discusses the requirements for a wrapper framework. Section 2.2 presents the internal structure of a wrapper and the general mechanisms supported by a wrapper framework.

The wrapper framework must address several particular problems. First, it must protect itself from some or all threats in the environment. Section 2.3 discusses protection. Second, it must allow a wrapper to be modified and/or extended while it is in use. Section 2.4 discusses extensibility.

2.1   Requirements

We expect that a wrapper framework will have the following properties:

- It must allow multiple protocols to run concurrently. These protocols can include communication protocols, either point-to-point or multicast protocols, and other kinds of protocols such as those used for clock synchronization.

- It must allow adaptation by changing protocols while in use and it must protect itself from disruptive protocols. Both these goals can be met with a framework that is *reflective*[KdRB91], i.e., one in which the properties and behavior of protocols can be analyzed and modified at runtime.

---

[1]In this report we use the word "framework" with its loose English meaning of "skeletal structure" rather than with its more precise meaning in object-oriented design.

- It must provide a hardware-independent platform on which wrapper extensions can be downloaded and run. This property of hardware independence suggests that the wrapper framework should provide a virtual machine for running its extensions. A simple way to get this virtual machine is to implement at least the wrapper extensions, and maybe the wrapper framework itself, in a machine-independent programming language such as Java[AG96]. Java, in particular, defines a Java Virtual Machine (JVM)[LY97] that guarantees a program will yield the same results when run on different hardware architectures.

2.2 Design

The wrapper framework consists of three parts:

1. A set of protocol peers. Each protocol peer is an executing instance of one role in a protocol. For instance, a client-server protocol typically has a client role and a server role, and each time a client-server connection is established there will be a client peer in one wrapper and a server peer in another.

2. A set of data shared between protocols. This data may be accessed concurrently by one or more peers in the wrapper. In this report, data shared between protocols will be called *global* because it is global within the wrapper (even though it is also local to the wrapper in the context of the complete system of which the wrapper is part). Data is *local* when it is accessible by only one peer.

3. A set of adapters that connect communication protocol peers to a wrapped component. The adapters allow switching between communication protocols in a way that is transparent to the component.

These parts of the framework are discussed further in the next three sections. A diagram showing the different parts of the wrapper design appears in the final section.

2.2.1 Protocol Peers

The wrapper framework will support several different kinds of protocol and their peers. These include:

- Communication protocols. These protocols transmit data between wrapped components.

- Clock synchronization and other wrapper coordination protocols. These protocols work in the background, not interacting directly with the wrapped component but supporting the operation of other protocols.

- Metaprotocols. These protocols download and install other protocols in an adaptable wrapper.

The main purpose of a metaprotocol is to facilitate one wrapper learning a new protocol from another wrapper. A metaprotocol will take some or all of the following actions:

- triggering – a metaprotocol is begun, or triggered, when one wrapper, A, decides that the set of protocols it is using should be changed. Several conditions can trigger the metaprotocol:

  - Another wrapper, possibly acting on behalf of a human user, explicitly instructs A to download a new protocol. This instruction starts the metaprotocol.

  - A currently running protocol in wrapper A detects that quality-of-service requirements are not being met and that protocol notifies the metaprotocol. Quality-of-service requirements can include sufficient throughput, sufficient reliability, or sufficient redundancy among replicated peers participating in the protocol.

- negotiating – Wrapper A decides that another wrapper, B, either has a new protocol needed by A or needs a new protocol that A has. A and B then negotiate to come to agreement about the properties of the protocol to transmit.

- downloading – One wrapper transmits the protocol data to the other. The protocol data includes an executable description of the protocol, plus optionally a specification of the protocol's properties and a proof that the properties hold.

- verifying – Some properties of the new protocol are verified, possibly by a decision procedure, possibly by proof checking.

- translating – The receiving wrapper converts the protocol to a form that can be executed by the wrapper.

- installing – The wrapper establishes one or more protocol peers to run the protocol, giving each peer the necessary access rights to data structures within the wrapper.

After these steps are completed the receiving wrapper is capable of engaging in the new protocol when it is needed.

Different metaprotocols will carry out the steps differently. In particular, not all these steps are necessary in every metaprotocol. Some metaprotocols, for example, will do less verification than others.

Protocol Bootstrapping

A metaprotocol is itself a protocol. So metaprotocols can be downloaded at runtime just like other protocols. This fact means that different wrappers may use different metaprotocols and that a single wrapper may run more than one metaprotocol at the same time.

Protocol bootstrapping is the process by which a wrapper learns new metaprotocols. Initially the wrapper has only a trivial metaprotocol to get started. The trivial metaprotocol does little, if any, verification of protocols it downloads and imposes no restrictions on the access rights of the new protocols. The trivial metaprotocol can be used to download more sophisticated metaprotocols. These can download other metaprotocols in turn, if necessary.

The possibility of protocol bootstrapping means two things for the wrapper:

1. The metaprotocols used by a wrapper do not need to be decided in advance. This allows better metaprotocols, with better algorithms for negotiating and verifying, to be installed later.

2. The wrapper framework needs to give metaprotocol peers the access rights to modify some of the data affecting other protocol peers within the same wrapper. These access rights are needed for installing new protocols.

2.2.2   Shared Data

Most protocols have state. In the Transmission Control Protocol (TCP)[Tan89], for example, each protocol peer must maintain the status of connections to other TCP peers. This state is local to each protocol peer.

If a wrapper is engaged in several protocols simultaneously, the peers for these protocols are likely to need to share data. This shared data differs from the local state of each protocol peer and it is often global to all protocol peers in a wrapper. The shared data includes:

- Information about other hosts and other wrappers. For example, the wrapper should maintain information about which remote hosts have failed. If a peer for one protocol has determined that a host has failed then this information can be shared with every other protocol peer that interacts with the failed host.

- Interconnections between different protocol peers in the same wrapper. For example, protocols are commonly layered (e.g., TCP over IP), so a peer for a lower layer protocol would be connected to a peer for a higher layer so that data can be passed from one to the other. This information about interconnections between peers needs to be global because metaprotocols may need to change the connections when installing new protocols.

- Protocol code and properties. This data must be accessible to metaprotocols in order that new protocols can be installed correctly.

- A real time clock. This data structure is shared by all real time protocols.

- Access rights data telling which peers may access which shared wrapper data. This access rights data needs to be global because metaprotocols may need to change it.

2.2.3 Adapters

Adapters connect the wrapped component to the wrapper. They provide the interface that the component expects, so that even if the component is legacy software it can be wrapped and still continue to function normally.

Because an adapter provides the interface the component expects, adapters must be specialized for particular components. For example, if a component expects to communicate using Unix sockets then the adapter must provide a Unix socket interface. If a component expects to communicate using DOS interrupts then the adapter must provide a DOS interrupt interface. There is no universal adapter.

Inside the wrapper, however, the adapter provides standard services. It must respond to requests from metaprotocols to switch between communication channels. It must do this switching in a way that preserves the semantics of the communication channel expected by the component. For example, if the channel needs to be reliable and sequenced, then the adapter must ensure that no data in the channel is lost or reordered during a switch between protocols.

Figure 2.1 shows the structure of a wrapper. The component being wrapped is shown in the upper left. In this case the wrapper needs to provide only a single communication channel for the component. The adapter allows the wrapper to switch from
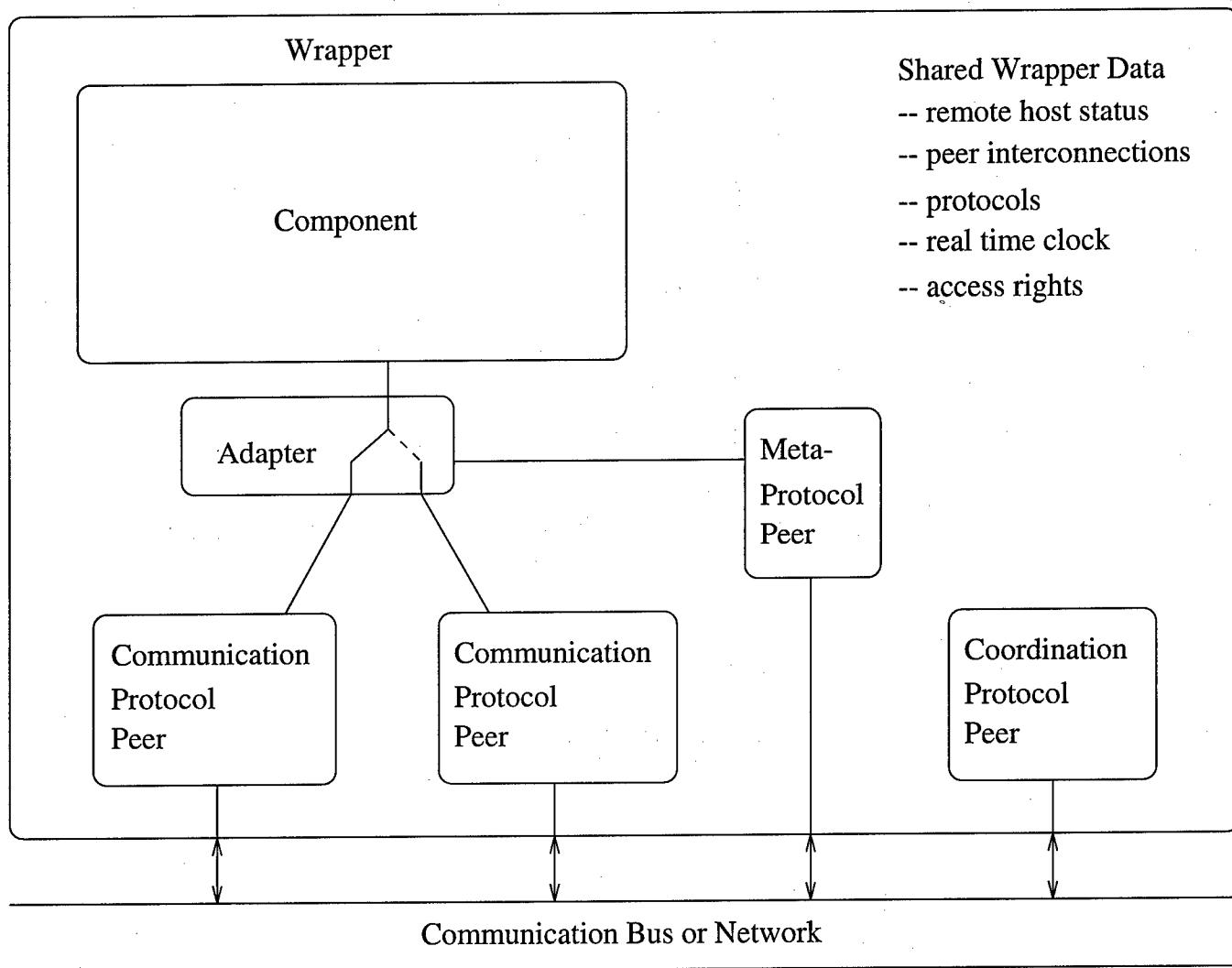
Figure 2.1: Structure of a wrapper

using the communication protocol on the left to using the communication protocol on the right instead. The switching is under the control of a metaprotocol, shown in the center. Also, a single coordination protocol (e.g., for clock synchronization among many wrappers) is shown on the right. Data that is shared between many protocols is shown in the upper right.

2.3 Wrapper Protection Mechanisms

A system that can download new code while running exposes itself to possible damage from that code. Downloaded code can cause some or all of the following kinds of damage:

- preventing the system from completing its tasks;

- making unexpected use of the system's functions;

- disclosing the system's data or code;

- corrupting the system's data or code.

Any of these kinds of damage could cause a system to behave incorrectly.

Downloaded code may cause damage either accidentally or maliciously. Accidental damage happens when code fails to perform as designed, because of programming mistakes. Malicious damage happens when code is designed for sabotage. A system's protection mechanisms should counter, if possible, both the threat of accident and the threat of malice.

The wrappers described in this chapter can download new code and must prevent damage from that code if they are expected to run continuously and autonomously for long periods of time. If possible, all damage should be prevented. If that is not possible, damage should be confined in some way by the wrapper protection mechanisms.

This section describes alternative approaches to wrapper protection. First, we describe the usual approaches to protection and the unusual protection needs of systems that download code. Then we classify the alternative protection mechanisms that satisfy those needs. Finally, we discuss these protection mechanisms as they would be implemented in Java.

2.3.1 Background

The threat to a system from downloading untrusted code is essentially the same as the threat faced by multiprogram operating systems that run untrusted programs: different programs running concurrently may interfere, and untrustworthy programs may interfere destructively. Therefore techniques used for protection in operating systems may also be useful for systems that download untrusted code.

Operating systems have countered the threat of program interference in a variety of ways. The most common way is to construct a *reference monitor* that prevents unauthorized interference between programs[And72]. The reference monitor allows only limited forms of program interaction and prohibits the rest. For example, two programs may be authorized to share a file but be prohibited from overwriting each other's code. A reference monitor makes its protection *uncircumventable* by ensuring that no set of authorized actions can enable unauthorized actions. For example, reference monitors usually prohibit themselves from being modified.

Reference monitors use two different kinds of protection mechanism.

1. With **access control**, a reference monitor rejects unauthorized requests to use resources. For example, if a program requests to open a file the request may be denied if it has not been given permission to access the file.

2. With **virtualization**, a reference monitor makes unauthorized resources invisible, i.e., a program may be unable even to supply a name for the resource. For example, virtual memory allows a program to access any memory it can supply a name for (i.e., the virtual memory address), but other memory may be unnameable (i.e., physical memory addresses that are outside the virtual address space).

These two kinds of protection mechanism are often used in complementary ways. For example, virtual memory can be used simultaneously to make another program's code invisible while permitting selective sharing of that program's data. The protection of the code depends on virtualization while the protection of the data depends on access control.

Historically, most reference monitors have used hardware mechanisms for protection. To implement access control, hardware mechanisms such as traps and exceptions are used to return control to the reference monitor when access control decisions are needed. To implement virtualization, hardware address translation is used to implement virtual memory and hardware support for threads is used to create a virtual processor for multiple threads running on a single physical processor.

Hardware-Independent Protection

Like an operating system, a system that downloads code can also implement a reference monitor using hardware mechanisms. Unlike most traditional operating systems, however, there are two reasons not to use hardware mechanisms for protection:

1. Hardware protection mechanisms are, of course, hardware-dependent and therefore some access restrictions on downloaded code will be nonportable. For example, one hardware architecture may support read-only access while another may support all-or-nothing access to memory segments. For another example, access control that depends on the segmented virtual memory of one processor may not be easily translated to another architecture that uses paged virtual memory. Because downloaded code cannot (normally) install new hardware at runtime to get the protection it needs, a protection mechanism that did not depend on hardware at all would be preferable.

2. Hardware protection mechanisms, while very efficient in controlling access, can have performance costs. These performance costs are paid during the management of hardware access control, i.e., checking of access rights, initializing the hardware protection mechanisms, and switching between contexts that have different access rights, rather than during the actual accessing of the protected objects. The management cost rises as the number of objects to be protected gets larger and the largest costs are paid for access control to very fine-grained objects. While this overhead cost for access control is not unique to hardware mechanisms, other kinds of protection that reduce this cost are preferable.

A designer can overcome the lack of portable hardware protection mechanisms in one of two ways:

1. The hardware mechanisms can be wrapped with a software layer to provide a standard interface. This approach is used by portable operating systems such as Unix[HS87]. Unfortunately this approach will not work on hardware architectures that have no protection mechanisms or mechanisms that are too limited. Such architectures are increasingly rare in general-purpose computers but can be found more commonly in embedded systems hardware.

2. No hardware mechanisms will be needed if the system runs only software that can be trusted not to perform unauthorized operations.

In this section we are particularly interested in designs that use no hardware protection and in which all software is trusted not to perform unauthorized operations.

This solution can be made portable if software is written in a machine-independent language and if programs written in that language can be analyzed at compile-time or download-time to determine whether unauthorized operations might be performed. This solution is also efficient because the costs of protection are paid at compile-time or download-time rather than at runtime[2]. This solution is also timely because it is the approach to security used for Java code that is downloaded over the Internet[Sun95].

Historically, protection without hardware enforcement has been considered impractical and this objection is still raised today for Java-based systems. The reasoning behind this objection is as follows. If hardware protection mechanisms are not used, then all code that runs in a system must be trusted to make only authorized use of resources. For practical purposes, and certainly for code downloaded at runtime, this implies that the code's trustworthiness must be verified without human intervention. This automatic verification of code poses two practical problems:

1. The property to be verified is typically undecidable. In other words, no automatic procedure can correctly determine whether the code is trustworthy or untrustworthy in every case.

   This problem can sometimes be handled by erring conservatively: the automatic verifier correctly identifies all untrustworthy code but may incorrectly reject some trustworthy code. The practical problem then becomes whether the verifier's rejection rate is too high.

2. It is difficult to show that the verifier is uncircumventable. For example, protection in the Burroughs 6700 depended on trusted compilers that accepted only input in a high-order language and emitted only code verified not to make unauthorized accesses. One published study showed the ease with which this protection scheme could be circumvented[W+81].

Java and some other modern programming languages base protection on type-safety. The type-safety property limits how one part of a program can interact with another and therefore is a kind of protection. Automatically verifying type-safety has been shown to be practical and useful for detecting errors in ordinary programming.

We will answer the following questions in later sections:

- can type-safety form the basis for more specialized kinds of protection needed in complicated systems?

- can type-safety protect against malicious programming?

---

[2]Compile-time analysis can have runtime costs if the software must be organized in an inefficient way to permit the analysis. We are assuming that these costs are insignificant.

- can Java be used for protection within a wrapper?

2.3.2 Protection in Typesafe Object-Oriented Systems

This section summarizes the concepts underlying object-oriented systems, protection, and type-safety. It relates some of these concepts. It introduces terminology that will be used throughout later sections.

Objects

An *object-based* system is organized as a collection of objects, each object of which is defined by the set of operations that it implements[Boo94]. An object, A, in an object-based system may *invoke* an operation implemented by another object, B, in which case A is called a *client* of B. Objects A and B may be different or identical, i.e., A is potentially the same object as B, in which case it can invoke one of its own operations and be its own client. Very complicated software architectures can be organized as object-based systems and many have been.

Each operation takes a set of parameters as input and returns a set of values as output. Some or all of the parameters and return values may be objects.

Each object, parameter, and return value in an object-based system belongs to one or more types. Then an operation can be partly described by its *signature*, which is the name of the operation along with the types of the parameters it expects as input and the types of the values it is expected to return. Two objects of the same type implement operations with the same signatures.

The set of signatures of a type is called the type's *interface*. If type A includes every signature of type B (and possibly others) then A is called a *subtype* of B.

A *class* is a particular implementation of a type. A type may have more than one class that implements it. Many different objects can be *instances* of a class, in which case they share the same implementation.

An *object-oriented* system is an object-based system with *inheritance*. One class, A, can inherit the implementation of a type from another class, B, and possibly extend that implementation. In this case A is called a *subclass* of B.

An object-oriented language is used to describe object-oriented systems. Java, for example, is a full object-oriented programming language offering classes, inheritance, and type interfaces[AG96].

23

Type-Safety

An invocation of operation O is called *typesafe* if the parameters supplied by the client and the return values supplied by the object each are of the types specified in O's signature. A system is called typesafe if every operation that can be invoked in any run of the system is typesafe.

The type-safety of a system can be deduced from a description of that system in an object-oriented language if the description contains enough information about the types of individual objects. If type-safety can be verified in this way, checking for type-safety of individual operation invocations can be avoided at runtime.

Security Policies

A *security policy* is a statement of a system's protection goals. The security policy typically tells what is authorized in the system and what isn't. *Subjects* in the policy are active agents that can perform *operations* on *objects*. The policy defines the *access rights*, i.e., which subjects are authorized to carry out which operations on which objects.

How can a security policy, expressed in terms of subjects, objects, and operations, be applied to a system described in an object-oriented language? It is obvious and natural to make the following identifications:

- The objects protected by the security policy are the objects described by the language.

- The operations governed by the security policy are the operations on objects described by the language.

What are the subjects of the policy? The most natural answer is:

- The subjects given access rights by the security policy are a subset of the objects described by the language. The subjects are viewed as independent agents. For example, the agents may run concurrently as separate threads of computation but this is not necessary. These agents access resources during their computation, and only some of the potential accesses are authorized.

This answer most resembles access control in operating systems, in which access rights are assigned to threads.

There are two common alternatives for implementing protection in operating systems: capabilities and access control lists (ACLs). Each alternative has been implemented using hardware support. The following sections will show that each alternative can instead be implemented without hardware support, in a typesafe object-oriented system that depends only on compile-time verification for protection.

2.3.3 Capabilities

A *capability* is a reference to an object that also carries access rights for that object. The reference will be expressed in some language, e.g., an object-oriented language such as Java. The holder of a capability has the right to access the object referred to by the capability. Each subject in a capability-based system possesses a set of capabilities that determine the subject's authority to invoke operations. The reference monitor in a capability-based system ensures that a subject may perform an operation on an object only if it has a capability for that operation and object.

Capabilities are a virtualization mechanism (in the sense of section 2.3.1). In a capability-based system, not only is every capability an object reference, but the only object references are capabilities, meaning that if a subject does not possess a capability for an object then it cannot even *refer* to that object properly. In other words, the capabilities form a virtual space of names for objects. Without a proper name, no access is possible.

Capability-based systems have been implemented using hardware mechanisms. A recent example is the IBM AS/400 architecture[Sol96]. In that architecture, processes can possess unforgeable hardware-enforced capabilities for a variety of system resources.

Capabilities can also be implemented without hardware support. For example, a reference to an object in a typesafe object-oriented system is a capability. To see this, consider that a subject normally gets a reference to an object in one of these ways:

- by creating a new object and a new reference to it;

- by accessing global data;

- by being passed the reference as a parameter of an operation;

- by being returned the reference as a value of an operation.

Each of these ways can be thought of as an authorized means for transferring access rights along with the object reference. The reference is a capability because it can-

not be gotten in any unauthorized way, i.e., other than the ones listed above. The type-safety property prevents new references from being generated in other ways, for example

- by converting a random integer into a reference;

- by "incrementing" a reference to point to other memory, as though it were an integer.

Typesafe operations that have these effects are not (normally) supplied in an object-oriented language. Therefore, a subject can get only the capabilities for which it is authorized.

The problem with this simple reference-as-capability scheme is that it offers too little control over transferring access rights. When a subject gets an object reference, it gets access rights to all that object's operations, including ones that return other capabilities. Better would be a mechanism for restricting the access rights that go with a reference.

Typesafe object-oriented languages can be used to limit the access rights that a subject has to a referenced object. The trick is to use language features to encode the access rights. Here are two basic schemes for limiting the access rights subject S has to the operations of object O, which is an instance of class C:

1. **Capability Interfaces:** In this scheme the access rights are encoded as an interface to a type in an object-oriented language. Suppose that interface I includes some subset of the signatures implemented in class C. Then C implements the type of I. If a subject has a reference to an instance of I, say object O, then it can invoke only those operations in I, even though O implements all of the operations in C. In this scheme the interface serves as a capability because access to object O is limited.

   Figure 2.2 shows two clients invoking operations on an object through two different capability interfaces. One client holds a capability that allows access to three of the four methods. The other client holds a capability that allows access to only two of the four.

2. **Capability Objects:** In this scheme the access rights are encoded as an object, separate from the object O. Suppose that object Q defines some subset of the signatures implemented in class C. Suppose Q implements each of its operations by invoking the corresponding operation on an instance Q' of class C. Then, just as in the Capability Interfaces scheme, a subject that has a reference to Q can invoke only those operations defined by Q and cannot invoke any other

26
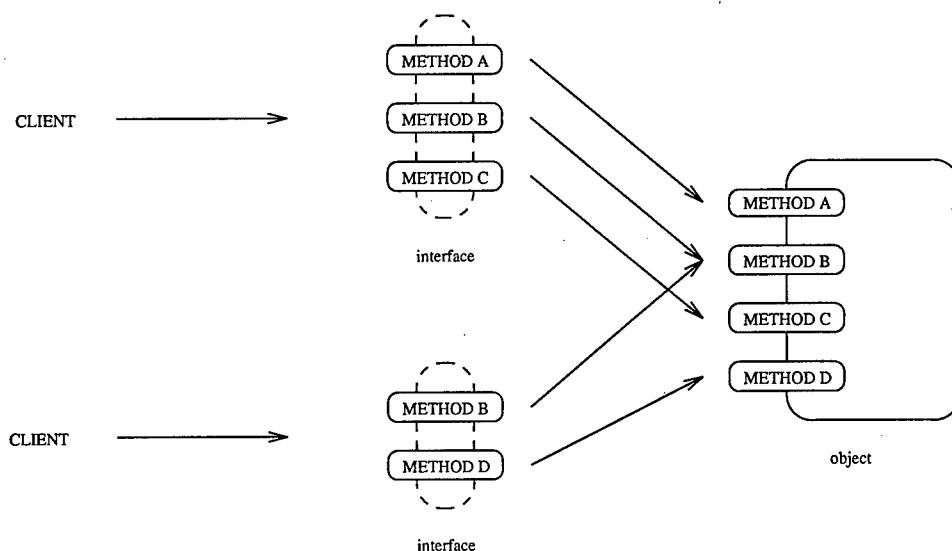
Figure 2.2: Capability Interfaces

operations implemented by Q'. In this scheme the object Q serves as a capability for Q'.

Figure 2.3 shows two clients invoking operations on an object through two different capability objects. Each client holds a different capability. When an operation is invoked on a capability object the invocation is passed through to be invoked on the actual object.

Each of these two schemes for implementing capabilities in a typesafe language has advantages over the other.

- The Capability Interfaces scheme is very efficient in both space and time. The structures used for protection, programming language interfaces, are needed only at compile time. In principle, these structures need not take *any* space or time in a running system, In practice, depending on how the language is implemented, they may take a small amount of space and time but these amounts should be negligible. Therefore the Capability Interfaces scheme is an ideal way to get protection with little or no runtime cost.

- The Capability Objects scheme is very flexible. Because the structures used for protection are programming language objects, they can be programmed to do useful tasks whenever a client invokes an operation. For example, a capability object can be programmed to allow the first n invocations, then deny the rest. This example is a kind of self-destruct mechanism for capabilities.
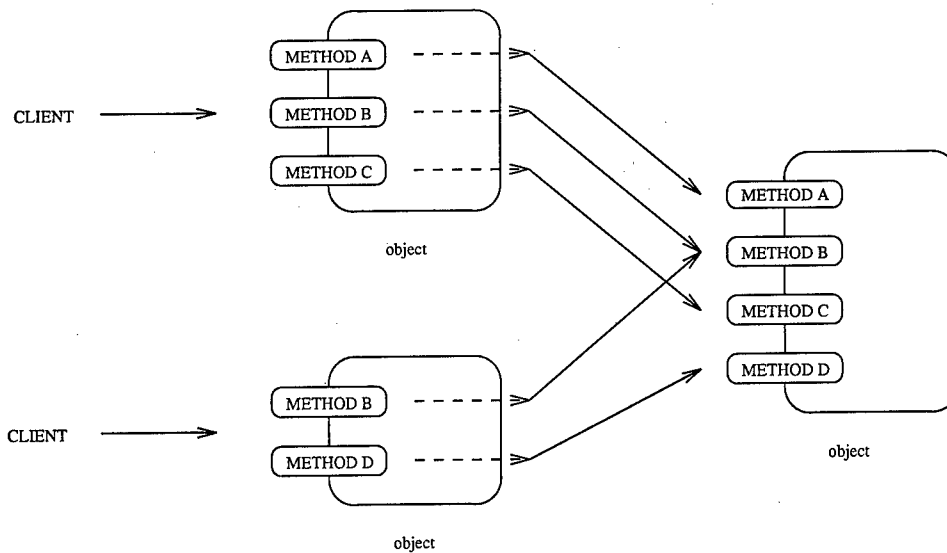
Figure 2.3: Capability Objects

This tradeoff between efficiency and flexibility is the essential difference between the two capability schemes.

The next two sections discuss how the Capability Interfaces and Capability Objects schemes can be implemented in Java. It will become clear that, in Java, the Capability Interfaces scheme has an additional drawback relative to the Capability Objects scheme: it is circumventable unless a more conservative model of type-safety than Java's is enforced. Details of the Java language will be introduced as they are needed in the next two sections[3].

Capability Interfaces Using Java

Suppose that some service, internal to the wrapper, needs protection. For example, the wrapper will offer a host information service. This service keeps information about remote hosts and shares this information among protocol peers within the wrapper. Some peers will be trusted to update the host information but other peers will not be so trusted. To ensure that the host information service provides trustworthy information to all protocol peers the service must be protected from modification by the untrusted peers.

Suppose that the host information service is implemented by a Java class:

```
class HostInfoService
```

---

[3]Some of the Java details may be specific to Java version 1.0.2.

```
{
    boolean isAlive (HostAddress addr) { ... }
    void recordLife (HostAddress addr, boolean alive) { ... }
}
```

HostInfoService is the name of this Java class. The class has two methods, isAlive and recordLife. A *method* is the Java term for an object-oriented operation. The first method takes a HostAddress as a parameter and returns a boolean value. The value returned by isAlive tells the state, alive or dead, most recently detected for the host at address addr. The second method takes HostAddress and boolean parameters and returns no value (i.e., void). This recordLife method is used by trusted protocol peers to set the state of the remote host known to this host information service. For example, after a peer detects that some host at HostAddress instance h has failed it would make the following invocation on some instance serv of class HostInfoService:

```
    serv.recordLife (h, false);
```

In Java, the object of the operation is written first, then the method being invoked, then the parameters of the method.

This Java description of the host information service is oversimplified for the purpose of exposition. The actual host information service in the wrapper framework will keep more information about each remote host than the single boolean value shown here. In fact, the host information service may need to be extended at runtime to allow new protocols to be downloaded, and this possibility will be discussed in section 2.4.

To protect the host information service, the wrapper framework will create a single, private, instance of the HostInfoService class. The wrapper framework will itself be a unique instance of a class, and the declaration of that class will look something like this:

```
class WrapperFramework
{
    ...
    private HostInfoService theHostInfoService =
        new HostInfoService ();
    ...
}
```

The Java code shown declares a single variable, theHostInfoService, which is a newly created instance of the class HostInfoService declared previously. The Java

keyword `private` means that the variable `theHostInfoService` cannot be named by other classes. In particular, it cannot be named within classes that implement protocols and therefore is not accessible by those protocols unless a reference to the-HostInfoService is supplied by the wrapper framework. This private variable cannot be named even if code in some other class holds a reference to the WrapperFramework. This constraint on the visibility of variables in Java is a means of protection enforced by the language above and beyond the protection offered by type-safety.

To make the host information service available to protocols the wrapper framework will supply a capability for the service rather than the service itself. For example, the wrapper framework may choose to give an untrusted protocol a read-only capability for the service. In the Capability Interfaces scheme, read-only access rights would be defined by an interface that declares only the `isAlive` method:

```
interface ReadOnlyCapability
{
    boolean isAlive (HostAddress addr);
}
```

And in this case the service `implements` the interface:

```
class HostInfoService implements ReadOnlyCapability
{
    public boolean isAlive (HostAddress addr) { ... }
    public void recordLife (HostAddress addr, boolean alive) { ... }
}
```

The Java keyword `public` indicates here that the methods are visible in all classes. Methods must be `public` if they implement signatures in a Java interface.

A protocol peer running in the wrapper may be an instance of a protocol class such as this one:

```
class UntrustworthyProtocol
{
    ...
    void initialize (ReadOnlyCapability capa, ...) { ... }
    ...
}
```

This protocol has a method, `initialize`, that is given a capability by the wrapper framework. The capability gives the protocol read-only access to the host information service and no other rights.

Is this capability-based protection circumventable? Unfortunately, in pure Java, it is. Java allows type-conversion operations, called casts, that can be used to coerce one type into another, and using a cast an interface can be coerced into an object having the type it protects. For example, in the UntrustworthyProtocol,

```
void initialize (ReadOnlyCapability capa, ...)
{
    HostInfoService serv = (HostInfoService) capa;
    serv.recordLife (...);
}
```

This code shows the initialize method casting the capability interface into a direct reference to the protected service, then invoking an operation for which it is not authorized. Java checks at runtime that such casts do not violate type-safety. In this case, because the capa parameter to the method really refers to a HostInfoService object, the cast will preserve type-safety. But it is not secure.

The Capability Interface scheme can be made noncircumventable by augmenting Java's type-safety verification with additional checks. Suppose that the goal is to protect some set of Java classes using the Capability Interface scheme. We will refer to these classes and instances of them as *protected*[4]. One must check that

- no untrusted program may cast an object of a Capability Interface type into an object of a protected class;

- no untrusted program may cast an object of a Capability Interface type into an object of a subtype interface.

The first of these checks prevents direct access to the underlying service being protected. The second of these checks prevents a program from increasing the set of access rights it holds by converting an instance from one interface into another interface that allows more methods to be invoked.

These additional checks can be implemented in an automatic verifier. This verifier will be called the *Capability Verifier* in this report. The Capability Verifier augments the Java Verifier that is part of the standard Java environment. Together, these two verifiers make the Capability Interfaces scheme uncircumventable.

---

[4]Not to be confused with the Java keyword of the same name, which always appears in boldface in this report.

31

Capability Objects Using Java

The host information service can also be protected with Capability Objects instead of Capability Interfaces. In this alternate scheme, one creates a class of capabilities:

```
class ReadOnlyCapability
{
    private HostInfoService reference;

    ReadOnlyCapability (HostInfoService ref) { reference = ref; }

    boolean isAlive (HostAddress addr)
    {
        return reference.isAlive (addr);
    }
}
```

Each object of this class contains a reference to another object of the protected HostInfoService class. The reference is private and so is inaccessible from other classes. The reference can only be set once, when the capability is created, by the constructor method ReadOnlyCapability. Every invocation of the isAlive method is then delegated (i.e., forwarded) to the method of the same name in the protected object.

Note that the Capability Objects scheme implemented in Java differs from the Capability Interfaces scheme in these ways:

- One Java capability object must be created per object to be protected. These capability objects take space, and delegating method invocations takes time. Typically the space and time taken will be greater than in the Capability Interfaces scheme.

- Java type-safety is uncircumventable protection. Untrustworthy code that tries to coerce an object of the ReadOnlyCapability class into an object of the HostInfoService class will cause a runtime error. No Capability Verifier is needed to enforce this protection, unlike in the Capability Interfaces scheme.

Because each Capability Object is an instance of the generic Java class called Object, each Capability Object will inherit methods defined in the Object class. These methods may allow, for example, an object to be cloned or a string representation of a hash code for the object to be returned. These methods do not seem to pose a security

32

risk in the Capability Objects scheme but for extra safety they can be overridden in any Capability Object class by new methods that do nothing.

Metaobject Capabilities

Neither the Capability Interfaces nor Capability Objects schemes explicitly protects a system's metaobjects. This section discusses how those schemes can be extended to gain this protection.

A *metaobject* is an object that represents an entire class of objects. For example, the HostAddress class used in the previous sections contains a set of instances, each of which is an object used as an address. If an operation were to affect every address in that set then that operation could be thought of as an invocation on a unique object, the metaobject, representing the class of all addresses. If, say, the metaobject stores a list of all addresses in the class then the operation of sorting the addresses would be an invocation on the metaobject.

Creation of new objects is the one metaobject operation that every system must have. Object creation cannot be an ordinary operation because the object it affects does not exist before the operation happens. Object creation is normally implemented by the metaobject for the class of the new object.

If the metaobject for, say, the HostAddress class is not protected then untrusted programs will be able to create new HostAddress objects at will, sort the host addresses, or invoke any other metaobject operation. In some cases it will not be a problem if untrusted programs can create their own instances of a protected class: then instances created by trusted clients will be handled in a trustworthy way by giving only capabilities to untrusted clients while instances created by untrusted programs might be handled in any way whatsoever. Whether this freedom is a problem depends on the particular application.

In general, metaobjects can be protected using the same techniques as ordinary objects. In other words, to protect the metaobject for a class, just use either the Capability Interface or Capability Object scheme. Direct references to the metaobject are prevented by type-safety. In order for a client to access a metaobject operation it must hold a capability for that operation.

In Java, though, neither the Capability Interface nor Capability Object scheme can be used immediately to protect metaobject operations, because Java provides no explicit metaobjects. Instead, Java provides special language constructs for metaobject operations of a class C:

- Most metaobject operations are written as methods of C, prefixed with the

keyword `static`. These `static` methods are available to every object of the class without restriction. They may also be available to other classes even if no object of class C exists.

- The metaobject operation that creates a new object of class C is written with another special construct, called a *constructor*, and is also part of class C.

The Capability Interface scheme cannot protect these metaoperations because Java interfaces are not permitted to describe either `static` methods or constructors. In Java, access to `static` methods and constructors is limited only by the language's restrictions on the visibility of names and not by type-safety. The Capability Object scheme cannot be used either because there is no explicit metaobject for the capability object to delegate operations to.

A crude way to protect `static` methods and constructors in Java is by using the language's `package` mechanism. This approach does not depend on type safety at all. For example:

```
package ProtectedStuff;

class ProtectedService {
    ...
    static void metaOperation ( ... ) { ... }
}

public class Capability {
    private ProtectedService reference;
    Capability () { reference = new ProtectedService (); }
    ...
}

public class MetaCapability {
    MetaCapability () {}

    public void metaOp ( ... ) {
        ProtectedService.metaOperation ( ... ); }
    public Capability create () { return new Capability (); }
}
```

This Java code is all written within a `package` called `ProtectedStuff`, which means that code outside this package cannot name classes and methods in the package unless

they are declared `public`. In particular, neither a `ProtectedService` object nor the `metaOperation` can be named outside the package. However, both a `Capability` for the service and a `MetaCapability` for its metaobject operation can be named outside. So if code outside the package is given a `MetaCapability` instance then it can invoke the operation `metaOp`, which in turn invokes the protected metaobject operation. Such a `MetaCapability` instance also authorizes a client to invoke `create`, which creates a new instance of `ProtectedService` and returns a Capability Object for that instance.

Could code outside the package construct its own `MetaCapability` and thereby get unauthorized access to the protected service's meta-operations? This loophole has been closed in the example by giving the `MetaCapability` class an explicit constructor, also called `MetaCapability`, that cannot be named outside the package. Note that the `MetaCapability` constructor has *not* been declared `public`. If it had been declared `public` it would have been visible outside the package. If it had not been declared at all the Java language would have implicitly generated a constructor and that implicit constructor would have been `public` by default. As shown, only clients within the package can construct a new `MetaCapability`.

While this example shows that one can protect Java meta-operations, the protection mechanism has these drawbacks:

- The Capability Interface scheme cannot be used.

- The objects to be protected must be confined to a set of protected Java packages. This makes the Java `package` construct do double-duty, first for grouping of related classes and second as a protection mechanism. While these duties may be consistent in some cases, in general they won't be. This fact adds another constraint to the system's design and may be cumbersome.

- To prevent untrustworthy software from adding its own code to the `ProtectedStuff` package a Capability Verifier must be implemented that prevents this possibility.

A more flexible solution is to construct a Java object for each class to be protected. These Java objects serve as explicit metaobjects. Then protection is arranged by the following steps:

1. Every meta-operation to be protected is written as an ordinary method of the metaobject class rather than as a `static` method of the class being protected.

2. Either the Capability Interface or Capability Object scheme is used to protect the metaobject.

35

3. If creating new objects of the protected class is to be a protected operation on the metaobject the Capability Verifier must be enhanced to prevent this protection from being circumvented. In addition to the restrictions previously discussed in section 2.3.3 for the Capability Interface scheme these restrictions must also be enforced:

- No untrusted program may directly invoke the constructor for a protected class.

- No untrusted program may invoke the newInstance method of the Java class Class to make new instances of a protected class.

The first restriction can be checked automatically. The second restriction is undecidable in general because the Java Class given to the newInstance method is a variable whose value may not be decidable at compile-time. Stronger restrictions can be checked automatically, though, e.g., preventing any untrusted access to newInstance. Use of newInstance is rare and so stronger restrictions may not be too strong for practical application.

The previous example can be implemented without using Java packages or static methods as follows:

```
class ProtectedService { ... }

class Capability {
    private ProtectedService reference;
    Capability () { reference = new ProtectedService (); }
    ...
}

class MetaObject implements MetaCapability1, MetaCapability2 {
    public Capability create () { return new Capability (); ... }
    public void metaOperation ( ... ) { ... }
}

interface MetaCapability1 {
    Capability create ();
}
interface MetaCapability2 {
    void metaOperation ( ... );
}
```

Here the `static` method `metaOperation` has been removed from the `ProtectedService` class and placed entirely within the `MetaObject` class. To access the `metaOperation` a client must have a `MetaCapability2`. Both the `ProtectedService` object and the `MetaObject` are protected, the former using the Capability Object scheme, the latter using the Capability Interface scheme. Either scheme, however, could have been used at either the object or the metaobject level.

The Capability Verifier ensures that the meta-capabilities cannot be circumvented. It also ensures that untrusted programs cannot construct their own instances of any of the three classes. Thus the only way an untrusted program can create a new `ProtectedService` is by being given a `MetaCapability1`.

This example assumes that every `MetaObject` will be constructed by a trusted program. If it is necessary instead to give an untrusted program a limited access right to construct its own `MetaObject`, then a meta-meta-object could be implemented using the same techniques. Fortunately this situation is not common.

2.3.4 Access Control Lists

An *access control list* (ACL) is a list of access rights for an object. Each access right tells which subject can access the object using which operations. The reference monitor in a system using ACLs will, for each access to a protected object X, search the ACL for X looking for the access rights needed by the subject. If the rights are not found, the reference monitor will deny the access.

In one sense, ACLs are analogous to capabilities. Both ACLs and capabilities store the complete information about access rights. An ACL stores the access rights for a particular *object* and a capability stores the access rights for a particular *subject*. The collection of ACLs for every object in a system contains the same information as the collection of capabilities for every subject in the system but organizes that information in a different way.

In another sense, though, ACLs differ greatly from capabilities because the protection they offer is enforced differently. ACLs are an access control mechanism, while capabilities are a virtualization mechanism (see 2.3.1). The ACL protection is enforced by trusted code that checks access rights; the checking mechanism is typically centralized. Capabilities, on the other hand, are decentralized, and, so long as they are unforgeable, can be handled securely by untrusted code. One advantage this means for ACLs over capabilities is that a centralized reference monitor can much more easily control the propagation of access rights from one subject to another using ACLs; capabilities, while they cannot be forged, can be easily copied.

One can implement a reference monitor for ACLs using hardware mechanisms. To use a current example: the Intel 80386 chip[Int90] (and all its successors, including the Pentium chip) has several different operating modes corresponding to different levels of protection. Some 80386 instructions can only be successfully executed at the highest level of protection. Transitions between different protection levels are carefully controlled. These features make it possible to implement a reference monitor, running at the highest level of protection, that controls access by untrusted programs to the protected instructions and thereby controls direct access to shared resources such as memory and external devices. Such a reference monitor can grant or deny indirect access based on an ACL lookup.

One can also implement a reference monitor for ACLs without hardware mechanisms using a typesafe object-oriented language with the following steps:

1. Create a unique object to implement the reference monitor. This object can be made globally accessible.

2. Store all ACL data in the reference monitor object.

3. Store all references to other protected objects in the reference monitor. The reference monitor will be designed to keep these references private, i.e., it will not give them out to untrusted programs. Because untrusted programs in a typesafe language cannot generate references to existing objects (i.e., these references are capabilities, see section 2.3.3) these objects will remain accessible only to the reference monitor.

4. Provide a reference monitor interface that allows untrusted programs to access the operations of protected objects indirectly. This interface lets the reference monitor intercept all accesses to a protected object and to decide, based on the ACL for that object, whether the access should be allowed. If the access is to be allowed, the reference monitor itself invokes the operation on the protected object and returns the result to the untrusted client.

In order for the reference monitor to decide whether to allow a request for access it must know both the subject and object of that request. The only way to do this in general is to give the identity of both subject and object as part of the request to the reference monitor. These identities cannot be object references, for reasons given shortly, but rather must be unique identifiers that the reference monitor can associate with an object reference. Consider the object and subject cases in turn:

- The object of a request cannot be specified by its object reference because no untrusted subject has that reference – only the reference monitor does. There-

fore the object must be specified by a unique object ID, which the reference monitor can convert to an object reference.

- The subject making a request must identify itself to the reference monitor so that the request can be approved or rejected based on a security policy as described in section 2.3.2. Suppose that the subjects are themselves objects, as discussed in section 2.3.2, but are not necessarily protected objects. Then a subject S cannot supply a reference to itself as part of the request because that reference might be known to other subjects which could then assume the identity of S and use its access rights. Therefore the subject must register itself with the reference monitor to get a unique, private, identifier which it supplies with all future requests.

  Each subject must be trusted not to divulge its unique, private, unforgeable, identifier. This trust can be gotten using type-safety plus other syntactic checks. If the type of the subject identifiers never appears in any signature except as a parameter used by the reference monitor for authentication, then type-safety guarantees that no subject can pass its private identifier to or forge the identifier of another subject.

An example of this ACL scheme implemented in Java is shown in the next section.

ACLs Using Java

To demonstrate ACLs using Java, consider again the host information service discussed in section 2.3.3. Suppose this service is implemented by a set of objects, one per remote host. The declaration of the host information might look like this:

```
class HostInfo {
    private HostAddress addr;
    HostAddress getAddress () { return addr; }

    private boolean alive = false;
    boolean isAlive () { ... return alive; }
    void recordLife (boolean live) { ... alive = live; }
}
```

Instances of the HostInfo class record only the address of a host and whether that host has been determined to be alive. This data is private but also exported to all classes via accessor methods.

Each object of this `HostInfo` class will be given unique identifier from the following class:

```
class ObjectID { ... }
```

Each subject will identify itself to the reference monitor using its own unique, private instance of the following class:

```
class SubjectID { ... }
```

Then the reference monitor will offer a protected host information service to all clients via an implementation similar to the following:

```
class ReferenceMonitor
{
    private static HostInfo lookup (ObjectID id) { ... }

    private static boolean hasAccessRights (SubjectID subj,
                                            ObjectID obj) { ... }

    public static boolean isAlive (SubjectID subj, ObjectID obj)
    {
        if (hasAccessRights (subj, obj)
            return lookup (obj).isAlive ();
    }
    public static void recordLife (boolean life,
                                   SubjectID subj, ObjectID obj)
    {
        if (hasAccessRights (subj, obj)
            lookup (obj).recordLife (life);
    }
}
```

The first reference monitor method, `lookup`, allows the reference monitor to convert an `objectID` to a `HostInfo` object. The second reference monitor method, `hasAccessRights`, allows the reference monitor to check whether a particular request is authorized. These methods are private because only the reference monitor should be allowed to use them; if another subject could use `lookup`, in particular, it could bypass the reference monitor to invoke operations directly on objects.

The reference monitor then provides a public operation for every operation possible on a protected object, in this case the `isAlive` and `recordLife` methods (`getAddress` could have been handled similarly), but these public operations have two new parameters: the subject and object identifiers. The reference monitor checks access rights on each request and looks up the object reference from the object identifier, using private operations in each case. If the subject has access rights the reference monitor delegates the operation to the `HostInfo` object.

This sample code does not allow access rights to depend on which operation is being invoked but could be trivially generalized for this possibility.

Note that the reference monitor in this scheme is essentially a single Capability Object for all the protected objects in the system. In other words, all clients can refer to the reference monitor but none can refer to a protected object.

Note also how this scheme depends on type safety: the `SubjectID` supplied with a request is known only to the subject and the reference monitor and no other subject has typesafe operations that generate this `SubjectID`.

2.3.5 Summary

This section has shown several schemes for wrapper protection that depend on type safety and can be verified when code is compiled or loaded. These schemes do not depend at all on hardware protection mechanisms and are therefore platform-independent. Either access control list or capability schemes can be implemented, and issues specific to a Java implementation were discussed.

2.4 Wrapper Extension Mechanisms

A system that can download new code needs mechanisms for integrating that new code with the code that is already running. In general, a system can make arbitrary self-modifications when new code is downloaded. The freedom of arbitrary self-modification, however, is dangerous because it implies that no checking is done to ensure that the modifications make sense. Then even trivial mistakes in the new code can wreck a running system. Better would be a mechanism, or a set of mechanisms, for incorporating new code in a structured way.

One of the simplest ways to add new code safely is to run it in a new thread or process that cannot interact with already-running threads. Then the original system is protected from the new code. Unfortunately, this mechanism is too simple for

most practical applications because the reason for adding code is to fix or improve the original system and this requires that the new code interact with or replace the old. So more flexible mechanisms are needed.

The wrappers described in this chapter need more complicated extension mechanisms. Wrappers will be upgraded by loading, installing, and using new code for protocols. These protocols govern the wrapper's interaction with other wrappers and system components, and wrapper upgrades will typically happen while some of these protocols are already in progress. When a new protocol is installed it may interact with already-running protocols in two ways:

1. The new protocol may replace an earlier version of the same protocol.

2. The new protocol may share data with other protocols, in which case the shared data may need to be modified for the new protocol without disturbing the operation of the old protocols.

This section explores a particular extension mechanism, called *dynamic extension*. This mechanism is a natural means of extension in object-oriented systems. It is explained in sections 2.4.1. Specific features of dynamic extension are discussed in sections 2.4.2 and 2.4.3. Section 2.4.4 shows how dynamic extension interacts with the capability protection mechanisms of section 2.3.3. All examples used in this section will be written in Java.

Other extension mechanisms are possible. In fact, any program transformation that might be used during the evolution of a system's *design* could also be used during the evolution of the system itself. Future versions of this report may consider other extension mechanisms.

2.4.1 Dynamic Extension

Object-oriented systems suggest a natural approach to upgrading data structures that are shared between new downloaded code and older code. The approach relies on subclassing, inheritance, and dynamic binding. In this approach, the shared data structures are represented originally as instances of a parent class, C. When new code needs to access instances of class C but with new fields or methods added, then the system extends the parent class C into a subclass, S, containing those new fields and methods. During the installation of the new code, instances of class C are converted to instances of the subclass S. After the conversion, both old and new code can operate correctly: new code will use fields and methods for subclass S while old code

will inherit the fields and methods it needs from class C and will continue to function as before the conversion.

We call this approach to upgrading *dynamic extension* because it extends a system just as a programmer might at compile-time but instead extends it dynamically, at runtime. The crucial step in dynamic extension is the conversion of existing class instances to new instances of a subclass. This conversion step is not usually supported by object-oriented languages. It requires that the system replace every potentially sharable instance of the parent class C by an instance of subclass S and that every reference to these class C instances be modified to refer to the new subclass S instances. This conversion step may be costly but it can be implemented in three different ways.

1. The system can search all of memory for instances and replace them. This straightforward approach has several drawbacks, though. First, few systems are built to allow such searches. For example, in the executable form of a Java application it may be difficult or impossible to tell whether a particular bit pattern represents an object reference. Only reflective[KdRB91] languages such as the Common Lisp Object System (CLOS)[GLS90] allow the necessary kind of program introspection and self-modification[5]. Second, a search of the entire memory of a large application can take a lot of time. This search operation is similar to garbage collection in languages such as Lisp and Java, operations that can often contribute to poor performance.

2. The system may create every object to be extendable. This can be done by chaining: the original object holds a pointer to its first extension, which holds a pointer to its second extension, and so on. The main drawback of this approach is that following a lengthy chain of pointers is an expensive operation for referencing a single field in an object, and a lengthy chain would result from repeated extensions. A lesser drawback is that the extensions are not invisible to the original code. In other words, a program can inspect an object to find out whether it has been extended or not. This unnecessary visibility follows from the fact that this approach duplicates the extension capability provided by subclassing in object oriented languages.

3. The system may provide an extra layer of indirection for every object. This layer lets the system switch from an older version of an object to a newer version. In other words, every object reference is implemented by a pointer to a reference, and extension is implemented by switching pointers. This approach has the drawback that it takes a lot of space: one new pointer must be allocated for every extendable object.

---

[5]A recent addition to Java allows some reflection but it is not sufficient for this purpose.

43

Any of these three approaches can be used to extend a single object. Dynamic extension extends every object in a class. Both the second and third approaches take extra memory for dynamic extension because a list of references to every class instance must be stored in the class metaobject. The third approach will be of special interest later in section 2.4.4 because it works well with the Capability Objects scheme of section 2.3.3.

Dynamic Extension Using Java

To demonstrate dynamic extension using Java, consider again the host information service and host information type discussed in section 2.3.4.

```
class HostInfo {
    protected HostAddress addr;
    public HostAddress getAddress () { return addr; }

    protected boolean alive = false;
    public boolean isAlive () { ... return alive; }
    public void recordLife (boolean live) { ... alive = live; }
}
```

Instances of the HostInfo class record only the address of a host and whether that host has been determined to be alive. Unlike in section 2.3.4, this data is now protected which means that it can be directly accessed by subclasses and other classes within the same Java package, but the data is also exported to all classes via accessor methods.

Suppose a new protocol needs not only the data on host liveness but also a record of when the host was determined to be alive (perhaps a client of the host information service must make a judgement of whether the liveness data is stale). Then the new protocol would find it convenient to record host data in the following subclass:

```
class HostInfo_1 extends HostInfo {
    protected Time alive_detection_time;
    public Time getDetectionTime () { return alive_detection_time; }
}
```

In this subclass, all the fields and methods from HostInfo are reused. A new field, alive_detection_time, has been added and a new method, getDetectionTime, is included to access the new field.

44

Then each instance of `HostInfo` would be extended to an instance of `HostInfo_1` in the following steps:

1. create a new `HostInfo_1`

2. copy the data from the `HostInfo` to the `HostInfo_1`

3. change every reference to the `HostInfo` to refer to the `HostInfo_1`

4. destroy the old `HostInfo`

Dynamic extension is useful in this case because the new protocol does not need a new representation for the existing data in `HostInfo`. The existing data was simply augmented with new data. Dynamic extension would be cumbersome, but still useful, in cases where the existing data structures must be changed rather than augmented. In these cases some or all of the existing methods would need to be overridden. Dynamic extension would be least useful in cases where the signatures of operations need to be changed.

Dynamic extension in Java allows existing methods to be enhanced as well as new ones to be added. This option is implemented by overriding existing methods and relying on dynamic binding to use the new version of the method. For example, suppose that an existing client, written to use the `HostInfo` class, calls `recordLife (false)` when it detects that a remote host is alive. Even after dynamic extension from `HostInfo` to `HostInfo_1` this client will ignore the `alive_detection_time` field in class `HostInfo_1` because it wasn't written to use it. The `recordLife` method can be overridden, however, to record the time at which liveness was detected in addition to the fact of its detection:

```
class HostInfo_1 extends HostInfo {
  protected Time alive_detection_time;
  public Time getDetectionTime () { return alive_detection_time; }

  public void recordLife (boolean live) {
    alive_detection_time = Time.now ();
    super.recordLife (live);
  }
}
```

In this second version of the `HostInfo_1` subclass the `recordLife` method of class `HostInfo` has been overridden. The new method records the current time when

45

it was called, then invokes the method it overrides in the superclass, i.e., method `recordLife` of class `HostInfo`.

Before dynamic extension, a client invoking `recordLife` on an instance of class `HostInfo` will get the old version of the method. During dynamic extension, every `HostInfo` instance is replaced by a `HostInfo_1` instance. After dynamic extension, dynamic binding (a feature of Java and other object-oriented languages) will cause the new `recordLife` method of class `HostInfo_1` to be called whenever a client invokes `recordLife`. This new method takes new actions appropriate to the subclass but also invokes the old version of the method. In this way, the dynamically extended objects implement new functionality but do so in a way that still satisfies invocations by clients that were not written to use the new functionality.

2.4.2 Conservative Dynamic Extension

Dynamic extension is *conservative* if it affects no existing client. In other words, even though newly downloaded clients may dynamically extend data structures used by existing clients, the latter will continue as though no extension had happened.

Not every extension is conservative. In particular, an extension that replaces existing methods by overriding them will not be conservative if the functionality used by existing clients is changed.

A dynamic extension can be verified in some cases automatically to be conservative. This question is in general undecidable but checking the following conditions is sufficient to prove that an extension is conservative (assuming that changes to the timing and performance of methods are negligible):

- no code in the extension modifies any field in the superclass;

- every method override invokes the overridden method in the superclass;

- all extension code will terminate normally, i.e., no exceptions will be raised and it must terminate.

The dynamic extension shown in Java in section 2.4.1 satisfies the above conditions and is therefore conservative.

2.4.3 Repeated Dynamic Extension

Over time, the instances of an original class C might be repeatedly dynamically extended to subclasses, S_1, S_2, ... These subclasses form a sequence, i.e., S_1 is a

subclass of C, S_2 is a subclass of S_1, and so on.

Independent attempts to dynamically extend class C, however, need to be coerced into a sequence of extensions. For example, suppose new code written by one author is used to extend from C to S_1. A second author, unaware of the extension to subclass S_1, writes new code to extend from C to subclass E. Class E is clearly not an extension of S_1 but needs to be made into one.

Repeated extension is, in effect, a form of multiple inheritance. In the example just used, what is needed is a new class, call it S_2, that inherits from both S_1 and E. A wrapper that implements dynamic extension must construct such a class S_2. In the examples that follow, subclass S_1 will be called the *first subclass* and subclass E will be called the *second subclass*. The goal is to construct a subsubclass S_2 from S_1 and E.

Repeated Extension Using Java

Java does not implement multiple inheritance[6]. Therefore to coerce repeated, independent, extensions into a sequence of extensions a mechanism outside the Java language must be used. The mechanism is a meta-level operation that takes Java code as input and produces new Java code as output.

To use the host information example again, suppose `HostInfo` is dynamically extended as before, making `HostInfo_1` the first subclass. Also suppose that newly downloaded code needs to dynamically extend `HostInfo` in some other way, such as:

```
class HostInfoWithLock extends HostInfo {
  protected boolean lock = false;
  public synchronized boolean Acquire () {
      if (! lock)
      {
          lock = true;
          return true;
      }
      else
          return false;
  }
  public synchronized void Release () { lock = false; }
}
```

---

[6]Java does have multiple inheritance of *interfaces* but what is needed is multiple inheritance of *classes*.

In this case `HostInfoWithLock` is the second subclass. This new subclass introduces a locking mechanism, perhaps because the new code can restart failed hosts remotely but it needs first to acquire a lock to warn other protocols from trying to restart the same host simultaneously.

The `HostInfoWithLock` extension is handled simply by modifying the second subclass to be an extension of `HostInfo_1` rather than of `HostInfo` as written. Then the new, modified, subclass will contain both the locking data structures from the second subclass and the previously existing data structures inherited from the first subclass, `HostInfo_1`. Note that although the new class "inherits" from two parents it is a Java subclass of only the first subclass, and has the same name as the second subclass. So this class modification has the same effect as multiple inheritance without needing multiple inheritance as part of the Java language.

Commutivity of Extensions

Repeated dynamic extension shares with multiple inheritance one potential pitfall: different extensions may conflict with each other. A conflict arises if both extensions seek to modify the same field or method in different ways. Then the result of doing both extensions will depend on the order in which the extensions happen. A way must be found to identify and resolve such conflicts.

Conflict is avoided if extensions commute. This means that the order of extensions does not matter because the resulting subclass will be the same in either case. For extensions to commute it is sufficient that:

- The extensions introduce fields with disjoint sets of names.

- If the extensions introduce or override a method then the extended method is identical in both cases.

The above conditions for commutivity can be checked automatically and so can be enforced at runtime for dynamic extensions. It is not necessary to enforce commutivity of extensions but it is desirable in some cases.

2.4.4 Protected Dynamic Extension

A dynamically extendable object may need protection just like any other. This protection has the following two goals:

1. The access rights to an object O held by a client after O is extended should be related to the access rights to O before extension. Typically the access rights should be unchanged by dynamic extension.

2. The operation of dynamic extension must itself be protected. If it were not, then any client could extend an object, override any of its methods and subvert the object in any way.

Protecting the dynamic extension operation itself is easy given the technique described in section 2.3.3. That technique depends on a capability scheme, enforced by compile-time checking, to encode access rights. Because the operation of dynamic extension is invoked on an entire class it is a metaobject operation and it can be protected by protecting the metaobject. As shown in section 2.3.3, the metaobject can be protected using either the Capability Interface or Capability Object schemes.

How to protect dynamically extended objects will depend on the mechanism used to implement dynamic extension. Suppose object O of class C is to be protected. The following steps describe an economical way to combine protection with dynamic extension:

1. Use the Capability Object scheme to protect objects. In this scheme a client can only access O indirectly via some other capability object it has been given. No untrusted client has direct access to O itself.

2. Implement both creation of new instances of C and dynamic extension of C in the metaobject using this approach:

   - The metaobject keeps a list of all the objects and capabilities it has created on behalf of clients.

   - When the metaobject creates a new object it stores a reference to that object and returns a capability for that object to the client.

   - When the metaobject performs a dynamic extension of class C it replaces every object of class C with an object of the subclass S.

3. Give the capability object for object O a new operation that will cause it to switch from protecting object O to protecting a new object O' of subclass S when class C is dynamically extended to S. So a client that holds a capability for O before dynamic extension continues to hold that capability after dynamic extension. That capability, however, has (transparently to the client) become a capability for object O'.

4. Protect the switching operation in the capability object. If the metaobject can cause a capability object to switch from one object to another, what prevents an arbitrary client from invoking that switching operation also? Use the Capability Interface scheme to protect the switching operation of the capability object. So a client is given access to an interface for the capability object rather than the object itself. This interface to the capability gives the client access to all operations except switching between object instances.

This approach to combining protection with dynamic extension depends on the flexibility of the Capability Objects scheme. A capability object is given two roles: first, to protect an object of class C; second, to switch to protecting an object of subclass S when the metaobject is dynamically extending the class.

2.4.5 Summary

This section has described one mechanism by which a system can be extended at runtime. This mechanism, dynamic extension, is built using standard features of object-oriented languages: subclassing, inheritance, and dynamic binding. Dynamic extension can be applied repeatedly to the same class, and it can be combined with the Capability Object scheme for protection. Dynamic extension can be implemented in Java, but some operations outside the Java language are necessary.

Chapter 3

Applications

The previous chapter presented the design of a wrapper framework. Most aspects of that design have been implemented under this project.

This chapter discusses applications that were made of the wrapper framework. Most, but not all, of these applications are protocols that can be run within a wrapper.

The implementation of the framework and of applications that use it were written in Java and run on a network of Solaris workstations. The version that was delivered under this project works with the Java Development Kit (JDK) 1.1.7.

The rest of this chapter is structured as follows. Section 3.1 lists the data structures created for use by protocols when a wrapper starts running. These structures are used by various applications. Section 3.2 explains the protocols, and section 3.3 the metaprotocols, that were implemented for this project. Section 3.4 discusses two examples of applications that must cooperate by sharing wrapper data structures. Finally, section 3.5 presents an application in which preexisting code was wrapped.

3.1   Shared Data Structures

At startup, the wrapper creates the following data structures:

- a clock variable that records the difference between the time on the host system's clock and the "true" time, as determined by synchronizing with the clocks in other wrappers;

- a list of hosts known to this wrapper;

- a list of wrappers known to this wrapper;

- a list of protocols known to this wrapper, including the code that implements each role in the protocol;

- a list of the protocol peers, i.e., instances of protocol roles running in this wrapper;

- a list of non-protocol tasks running in this wrapper;

- a list of communication ports known to this wrapper;

- a history of events logged by peers and tasks in this wrapper.

These structures are used by the protocols described in the next section.

3.2   Protocols

The wrapper framework is intended to support the operation of many concurrent protocols. These protocols form an interface to the component being wrapped, and in some cases form part of the application itself.

To show the flexibility of the wrapper framework, we implemented a variety of protocols, described in this section. Each protocol is implemented in Java, although some make use of preexisting facilities that may be written in other languages.

3.2.1   Ping

The Ping protocols collect information about the status of Internet hosts. They are simple protocols, designed to demonstrate protected dynamic extension in a simple context. They use the standard Internet Control Message Protocol (ICMP)[Pos81], also called "ping", to gather information.

There are two protocols:

1. The Ping Seek protocol searches for Internet hosts that respond to ICMP pings. The protocol chooses Internet addresses randomly, pings each address (using ICMP), and if the remote host responds, it is added to a list.

2. The Ping Visit protocol periodically pings each host on the list created by Ping Seek, adding the result to previous data collected about that host.

The Ping Seek and Ping Visit protocols each consist of a single role. These protocols are unusual in that the peers that implement these roles interact with no other wrapper peers: they just use ICMP to interact with Internet hosts directly.

Ping Seek and Ping Visit share a list of objects, each object of which holds data about one host. For Ping Seek, these objects need not have any internal structure: it is enough to record that a host exists and responded to a ping. Ping Visit, however, needs each object to have structure for recording statistics about when and how often a host failed to respond to pings. Before Ping Visit can be started, therefore, the shared list of host objects must be dynamically extended to have this structure. After the list is dynamically extended, Ping Seek and Ping Visit have different capabilities for accessing the list: Ping Seek needs only a capability to create new objects on the list; Ping Visit needs capabilities to look up hosts and add data to the objects that represent them.

Protected dynamic extension is demonstrated by the following scenario:

1. Create an empty list of host data objects.

2. Start Ping Seek, giving it a capability to extend the list.

3. Dynamically extend the list, adding structure to each object.

4. Start Ping Visit, giving it a capability to access the new structure.

Ping Seek will continue to access the list, unaffected by the replacement of the list with a new, dynamically extended list with new structure and new capabilities for accessing that structure.

3.2.2   Distributed Logging

The Distributed Logging protocol creates a single log of the activities of many wrappers distributed across a network. In this protocol, events recorded by each wrapper are sent to a central location and interleaved there with events recorded by other wrappers. The events are interleaved according to the time they arrive at the central location.

Distributed Logging was implemented to help with debugging other protocols, most of which in involve two or more wrappers: when a protocol is doesn't work, a single, unified log of events from each peer in the protocol helps in understanding why it fails. Distributed Logging also demonstrates, however, controlled sharing of wrapper data in a way that will be described in this section.

The Distributed Logging protocol consists of two roles:

1. a Log Relay role, in which a peer collects log data from the wrapper in which it runs and forwards that data to a central location;

2. a Log Merge role, in which a peer runs at the central location, collecting data relayed to it and writing that data to a file.

To create a single log of activity in a network of many wrappers, one runs a Log Relay peer in each wrapper and a Log Merge peer in a single wrapper whose address is known to all the others.

Each Log Relay peer collects data from its wrapper using a shared event buffer. This buffer is shared with each task or peer that needs to log events. This sharing is controlled by capabilities: a task or peer can log events only if it has a capability to append to the event buffer, and the Log Relay peer can only forward events to the Log Merge peer if it first has a capability to extract the events from the shared event buffer.

3.2.3 Clock Synchronization

The Clock Synchronization protocol creates a *distributed clock*. In other words, wrappers running this protocol share a consistent view of real time. A distributed clock is useful support for other protocols, such as reliable atomic multicast, that may need a real time clock for sequencing their actions.

The Clock Synchronization protocol consists of a single role. Each peer in this role periodically measures the difference between its clock and the clocks maintained by peers in other wrappers and sets its own clock to the average of the differences. A clock difference is measured by passing timestamped messages back and forth between two peers: suppose peer 1 sends a message timestamped at its time t_1, peer 2 receives it, then immediately replies with a message timestamped at its time t_2, which peer 1 receives at its time t_3. Then peer 1's clock leads peer 2's clock by

$$(t_3 - t_1)/2 - t2$$

assuming that messages are delayed equally in both directions. To minimize the effect of random delays, the protocol repeats the back-and-forth a number of times for each measurement, averaging the results.

Clock Synchronization tolerates crash failures. Peers that do not respond are ignored when computing the average of clock differences. A Byzantine fault-tolerant version of this protocol was planned but was never implemented[Sch87].

Each wrapper that needs access to a distributed clock runs a single Clock Synchronization peer. This peer maintains a shared clock variable that tells difference between the host's clock and the distributed clock time. (The peer could also modify the host's clock itself, but most operating systems require the peer to have special privilege to do this.) The Clock Synchronization peer is given a capability to read and write the shared clock variable; peers in other protocols that need a distributed clock may have a read-only capability to the shared clock variable.

3.2.4 Stable Sharing

The Stable Sharing protocol enables a group of wrappers to maintain an up-to-date picture of the status of the entire group, and can do so in the presence of failures, both transient and permanent, in the members of the group. It is used to inspect part of an application's state, by finding which wrappers are working and which are not. It can be used either by a system administrator or by other applications.

Stable Sharing creates a self-stabilizing software bus for wrapper status data. A *bus* distributes data to a collection of processes, any of which can write new data to the bus and any of which can read the most recent data written. A *software bus* implements a bus using software. A system is self-stabilizing[Sch93], or more concisely, *stable*, if it is guaranteed to converge to one of a predefined set of states even after being forced into some state outside that set.

Each wrapper that participates in a Stable Sharing group will write its own status to the software bus, and will read the status of all other wrappers in the group. If wrappers join the group, leave, or fail, the protocol is guaranteed, because it is stable, to recover by converging to a state in which all working wrappers in the group agree on each other's status.

The Stable Sharing protocol consists of one role. Each peer in this role organizes its data about wrappers into a ring, ordered by wrapper address. (Wrapper addresses can be ordered in this way because each one consists of an Internet address and a wrapper number, both of which can be ordered.) Each peer forwards its data to its successor in the ring, and updates its own data when its predecessor in the ring forwards data to it. The protocol uses timeouts and acknowledgements in some situations to detect failures and reconfigure the ring accordingly.

Each wrapper that participates in Stable Sharing runs a single protocol peer. This peer is given a capability to modify the data about other wrappers. Other protocols that need to use this data are given a read-only capability to it.

3.2.5 Best Effort Multicast

Message-passing communication in computer systems can take several forms. Typically, messages are passed from a single sender to a single receiver. This type of communication is called *unicast*. In some situations, though, a sender must pass a message to many receivers. This is called *multicast*. More generally, multicast may even involve several senders agreeing on a message to pass to one or more receivers.

The wrapper framework supports multicast communication (and unicast as a special case). This support includes both specialized shared data structures and protocols to use them.

A *port* is an endpoint for multicast communication. So a multicast is sent from one port and received at another, or possibly the same, port. Each port has a unique address. For several peers to participate in a multicast, either as senders or receivers, the port they use must be distributed. So each wrapper that uses a port must maintain data about it; this maintenance is done by metaprotocols such as "Channel Control", discussed later in section 3.3.4.

The simplest wrapper protocol for multicast is called Best Effort. It generalizes the Internet's unicast protocol, UDP, which is also "best effort" in that it does not guarantee that messages will be delivered, only that a "best effort" will be made toward delivery.

Best Effort consists of one role, used for both sending and receiving. Each peer in this role is given at startup:

- a unique UDP endpoint (also called a "port");

- a read-only capability to a list of data about multicast ports.

To send, the Best Effort peer uses its read-only capability to look up the UDP endpoints associated with the destination port. It then sends UDP messages to those endpoints from its own. To receive, the peer listens for incoming UDP messages. When one is received, it uses its read-only capability to look up which peers or tasks in the wrapper are waiting for a Best Effort message on the destination port, and it delivers the message to them.

3.2.6 Reliable Multicast

The Reliable Multicast protocol improves on Best Effort by guaranteeing that messages will be delivered. Even if some of the underlying UDP messages are lost or

garbled, Reliable Multicast will eventually deliver every message sent.

Reliable Multicast is like Best Effort in several ways:

- it sends and receives from ports;

- it has a single role for both sending and receiving;

- every peer uses a unique UDP endpoint;

- every peer has read-only access to a list of data about multicast ports.

Reliable Multicast differs from Best Effort only in that the receiver sends an acknowledgement of every message, and the sender will periodically resend a message until the acknowledgement is received.

This multicast protocol ensures that messages eventually arrive, but it does not ensure that they arrive in any particular order. A reliable *atomic* multicast would guarantee that messages arrive in the same order at all destinations. Such a protocol was planned to support fault tolerant group communication[Sch90] but was never implemented.

3.2.7 Encapsulating TCP

For some applications, it is better to use the Internet TCP protocol for communication, rather than one of the wrapper multicast protocols described previously. TCP has the disadvantage that it is a unicast protocol rather than multicast, but multicast is not always needed. On the other hand, TCP has a big advantage in that it is both reliable and very efficient, much more efficient than using the Reliable Multicast protocol to send unicasts.

So an encapsulation of TCP was implemented in the wrapper framework. This encapsulation gives TCP the same communication interface as offered by Best Effort and Reliable Multicast, e.g., applications send and receive messages at multicast ports. Offering the same interface allows switching between protocols to be done easily, perhaps even automatically (see section 3.4.2).

Although the encapsulation of TCP gives TCP the same interface as other wrapper communication protocols, it differs from those protocols in the following ways:

- multicast is impossible, even if the sender or receiver ports are distributed;

- no protocol peers are needed because TCP is built into every Java environment;

- TCP handles its own set-up and shut-down of communication channels, unlike Best Effort and Reliable Multicast, which depend on a metaprotocol to do the same.

3.2.8   THETA Object Managers

One approach to fault-tolerant distributed computing was taken in the THETA secure distributed operating system[ORA92] and in the Cronus operating system of which it was a variant. In this approach, object *managers* handle all requests for access to objects. More than one manager may handle the same object, in which case the object is replicated. Typically replicated objects are handled by managers distributed on different hosts. Several managers must coordinate access to replicated objects, and this coordination makes access to the object tolerant to some host failures and to some network partition failures.

A simple version of the THETA object management protocol was implemented in the wrapper framework. The protocol has two roles:

1. a Client role, which locates a manager for an object, and forwards requests for access to the object to the manager;

2. a Manager role, which handles requests for access by coordinating with other Manager peers handling the same object at different locations.

Wrappers that participate in this protocol may run one or more Manager peers and one or more Client peers, depending on the set of objects to be handled and the needs of the application that generates requests for object access.

The THETA protocol does not share any wrapper data structures with other protocols. However, a more sophisticated version of the protocol could make use of the data maintained by the Stable Sharing protocol, to locate objects. It would need read-only access to the shared data for this purpose.

3.3   Metaprotocols

A metaprotocol is a protocol that must access another protocol as though it were data. For example, a protocol that downloads the code for another protocol and then runs that protocol is itself a metaprotocol.

Metaprotocols allow a wrapper to be configured by a *bootstrapping* process: an initial metaprotocol is loaded, which loads other protocols and metaprotocols, which themselves may load others, until finally the required configuration of protocols is reached. This bootstrapping process determines not only which protocols are eventually loaded, but also which capabilities each protocol is given. In a well-defined bootstrapping process, metaprotocols would give capabilities to each new protocol based on the trustworthiness of that protocol.

The set of metaprotocols implemented in the wrapper framework is basic. It includes only those metaprotocols needed to support the protocols listed previously, in section 3.2. It includes metaprotocols for beginning the bootstrap process, for user control of wrappers, and for creating and connecting multicast ports. One other metaprotocol, for negotiating and downloading a protocol from another wrapper, was designed but was never implemented.

3.3.1 Boot

The Boot metaprotocol initializes the wrapper. It begins the bootstrapping process that allows other protocols and metaprotocols to be loaded.

Boot consists of one role. Exactly one peer is created in this role. The Boot peer does the following:

1. assigns the wrapper a unique address;

2. creates an initial version of each of the standard shared data structures, including clock data and lists of hosts, wrappers, protocols, peers, tasks, ports, and log events;

3. adds itself to the peer list;

4. starts a peer or peers for Distributed Logging;

5. starts peers for any other protocols specified on the command line.

The Boot peer has full access to all shared data structures in the wrapper.

3.3.2 User Interface

The User Interface metaprotocol allows user control over the wrappers in an application. The interface it provides is useful both for debugging applications and for monitoring their behavior.

The User Interface metaprotocol consists of two roles:

1. a Server role, in which a peer carries out commands from the user and from other User Interface Server peers in other wrappers;

2. a Client role, in which a peer interacts with the user and relays commands to a Server peer.

To install the User Interface, a single Server peer is run in each wrapper. The Client, unlike most other peers, runs in a separate program outside the wrapper and relays commands to the Server via UDP. One or more Clients can be run; several Clients allow communication with several Servers.

User Interface implements commands for starting and stopping protocol peers and other tasks, for displaying the state of the shared wrapper data structures, and for shutting down the wrapper. Each of these commands can be issued in a "flooding" mode, in which each Server peer relays the command to every other Server peer it knows about, until every wrapper in an application has handled the command. Flooding is useful for making every wrapper in an application do the same thing, for example, having every wrapper run the Clock Synchronization protocol or having every wrapper shut down.

Like the Boot peer, every Server peer is given full access to all shared data structures in the wrapper. This degree of access is not strictly necessary, but it is convenient because the User Interface typically starts many of the other protocols, which themselves need to be given varying degrees of access to the shared data.

3.3.3 Web Server Interface

The Web Server Interface metaprotocol, like the User Interface, allows a user to monitor and control wrappers. It offers the same functions as the User Interface, but in a prettier package.

The Web Server Interface metaprotocol, like the User Interface, consists of two roles:

1. a Server role, in which a peer responds to HyperText Transfer Protocol (HTTP) requests;

2. a Client role, whose peer is a Web browser, such as Netscape, and is outside the wrapper.

Like the User Interface Server peer, the Web Server Interface peer has full access to shared wrapper data structures, enabling it to start and stop other protocol peers and to inspect the wrapper state.

3.3.4 Channel Control

The Channel Control metaprotocol is used to make and to break multicast communication channels. A communication *channel* connects two ports so that messages can be sent between them. Multicast protocols, such as the Best Effort and Reliable Multicast protocols described previously, use these channels to direct a stream of messages from a sender to a receiver. Channels are bidirectional, so a connected port can be used simultaneously for both sending and receiving.

Channel Control consists of a single role. A peer in this role has the following functions:

- creating and destroying ports;

- connecting and disconnecting pairs of ports.

For these functions, the peer must multicast a request to other Channel Control peers running on one or more wrappers where the target ports are located. The multicast is made reliable using acknowledgements and retransmissions. When creating a new port, the Channel Control peer randomly selects wrappers to which the port will be distributed, based on the set of wrappers currently running Channel Control peers.

The Channel Control protocol peers are given capabilities for reading the list of known wrappers, and for reading and modifying the list of known ports.

3.4 Cooperating Protocols

A protocol's design can sometimes be simplified if it can share data maintained by another protocol. Several examples of this sharing can be noted in the previous discussion of protocols, e.g., many of the protocols make use of a shared list of known wrappers.

This section discusses two cases of cooperation between protocols not mentioned previously. Both involve shared data structures. Both cases show dynamic features of the wrappers that were developed on this project.

3.4.1 Protected Dynamic Extension

Suppose a wrapper runs both the Clock Synchronization and Stable Sharing protocols. Both these protocols interact with peers in other wrappers.

Ideally, each protocol should have peers running in every one of a set of mutually trusting wrappers: using Clock Synchronization, one would like every wrapper's clock to read the same time, and using Stable Sharing, one would like every wrapper to be aware of the status of all other wrappers. In most situations, the set of mutually trusting wrappers will be the same for both the Clock Synchronization and Stable Sharing protocols.

Both protocols need to maintain data about their interactions with peers in other wrappers:

- a Clock Synchronization peer needs to maintain a history of measurements of clock differences with each of its peers;

- a Stable Sharing peer needs to maintain the status of each other wrapper, as reported by its peer on that wrapper.

Both protocols are designed to dynamically extend, with their own data structures, a shared list of wrapper data. In a situation in which the set of mutually trusting wrappers is the same for both protocols, both protocols must dynamically extend the same shared list.

A typical scenario for using both protocols might have the following steps:

1. wrapper W is started;

2. the Boot peer in W creates a wrapper data list, initially containing an object for W's status;

3. the User Interface peer, at the command of a user, starts the Stable Sharing protocol with these steps:

   (a) dynamically extend the wrapper data list with a status data structure needed by Stable Sharing;

   (b) create a capability with which the list can be read, new wrappers can be added, and wrapper status data changed;

   (c) start the Stable Sharing peer, giving it the new capability.

4. the Stable Sharing peer joins the group of already-running peers in other wrappers (unless it is the first such peer), and begins adding new wrappers and their status data to the list;

5. the User Interface peer starts the Clock Synchronization protocol with these steps:

   (a) dynamically extend the wrapper data list with a clock difference data structure needed by Clock Synchronization;

   (b) create a capability with which the list can be read, and the clock difference data changed;

   (c) start the Clock Synchronization peer, giving it the new capability.

6. the Clock Synchronization peer begins synchronizing its clock with peers in wrappers on the list;

7. a new wrapper, X is started;

8. when X starts its Stable Sharing peer, an object representing X is added to W's wrapper list;

9. W and X begin synchronizing clocks.

This scenario shows several important features:

- repeated dynamic extension as discussed in section 2.4.3, first for one protocol then the other;

- protected dynamic extension as discussed in section 2.4.4, giving the protocols different capabilities to access the same data;

- cooperation between protocols, as the Stable Sharing protocol dynamically changes the set of wrappers used by the Clock Synchronization protocol.

Does the combination of these protocols undermine the correctness of either? No, because:

- The behavior of Stable Sharing is unaffected by Clock Synchronization because the former has no access to data the latter can modify.

- The synchronization of wrapper clocks will certainly be affected when new wrappers are added, but, once the set of wrappers becomes stable, the Stable Sharing protocol will eventually converge to a state in which all wrappers agree about wrapper status, and then all wrapper clocks will eventually converge to synchronization.

63

3.4.2 Channel Replacement

If a wrapper is to replace one communication protocol with another, say, replacing Reliable Multicast with an improved version of the same protocol, and to do the replacement at runtime, then several requirements must be met:

- the old and new protocols must offer the same interface to software that uses the communication channels it provides;

- data about communication ports and their connections must be accessible by protocols that need it;

- the Adapter software, described in section 2.2.3, must have access to the peers running the communication protocols.

These requirements imply sharing within the wrapper: of ports, and of protocols. They also imply that the Adapter, having low-level control over how communication channels are used, must be trusted software. Because the Adapter is part of the wrapper framework, though, and not subject to modification or upgrade by users, it is possible to ensure its trustworthiness.

3.5 Distributed CLIPS

The wrapper framework applications described so far in this chapter have all been new code: protocols, written in Java, specifically for use in the wrapper framework. As explained in the Introduction, though, one important use of the wrapper framework is to wrap preexisting, "legacy" code to incorporate it into a distributed system. This section discusses such an application.

CLIPS is an expert system shell created by NASA[NAS99]. It implements a rule-based approach to expert system programming. In this approach, a program consists of facts and rules of inference for manipulating and reasoning about those facts. A program is run by applying an inference engine to process the rules. CLIPS provides both a language for expressing facts and rules and an inference engine for executing them.

As part of this project, a wrapper for CLIPS was created. This wrapper allowed one instance of CLIPS to send facts and/or rules to another instance and have the remote inference engine execute them. The sending and receiving of CLIPS data was handled by the wrapper's Encapsulated TCP protocol. (Multicast protocols could have been

64

used instead, but this experiment was never tried.) The wrapping of CLIPS created, in effect, a new distributed CLIPS application out of preexisting components.

Chapter 4

Conclusion

The accomplishments of this project were as follows:

- We designed a framework for software wrappers. These wrappers allow a distributed system to be glued together from wrapped components using pluggable communication protocols. In our design, protocols may be added to a wrapper or replaced while the wrapper is in operation – this allows for adaptation. Protocols may also interact via shared data structures within the wrapper – this encourages efficiency and supports continuous operation during upgrades.

- We analyzed the options available for protection within a wrapper. No option analyzed depends on hardware-specific protection mechanisms. We specifically showed how to implement all the protection options in Java.

- We invented a mechanism, called dynamic extension, for extending the shared data structures within a wrapper while maintaining protection.

- We implemented the wrapper framework in Java. Our implementation includes some supporting tools:

  - a Capability Generator tool: this tool takes the Java source for data structures to be protected and generates new Java source for capabilities to those data structures;

  - various tools for manipulating Java bytecodes: these tools were intended for use by metaprotocols when downloading bytecodes from other wrappers, but they were never actually used for this purpose;

  - a basic Capability Verifier: this tool, described in section 2.3.3, was implemented using the Java bytecode manipulation tools;

66

These supporting tools were also written in Java.

- We populated the wrapper framework with a variety of protocols, also written in Java.

- We wrapped CLIPS, an expert system shell, as an example of legacy code, and built a distributed CLIPS system.

The project received less funding than originally planned, and as a result several tasks that we proposed were left unfinished:

- Adapters, discussed in section 2.2.3, were designed to switch between communication protocols while a communication channel was in use. This functionality was never completed.

- A negotiation metaprotocol, described in section 2.2.1, was designed but never implemented.

- Three specification languages, for wrapper interfaces, for component properties, and for protocols, were proposed. Some language design work was done, but was never completed. Instead, we simply used Java to specify interfaces and properties, and to implement protocols, but this approach is inferior to the one we proposed.

One final comment: Our implementation approach, using Java, depends on Java byte-code manipulation tools for code analysis and modification. Bytecode manipulation tools now exist that are better than the ones we wrote for this project[CCK98]. If this work were continued, those other tools should replace ours in metaprotocols that need them.

Bibliography

[AG96]   Ken Arnold and James Gosling.   *The Java Programming Language.*
           Addison-Wesley, 1996.

[AK84]   A. Avizienis and J.P.J. Kelly. Fault tolerance by design diversity. *IEEE
           Computer*, 17(8), August 1984.

[And72]   James P. Anderson. Computer security technology planning study. Tech-
           nical Report ESD-TR-73-51, vols I and II, AD-758 206, USAF Electronic
           Systems Division, October 1972.

[Ben94]   K. Benner. Knowledge-based software assistant – Advanced Development
           Model demonstrations. In *9th Knowledge-Based Software Eng. Conf.*, 1994.

[BM95]   S. Bradner and A. Mankin. The recommendation for the IP next generation
           protocol. Technical Report RFC 1752, Internet Engineering Task Force,
           http://www.rfc-editor.org/rfc.html, January 1995.

[BMD93]   M. Barborak, M. Malek, and A. Dahbura. The consensus problem in fault-
           tolerant computing. *ACM Comput. Surv.*, 25(2), 1993.

[Boo94]   Grady Booch. *Object-Oriented Analysis and Design.* Benjamin/Cummings,
           1994.

[BSS91]   K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic
           group multicast. *ACM Trans. Comput. Syst.*, August 1991.

[CCK98]   Geoff A. Cohen, Jeffrey S. Chase, and David L. Kaminsky. Automatic pro-
           gram transformation with JOIE. In *USENIX Annual Technical Conference*,
           June 1998.

[Cri85]   Flaviu Cristian.   Atomic broadcast: From simple message diffusion to
           Byzantine agreement. In *IEEE Symp. Fault Tolerant Comp.*, 1985.

[Ech86]   Klaus Echtle. Fault-masking with reduced redundant communication. In *IEEE Symp. Fault Tolerant Comp.*, 1986.

[Fra94]   Robert W. Franklin. Is open architecture and type-1 encryption an oxymoron? *Defense Electronics*, August 1994.

[GLS90]   Jr. Guy L. Steele. *Common Lisp*. Digital Press, 1990.

[HS87]   Keith Haviland and Ben Salama. *Unix System Programming*. Addison-Wesley, 1987.

[Int90]   Intel Corporation. *386 DX Microprocessor Programmer's Reference Manual*, 1990.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[Key95]   Key Software, Inc. The fault tolerance in ADM project: Final report. Technical Report F30602-93-C-0237, USAF Rome Laboratory, September 1995.

[LSP82]   L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3), 1982.

[LY97]   Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[Mic91]   Microsoft Press. *MS-DOS Programmer's Reference: version 5.0*, 1991.

[NAS99]   CLIPS: A tool for building expert systems. Internet URL http://www.ghg.net/clips/CLIPS.html, 1999.

[Net96]   Netscape Communications Corp. The SSL protocol. Internet URL http://home.netscape.com/newsref/std/SSL.html, 1996.

[NT94]   B. Clifford Neuman and Theodore Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications Magazine*, 32(9):33–38, September 1994.

[ORA92]   ORA Corporation. The secure distributed operating system (THETA). Technical Report F30602-88-0146, USAF Rome Laboratory, 1992.

[Pos81]   J. Postel. Internet control message protocol. Technical Report RFC 792, Internet Engineering Task Force, http://www.rfc-editor.org/rfc.html, September 1981.

[Rei96] Michael K. Reiter. Distributing trust with the Rampart toolkit. *Commun. ACM*, 39(4):71–74, April 1996.

[S⁺93] O.S. Saydjari et al. Synergy: A distributed, microkernel-based security architecture. Technical Report version 1.0, National Security Agency IN-FOSEC Research and Technology, November 1993.

[Sch87] Fred B. Schneider. Understanding protocols for Byzantine clock synchronization. Technical Report 87-859, Dept. Comp. Sci. Cornell U., August 1987.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4), December 1990.

[Sch93] Marco Schneider. Self-stabilization. *ACM Comput. Surv.*, 25(1), 1993.

[Sol96] Frank G. Soltis. *Inside the AS/400*. Duke Press, 1996.

[Sun95] Sun Microsystems. HotJava(tm): The security story. Internet URL http://java.sun.com/1.0alpha3/doc/security/security.html, 1995.

[Tan89] Andrew Tanenbaum. *Computer Networks*. Prentice Hall, 2nd edition, 1989.

[W⁺81] A.L. Wilkinson et al. A penetration analysis of a Burroughs large system. *ACM Operating Systems Review*, 15(1), January 1981.

DISTRIBUTION LIST

addresses	number of copies
DOUGLAS A. WHITE AFRL/IFT 525 BROOKS RD ROME NY 13441-4505	10
KEY SOFTWARE 131 HOPKINS RD ITHACA NY 14840	5
AFRL/IFOIL TECHNICAL LIBRARY 26 ELECTRONIC PKY ROME NY 13441-4514	1
ATTENTION: DTIC-OCC DEFENSE TECHNICAL INFO CENTER 8725 JOHN J. KINGMAN ROAD, STE 0944 FT. BELVOIR, VA 22060-6218	2
DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 3701 NORTH FAIRFAX DRIVE ARLINGTON VA 22203-1714	1
ATTN: NAN PFRIMMER IIT RESEARCH INSTITUTE 201 MILL ST. ROME, NY 13440	1
AFIT ACADEMIC LIBRARY AFIT/LDR, 2950 P.STREET AREA B, BLDG 642 WRIGHT-PATTERSON AFB OH 45433-7765	1
AFRL/MLME 2977 P STREET, STE 6 WRIGHT-PATTERSON AFB OH 45433-7739	1

```
AFRL/HESC-TDC                                              1
2698 G STREET, BLDG 190
WRIGHT-PATTERSON AFB OH   45433-7604


ATTN:  SMDC IM PL                                          1
US ARMY SPACE & MISSILE DEF CMD
P.O. BOX 1500
HUNTSVILLE AL 35807-3801


TECHNICAL LIBRARY D0274(PL-TS)                             1
SPAWARSYSCEN
53560 HULL ST.
SAN DIEGO  CA  92152-5001


COMMANDER, CODE 4TL000D                                    1
TECHNICAL LIBRARY, NAWC-WD
1 ADMINISTRATION CIRCLE
CHINA LAKE  CA  93555-6100


CDR, US ARMY AVIATION & MISSILE CMD                        2
REDSTONE SCIENTIFIC INFORMATION CTR
ATTN: AMSAM-RD-OB-R, (DOCUMENTS)
REDSTONE ARSENAL AL 35898-5000


REPORT LIBRARY                                             1
MS P364
LOS ALAMOS NATIONAL LABORATORY
LOS ALAMOS NM 87545


ATTN:  D'BORAH HART                                        1
AVIATION BRANCH SVC 122.10
FOB10A, RM 931
800 INDEPENDENCE AVE, SW
WASHINGTON DC  20591

AFIWC/MSY                                                  1
102 HALL BLVD, STE 315
SAN ANTONIO TX 78243-7016


ATTN:  KAROLA M. YOURISON                                  1
SOFTWARE ENGINEERING INSTITUTE
4500 FIFTH AVENUE
PITTSBURGH PA 15213
```

```
USAF/AIR FORCE RESEARCH LABORATORY                    1
AFRL/VSOSA(LIBRARY-BLDG 1103)
5 WRIGHT DRIVE
HANSCOM AFB  MA  01731-3004


ATTN:  EILEEN LADUKE/D460                             1
MITRE CORPORATION
202 BURLINGTON RD
BEDFORD MA 01730


OUSD(P)/DTSA/DUTD                                     1
ATTN:  PATRICK G. SULLIVAN, JR.
400 ARMY NAVY DRIVE
SUITE 300
ARLINGTON VA 22202

AFRL/IFT                                             1
525 BROOKS ROAD
ROME, NY 13441-4505


AFRL/IFTM                                           1
525 BROOKS ROAD
ROME, NY 13441-4505
```

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.