Generalizing mobility for the Hurd

Fredrik Hammar

Examensarbete för 15 hp
Institutionen för datavetenskap, Naturvetenskapliga fakulteten, Lunds universitet

Thesis for a diploma in computer science, 15 ECTS credits
Department of Computer Science, Faculty of Science, Lund University

Abstract

The GNU Hurd features mobile objects in its implementation of filesystem backing stores. This thesis investigates the limitations and security concerns these objects present, and how they can be overcome. This is done in preparation for new applications that feature mobile code and mobile objects. In addition, one such application is studied and implemented, in which mobile code is used to make the `ioctl` system call more extensible.

Sammanfattning

I GNU Hurd förekommer rörliga objekt i dess implementering av lagring för filsystem. Denna rapport beskriver de begränsningar och säkerhetsfrågor som rör dessa objekt och hur de kan övervinnas. Detta görs i förberedning för nya applikationer av rörliga objekt och rörlig kod. Dessutom studeras och implementeras ett sådant användningsfall, där rörlig kod används för att göra systemanropet `ioctl` mer utvidgningsbart.

Contents

List of Figures

List of Listings

Acknowledgements

I would like to thank the Hurd community, especially Olaf Buddenhagen who has suffered through very long and often confusing discussions with me on this topic. I also give thanks to my supervisor Jonas Skeppstedt, and also to my examiner Ferenc Belik who helped me with both getting started and actually getting done with this report. Finally, I thank my brother Peter Arvebro, my father Claes Hammar, and fellow Hurd contributor Zheng Da, for their help in proofreading this report.

Chapter 1

Introduction

The GNU Hurd is a Unix-like kernel that features a distinctive architecture. Traditional Unix-like systems such as Linux are *monolithic*, which means that all their services are implemented inside the kernel itself. *Single-server* operating systems make use of a microkernel that implements a core set of services, on top of which they run a user-space server that implements the remaining services a monolithic kernel provides. The Hurd's *multi-server* architecture also builds on a microkernel, but the other system services are implemented by several separate servers.

This multi-server architecture has a number of advantages. The most interesting of these is that it allows unprivileged users to run arbitrary system services of their own, which allows them to extend the system.

The catch is that the different components of the system can no longer communicate directly by manipulating memory and running code. Instead, the server processes have to use IPC (*inter-process communication*), where processes can only communicate through messages. This indirection makes the implementations of the components more clumsy and less efficient.

Mobile code is a technique which can circumvent this issue in certain situations. The idea is simple: instead of requesting which actions a server should take, the client can request program code that performs those same actions when executed. This code will now have direct access to the client's resources and can therefore be simpler and more efficient. A *mobile object* is an object that not only has state that can be transferred between processes, but also has a code base consisting of mobile code.

In this thesis the foundations for securely implementing mobile code and objects in the Hurd will be explored. Since many readers will be unfamiliar with the Hurd, an introduction to the parts that are necessary to be able to follow this thesis is given in chapter 2.

The Hurd already supports one type of mobile objects: *stores*, which are used to implement backing stores for filesystems. They are very specialized to their purpose, but I will investigate how it deals with the issues presented in this thesis to see if the techniques used there can be generalized. This is done in preparation for future applications of mobile code and objects; one of which is implemented in this thesis.

This thesis is divided into three main chapters, which are outlined below.

3 **Generalizing dependency transfer** tackles the problem of how sensitive information and references to other objects that a mobile object depends on can be transferred in a secure manner. It starts of by analysing how dependencies are handled for various types of stores, and tries to distill them into a few general methods.

4 **Generalizing code transfer** deals with the issue of specifying and loading the required code in a secure manner. Issues in the method used for stores are identified, and a new mechanism is proposed which solves them.

5 **Server provided IOCTL handlers** puts the lessons learned from chapter 4 to use in a novel application: using mobile code to make the Hurd's implementation of the `ioctl` (*IO control*) system call extensible at run time, which is a major feature missing from the current implementation.

I will not implement anything that requires changes to existing IPC protocols or servers; only additions are considered. This is because such changes can cause incompatibilities, make upgrades more difficult, and are more difficult to ultimately get accepted into the Hurd's code base. I will, however, investigate such possibilities and make suggestions for such changes, which hopefully will be of use in future developments.

1.1 Mobility example

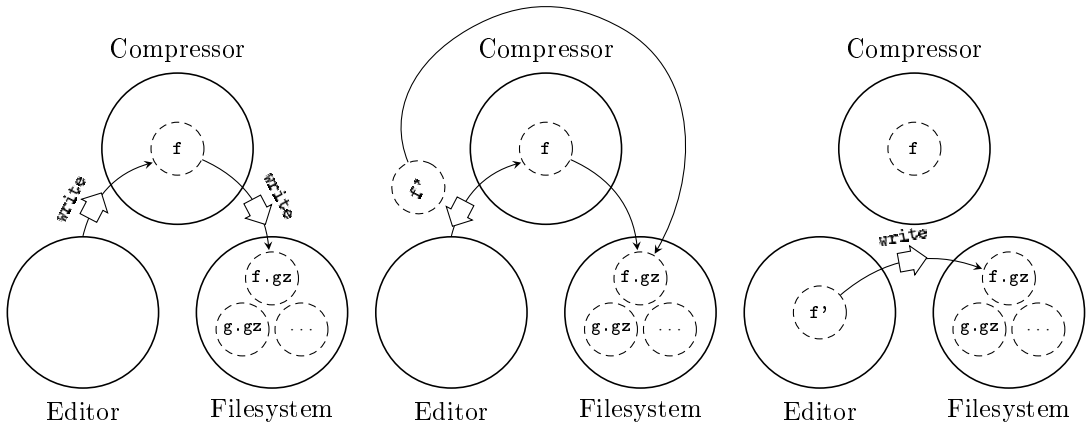
Alice wants to edit her file `f.gz`, which she has compressed to save storage space. The normal way to do this is to decompress `f.gz` into a new file `f`, edit `f`, and then recompress it back into `f.gz`.

Alice thinks decompressing and recompressing the file is tedious. Instead, she decides to make use of the Hurd's extensible filesystem. She starts a compressing filesystem server, which exports a filesystem that contains the virtual file `f`. When Alice's editor tries to write data to `f`, the data is compressed and written to `f.gz` instead. The situation is illustrated in figure 1.1a.

Mobile code can be used to optimize this use case. Instead of reading and writing to `f`, the editor can request a reference to `f.gz`, and the code required to do the compression. In other words, a copy of `f`'s implementation can be transferred from the compressor to the editor, as shown in figure 1.1b. After this, the editor can make use of the copy to read and write to `f.gz` directly, which is illustrated in figure 1.1c.

But how can the editor know that the code will not hijack the editor and make it do something completely different? How can the compressor know that the editor is allowed to use `f.gz` directly? These are the questions this thesis will try to answer.

Using mobile code for optimization like this, is strictly an example of what mobile code could be used for in the future. This particular application will not be investigated any further in this thesis. Instead, the focus will be on the underlying mechanisms needed to answer the aforementioned questions.



(a) *The editor writes to the compressed file* `f.gz` *indirectly though the compressor's file* `f`*, which compresses data before writing it to* `f.gz`*.*

(b) *The compressor returns a copy of* `f`*, after being requested by the editor, along with the needed reference to* `f.gz`*, and code that can compress data.*

(c) *The editor makes use of the copy of* `f` *to write directly to* `f.gz`*, thereby reducing the total number of writes.*

Figure 1.1: *Example of mobility*

Chapter 2

Background

2.1 Mach

The microkernel Mach forms the foundation on which the Hurd is implemented. It provides the means to multitask, the means for different processes to communicate with each other, and also provides low-level device drivers.

This section is based on the *Mach 3 Kernel Principles*[14] and *Mach 3 Kernel Interfaces*[13] manuals, unless otherwise noted.

2.1.1 Tasks and threads

Mach does not support the full-blown notion of the *process* as understood by Unix-like systems. Instead it provides *tasks*, which are much more narrowly defined.

A task is the smallest unit to which one can allocate resources in Mach. A task is a collection of resources, such as virtual memory and *port rights*—which will be explained in 2.1.2. It also contains one or more *threads*, each running program code stored in the tasks memory, potentially in parallel.

While tasks can do nothing on their own—as actions are always taken by threads—it is more convenient to say that a task took an action than it is to explicitly say that one of its threads took it, even though it is not technically true.

2.1.2 Ports

To enable tasks to communicate with each other, Mach provides message queues called *ports*. Messages added to ports can later be read back in the order they were added.

Tasks can access ports through *port rights*, which are kernel-protected references that only allow a particular form of access. The kinds of port rights that are relevant in this thesis are: *send rights* that let a task add messages to a port, and *receive rights* that let a task read messages from a port.

Each task stores their port rights in their own *port name space*, and are specified through an index number called *port name*. These port names are then used when invoking a particular port right. The port name space is local to a task, and cannot be used by other tasks—unless it has access to its *task*

port, which will be presented in 2.1.6. Send and receive rights for the same port are given the same name, so tasks can detect if they are given rights for the same port more than once.

When a task requests that a port is created it is given a receive right to that port, and this is the only receive right that will ever exist for this port. Holding the receive right or any send rights for a port also allows a task to create new send rights for that port.

As a convenience, details regarding port rights and port names will be omitted where there is no confusion. Instead ports will be treated as directed communication channels to the tasks that holds their receive right, and no distinction is made between a port name and the port it refers to. That is, the convoluted saying "task A has a send right for a port for which task B has a receive right", becomes "task A has a port to task B", and "the function takes the port name of this port as an argument" becomes "the function takes this port as an argument".

2.1.3 Messages

Messages sent through ports can not only carry data but also port rights, which enables tasks to discover new ports from other tasks. While receive rights can only be moved to other tasks because there is only one per port, send rights can be both copied and moved.

Messages must follow a certain protocol so the receiver can make sense of them. The protocol states what a message should contain, and how the receiver should respond. For this purpose, messages always have ID numbers that distinguish them from other message types. It is also common for a message to carry a reply port, which specifies where the receiver should send a reply if one is required.

2.1.4 Remote procedure calls

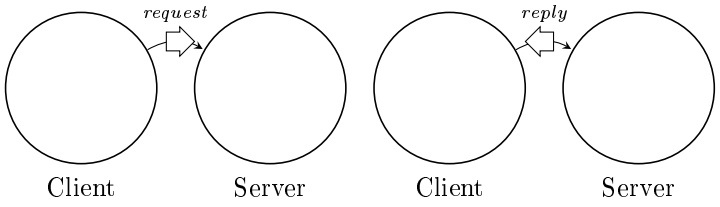
A basic pattern used when building protocols is the *remote procedure call* (RPC), which is used to invoke procedures implemented in other tasks by using messages.

To make an RPC, a task begins by creating a reply port. It then sends an RPC request message with the reply port and any additional arguments. The receiving task then performs the specified action and sends a reply with the results to the reply port, which the calling task is waiting to receive. The reply port is typically only used for a single RPC, as reusing the reply port would allow previously called tasks to reply to later RPCs made to other tasks.

Figure 2.1 shows the notation used for RPCs in the figures of this thesis. Note that the details of the reply port is not shown.

2.1.5 Object system

Each object implemented by Mach is associated with a distinct port, so that RPCs to that port will be interpreted as an operation on that particular object. Most objects provided by Mach are manipulated in this fashion, including the tasks themselves.



(a) *Client requesting an RPC.* **(b)** *Server replying to an RPC.*

Figure 2.1: *Remote procedure call*

Mach itself holds the receive right for such objects, but tasks cannot distinguish a port to Mach from a port to another task. This fact makes it possible for user-space tasks to implement such objects themselves.

It also makes it possible to extend existing Mach objects, by interposing a proxy object between the client task and the object. This proxy can then selectively forward messages from the client to the proxy, alter the messages on the fly, and even respond to new types of messages.

While ports are assigned to tasks and not to the objects themselves, it is often convenient to treat them as such. However, objects within the same task are not protected from one another and can access each other's ports, unlike tasks which are protected from each other by Mach.

2.1.6 Task ports

Each task can be manipulated by other tasks through its *task port*. In fact, a task can only manipulate its own properties—other than memory values—through its task port.

A task's port name space can be manipulated through its task port, and can be used to transfer port rights from one task to another. Transferring port rights to and from other tasks is called *injection* and *extraction*, respectively. The same applies to a task's memory mappings, which can also be manipulated in a similar fashion, so that memory of one task can be made available to others.

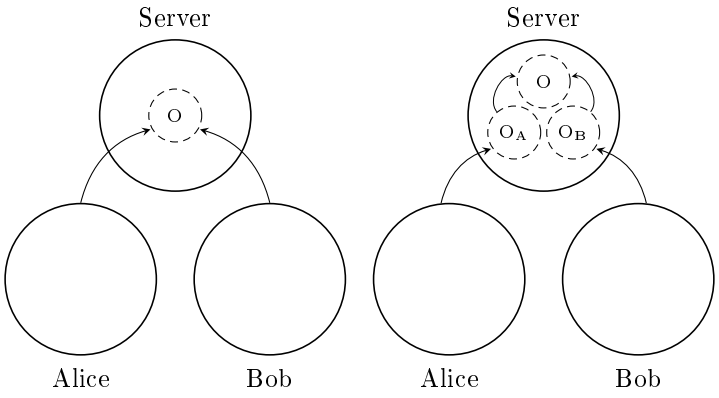
A task port can also be used to create new tasks, which may inherit some or all of the ports and memory mappings from the task referenced by the port.

2.1.7 Access control

Because ports can only be shared explicitly by sending messages to a task—or by manipulating it through its task port—sharing objects can also only be done explicitly. Access to an object is prevented by simply not sharing its port.

Note that access to an object is all or nothing. If a port is given out there is nothing stopping the receiving task from invoking any of the object's operations, and can even propagate this access to other tasks. Once access to an object has been given, the only way to revoke it is to destroy the object's port—or simply ignore all messages sent to it, which would in principal be the same.

The only way to restrict access to an object is to proxy it. That is, instead of giving out a port to the object, a proxy object is created. This proxy responds to a restricted RPC with an error, and forwards all other RPCs to the real



(a) *Without any handles for the underlying object* O*, the server cannot differentiate Alice and Bob.*

(b) *With* O_A *and* O_B *as handles that are specific for Alice and Bob, respectivelly, the server can differentiate them.*

Figure 2.2: *Access control with handles*

object. Similarly the only way to selectively revoke access to an object is if it is used through a proxy. When used in this manner to distinguish clients, proxies are usually referred to as *handles*. This use is illustrated in figure 2.2. If a handle has been destroyed, the referenced object can clearly no longer be accessed through it.

2.2 The Hurd

The GNU project aims to create a Unix-like operating system consisting entirely of free software, and the Hurd is the GNU project's replacement for the Unix kernel. This section is an introduction to the Hurd, and outlines the parts of its architecture that are relevant to understand the difficulties faced in this thesis.

This section is based primarily on the excellent presentation of the Hurd's design goals and architecture outline given by Walfield and Brinkmann in *A Critique of the GNU Hurd Multi-Server Operating System*[17]; and to a lesser degree on the outline given in *Towards a New Strategy of OS Design*[9] by Bushnell, who was the original architect of the Hurd.

The Hurd is implemented on top of Mach, and extends the primitive objects exposed by it to provide Unix-like objects such as files, network connections, and processes.

The Hurd consists of several servers and protocols. The servers are normal user-space processes that implement Hurd objects. Their typical function is to decompose some large object into smaller ones. For instance, a filesystem server takes a backing store, and exposes this flat array of bytes as a hierarchy of files and directories.

2.2.1 The Hurd's name

Hurd stands for *Hird of Unix-Replacing Daemons*, and in turn *Hird* stands for *Hurd of Interfaces Representing Depth*. Though it is an acronym, it is treated as a concrete noun, and as a title rather than a name. So it should be preceded with a definite article, as in *the Hurd*. [7]

The names *Hurd* and *Hird* where invented as alternate spellings of the English word *herd*, and are pronounced the same. The intention was to make *GNU Hurd* allude to a herd of gnus. [7]

2.2.2 Credentials

The Hurd has two types of entities to which one may grant access: *users* and *groups*. A user in the system might not coincide with a physical user. For instance, a physical user may use several system users, each for a different purpose, or may share a single system user with others. System users might also be allocated for use by system services. Since physical users are not interesting in this context, system users will be referred to unless otherwise specified. Users can be members of groups, which are associated with permissions that are usually weaker than those associated with users.

Each user is identified by a *user identifier*, and each group by a *group identifier*. These identifiers are unique numbers, which are typically small. Users and groups are normally given human-readable names as well, so that they are more memorable. However, these names are implemented by higher-level parts of the system and are only loosely connected with the identifiers. The same identifier can even be given several names, though this is not typical. [3]

Each process in the Hurd can be run on behalf of several users and groups, and other processes may grant access to it by virtue of being run by these *principals*. To invoke the authority of its principals a process makes use of its *credentials*, with which it can prove the identity of its principals to other processes. The set of credentials is a protected object implemented by the *authentication server*. The credentials of a process are passed to it by its parent process on startup, and they are usually the same as those used by its parent.

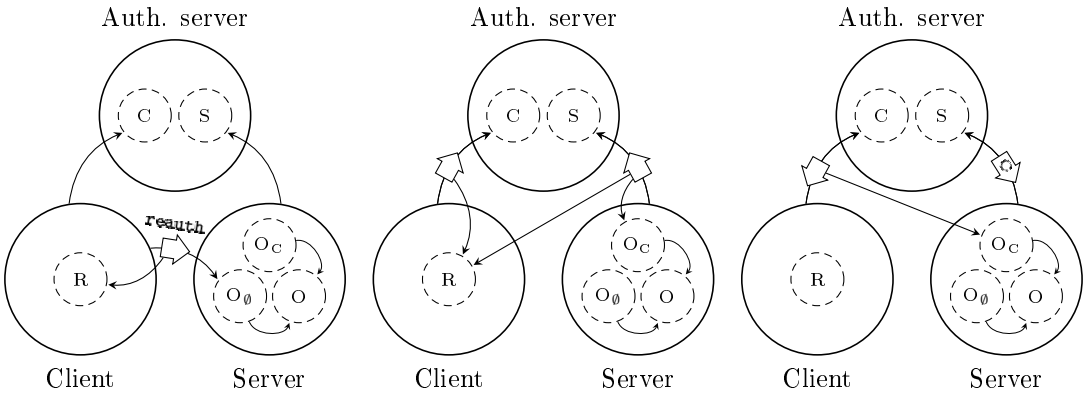
The credentials are nothing more than a protected collection of user and group identifiers. The authentication server has operations that allow the creation of new credentials as long as they are subsets or unions of other credentials. If the credentials contain the special root user identifier, then it can also be used to create credentials that contain arbitrary identifiers.

In more traditional Unix-like systems, each process can only have a single user identifier in its credentials. In the Hurd, credentials can have any number of user identifiers, including zero. This also allows processes to drop credentials when it no longer needs them, which lessens the impact if a process is compromised.

2.2.3 Authentication protocol

For a process to prove its authority to another process, it must prove that it has the necessary credentials containing the required principal's identifiers.

The naive way to do this would be for the client to send the port to its credentials to the other process for inspection. However, this would allow a



(a) *The client requests that the server reauthenticates the* $O_\emptyset$ *handle for the underlying object* O*. It sends along a* rendezvous *port to the object* R*, which exists only for this purpose.*

(b) *Both the client and server sends the* rendezvous *port to their respective credential objects* C *and* S*. The server also sends a port to the* O_C *handle.*

(c) *The authentication server pairs the two requests using the* rendezvous *port. It replies with the port to* O_C *to the client, and the identifiers in* C *to the server. The server now knows the identity of the client.*

Figure 2.3: *Authentication protocol*

malicious server to steal the credentials by passing them on to other processes and claim that these credentials are its own.

Instead, the authentication server provides a three-way handshake, where the two parties independently contact the authentication server through their own respective credentials. The authentication server then pairs the requests, using a *rendezvous* port, which the parties previously exchanged before passing it along to the authentication server. The authentication server then responds to the client with a port provided by the server, and responds to the server with a list of identifiers stored in the client's credentials. The protocol is illustrated in figure 2.3.

The server never comes in direct contact with the client's credentials, but it can still rely on them being correct, since the information came from the authentication server.

Note that only the port that was passed through the authentication server is authenticated. If the server associates the port on which it got the authentication request with the credentials that resulted from the authentication, then another malicious server could fool it by forwarding an authentication request it got from one of its own clients. The malicious server would then be authenticated with its client's credentials. This is an example of a *man-in-the-middle attack*.

2.2.4 Virtual filesystem

The protocols for filesystems are especially central to the Hurd. They can be implemented by arbitrary processes and it is possible to connect two filesystems so that they are presented as a single large filesystem tree. This makes it

possible for normal users to make arbitrary extensions to filesystems. The only requirement is that the user owns the filesystem node that the other filesystem will be attached to.

Servers that implement filesystems are called *translators* because they translate filesystem operations to domain specific operations. For traditional filesystems, this means operating on the storage that contains the filesystem, but filesystems could just as well use network communication, or even be completely virtual and only use pure computation.

Most of the Hurd's components are translators, even if they do not have any operations that can be expressed through the filesystem interface itself. Such translators typically implement filesystems that contain just a single file, which cannot be read or written to. In these cases the filesystem's only function is to make the translator available through the filesystem. In other words, the filesystem acts as a namespace for the Hurd's various operating system services. Examples of such translators are the network stack and the password server.

2.2.5 File permissions

Each file has four sets of permissions that apply to clients based on which users and groups are in its credentials.

Owner permissions used for credentials with the user that owns the file.

Group permissions used for credentials with the owning group.

Others permissions used for credentials with any users.

Unknown permissions used for credentials without any users at all.

The unknown permission set is an extension specific to the Hurd, since processes in traditional Unix-like systems always have a user identity.

Each permission set can contain permissions to *read*, *write*, and *execute* a file. When the file is a directory, the permissions have different meanings: read permission allows the entries to be read, write permission allows entries to be added or removed, and execute permission allows the entries to be opened, but is usually called *search permission* when referring to directories.

2.2.6 Process server

Every Hurd process has a port to the *process server*, which keeps track of all running processes in the system. Its primary purpose is to allow users to manage their processes and obtain information such as which program a process is running and which process started it.

It also keeps track of the task port of each process, and hands it out on request if the client has the necessary authorization. This task port can then be used to access the memory and ports of the target process, which is useful when debugging. This access to the resources of other processes will prove useful in later parts of this thesis.

2.2.7 Sub-Hurds

At start-up, a process gets a port to the root directory, through which it makes lookups in the filesystem. It is possible for the parent process to give the new process a completely different root than the one it uses itself.

This together with the fact that most operating system services are reached through the filesystem, or inherited in the same manner as the root directory, makes it possible to start sub-Hurds. A sub-Hurd is an instance of the Hurd running under another instance of the Hurd, which can replace some or all of this *super-Hurd's* services.

The fact that different processes can be in different but partially overlapping environments, is one of the Hurd's most interesting and distinctive features. I will therefore take special consideration of this in my work.

2.2.8 Stores

Mobility is already present in the Hurd, but only for block devices implemented through the *store* abstraction. A block device is a special file that represents the content of a block divided medium, such as a hard drive, and stores are mobile objects used to implement block devices.

Block device translators can implement the `file_get_storage_info` RPC, which returns a marshaled version of the store. This marshaled store can then be unmarshaled by clients and used directly in a manner similar to the example in 1.1.

The most often used stores are just simple wrappers around ports to device drivers implemented by Mach, so called `device` stores, or wrappers around ports to regular files, which are called `file` stores.

Other more interesting store types include `zero` stores, which are totally self-contained immutable stores that only contains zeroed bytes, and do not rely on external servers to implement the actual storage functionality. The `concat` and `stripe` stores are also interesting, as they allow users to combine several stores into larger ones, in different combinations.

2.2.9 Channels

Character devices are files that correspond to input and output devices such as terminals, mouses, and sound cards. I made an attempt to create a *channel* abstraction, which was to be similar to stores, but for character devices instead of block devices.

I abandoned the attempt because the semantics of different kinds of character devices differ too much to be used through a single interface. It became clear that it was just as well to generalize the problem to arbitrary objects, which led me to the subject of this thesis.

Chapter 3

Generalizing dependency transfer

To transfer a mobile object from one process to another, any dependencies it has must also be transferred, or somehow be substituted. These dependencies can be data, ports to objects, or other mobile objects.

Technically, this is trivial as Mach's message passing facilities provide the means to transfer the data and ports that make up a mobile object. However, once an object has been transferred it is no longer protected from the client, and the client will gain direct access to all of its private dependencies. The real question is whether the transfer can be done securely.

The goal of security is to prevent any unauthorized access, and a secure system will not grant any access unintentionally. A situation where a process gains unauthorized access is usually referred to as a *privilege escalation*. For instance, consider a translator that exposes a stored filesystem. File objects in such a translator depend on a store object. If a process that has access to one of these files were given access to the entire store by loading the file object, then it would also be able to access all other files in the filesystem.

Since access is all or nothing unless an object is proxied—which would mean adding indirection when the aim is to remove it—the only option is to demand that the client *already* has access to the required dependencies.

Any dependencies that are mobile objects are not interesting here, since any method that can transfer the dependant object securely can also be used to transfer its mobile object dependencies. Also, it will be assumed that the server has no knowledge of the implementation of any object dependencies, which is a prerequisite for handling them in a generic manner. Such dependencies will be referred to as being *external*. When it comes to data, focus will be on sensitive information such as passwords, because generic data does not pose any security threats.

Which brings us to the problem addressed in this chapter: how can a server determine whether the client can access an object that is not implemented by the server itself?

The objects a process may access are determined mainly by the ports it holds, and which additional ports it can acquire by using them. Access can also be gained through passwords, or similar pieces of information such as cryptographic

keys. For instance, a process could use one to log in as a different user, log in to a remote system, or decipher an encrypted file.

Finally, a process can also extend its access through indirect means. For instance, it might be able to change the permissions of a file, be able to read a file containing passwords, or even be able to request a user to manually grant it the needed access. But these cases will not be considered, as manual intervention should not be required, and because changing file permissions could inadvertently grant access to other processes as a side-effect.

To tackle the main issue of this chapter, I will first analyse how it is handled by the existing store framework, then use the information obtained to derive two complementary ways of determining that the client has access to the dependencies. The first—in section 3.2—makes use of privileged access to the server itself if available, and the second—in section 3.3—attempts to re-acquire the dependencies from the ports already available to the client, though ultimately this way will be found too uncertain and complicated to be followed through.

3.1 Securely returnable stores

The mechanism for determining whether a store can be securely loaded by a client is divided between the store itself, the `libstore` library, and the translator providing the store. As this process is not documented, I have studied the Hurd's source code[5] directly and will document my findings here.

The `file_get_storage_info` RPC is used to load the underlying store of a file. When it is called, the translator marshals the store into a format that can be sent in a reply message. When the client gets the reply, the store is unmarshaled, and is then ready to be used directly.

But before the store is returned, the translator must make sure that the store is *securely returnable*. Because the `storeio` translator does nothing else than expose a single store as a single block device file, it will serve as the reference implementation for what a proper store translator should check on a `file_get_storage_info` call.

If the client is the root user, `storeio` always returns the store. For users other than the root user, `storeio` makes use of the `store_is_securely_returnable` function provided by `libstore`, which takes the store and the open mode as arguments—the open mode being whether the file has been opened for reading, writing, both, or neither. This function examines the open mode and various store variables, and returns an error if the client should not be allowed to load the store.

I have broken down the various properties of a store that makes it considered securely returnable or not, and will present them below. I will also analyse why they are used, and if there are any problems with them.

3.1.1 Innocuous stores

A store is *innocuous* if it is impossible to do anything harmful by possessing it, as the comment of the `innocuous` flag states. The only store like that in `libstore` is the `zero` store, which is a store that only contains bytes that are zero, and silently ignores all writes. [12, Section 7.3.4.6]

Analysis

It is easy to see why a `zero` store is safe to return. As it is completely virtual, it would be possible for the client to implement the store without any contact with other processes. But because `zero` stores do not have any dependencies, they are not very interesting in this context.

Rather, the question is whether an object with external dependencies can be considered innocuous. Answering this question requires intimate knowledge of the dependencies, and such information is only available to the implementer of the dependency. To get this information, the server must either implement the dependency itself, in which case it is not external, or query the information from the dependency itself.

There is no generic interface that supports such a query in the Hurd, and extending existing objects to support such an interface is beyond the scope of this thesis, so this property will not be useful here.

3.1.2 Stores with an enforced range

Every store has a *range* that specifies which portions of the store access should be restricted to. The range is encoded in the store variable that is—somewhat confusingly—named `runs`. A store is *enforced* if it is not possible to access areas outside of the range, even if the range is later modified. For instance, `device`, `file`, and `task` stores only set the enforced flag if the range already covers the entire backing store, in which case there simply are not any areas outside the range. If the range is enforced, the store is considered securely returnable if it has been opened for both reading and writing.

A *hard read-only* store is one that cannot be made writable, unlike regular read-only store, which can be made writable by the holder of the store. If a store that has an enforced range is also hard read-only, then it only needs to be opened for reading to be securely returnable.

Analysis

The justification for considering enforced stores securely returnable was presumably that the client already has access to the entire range when using it remotely through the normal filesystem interface, or stated more generally, that there is no access to the dependency that is not already exposed by the dependant.

I believe this reasoning is flawed for the same reason stores with external dependencies cannot be considered innocuous. That is, answering it requires intimate knowledge of the dependency's implementation, in which case the dependency is not external, or the necessary information must be queried from the dependency, which there is currently no generic operation for.

For instance, files support other operations than just reading and writing, such as changing the file's permissions, but this functionality is not exposed by `file` stores.

3.1.3 Inactive stores

An *inactive* store is one that only carries symbolic descriptions of its dependencies. When activated, it attempts to open any resources it requires, for instance, by opening files specified by paths. A store may be deactivated again later, at

which point it drops all of its external resources. I deduced the meaning of the `inactive` flag by examining under which conditions `device`, `file`, and `task` stores set or clear it.

An inactive store is always considered securely returnable. If a store is not securely returnable, `storeio` tries to make an *inactive* copy of the store and return that instead.

Analysis

The reason inactive stores are securely returnable is presumably because the client is then forced to reopen any needed external resource on its own, which it fails to do if it lacks the required access.

The main problem is that this assumes that the symbolic descriptions will resolve to the same object. For instance, this is not the case for paths if the client and server do not share the same root directory. But even then, the original file may since have been replaced by another file.

Another problem is that this also assumes that the symbolic description is already known by the client. For instance, if one of a path's components is in a directory that is readable but not searchable by the client, then giving it the path grants it access to the file named by the component. That is, the path component would act as a password for accessing the file. This might sound exotic, but it is just a manner of setting the read permission bit and clearing the search permission bit of the directory, which is possible on any Unix-like system. The Hurd also makes it possible to implement translators that support hidden files that do not show up when listing directories but could still be opened. This is an interesting future possibility which this assumption undermines.

3.1.4 Nontransferable stores

When a store is to be transferred, it is first marshaled into a format suitable for the task by calling its `encode` method. This offers stores a chance to categorically refuse any transfer by not supporting the `encode` method.

The `copy` store is an example of such a store. It implements a copy-on-write copy of another store, storing blocks that are modified in memory. It is clear that if such a store is loaded into the client, the data structure for in-memory blocks can not just be copied to it, otherwise they will differ as soon as either of the copies are written to.

Analysis

This does not give any additional cases under which conditions dependencies are transferred; it is only here to complete the description of when stores can be securely returnable.

3.2 Criteria for secure dependency transfer

Store transfers are always allowed if the client is the root user, which is not an uncommon policy for a Hurd server, as the root user represents the owner of the system. For instance, the authentication server allows the root user to create arbitrary credentials. Because of its dominant role it is easy to see why it is

pointless to deny any access to the root user. This raises the question: can such dominance be found in processes run by regular users?

The server is trying to protect the data and ports of the mobile object. As explained in 2.1.6, the memory and port name space of a process can be accessed by other processes through its task port. It should be clear that if the client has this access then there is no point for the server to protect any of its resources from it. In other words, the client can completely bypass the server's defenses.

This privileged access is not as uncommon as one might expect. In fact, it is even used by the debugging program `gdb`.[6] The access is granted by the process server—presented earlier in 2.2.6—which does so by handing out the task port of a named process. It grants this access to the root user, the owner of the process, or if the process has no owner, to any other process in the same login collection.

If the client has authenticated against the server, then the server can easily tell whether the client is run by the root user or the owner of the server's process. This provides a very straightforward way of determining when dependencies can safely be transferred.

Checking whether the client is in the same login collection is harder, as login collections can only be specified by an easily forgeable ID number. To check this properly also requires extensions to the process server, which is beyond the scope of this thesis.

A better alternative is for the server to ask the process server directly whether the client has access to the server's task port. This also allows for the possibility that the access policy of process server changes. While this option also requires extensions to the process server, it could be a good follow up project.

3.3 Substitute dependencies

If it is not possible for the client to extract the dependencies directly from the server, it might be possible for the client to find suitable substitutes on its own.

I got this idea from how inactive stores are handled, where instead of sending dependencies directly, a description of the dependencies is sent. This forces the client to reopen the dependencies using its own authority.

The main problems with inactive store transfer—as explained in 3.1.3—is that there are no checks whether the description actually resolves to the same object, and that it assumes that file paths are not sensitive information, which may not be the case.

In the following sections, I will examine when two objects can be considered equal, and whether they can safely be compared with each other without exposing the server's object to the client, or the client's object to the server. I will then examine when two file handles can be considered equal, and if it is possible to resolve file paths while testing that the path is readable to the client. As will be shown, the Hurd is currently not well equipped to deal with this properly, which will lead to the conclusion that substituting dependencies is not yet viable. It does, however, present future opportunities for study.

3.3.1 Object equality

To find a proper substitute for the dependency of an object to be transferred, there must be a way to determine whether two objects can be considered equal.

The ideal scenario is if the exact same object is used, because then both the client and the server would use it through the same port. Since port rights to the same port are given the same port name when in the same process, the test is just a simple integer comparison.

However, as I explained earlier in 2.2.3, most objects are referenced indirectly through handles that remember the credentials of the client. Also, many handles carry additional state, such as the current position when reading or writing a file. Given this, it is unlikely that the server gives out the same port twice.

For two handles to be equal they must permit the same operations. The only general mechanism to test whether an object permits a certain operation, is to actually call it. Which means that the test is postponed until the procedure is needed, in which case it may be too late to recover. More importantly, it is not possible to detect that an operation has succeeded where it would have failed using the original dependency, which would mean that the dependency behaves differently in the client than in the server.

In summary, the only way two objects can be determined to be equal in general, is if they are in fact the same object. As this is a fairly rare occurrence, there is little point in going further with this idea.

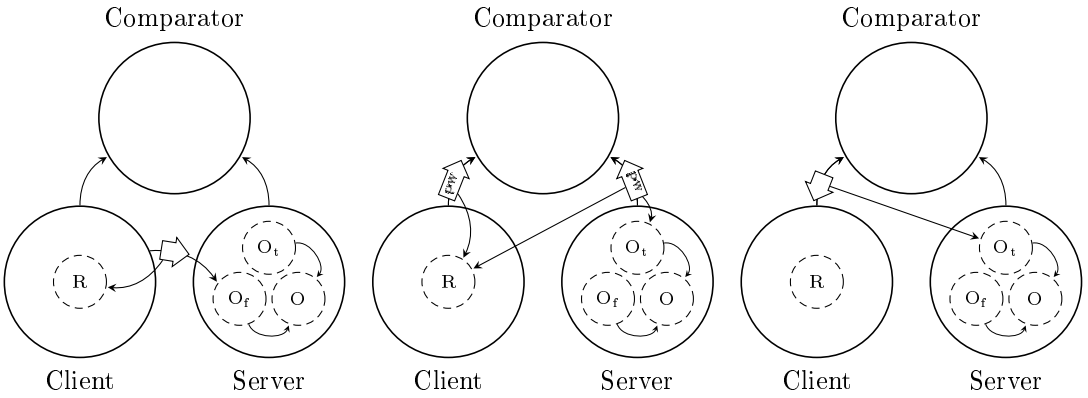
3.3.2 Secure comparison

Though there seems to be little reason in comparing objects when there are only a few cases they can be determined to be equal, it is interesting to consider how two objects can be compared as it poses interesting problems, and can still be useful in the future for both existing and future special cases.

The problem is that the comparison cannot be made in either the server or the client. This is because if neither can access the other's object now, granting any access to do the comparison will lead to privilege escalation. The same is true for passwords: sending a password to anyone who does not know it, is a poor test to see if they have it.

The answer is to do the comparison in a server trusted by both parties. The same technique that the authentication server uses can be used here to establish a three-way handshake. Each participant sends their port to the object, along with a previously shared *rendezvous* port, to their own trusted comparison server which then matches up the requests using the *rendezvous* port, and proceeds to compare the two objects. It then returns the result of the comparison to the server, and a new handle to the object to the client. After this, the server knows that holder of the new handle passed the test.

The figure 3.1 illustrates a similar protocol for passwords. Note the similarity with figure 2.3, which depicts the authentication protocol. The main difference is that the comparison server does not keep track of the clients identity and no results are returned to the server since it is enough to know that messages to the new handle must be from someone who knows the password.



(a) *The client requests a comparison to see if it has access to the same password as the server, it includes a* rendezvous *port unique for this comparison.*

(b) *Both the client and server sends the* rendezvous *port and the password to the comparator. The server also sends a port to the O_t handle.*

(c) *The comparator pairs the two requests using the* rendezvous *port. It replies with the port to O_t to the client. The server needs no reply since it knows any messages to O_t must be from someone that knows the password.*

Figure 3.1: *Example password comparison protocol*

3.3.3 File handle equality

File handles are almost certainly the most common form of object in the Hurd, so it would be useful to be able to compare them. However, because each client of the same file has a different handle, comparing ports are not enough.

The `io_identity` procedure of the `io` interface—which is implemented by all file handles—returns a send right to a port that is unique for the *underlying IO object*. However, it does not follow that file handles with the same identity port can be used as substitutes for each other.

This is mainly because what constitutes an *underlying IO object* is not well defined for any file that has different semantics than regular files. I know of no example where file handles to different files would return the same identity port, but it is definitely possible to implement such files in the Hurd. Arguably handles to such files should not return the same identity ports, but that is a new requirement of the interface, and changing existing operations is beyond the scope of this thesis.

Another problem is whether the two handles permit the same operations, which one can only be certain of if both have been authenticated using the same credentials. This is a very strong requirement that cannot be examined using existing interfaces. But if the handles were authenticated using empty credentials, then there is no possibility of privilege escalation, since empty credentials are available to everyone.

Only using empty credentials may sound like a severe limitation to the operations that are permitted, but permission for the most common operations—read and write—is only checked when opening the file and can later be examined with the `io_get_openmodes` procedure from the `io` interface. If the file handle is

reauthenticated *after* it has been opened, the handle will only permit operations whose permission can be examined, or is already available to everyone.

Still, the limitations of establishing whether two file handles are equal in the first place means that this is mostly a curiosity at this point.

3.3.4 Secure file resolution and comparison

While being able to compare two file handles is not very useful when it is not even practical to do so without the requirement of security, I have again examined the issues involved in the hope it will help future efforts on the subject.

For the client to know which object to send to the comparison server, the server must first name it somehow. For files this is naturally done with a path. However, as explained in the analysis of inactive stores (3.1.3), paths can be considered passwords.

To get around this, the entire lookup can be done in a comparison server for files. The server can send the path and a port to the file it should lead to, and the client can send a port to its root directory. The comparison server can then lookup the path in the client's root directory and compare the resulting file port with the server's file port. In addition, during the lookup, the comparison server should check that each component of the path is listed in its containing directory. This makes absolutely sure that the component does not refer to a hidden file.

Note that it is not necessary that the directories on the path are equal to each other. It is possible for the client to use a different root directory but still have access to the file under the same path. For instance, this occurs if the same translator is mounted in the same location on both the client's and the server's root file system. Only the final component needs to be directly compared with the file used by the server.

Chapter 4

Generalizing code transfer

In this chapter, I will examine how mobile code can be transferred from the server to the client, and how the client can determine if it can trust the code enough to load it without fear of being hijacked.

I will first give some background on how code linking and loading is handled in the Hurd, which should give the reader an overview of the available options.

Then, I will present how code loading is handled by stores, which is secure, but inadequate if the client and server are running in different environments.

Finally, I will then propose using ports to reference the required code modules directly, explain why doing so unconditionally is insecure, propose a condition for when it is secure, and propose a method for testing this condition.

4.1 Linking in the Hurd

In this section, I will present the different methods of linking together modules to form a single program that are supported by the Hurd.

Each file of source code in a program is typically compiled into a single *object file*. Such a file contains the named binary objects that correspond to the procedures and variables of the program. As a program is typically made up of several modules which reference each other, these object files need to be combined into a single executable program.

This process is called *linking*. There are three types of linking supported by the Hurd, which are listed below. They are distinguished by the time at which they are performed. Note that they are not mutually exclusive and often complement one another.

Static linking is done as part of the compilation. It links together object files into a single executable file.

Dynamic linking is done by the system when a program is to be run. It is typically used to link objects that are shared with other programs so that they do not need to be relinked each time an object file is modified.

Dynamic loading is done at the request of the program itself when it is running. It is typically used to add extra functionality to the program in a plug-in fashion.

An object file that can be linked or loaded dynamically is called a *shared object file*. By convention, normal and shared object files end their file names with `.o` and `.so`, respectively.

Shared object files are searched for in the directories `/lib` and `/usr/lib` by default, and additional directories can be specified in the environmental variable `LD_LIBRARY_PATH`. [2]

Dynamic loading is done with the `dlopen` function provided by the GNU C Library. Shared object files at any location can be loaded, in addition to searching for them in the same way as dynamic linking. [1]

4.2 Code lookup for stores

In this section, I will examine how a store's code is loaded into the client's program, and present the main disadvantages with this approach.

Each store class is identified with an ID number, and this number is included in a store's marshaled form. When a store is to be loaded, a lookup is first made in a collection of standard store classes provided by the `libstore` library. This lookup is done by the `store_decode` function[5].

If this lookup fails, an attempt is made to dynamically load a shared object file containing the class. It tries to load a file named:

`libstore_type-`*id*`.so.`*version*

where *id* is the ID number and *version* is current version number of the Hurd. This part is handled by the `store_module_decode` function[5].

As the lookup is done using `dlopen`, normal users can install their own store classes if they set the environment variable `LD_LIBRARY_PATH` so that it lists a directory containing the shared object files.

4.2.1 Analysis

I have not been able to identify any security issues with the code lookup mechanism used by stores. If the server can write to the directories where the client searches for modules, then it would have already been able to replace any shared object file the client program relies on, such as the GNU C Library. And there is no risk of the server tricking the client into loading a shared object file that is not a store module, since they do not stick to the store naming convention.

However, there is a problem associated with using symbolic names such as ID numbers: they may not refer to the same file over time, or in different environments. In particular, it is possible that the client has installed its own version of the requested store class. This version could be out-dated, which may lead to errors. It could even be a completely different store type which has mistakenly been assigned the same ID number, which could happen if the clashing classes were developed independently from each other.

4.3 Port designated modules

A straightforward way to avoid the problems of symbolic names is to use direct references, which means ports in the Hurd. So instead of transferring an ID

number, the server can transfer a port to the shared object file the client is to load. Since the object file is directly referenced, it is guaranteed that the server and client will not use different modules by mistake. For the same reason, different environments for the client and server does not pose problems. The server can avoid escalating the client's privileges by using the dependency transfer techniques developed in chapter 3.

But as I will explain, while loading code specified by ports does away with the practical problems associated with symbolic names, it instead poses new security issues. In this section, I will examine these issues and propose methods to overcome them.

4.3.1 Trusting port designated code

The security issue of trusting port designated modules is that the server can now specify arbitrary code, instead of naming modules that the client already trusts.

From a security perspective, loading arbitrary code specified by the server is equivalent to the server having direct access to the client's ports and memory. This is easy to see. If the server can modify the client's memory, it can simply write the program code into a location of the client's memory where it will be executed. If the server can make the client load arbitrary code, the code can read any data and ports the client possesses, and send them back to the server. The last part assumes that the client can still communicate with the server when the code is run, because even if the connection used to request the code has been closed, it should not be assumed that the code cannot reopen the connection again by using other open ports.

Because they are equivalent, the server having access to the memory and ports of the client, can be used as a criterion for loading code. In chapter 3.2, I established that the client has access to the server's memory and ports, if it is run by the same user or the root user, which in turn can be found out by using the authentication protocol.

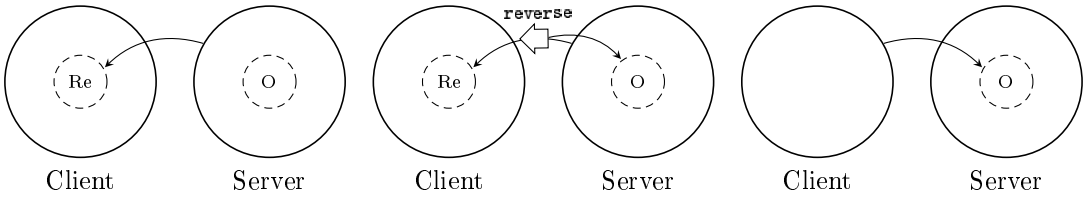
There is only one problem with this: the authentication protocol is designed to let a server establish the identity of a client, and not the other way around.

Reverse authentication

The client can establish the identity of the server using the normal authentication protocol with the roles reversed. That is, the client gets the server's credentials, while the server gets a port supplied by the client, as illustrated in figure 4.1a. But the goal is to supply the client with a port as well as the server's credentials.

My solution to this problem is very straight forward: the client only serves a single request in which the authenticated process returns a port on which it later services requests. That is, a reply port is given to the server instead of an object, as shown in figure 4.1b. After this their roles have been reversed, so that the client can act as a client, and the server can act as a server, as figure 4.1c shows. I will refer to this variant of the authentication protocol as *reverse authentication*.

Since the client knows the credentials of the process that received the reply port, it also knows that only that process—or another process that process has



(a) *After authenticating the server, the roles of client and server have been reversed.*

(b) *The client serves a single request, in which the server provides a port to an object.*

(c) *Now that the roles have been corrected, the client can drop the temporary reply object.*

Figure 4.1: *Role reversal*

delegated to—could have replied the new server port. That is, the client can now trust that the server port has been vouched for by the principals specified in the credentials.

The client can now safely load any code returned by the server port, without the risk of escalating the server's privileges, as long as the credentials specify a user that already has access to its memory and ports.

4.3.2 Indirect port designated modules

Port designated modules have one practical disadvantage: it requires that the server is run by a user that has privileged access to the client's memory and ports. This means two normal unprivileged users cannot make use of each other's mobile code services.

This can be mitigated by using *privilege separation*[16], in which privileged components of a program are factored out as separate programs. Here, that means separating the privileged service of designating trusted code from the server into a code providing server. The server can now simply delegate a request for mobile code to the code provider which has privileged access to the client's memory and ports. Two unprivileged users can now collaborate with the help of a privileged user that runs the provider.

In fact, the request is already indirect in that the server sends a port to a shared object file that most likely resides in another server.

Unfortunately, it is not possible to determine which user controls the file, as the Hurd's filesystem interface does not provide any operations for this. It is possible to get the user identifier of the file's owner, but as this is just a plain integer, it is no proof that this user actually controls the file. This could be made possible by implementing reverse authentication in existing filesystems, but that is beyond the scope of this thesis.

For now, it is possible for servers to simply forward the request for a code module to specialized code providers run by privileged users, as shown in figure 4.2.

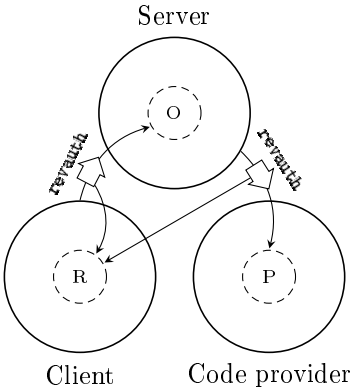


Figure 4.2: *Reverse authentication requests can be forwarded by the server to a code module provider that is more privileged than itself, along with the* rendezvous *port.*

Chapter 5

Server provided IOCTL handlers

Implementing the `ioctl` (*IO control*) system call in a user-space server is problematic. This is because `ioctl` has a variadic argument that may be a memory address to an arbitrarily complex data structure spread out in memory. To read or write to this data structure, the server needs access to the client's memory. But as such access is privileged, only privileged servers would be able to implement `ioctl`.

To get around this, the Hurd relies on the support of the GNU C Library, which does all the necessary memory accesses, and marshals the `ioctl` call into an RPC specific to the requested IOCTL operation. The functions responsible for this are called *IOCTL handlers*.

This fixes the problem at hand, but leads to a new limitation: adding a new IOCTL operation requires that either the client program or the GNU C Library is modified.

This is where mobile code comes in. Instead of relying on hard-coded IOCTL handlers, new handlers can be dynamically loaded from the server. The only requirement then, is that the client trusts the provided code. This allows an unprivileged user to add new IOCTL operations without modifying the GNU C Library or the client program. It also makes it possible to serve other users as long as the code is provided by a source that is trusted by those users.

I will start by giving background on how the `ioctl` system call is implemented in a traditional monolithic kernel, namely Linux. Then, I will give background on how it is currently implemented in the Hurd, and present the limitations of this implementation compared with the implementation in Linux. I will then show how a mobile code implementation is equivalent to a monolithic implementation, with the help of an intermediate hypothetical multi-server implementation. Finally, I will go through the specifics of my mobile code implementation, and the difficulties I encountered during the course of implementing it.

```
#include <ioctl.h>        /* For ioctl() and FIONREAD.  */
#include <fcntl.h>        /* For open() and O_RDONLY.   */

int amount;
int fd;

fd = open ("foo", O_RDONLY);      /* Open file foo.  */
ioctl (fd, FIONREAD, &amount);    /* Call FIONREAD to get how
                                     many bytes can be read
                                     without blocking.  */
```

Listing 5.1: *Example IOCTL call*

5.1 IOCTLs in monolithic kernels

In this section, I will provide an overview of the `ioctl` system call, and how it is implemented in the monolithic Unix-like system Linux, as described in the book *Linux Device Drivers*[10].

The `ioctl` system call first appeared in *Seventh Edition Unix*[8]. Its primary purpose is to provide device specific operations to files that are hard to express through normal file operations, such as reading and writing. For instance, the *GNU C Library manual*[11] lists the following example operations:

- Change the font of a terminal
- Ejecting a disk from a drive
- Play an audio track from a CD-ROM drive
- Change routing tables for a network

Such an operation is referred to as an *IOCTL*. Each IOCTL is specified by a code number, which is passed to the `ioctl` system call when invoking it. These codes are named by macros, which are defined in header files, so that calling programs can include them. Listing 5.1 demonstrates how an IOCTL is called.

Each IOCTL also takes an integer argument, which it can either ignore, use as a value, or use as a memory pointer. If it is a pointer, then the IOCTL can use it to return data by writing to the memory location. It can also use it to access more data than can be encoded into a fixed-width integer. This data may contain additional pointers, which enables the IOCTL to manipulate arbitrarily complex data structures.

The details of the code number varies from system to system, but in general it consists of:

Device class A number specifying the type of device that the IOCTL can be applied to. Sometimes referred to as the IOCTL's *type*, or *magic number*. The number is usually derived from the first letter of the device class, such as *t* for terminal, or *u* for USB.

Operation A number that specifies the requested operation. They are usually assigned sequentially.

Direction This determines in which direction the data is to be transferred. It can be none, read, write, or both. The direction is none if the argument is not a pointer.

Size The size of the referred data to be read or written. It is only applicable if the argument is a pointer.

When the system call is made, the IOCTL code is passed directly to the device driver. It is up to the driver to give the code a meaning. In particular, there's nothing stopping two different device drivers from responding to the same code in completely different ways. This situation will be referred to as an *IOCTL clash*.

The driver is also responsible for interpreting the IOCTL argument, though supporting procedures can make use of the size information in the IOCTL code to handle reading or writing to the referenced memory in a generic manner. However, the size can only describe flat data and not arbitrarily complex data structures, so drivers must have complete access to the memory of the calling process if the argument's type cannot be described in the IOCTL code.

All this enables a device driver to define arbitrary operations without changing the kernel or any other device drivers. This is the key feature of the `ioctl` system call, as it allows third-party developers to independently add new device interfaces.

5.2 IOCTLs in the Hurd

Here, I will describe how the `ioctl` system call is currently implemented in the Hurd, which I discovered by studying the source code of the Hurd[5] and the GNU C Library[4].

Accessing client memory is not a problem for drivers implemented in the kernel, as is normally the case for traditional Unix-like kernels. But for device drivers implemented in user-space—as is possible in the Hurd—only a privileged process has access to the memory of another process. And even if the device driver is privileged, it is not aware which process is making the request. Note that the user-space device drivers are typically not self-contained, rather they most likely extend some Mach driver to support a more Unix-like interface.

Therefore, in the implementation of the `ioctl` system call in the Hurd, each IOCTL is converted to a proper RPC by the GNU C Library. The Hurd has a range of RPC message IDs reserved for just this purpose. The range is split so that each possible device class has its own block big enough to hold one RPC per possible operation in the class. The IOCTL argument is also marshaled into the RPC message using the size information stored in the code of the IOCTL. This conversion is enough to handle most IOCTLs, but there are three cases that cannot be handled this way.

Firstly, some IOCTLs operate on functionality which in the Hurd is implemented by the GNU C Library itself, and not in the kernel as would be the case on other Unix-like systems. For instance, calling `FIOCLEX` on a file descriptor makes the GNU C Library close it if the `exec` system call is later invoked. However, such operations cannot be provided by a device driver, since their implementations depend on the internals of the GNU C Library.

Secondly, some IOCTLs take arguments that cannot be encoded in their code. For instance, if the size of the argument is too large to fit in the code, or if it contains additional pointers which must also be dereferenced.

Thirdly, if the IOCTL can be implemented using RPCs that are already defined in other interfaces, then using the standard RPC conversion would lead to two types of RPCs that do the same thing. For instance, `FIONREAD` is implemented by calling the RPC `io_readable`, which return the number of bytes that can be read without blocking the caller.

In these cases the GNU C Library has a list of *IOCTL handlers*, which are functions that can handle a range of such IOCTLs properly. The list also associates each handler with the range of operations it can handle.

The GNU C Library's `ioctl` function will call the first handler it finds whose range contains the IOCTL code. If the handler cannot handle the operation after all, it can return an error that states this. If the IOCTL was not handled by any handler, the function will try the standard RPC conversion.

5.3 Limitations of IOCTLs in the Hurd

In this section, I will list which features I have found in the `ioctl` system call of a traditional Unix-like system, but are not present in the Hurd's current implementation.

5.3.1 Automatic inclusion

As the GNU C Library is responsible for converting IOCTL requests into RPCs, the only way for a user to provide additional IOCTL types is to override the library, and add the required handlers. Doing this requires control over which libraries the client program is linked to, which servers generally do not have.

Such overrides can be done by using the `LD_PRELOAD` environment variable.[2] And doing it this way does not require that the client's program is modified or relinked in any way. The problem with this is that setting `LD_PRELOAD` only takes effect when a process is started, so it cannot be used to add handlers in already running clients.

But even if that was possible, no such action is required in traditional Unix-like systems. That is, the fundamental problem is that support for new IOCTL types in the Hurd is not *automatic*.

5.3.2 Not device specific

The GNU C Library assumes that there are no clashes. It always uses the first handler it finds that claims that it can handle a particular IOCTL. This means that if two handlers that handle two clashing IOCTLs, one of the handlers is always preferred over the other.

In Linux, the device driver is responsible for interpreting the IOCTL. And so, the call will always be interpreted as the IOCTL that applies to the called device. [10]

5.4 From monolithic to mobile code, step by step

Here I will demonstrate how mobile code can be used to remove the limitations listed above, while still allowing unprivileged users to implement arbitrary IOCTLs.

I will show this by first analysing a hypothetical implementation modeled after a monolithic kernel implementation, which does not have these limitations, but which other users cannot use if run by an unprivileged user.

I will then present a second hypothetical implementation, which uses *privilege separation* to break out the parts that require privileges into a separate server.

Finally, I will show how mobile code can simplify that implementation, and accomplish the same thing more conveniently, with no new requirements, and virtually no new disadvantages. Naturally, it supports the missing features, since it is functionally equivalent to the traditional implementation.

5.4.1 Privileged client-aware servers

An obvious implementation would be to mimic a traditional Unix-like system, that is, make the server aware of the client process, and require that the server has access to its memory.

One disadvantage compared to the translation done in the GNU C Library, is that access to the memory of the client is indirect. It needs to make use of virtual memory operations to make the relevant memory pages of the client available to the server, making it less efficient. In addition, this memory will most likely be mapped to a different address, so for each memory access the server must recompute the address, which is much more awkward.

However, the most glaring disadvantage is that it goes against one of the central goals of the Hurd, which is to allow normal, unprivileged users to extend the system. This is actually worse than the current situation where normal users can implement a predefined class of IOCTLs, but not arbitrary ones.

5.4.2 Privileged client-aware translation servers

It is clear that memory access is a requirement to implement IOCTL operations which take pointer arguments. However, like the IOCTL handlers do in the GNU C Library, a separate server can marshal the IOCTL request into an RPC message, and then forward it to the server that actually implements the IOCTL.

Now, only this translation server needs the privilege to access the memory of the client process. This is an example of a technique known as *privilege separation*, which has been successfully used elsewhere to factor out required privileges into separate components—though for a different reason.[15]

The obvious disadvantage is that an extra RPC now has to be made to the translation server with the unmarshaled IOCTL request. That is, there is an additional indirection per request.

Also the client must now maintain a separate port to the translation server. It is not possible to send an unmarshaled IOCTL to an unprivileged server that then forwards it to the privileged translation server, without also making

it possible for the server to make arbitrary accesses to the client's memory by sending bogus request. That is, it can escalate the server's privileges.

5.4.3 Mobile code IOCTL handlers

The translation server requires access to the client's memory, which is the same requirement that is needed to load code into another process, as was argued in section 4.3.1. Therefore, the translation server can instead provide IOCTL handlers—like those already present in the GNU C Library—as mobile code.

Since the translation code now runs inside the client, it now has direct access to its memory. And since the IOCTL request is also directly available, there is no longer any need for the extra RPC indirection.

The port to the translation server can be discarded once the handler code has been loaded. However, a reference to the loaded code must still be associated with the file it is specific to, so this is not an advantage in itself.

The only disadvantage is that the client must now take time to load the code, which is an additional overhead. This should be weighed against the decreased cost per IOCTL request. But since performance is not critical, I have not done any measurements to determine whether mobile code leads to increased performance in all cases, or only once a certain number of requests have been made.

5.5 Implementation

In this section, I will describe how I made it possible for servers to provide mobile IOCTL handlers. The implementation required adding the new `ioctl_handler` protocol to the Hurd, as well as several changes to the GNU C Library.

5.5.1 Protocol

The protocol for obtaining a port to a file's IOCTL handler module uses reverse authentication, as was described in section 4.3.1. It uses three new messages in addition to the four messages used by the standard authentication protocol. These are: `ioctl_handler_request`, `ioctl_handler_acknowledge`, and `ioctl_handler_reply`.

Essentially the protocol uses the authentication protocol to establish a secure channel on which the module provider can return a port to the IOCTL module. Not only is this channel safe from man-in-the-middle attacks, but the authentication server also provides the user and group identifiers of the module provider. This enables the client to determine whether it can trust the server to provide a module that is safe to load.

The protocol is as follows:

1. The client sends an `ioctl_handler_request` message to the server, which carries a *rendezvous* port.

2. The server sends an `ioctl_handler_acknowledge` message in reply. This is needed so that the client does not wait indefinitely for a reply to the `auth_server_authenticate` message if the server does not support this protocol.

3. The client sends an `auth_server_authenticate` message with the *rendezvous* port and a reply port to the authentication server. Note the reversal of the roles of client and server from the normal authentication protocol.

4. The server sends an `auth_user_authenticate` message with the *rendezvous* port it got in step 1 to the authentication server.

5. The authentication server matches up the requests using the *rendezvous* port, and returns the reply port to the server and the server's credentials to the client.

6. The server finally sends an `ioctl_handler_reply` message to the reply port with a port to a code module that exports an IOCTL handler function named `hurd_ioctl_handler`.

After this the client can examine the server's credentials to determine whether it can trust it. The policy implemented in the `ioctl` function is to trust the server if it is run by the root user or by the same user as the client.

It is also possible for the server to delegate this task to a privileged server by simply forwarding the initial `ioctl_handler_request` message.

5.5.2 Changes to the GNU C Library

The changes I made to the GNU C Library were mostly done to the `ioctl` function itself. It now uses the `ioctl_handler` protocol to get a port to the module file from the server. If no module is provided, or the IOCTL handler does not accept the IOCTL, it is handled as before.

Changes were also made to the *file descriptor* table. File descriptors are in most other Unix-like systems kernel protected references to open files, and have characteristics similar to Mach's ports. In the Hurd, file descriptors are implemented by the GNU C Library, and consists primarily of a port to the server implementing the file, but also an alternative port and auxiliary information used for purposes that are not relevant for this thesis.

Module management

I extended the `ioctl` function to retrieve and load a file's IOCTL handler module when it is called for the first time on a particular file descriptor. An opaque reference to the module is then stored in the file descriptor table, so it can later be unloaded when the file descriptor is closed.

If the module fails to load for any reason, a dummy IOCTL handler that rejects all requests is installed in the file descriptor table. I did this so that no further attempts are made to request a handler module on later calls to `ioctl`.

If the port stored in the file descriptor table is ever exchanged for any reason, the module is unloaded and the reference cleared so that later calls to `ioctl` loads the module provided by this potentially different file.

Consistency

The GNU C Library can be used by multiple threads concurrently. This means that the port to the underlying file server stored in a file descriptor can be changed whenever the file descriptor is not locked.

However, locking a descriptor for extended periods harms concurrency. Not only does it block other uses of the same file descriptor, it also blocks asynchronous signals from other processes. In particular, a file descriptor should not be locked while waiting for another process, for instance, when waiting for a reply to an RPC.

To reduce the duration of locks, GNU C Library code typically makes a local copy of the file descriptor's contents. The copy can then be used as a snapshot of the file descriptor's state at the time of the copy, and as long as the file descriptor is not modified by the current operation after the snapshot, it will act as if the file descriptor was locked during the entire operation. I will refer to this process as *resolution* of the file descriptor.

The file descriptor must be resolved before searching for an accepting handler. This is because the code module is specific to the file descriptor's underlying port. If done after, the handler that accepted the IOCTL may no longer be consistent with the file descriptor's port, as it can change in the meantime.

This is problematic since some handlers operate on the file descriptor itself and therefore must have it locked. However, relocking the descriptor means an additional resolution of the file descriptor, which opens the possibility of the port being changed, which in turn could affect the choice of the handler.

For instance, consider a thread making an IOCTL call to a file which does not provide any mobile handlers, instead it accepts the RPC that the call is converted to by default. It has already been determined by the `ioctl` function that this file provides no handlers for this IOCTL, but the RPC has not yet been sent. In the meantime, a second thread changes the file descriptor to refer to a different file that does override the IOCTL with a handler, which converts the IOCTL to a different RPC. The first thread then proceeds with the default conversion to RPC. However, the new file will not accept the default RPC and the IOCTL fails, whereas it would have succeeded if the file descriptor had been properly locked either before or after it was modified by the second thread.

To deal with this issue, I have changed the `ioctl` function to leave the resolution to the accepting handler, and keep the file descriptor locked while searching for one. If a handler accepts the IOCTL it should unlock the descriptor before it does any RPCs. This required changing the signature for handler functions, and changing existing handlers in the GNU C Library accordingly.

Distinguishing rejection from failure

When a handler does not accept an IOCTL, it fails with the error code `ENOTTY`. This is problematic if a handler makes an RPC that fails with the same error code, as even if the handler does accept the IOCTL, the remaining handlers are still called.

The problem is aggravated when the accepting handler is responsible for unlocking the file descriptor, as now it will be passed unlocked to the remaining handlers and the default RPC conversion.

To work around this issue, I have changed handlers to return two separate error codes. The first one for whether the IOCTL was rejected, and the second one for whether the IOCTL failed.

Code listing 5.2 shows how the final type used for IOCTL handler functions, and how it has changed since the original.

```
int (*ioctl_handler_t) (int fd, int code, void *arg);
```

(a) *The original IOCTL handler type, which has the same type as the* `ioctl` *function itself. Argument* `arg` *is a pointer, but its value may still be used as an integer.*

```
error_t (*ioctl_handler_t) (int fd,
                            struct hurd_fd *d, void *crit,
                            int code, void *arg,
                            int *result);
```

(b) *The new IOCTL handler type, where* `d` *is the locked file descriptor and* `crit` *is a critical section lock needed to lock* `d`*. The return value is now the rejection error, while the original return value will be stored in* `result`*.*

Listing 5.2: *IOCTL handler function type*

Chapter 6

Conclusions

In this chapter I will conclude my thesis by summarizing the results of the three main parts of my thesis, as well as pointing to future directions for mobility within the Hurd.

6.1 Transferring dependencies

I have argued how the dependency transfer for stores is deficient to use for more general mobility use-cases, and perhaps for stores as well. The assumption that an external dependency is innocuous or enforced does not hold in all cases, and the practise of transferring inactive stores assumes that symbolic names resolve to the same objects, which may not be the case.

Instead, I followed up on the assumption used by stores that the root user will already have access to the dependencies, but argued more specifically that if the client has access to the server's memory and ports, then it does indeed already have access to the dependencies. This lead to the conclusion that it is reasonably safe to assume that the client has access to the dependencies, if also run by the same user as the server. It is an assumption that does not hold in all cases, but this could be fixed by querying the Hurd's process server directly, which will be possible if it in the future is extended to support such queries.

I also explained that comparing objects in the Hurd is currently only possible if they are identical, which is rare in the Hurd as objects are most often referenced indirectly through object handles. Comparing handles in general requires cooperation from the objects themselves, which no interface in the Hurd currently supports. And even if this was possible, doing so safely is hard for file handles, which are the most common objects found in the Hurd. This is because determining whether a process can access a file is problematic, since the path to the file can be considered a password, which itself needs protection.

6.2 Transferring code

The code loading mechanism used by stores was found unsuitable for use-cases where loading code should work even if client and server are executing in different environments.

The solution was to reference which code modules to load directly with ports instead of using symbolic names. As loading arbitrary code supplied by the server can be used to gain access to the client's memory and ports, and vice versa, this became the criterion for when loading the code is secure. But as the authentication protocol is not designed to be used to authenticate the server to the client, I had to devise a workaround solution: reverse authentication.

I also found that the task of supplying the code can be delegated to a separate third server. This makes it possible for two unprivileged users to cooperate using mobile code, as long as they are supported by a third, privileged user.

6.3 Server provided IOCTL handlers

I used mobile code to make the `ioctl` system call more extensible. This is a concrete implementation of loading code by port and reverse authentication. It makes the Hurd's `ioctl` system call as powerful as the traditional implementation needed in monolithic Unix-like systems such as Linux. The privilege required to provide code modules that add new types of operations is comparable to that needed in those systems, but the ability to implement these operations are available to regular, unprivileged processes.

While the infrastructure is somewhat complex, the IOCTL handlers that add new operation types are themselves pretty straightforward, making it easy to make use of it.

But there is still plenty of room for improvement. In particular, if the requirement to pass a locked file descriptor to the IOCTL handlers can be relaxed, they can be made even simpler.

My patches to the Hurd and the GNU C Library work, but as the GNU C Library in particular is complicated, they need further review before they can be included in a release.

6.4 Future directions

Overall this thesis has laid the groundwork for future use-cases that make use of mobility. One interesting possibility is to use mobility for optimizing the use of Hurd objects by loading them into the client instead of using them remotely through IPC. The main issue remaining is how to make a framework that allows this, while still being similar enough to how servers currently implement objects, so it becomes easy to port existing implementations to the new framework.

Of course, the techniques used in this thesis should also be applied the existing mobile objects, namely stores, as they too would benefit from them.

Bibliography

[1] *dlopen(3) Manual Page*. From the Debian GNU/Hurd package `manpages-dev` version 3.22-1.

[2] *ld.so(8) Manual Page*. From the Debian GNU/Hurd package `libc0.3` version 2.9-23.

[3] *passwd(5) Manual Page*. From the Debian GNU/Hurd package `passwd` version 1:4.1.4.2-1.

[4] The GNU C Library Source Code. From the Debian GNU/Hurd source package `eglibc` version 2.9.18.

[5] The GNU Hurd Source Code. From Git repository, commit `08aa7edb495445c0bfa54cb5d207e85c1df9008a`.

[6] The GNU Project Debugger Source Code. Version 7.0.

[7] Grammatically speaking... `http://www.gnu.org/software/hurd/hurd/what_is_the_gnu_hurd/gramatically_speaking.html`, 2008. Verified: 2009-11-08.

[8] Bell Telephone Laboratories, Inc. *UNIX Time-sharing system: Unix Programmer's Manual*, 7th edition, January 1979.

[9] M. I. Bushnell. Towards a new strategy of OS design. *GNU's Bulletin*, 1(16), January 1994.

[10] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*, chapter 6, pages 135–147. O'Reilly Media, Inc., 005 Gravenstein Highway North, Sebastopol, CA 95472, 3rd edition, February 2005.

[11] Free Software Foundation, Inc. *The GNU C Library*, 0.12 edition, March 2009. For version 2.9 of the GNU C Library.

[12] Free Software Foundation, Inc. *The GNU Hurd Reference Manual*, 2009. For version 0.2 of the GNU Hurd.

[13] K. Loepere. *Mach 3 Kernel Interfaces*, 1992.

[14] K. Loepere. *Mach 3 Kernel Principles*, 1992.

[15] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the 12th USENIX Security Symposium*, pages 231–241, Berkeley, CA, USA, 2003. USENIX Association.

[16] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, September 1975.

[17] Neal H. Walfield and Marcus Brinkmann. A critique of the GNU Hurd multi-server operating system. *SIGOPS Oper. Syst. Rev.*, 41(4):30–39, 2007.