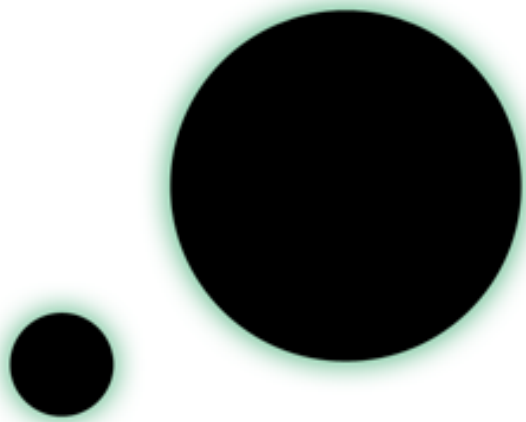
Trustworthy Systems
http://ts.data61.csiro.au/



RefOS — Reference Design For A Microkernel Based Operating System

Kevin Elphinstone
Kevin.Elphinstone@data61.csiro.au

August 31, 2016

Abstract

The design of a multi-server operating system is crucial to its efficiency, security and dependability. Increasing usage of embedded systems and microcontroller chips has increased the interest from both industrial and academic communities in microkernel-based component-based software systems. This has resulted in heightened interest in developing better systems and more advanced microkernels.

This document outlines a basic reference design for system components and interfaces for an operating system. The protocols have been designed with an L4 like microkernel in mind and assume functionality such as interprocess communcation, capability-based access models and virtual memory support. This reference design aims to standardise the protocols for tasks often required to build a system such as resource sharing, virtual memory management and naming. Operating systems may implement these protocols and extend them to incorporate more features.

A sample implementation of the following protocols is provided in addition to this document in the form of a simple operating system which runs simple client programs. The sample implementation, named RefOS, runs on the seL4 microkernel, which adds an extra dimension of dependability. Although this document outlines the major design strategies that RefOS employs, the doxygen code documentation provides additional details about RefOS.

Contributors

The following people have contributed to this document:

- Kevin Elphinstone
- Andi Cheng
- Xi Chen
- Alexander Wharton

Contents

Chapter 1

Introduction

This document explains the abstract protocols for a multi-server microkernel based operating system.

The overall design of RefOS revolves around the abstraction of a dataspace. A dataspace is a memory space (a series of bytes) representing anything from physical RAM to file contents on a disk to a device or even to a random number generator. The concept is analogous to UNIX files which may represent anything from `/usr/bin/sh` to `/dev/audio` to `/dev/urandom`.

The dataspace abstraction which dataspace servers provide paves the way for more complicated infrastructure such as sharing with complex trust relationships, memory management, file systems, and distributed naming. On top of this additional infrastructure, an operating system could implement features such as POSIX standard system calls, ports of existing software, drivers and display servers.

1.1 Terms

This section defines a number of terms that are used in this document.

`Methods` are conceptual "function calls" although they are usually implemented via communication with another process in which case they are actually remote procedure calls. Regardless of the underlying implementation, methods here refer to the actual procedures of an interface which get invoked.

`Interfaces` are a collection of (usually related) methods that usually serve to abstract a conceptual object. For instance, the `C` standard library functions `fopen`, `fscanf`, `fprintf`, `fclose` and so on form an abstract interface and provide abstraction for a file object.

`Message sequences` are ordered sequences of method invocations, replies, event notifications or operations which may achieve some overall result or goal. Example message sequences using the described protocols for commonly needed goals are presented in this document.

`Protocols` here refer to a collection of interfaces, their methods, message sequences and system components.

`Dataspaces` are abstractions over memory objects, analogous to UNIX files. A dataspace may be opened and closed, written to and read from, mapped into client memory and may represent a device, a file, or simply anonymous memory.

`Servers` are normal processes which have registered themselves using the naming service so that other client processes may find them. Once a client finds a process, the client may connect and establish a session with the server and the server is given access to the notification of client deaths in order to do any client bookkeeping that it may require.

`Dataspace servers` (henceforth also referred to as *dataservers*) are servers which implement the dataspace interface (or a subset of it) in order to provide a dataspace service to clients. Dataspace servers are often designed to provide dataspaces which represent a common concept. For instance, a file server may implement the dataspace interface for files stored on disk, and an audio driver may serve audio bytestream dataspaces.

`Process servers` are servers which implement the process server interface and provide process abstraction. In practice, the process server may also need to implement additional interfaces and provide

additional functionality in order for a system to start. For instance, in RefOS the process server also does naming and console input and output, implements the dataspace interface for anonymous memory and acts as the memory manager.

Pagers are servers which implement the pager service interface, which is very closely related to the dataspace interface and can be considered an optional feature of a dataspace server. Dataspace servers which act as pagers may have their dataspaces mapped directly into client virtual memory.

1.2 Microkernel Design

RefOS is designed with an L4 family microkernel in mind and assumes the existence of features such as interprocess communication and capabilities. This section describes these assumed features. These features are available in advanced L4-based microkernels such as seL4.

1.2.1 Interprocess Communication

- **Synchronous Calls** are used to invoke methods implemented by a server. They involve a badged capability invocation of a synchronous endpoint that identifies the server and represents the authority of the caller. The caller blocks until the server responds. A caller implicitly trusts the server it calls. Calling an untrusted server is possible, but doing so should be avoided in general.

  Exceptions, such as page faults, are propagated via sending and blocking on a synchronous endpoint.

- **Reply** protocols occur via reply capabilities. A call via invoking a synchronous endpoint generates a reply capability for the server to respond to the caller. The callee can send a response to the original caller and unblock the caller by replying with the reply capability. Reply capabilities are guaranteed either to succeed or to fail and not to block. Non-blocking replies are required for a trusted server to safely reply to an untrusted client.

- **Asynchronous Notifications** are sent via a send to an asynchronous endpoint capability. Notifications are non-blocking with at-least-once semantics (the bits are OR'd together). The asynchronous endpoint can be used to notify an untrusted client. To send messages (as opposed to just notifications) to an untrusted client, shared memory must be set up between the server and the client. Care must be taken to avoid unfavourable results like blocking and denial of service on shared memory buffers which may fill up.

1.2.2 Capability Access Model

RefOS assumes the microkernel that it runs on provides support for a capability-based access model, and more importantly, the ability to transfer badged capabilities via interprocess communication.

The basic concept of a capability is that a capability is an object that contains the permissions to access a particular object. This is a similar concept to UNIX file descriptors which indicate the access that a process has to a file. Just like UNIX file descriptor tables, capability objects are kept in kernel memory. User programs only have handles to such objects and cannot directly manipulate them.

RefOS is designed under the assumption that synchronous call and reply interprocess communication messages may contain capabilities. RefOS also assumes that there is some way to store a minimal amount of immutable information in such capabilities and that receiving processes may read the immutable information from the capabilities that they receive via interprocess communication.

In the sample implementation given using the seL4 microkernel, these assumptions are implemented as kernel interprocess communication endpoint *badges*. Endpoint capabilities may be badged with an integer, and the badge of an endpoint once it has been set is immutable. When a badged endpoint capability is sent along the endpoint that it is pointing to, the kernel *unwraps* the badge to be read. Only

the process which created the badge can read it back via interprocess communication. RefOS makes heavy use of these badges in order for servers to track server objects (such as windows and dataspaces).

1.2.3 Virtual Memory Support

RefOS assumes that the underlying kernel exposes virtual memory management. This is a standard feature of microkernels. RefOS also assumes that the kernel is able to deliver page fault exceptions via interprocess communication to a memory manager thread and is able to resolve the fault at a later stage by replying to the interprocess communication.

1.3 Notation

This section outlines the notation used to describe RefOS's protocols. Specifically, it looks at the messaging among components that is used to develop higher-level protocols.

1.3.1 Operation Types

Figure 1.1 presents the four basic operations RefOS employs (and indirectly attempts to limit itself to). They are synchronous calls (a method invocation on an object), replies, asynchronous notifications, and kernel system calls that asynchronously modify the state of another process (such as mapping memory directly into the virtual address space of another process).

Synchronous and asynchronous call and reply operations may be implemented directly using the microkernel features described above, such as interprocess communication, capabilities, capabilities via interprocess communication and virtual memory operations.

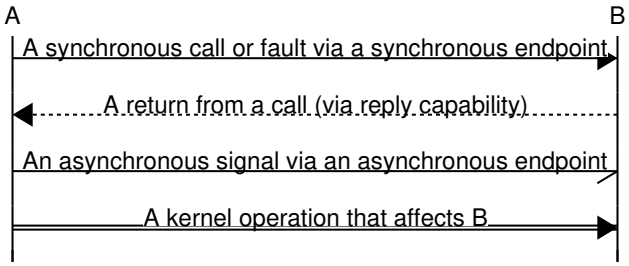


Figure 1.1: Message Type Arrows.

1.3.2 Message Notation

Protocols are shown as series of messages. Messages are implemented by operations, and the types of different messages directly correlate to the operation types. This section outlines the notation which this document will henceforth adhere to when describing the various forms of messages that form RefOS's protocol.

Method Invocation

Most of the interface methods are synchronous. The client making the call is considered to be less trusted than the server that is receiving the call. When a client makes a call, it is blocked until the server finishes handling the call and replies via the reply operation. When the server replies, the call is said to be finished and the client resumes execution. Method invocation is implemented with a synchronous call operation via a synchronous endpoint. Capabilities may be passed via method invocation and also via the corresponding reply.

For synchronous method invocations by a less trusted client to a more trusted server, the following notation is used:

```
server_interface_C.method(arg1, arg2, ...)
        ⇒ (val1, val2, val3 ...)
```

Where:

- `server` indicates the name of the server that is receiving and handling the method invocation via an endpoint. For instance, `procserv` and `dataserv` in method descriptions refer to a process server and a data server respectively
- `interface` refers to the name of the interface that the server implements
- `method(arg1, arg2, ...)` refers to the name of the interface method and the arguments of the method call
- `(val1, val2, val3 ...)` represents the set of return values, output variables and/or reply capabilities of the method invocation

For example, the notation `dataserv_dataspace_C.open` means an invocation of the `open` method of the `dataspace` interface on the server called `dataserv`. Note that any `server` can implement a particular `interface`. The encoding of method and arguments in the actual message is left as an implementation detail.

Figure 1.2 describes an example protocol description (in the form of a message sequence diagram) of a client process invoking a method via a synchronous message operation:

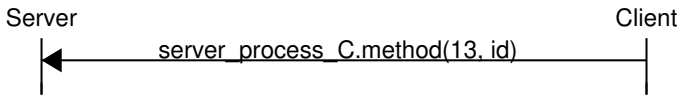


Figure 1.2: Example protocol message sequence diagram

Event Notification

Some servers need a method of notifying another process without blocking and waiting for the other process to finish processing the notification. Asynchronous signal operations via asynchronous endpoints are used to send such notifications to other servers. Capabilities may not be transferred via a notification.

Note that RefOS does not make the assumption that the kernel is able to pass message contents via the notification itself, just that the kernel is able to pass just the notification. In order for the actual message to be delivered along with the notification, the implementation of the protocol may for instance make use of a shared buffer.

The following notation is used for event notifications:

```
server_event_C.notify(arg1, ...)
```

This notation is purposely very similar to a method invocation as those two are similar in concept in that they are both interprocess messages. The different message sequence diagram arrows used for synchronous calls and asynchronous notifications clearly distinguish the two.

Abstracted Protocols

Sometimes protocols are of different levels of abstractions. In order to separate two closely related protocols in one's sequence diagram, the representation as indicated in Figure 1.3 is used:

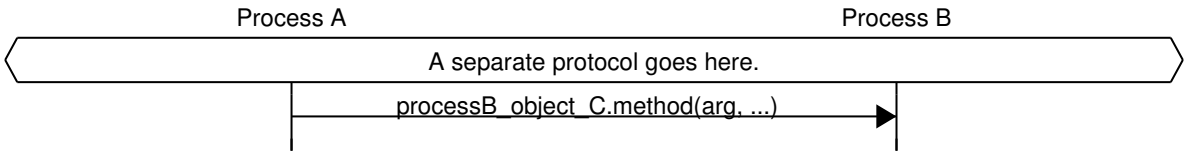


Figure 1.3: Example of separation of protocols

Chapter 2

Interfaces

This chapter explains the interfaces which make up the RefOS protocol. These interfaces set the language with which components in the system interact. Interfaces are intended to provide abstractions to manage conceptual objects such as a processes, files and devices.

Further implementations beyond RefOS may choose to extend these interfaces to provide extra system functionality and to attach additional interfaces.

2.1 Objects

Servers implement interfaces which provide management (creation, destruction, monitoring, sharing and manipulation) of objects. For example, an audio device may implement the dataspace interface for UNIX `/dev/audio` which gives the abstraction to manage audio objects.

In RefOS, the access to an object (the permission to invoke the methods which manage the object) is implemented using a badged endpoint capability. The badge number is used to uniquely identify the instance of an object. Since most expected implementations of these protocols likely will use a similar method to track access permissions, the term capability may be used interchangably with object in this document. For example ”window capability” means a capability which represents the access to a window object.

Note that in implementations, objects may be merged in some cases. For example, the process server's `liveness`, `anon` or `process` object capabilities may be merged for simplification.

`process` is an object which is most likely maintained by the process server (`procserv`) and represents a process. If something has access to a process object, it may perform operations involving that process such as killing that process and calling methods on behalf of that process.

`liveness` is an object representing the identity of a process. If something has access to a liveness object of a process, it can be notified of the process's death and can request an ID to uniquely identify the process, but it cannot kill the process or pretend to be the process.

`anon` is an object representing the ”address” of a server, and it is used to establish a session connection to the server.

`session` is an object representing an active connection session to a server. If something has access to a session object, it can invoke methods on the server on behalf of the session client.

`dataspace` is an object that represents a dataspace. The dataspace itself may represent anything that may be modeled as a series of bytes including devices, RAM and files. If something has access to a dataspace object, it may read from the dataspace object, write to the dataspace object and execute the dataspace object by mapping a memory window to the dataspace, closing the dataspace, or deleting the dataspace. Performing these operations is dependent on the dataspace permissions.

`memwindow` is an object that represents an address space range (i.e. a memory window) segment in a process's virtual memory. If something has access to a memory window object, it may perform operations on the memory window object such as mapping the memory window to a dataspace and mapping a frame into the memory window.

2.2 Protocols

This section describes a number of important protocols that RefOS employs. As noted in Section 1.3.2 of this document, each protocol description consists of the server that is receiving and handling the method invocation via an endpoint, the name of the interface that the server implements, the name of the method call and the arguments that are passed to the method call and the return values, output variables and/or reply capabilities of the method invocation. Note that for simplification some method names differ slightly between this document and RefOS's implementation.

2.2.1 Process Server Interface

The process server interface provides the abstraction for managing processes and threads. The abstraction includes management of processes' virtual memory, memory window creation and deletion, process creation and deletion, death notification of client processes and thread management. Note that in implementations `procserv_session_C` could be the same capability as `procserv_process_C` in which case the process server is connectionless. `procserv_session_C` may also be shared with `procserv_anon_C` for simplification.

`procserv_session_C.watch_client(procserv_liveness_C, death_notify_ep)`
⇒ `(Errorcode, death_id, principle_id)`
Authenticate a new client of a server against the `procserv` and register for death notification.

procserv_liveness_C The new client's liveness capability, which the client has given to the server through session connection

death_notify_ep The asynchronous endpoint through which death notification occurs

death_id The unique client ID that the server will receive on death notification

principle_id The ID used for permission checking (optional)

`procserv_session_C.unwatch_client(procserv_liveness_C)`
⇒ `(Errorcode)`
Stop watching a client and remove its death notifications.

`procserv_session_C.create_mem_window(base_vaddr, size, permissions, flags)`
⇒ `(ErrorCode, procserv_window_C)`
Create a new memory window segment for the calling client. Note that clients may only create memory windows for their own address space and alignment restrictions may apply here due to implementation and/or hardware restrictions. In the RefOS client environment, a valid memory window segment must be covering any virtual address ranges before any mapping can be performed (including dataspace and device frame mappings).

base_vaddr The window base address in the calling client's VSpace

size The size of the memory window

permissions The read and write permission flags (optional)

flags The extra flags bitmask - cached versus uncached window for example (optional)

`procserv_session_C.resize_mem_window(procserv_window_C, size)`
⇒ `ErrorCode`
Resize a memory window segment. This is an optional feature, which may be useful for implementing dynamic heap memory allocation on clients.

size The size of the memory window to resize to

`procserv_session_C.delete_mem_window(procserv_window_C)`
⇒ `ErrorCode`
Delete a memory window segment.

`procserv_session_C.register_as_pager(procserv_window_C, fault_notify_ep)`
⇒ `(ErrorCode, window_id)`
Register to receive faults for the presented window. The returned `window_id` is an integer used during notification in order for the pager to be able to identify which window faulted. `window_id` must be unique for each pager, although each `window_id` may also be implemented to be unique across the entire system.

fault_notify_ep The asynchronous endpoint which fault notifications are to be sent through

window_id The unique ID of the window which is used to identify which window faulted. The server most likely has to record this ID to handle faults correctly

`procserv_session_C.unregister_as_pager(procserv_window_C)`
⇒ `(ErrorCode)`
Unregister to stop being the pager for a client process's memory window

`procserv_session_C.window_map(procserv_window_C, window_offset, src_addr)`
⇒ `ErrorCode`
Map the frame at the given VSpace into a client's faulted window and then resolve the fault and resume execution of the faulting client. This protocol is most commonly used in response to a prior fault notification from the process server, and it also may be used to eagerly map frames into clients before they VMfault.

window_offset The offset into the window to map the frame into

src_addr The address of the source frame in the calling client process's VSpace. This address should contain a valid frame, and page-alignment restrictions may apply for this parameter

`procserv_session_C.new_proc(name, params, block, priority)`
⇒ `(ErrorCode, status)`
Start a new process, blocking or non-blocking.

name The executable file name of the process to start

params The parameters to pass onto the new process

block The flag stating to block or not to block until the child process exits

priority The priority of the new child process

status The exit status of the process (only applicable if blocking)

`procserv_session_C.exit(status)`
⇒ `(ErrorCode)`
Exit and delete the calling process

status The exit status of the calling client process

`procserv_session_C.clone(entry, stack, flags, args)`
⇒ `(ErrorCode, thread_id)`
Start a new thread, sharing the current client process's address space. The child thread will have the same priority as the parent process.

entry The entry point vaddr of the new thread

stack The stack vaddr of the new thread

flags Any thread-related flags

args The thread arguments

thread_id The thread ID of the cloned thread

2.2.2 Server Connection Interface

The server connection interface enables client session connection and disconnection. It may be a good idea during implementation to extend this interface onto any other extra operating system functionality that is common across servers. This could include debug ping, parameter buffer setup and notification buffer setup.

`serv_anon_C.connect(procserv_liveness_C)`
⇒ `(ErrorCode, serv_session_C)`
Connect to a server and establish a session

`serv_session_C.disconnect()`
⇒ `(ErrorCode)`
Disconnect from a server and delete session

2.2.3 Data Server Interface

The data server interface provides the abstraction for management of dataspaces including dataspace creation, dataspace access, dataspace sharing and dataspace manipulation.

`dataserv_session_C.open(char *name, flags, mode, size)`
⇒ `(ErrorCode, dataserv_dataspace_C)`
Open a new dataspace at the dataspace server, which represents a series of bytes. Dataspace mapping methods such as datamap() and init_data() directly or indirectly map the contents of a dataspace into a memory window after which the contents can be read from and written to. The concept of a dataspace in RefOS is similar to a file in UNIX: what a dataspace represents depends on the server that is implementing the interface.

name The name of the dataspace to open

flags The read, write and create flags to open the dataspace with

mode The mode to create a new file with in the case that a new one is created

size The size of dataspace to open - some data servers may ignore this

`dataserv_session_C.close(dataserv_dataspace_C)`
⇒ `ErrorCode`
Close a dataspace belonging to the data server.

`dataserv_session_C.expand(dataserv_dataspace_C, size)`
⇒ `ErrorCode`
Expand a given dataspace. Note that some dataspaces may not support this method as sometimes the size of a dataspace makes no sense (serial input for instance).

size The size to expand the dataspace to

`dataserv_session_C.datamap(dataserv_dataspace_C, procserv_window_C, offset)`
⇒ `ErrorCode`
Request that the data server back the specified window with the specified dataspace. Offset is the offset into the dataspace to be mapped to the start of the window. Note that the dataspace has to be provided by the session used to request the backing of the window.

procserv_window_C Capability to the memory window to map the dataspace contents into

offset The offset in bytes from the beginning of the dataspace

`dataserv_session_C.dataunmap(procserv_window_C)`
⇒ `ErrorCode`
Unmap the contents of the data from the given memory window.

procserv_window_C Capability to the memory window to unmap the dataspace from

`dataserv_session_C.init_data(dest_dataspace_C, dataserv_dataspace_C, offset)`
⇒ `ErrorCode`
Initialise a dataspace by the contents of a source dataspace. The source dataspace is where the content is, and the source dataspace must originate from the invoked dataserver. Whether the destination dataspace and the source dataspace can originate from the same dataserver depends on the dataserver implementation: one should refer to the dataserver documentation. One example use case for this is a memory manager implementing the dataspace for RAM having a block of RAM initialised by an external data source such as a file from a file server.

dest_dataspace_C The dataspace to be initialised with content from the source dataspace

dataserv_dataspace_C The dataserver's own dataspace (where the content is)

offset The content offset into the source dataspace

`dataserv_session_C.have_data(dataserv_dataspace_C, fault_ep)`
⇒ `ErrorCode, data_id`
Call a remote dataspace server to have the remote dataspace server initalised by the contents of the local dataspace server. The local dataspace server must bookkeep the remote dataspace server's ID. The remote dataspace server then will request from the given endpoint content initialisation with the remote dataspace server providing its ID in the notification.

fault_ep The asynchronous endpoint to ask for content initialisation with

data_id The remote endpoint's unique ID number

`dataserv_session_C.unhave_data(dataserv_dataspace_C)`
⇒ `ErrorCode`
Inform the dataserver to stop providing content initialise data for its dataspace.

`dataserv_session_C.provide_data(dataserv_dataspace_C, offset, content_size)`
⇒ `ErrorCode`
Give the content from the local dataserver to the remote dataserver in response to the remote dataserver's earlier notification asking for content. The content is assumed to be in the set up parameter buffer. This call implicitly requires a parameter buffer to be set up, and how this is done is up to the implementation. Even though the notification from the dataserver asking for content uses an ID to identify the dataspace, the reply, for security reasons, gives the actual dataspace capability. The ID may be used securely if the dataserver implementation supports per-client ID checking, and in this situtation a version of this method with an ID instead of a capability could be added.

offset The offset into the remote dataspace to provide content for

content_size The size of the content

`dataserv_process_C.datashare(dataserv_dataspace_C)`
⇒ `ErrorCode`
Share a dataspace of a dataserver with another process. The exact implementation of this is context-based. For example, a server sharing a parameter buffer may implement this method as `share_param_buffer(dataspace_cap)`, implicitly stating the context and implementing multiple share calls for more contexts. Although this method of stating the sharing context is strongly encouraged, the exact method of passing context is left up to the implementation.
There are also a few assumptions here:

- The dataspace is backed by somebody the receiving process trusts (the process server for instance)
- The dataspace is not revocable, so the receiving process does not need to protect itself from an untrusted fault handler on that memory

`dataserv_event_C.pagefault_notify(window_id, offset, op)`
⇒ `ErrorCode`
Send a notification event to a data server indicating which window a page fault occurred in, the offset within that window and the operation attempted (either read or write)

`dataserv_event_C.initdata_notify(data_id, offset)`
⇒ `ErrorCode`
Send a notification event to a data server indicating that a dataspace needs its initial data

`dataserv_event_C.death_notify(death_id)`
⇒ `ErrorCode`
Send a notification event to a data server indicating a client has died and that the resources associated with the client can be cleaned up

2.2.4 Name Server Interface

The name server interface provides a naming protocol. RefOS employs a simple hierachical distributed naming scheme where each server is responsible for a particular prefix under another name server. This allows for simple distributed naming while allowing prefix caching. In RefOS, the process server acts as the root name server.

`nameserv_session_C.register(name, dataserv_anon_C)`
⇒ `ErrorCode`
Create an endpoint and set this endpoint on the root name server in order for clients to be able to find the root name server and connect to it. The anon capability is given to clients looking for the root name server, and then clients make their connection calls through the anon capability to establish a session. Re-registering replaces the current server anon capability.

name The name to register under

dataserv_anon_C The anonymous endpoint to register with

`nameserv_session_C.unregister(name)`
⇒ `ErrorCode`
Unregister a server under a given name so clients are no longer able to find the server under that name. This method invalidates existing anon capabilities.

name The name to unregister for

`nameserv_session_C.resolve_segment(path)`
⇒ `(ErrorCode, dataserv_anon_C, resolved_bytes)`
Return an anon capability if the server is found. This method resolves part of the path string returned as an offset into the path string that has been resolved. The rest of the path string may be resolved by other dataspaces until the system reaches the endpoint to the server that contains the file that it is searching for. This allows for a simple hierachical namespace with distributed naming.

path The path to resolve

resolved_bytes The number of bytes resolved from the start of the path string

2.3 Server Components

RefOS follows the component-based multi-server operating system design. The two main components in the design are the process server and the file server. RefOS implements an additional operating system server which is a dataserver that is responsible for basic device related tasks.

2.3.1 Process Server

The process server is the most trusted component in the system. It runs as the initial kernel thread and does not depend on any other component (this avoids deadlock). The process server implementation is single-threaded. The process server also implements the dataspace interface for anonymous memory and acts as the memory manager.

In RefOS, the process server implements the following interfaces:

- Process server interface (naming, memory windows, processes and so on)
- Dataspace server interface (for anonymous memory)
- Name server interface (in RefOS, the process server acts as the root name server)

2.3.2 File Server

The file server is more trusted than clients, but it is less trusted than the process server (this avoids deadlock). In RefOS, the file server does not use a disk driver and the actual file contents are compiled into the file server executable itself using a cpio archive. The file server acts as the main data server in RefOS.

In RefOS, the file server implements the following interfaces:

- Dataspace server interface (for stored file data)
- Server connection interface (for clients to connect to it)

2.3.3 Console Server

The console server provides serial and EGA input and output functionality, which is exposed through the dataspace interface. The console server also provides terminal emulation for EGA screen output.

In RefOS, the console server implements the following interfaces:

- Dataspace server interface (for serial input and output and EGA screen devices)
- Server connection interface (for clients to connect to it)

2.3.4 Timer Server

The timer server provides timer get time and sleep functionality, which is exposed through the dataspace interface.

In RefOS, the timer server implements the following interfaces:

- Dataspace server interface (for timer devices)
- Server connection interface (for clients to connect to it)

Chapter 3

Message Protocol

3.1 Message Sequences

This section provides a series of example message sequences for common tasks using the protocol interfaces described in Section 2.2 of this document.

3.1.1 Server Registration

Figure 3.1 illustrates the protocol a server uses to register itself under a name.

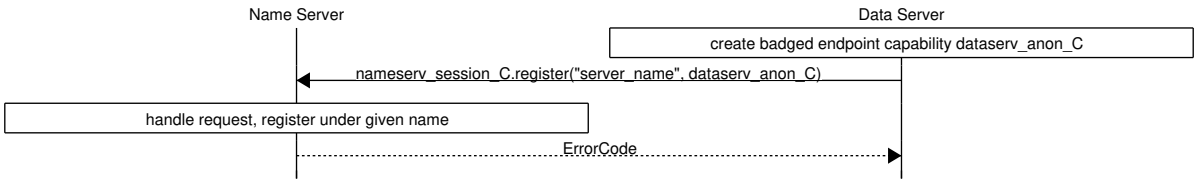


Figure 3.1: Server registration protocol

3.1.2 Establish a Session to a Server

Figure 3.2 illustrates that to interact with a server one must establish a session with the server through the server's anonymous endpoint capability. An established session with a server is represented by a `session` capability with the server knowing the security subject of the client and optionally death notifications set up so that the server is informed when the client exits and at this time can modify its bookkeeping accordingly.

Note that the (optional) argument principle_id is intended to be used for authentication assuming an ACL-like security model in the file system.

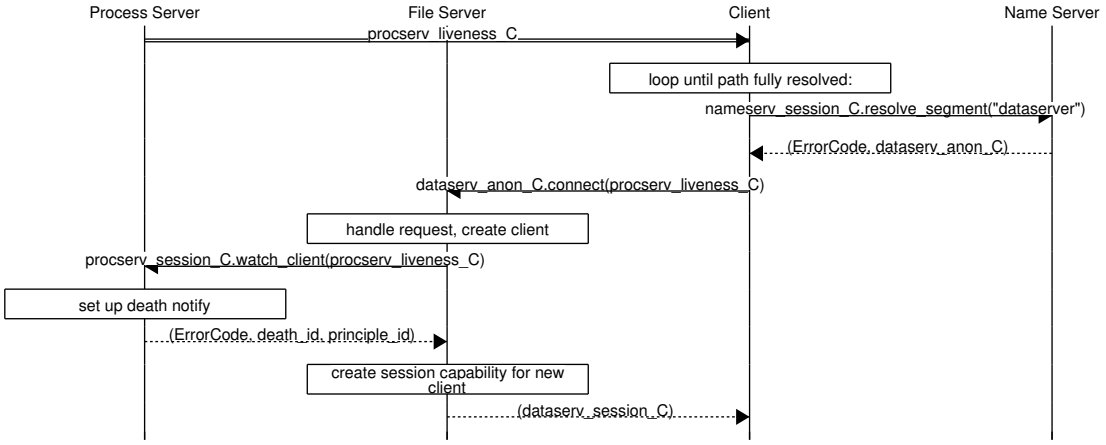


Figure 3.2: Example message sequence for finding and establishing a connection with a server

3.1.3 Opening and Closing a Dataspace

Figure 3.3 shows how a client opens a dataspace which could subsequently be used for mapping in a window. This example also shows how a dataspace is closed.

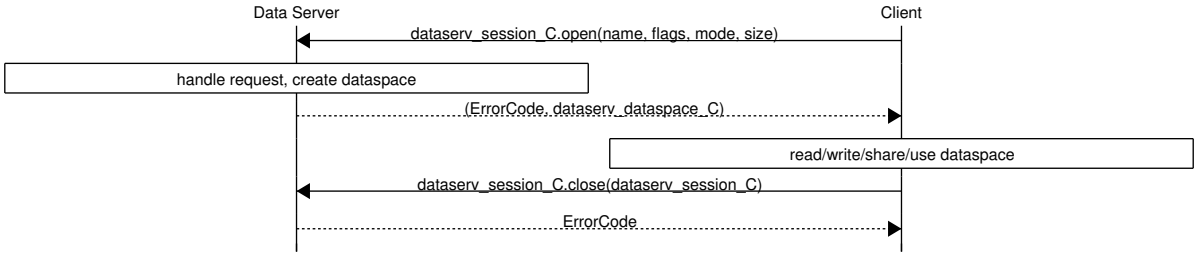


Figure 3.3: Sample protocol for opening and closing a dataspace

3.1.4 Map a Dataspace into a Window

Figure 3.4 shows mapping an open dataspace into a given memory window. The initialisation sets up the components for sample fault delegation in the system. Note that the process server is acting as the memory manager. The paging could instead be implemented through a separate memory manager process.

3.1.5 Page Fault

Figure 3.5 shows the page fault resolution protocol in the case of an external dataspace server backing a window in the client's address space. In `RefOS`, the `seL4` microkernel sends the page fault notification to an endpoint capability pretending to be the faulting process.

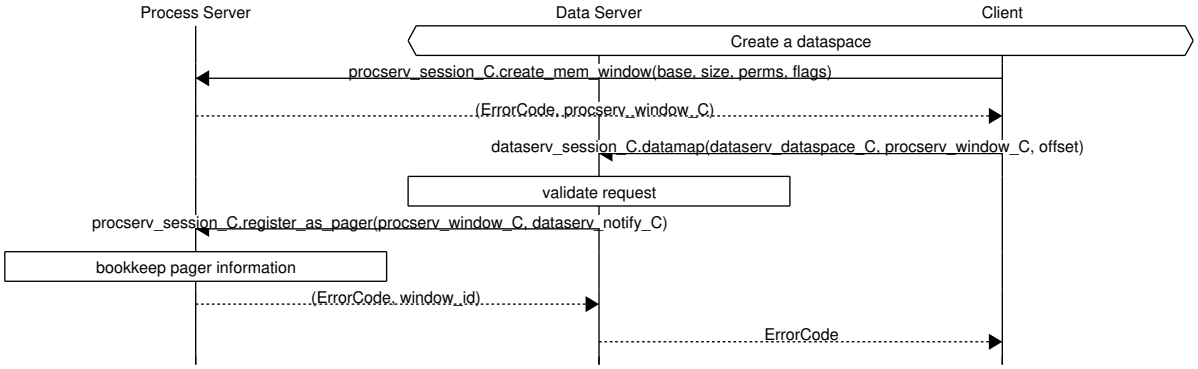


Figure 3.4: Sample protocol for mapping a dataspace into a given window

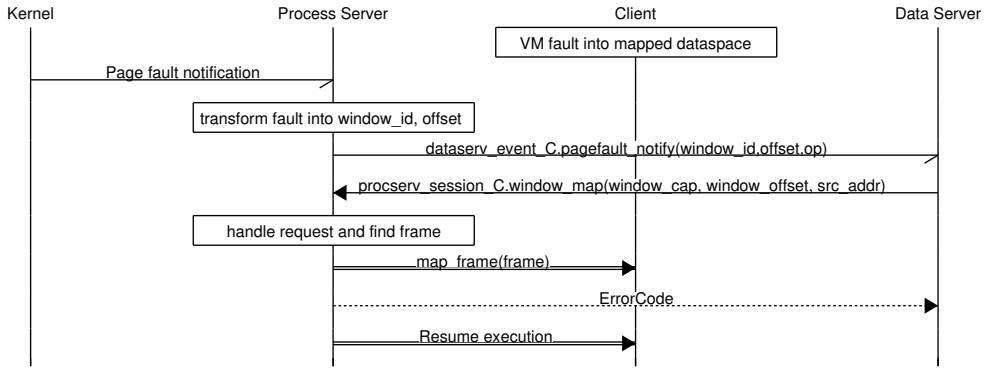


Figure 3.5: Page fault protocol

3.1.6 Death Notification

Figure 3.6 illustrates an example death notification protocol. Note that it is assumed that the client is active and has already established a connection with the data server (see Figure 3.2) and that the data server has subsequently set up death notification.

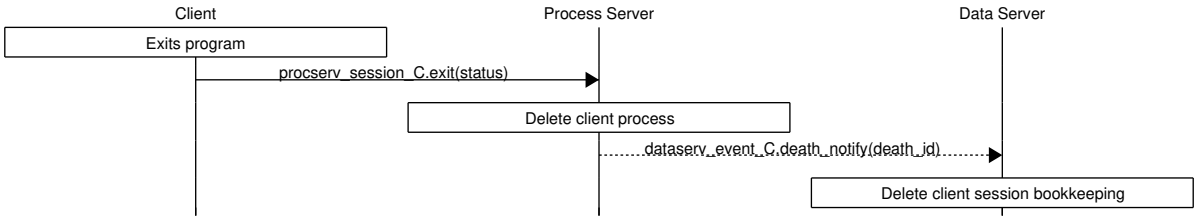


Figure 3.6: Example death notification protocol

3.1.7 Dataspace Sharing

Figure 3.7 is an example of how to set up a shared dataspace. Note that it is assumed that the share is not revocable and that sharing is with an entity that is trusted. Therefore, given two clients, at least one of the clients must trust the other client. Mutual distrust is not supported yet because capability transfer to an untrusted entity requires more than a direct call.

In this example, the shared dataspace is implemented by the process server (acting as the memory manager) and is shared with a server. This is typical for a shared memory communication buffer.

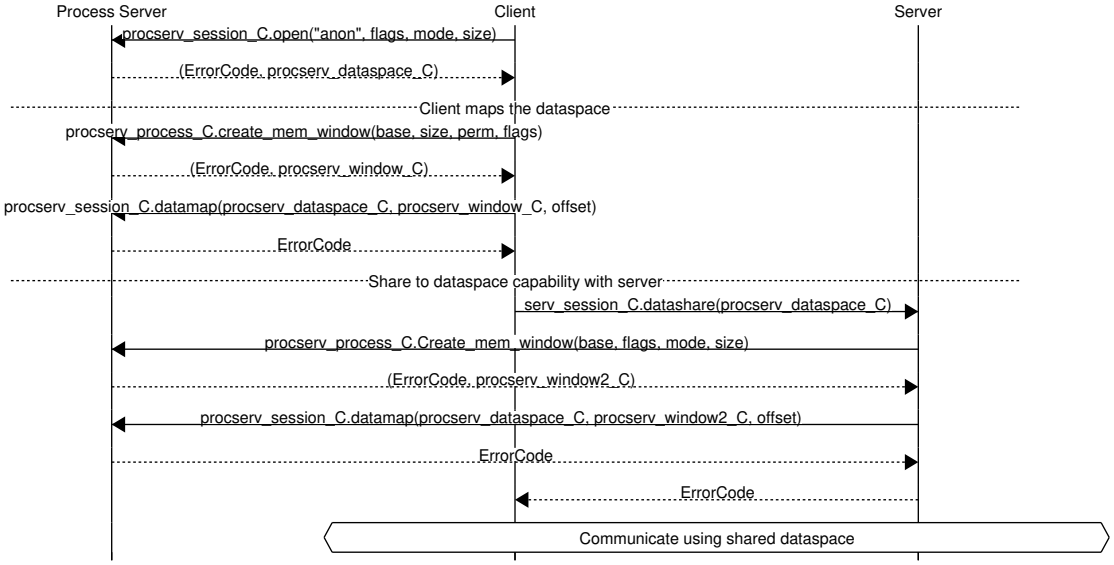


Figure 3.7: Sample protocol for sharing an anonymous dataspace

Figure 3.8 provides another example of dataspace sharing. In this example, the dataspace is implemented by an external data server instead of by the process server. A third party is involved in this case. This is a more general example of Figure 3.7.

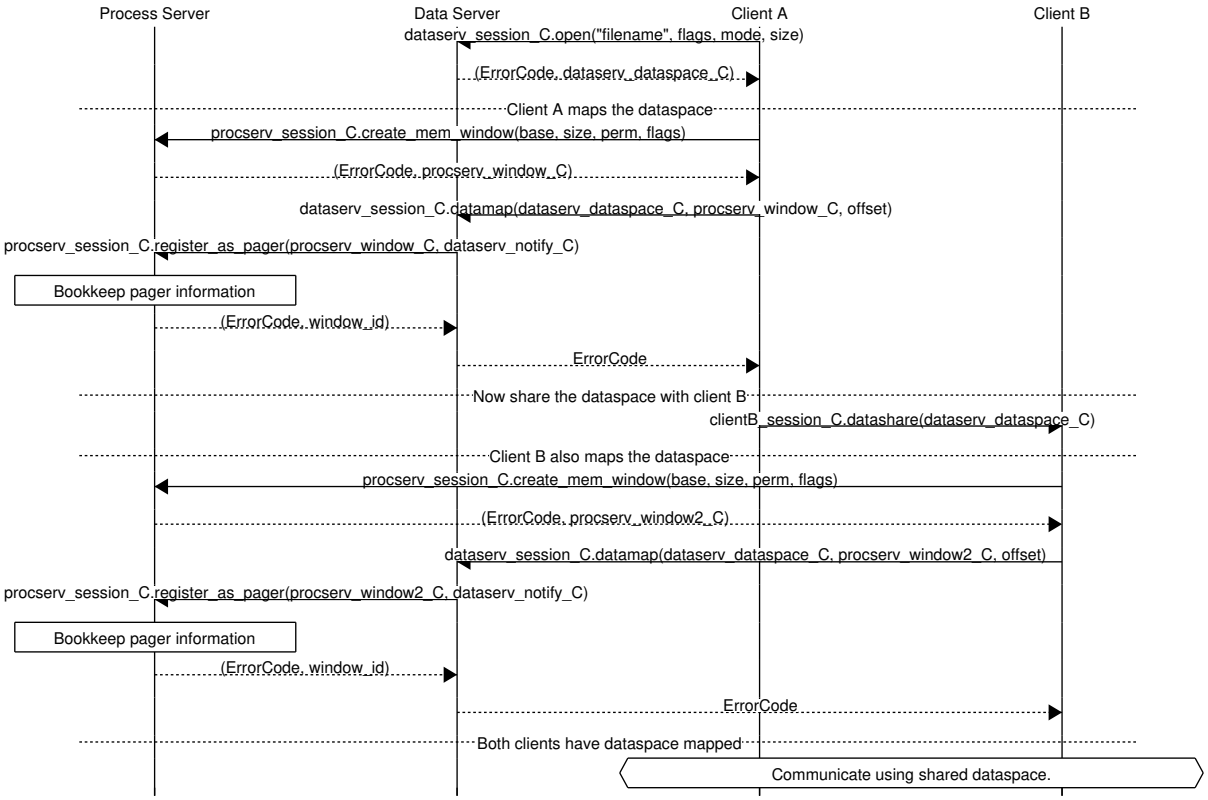


Figure 3.8: Sample protocol for sharing a dataspace provided by a dataspace server

3.1.8 Content Initialise a Dataspace with Another Dataspace

Figure 3.9 shows creating a dataspace and setting the dataspace so that it is initialised with the contents of another dataspace. This example initialises a RAM dataspace (with the process server acting as its data server as per RefOS's implementation) with the contents of an executable file from another dataspace.

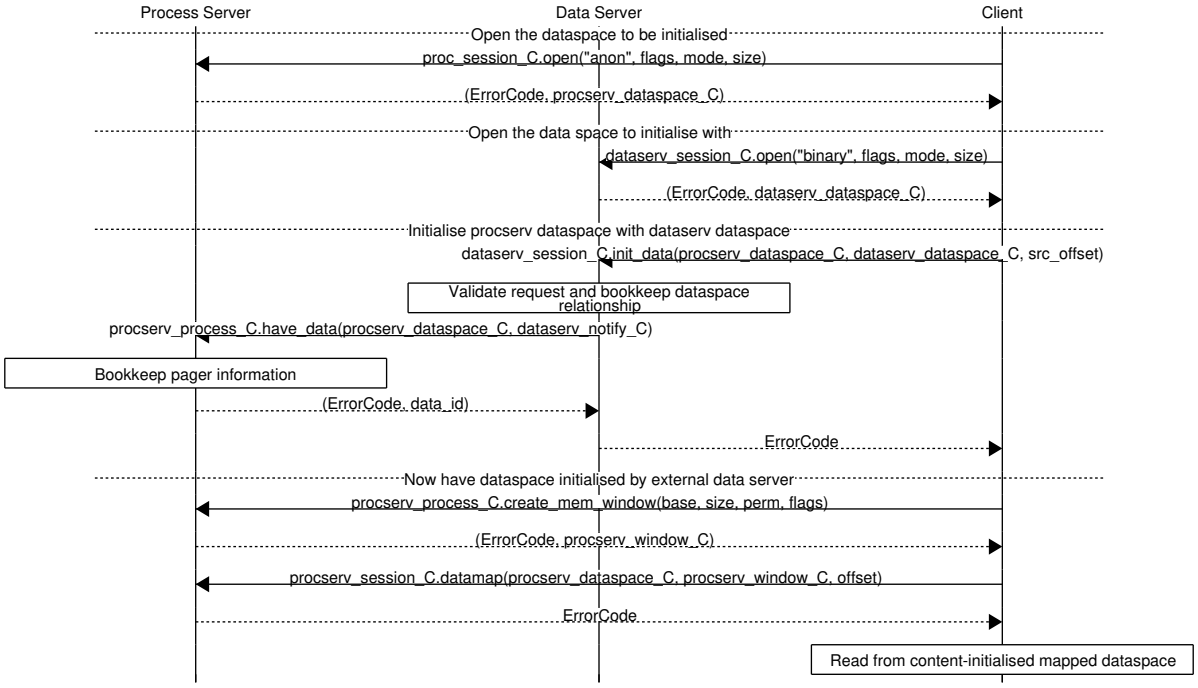


Figure 3.9: Sample protocol for creating a dataspace initialised with the contents of another dataspace

3.1.9 Content Initialised Page Fault

Figure 3.10 illustrates the page fault resolution protocol in the case of an external dataspace server providing the initial data for the dataspace. Note the similarity with Figure 3.5. The difference between the two protocols is that in the content initialised page fault protocol the content in the frame provided by the data server is not used by the client; instead the content of the frame is copied to the receiving dataserver's dataspace. In contrast, the regular page fault protocol directly maps the physical frame into the client's address space.

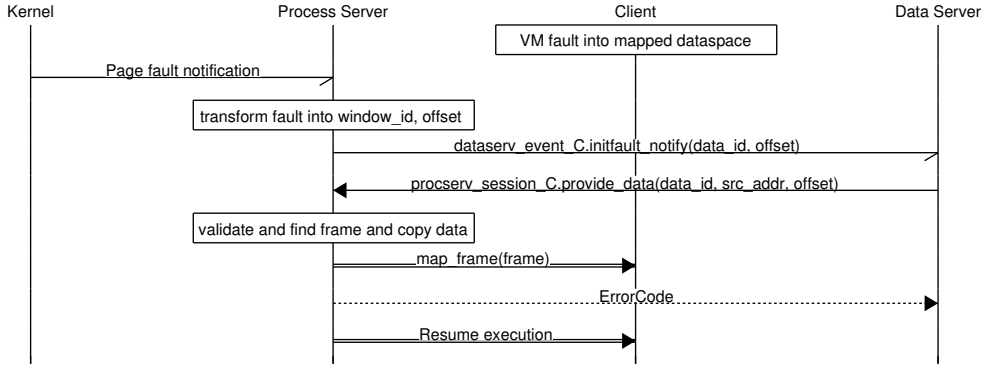


Figure 3.10: Data content initialisation fault protocol

3.1.10 ELF loading

Figure 3.11 shows the elf loading protocol for bootstrapping a user process.

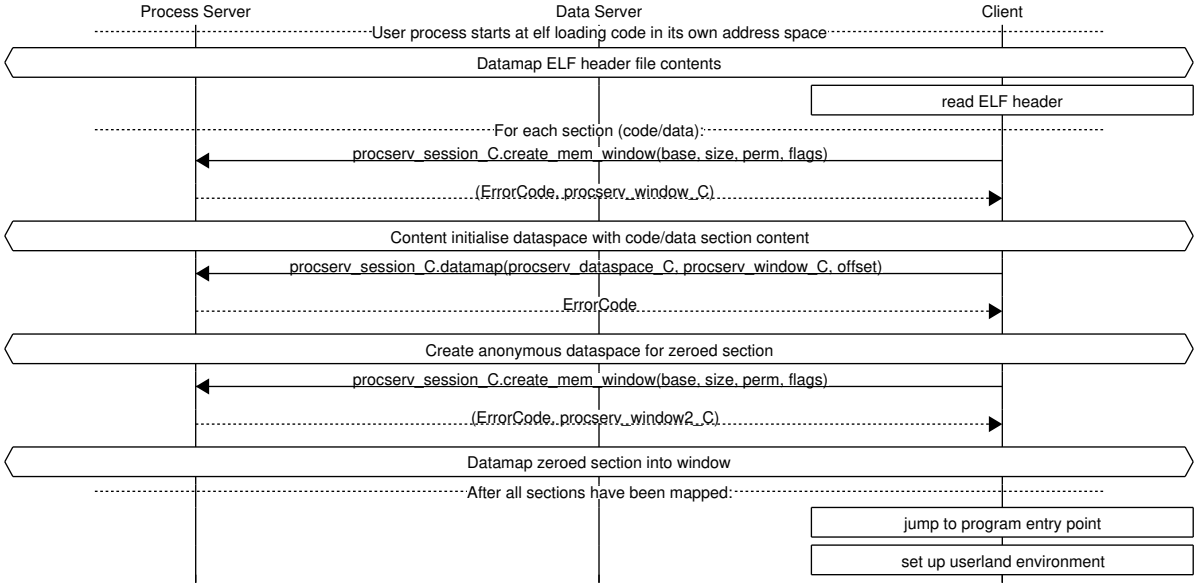


Figure 3.11: Elf loading protocol

3.2 Data Structures

This section provides an approximate representation of a number of data structures that RefOS employs. These data structures are only meant to be a guideline and are presented in the C language. These data structures aim to be the minimal information such a structure should store and intend to convey an idea of what these objects represent. Implementations may need to add extra members and references to these data structures to allow for internal bookkeeping.

3.2.1 Memory Window

A memory window should bookkeep its state, which vspace it belongs to, where and how long the memory segment is and the state of the memory window's contents. A memory window may be unmapped or mapped to a dataspace, and that dataspace may or may not be content initialised. In RefOS, the process server is the anonymous memory dataserver, so in RefOS the memory window data structure has extra bookkeeping information for the internal anonymous dataspace mapping state. Also, RefOS separates the memory window's size and offset into multiple windows.

```
struct w_window {
    vspace_t *vspace;
    vaddr_t offset;
    vaddr_t size;
    uint32_t permissions;
    bool cacheable;
    cspacepath_t capability;
    pager_t pager;
};
```

3.2.2 Process Control Block

A process control block data structure should bookkeep its address space (capability space and virtual memory space), its threads and the clients it is watching. RefOS also contains parameter buffers, notification ring buffers, the process operating system capabilities bitmask, the parent process's ID, the debug name and a number of other bookkeeping parameters.

```
struct proc_pcb {
    struct vs_vspace vspace;
    proc_tcb threads[];
    struct proc_watch_list clientWatchList;
};
```

3.2.3 Anonymous Dataspace

An anonymous dataspace data structure needs to bookkeep its capability, the list of frame pages it is representing and its content initialisation data. The content initialisation bitmask records which pages in the dataspace have been initialised and which still need to be initialised. Initialisation happens lazily, on the first VM fault, so a waiting list of clients that are VM fault blocked waiting for content to be initialised needs to be recorded.

```
struct ram_dspace {
    cspacepath_t capability;
    vka_object_t *pages;
    uint32_t npages;
    cspacepath_t contentInitEP;
```

```
    uint32_t *contentInitBitmask;
    cvector_t contentInitWaitingList;
};
```

3.2.4 Archived File Server Dataspace

An archived file server dataspace assumes that there is a char* storing the file data somewhere in the process server's address space. In RefOS, this comes from a cpio format archive linked into the process's ELF sections.

```
struct fs_dataspace {
    seL4_CPtr dataspaceCap;
    seL4_Word permissions;
    char *fileData;
    size_t fileDataSize;
    content_init_info_t contentInitInfo;
};
```

3.2.5 Userland Dataspace Mapping

A userland dataspace mapping data strucure must bookkeep its window, dataspace and session and the virtual address and size of the memory window segment.

```
typedef struct data_mapping {
    seL4_CPtr session;
    seL4_CPtr dataspace;
    seL4_CPtr window;
    char* vaddr;
    int windowSize;
    int dspaceSize;
} data_mapping_t;
```

Chapter 4

Implementation Notes

This section contains a brief overview of RefOS's implementation. For more details, refer to the doxygen code documentation.

4.1 Bootstrapping

Bootstrapping System Processes

While implementing a multi-server operating system, one quickly runs across the problem of trying to start operating system infrastructure with no operating system infrastructure in place. The executables are started via a bootstrapper app talking to the file server, but it can be difficult to start the file server.

In RefOS, the process server, being the root task and the most trusted component in the operating system, acts as a mini ELF loader to start the file server directly from its own cpio archive. There are several other methods that one could employ to start the file server. For instance, the process server could fork a copy of itself to act as the initial file server, or the process server could use a kernel which supports multiple initial tasks.

Bootstrapping Client Processes

After the system processes have been started, the infrastructure is in place to start userland processes. Starting userland processes requires interprocess communication, and in order to avoid the process server directly communicating with file servers (which causes backwards dependency), RefOS employs an external bootstrapper. The bootstrapper may be a short-lived process or thread which reads the new userland process's ELF file and maps the sections.

In RefOS, a small bootstrapper called self-loader is linked into a high reserved virtual address using a linker script. The bootstrapper then runs in the same address space as the starting client process and sets up the ELF sections in its own address space before directly jumping into the client ELF's entry point.

4.2 Communication

RefOS uses kernel communication mechanisms to implement server interface calls. It runs a simple python script remote procedure call stub generator which takes the RefOS interface (extended for extra functionality and to make implementation easier) in a simple XML format and generates corresponding C stub code. Parameters are marshalled and unmarshalled over the kernel thread interprocess communication buffer. This algorithm is not very efficient, but it requires the least amount of infrastructure. For long string parameters which are unsuited to the interprocess communication buffer and notifications, RefOS takes advantage of a shared dataspace buffer.

For asynchronous notifications, a shared dataspace with a one-way ring buffer protocol is used between the sender and the reciever. A kernel asynchronous endpoint provides synchronisation for the buffer.

4.3 Anonymous Memory

The process server recieves all the virtual memory fault notifications from the kernel and delegates them accordingly. In RefOS, the process server acts as a data server to implement anonymous memory. Under this implementation, page faults in content initialised memory only need one delegation indirection as opposed to two levels of delegation (first from the process server to the memory server and then from the memory server to the data server). Alternatively, the process server interface and the memory server interface may be separated, and the task may be distributed to another process if need be, and this memory server may act as the VM fault reciever.

Chapter 5

Conclusions

This paper presents a reference design for a multi-server operating system designed for an L4 based microkernel. It outlines a set of minimal design abstractions and methodologies which provide a standard protocol for common tasks and sample message sequences. In addition to this document, RefOS provides a documented sample implementation of this protocol. For more information about RefOS, refer to the doxygen code documentation.