

The Hurd-NG Interfaces

Marcus Brinkmann, Neal H. Walfield

March 2006

DRIFT
March 20, 2006-- 21:38

DRAFT
March 20, 2006-- 21:38

Contents

1	Introduction	1
2	Types	3
2.1	Basic Types	3
2.2	Capability Types	3
2.3	Compound Types	3
2.4	Buffer Types	3
3	Transaction Protocol	5
3.1	Remote Procedure Calls	5
3.1.1	The Client	5
3.1.2	The Server	6
3.2	Atomicity	6
3.3	Blocking Operations	6
4	Object Reference	9
4.1	Type Hierarchy	9
4.2	Input/Output Objects	9
4.3	Directory Objects	13
4.4	Server Control Objects	16
4.5	Directory Server Control Objects	17
4.6	File Server Control Objects	17
4.7	Pipe Server Control Objects	17
4.8	Signal Objects	18
4.9	Process Control Objects	18
4.10	Process Exit Objects	18
4.11	System Logging	19
4.11.1	System Log Control Objects	19
4.11.2	System Log Objects	20

Chapter 1

Introduction

The Hurd is a general purpose multi-server operating system running on a protected capability system. The functions of the operating system are split up into many different services and instances of these services. The high degree of *compartmentalization*, together with the careful choice of dependencies among the components, increases robustness and fault tolerance. It also allows to implement fine-grained security policies to increase isolation between components.

By choosing generic mechanisms over specific ones, and by carefully avoiding mutual trust between components as much as possible, we allow the user to reuse the existing components in different environments, and to add their own implementations of services that work together with the ones described here. This provides the basis for *extensibility*.

Chapter 2

Types

2.1 Basic Types

2.2 Capability Types

2.3 Compound Types

2.4 Buffer Types

A *buffer* is a continuous memory region of a fixed size. The maximum supported size depends on the IPC mechanism. Buffers can be used, logically, as input arguments or as output arguments to RPCs.

An *input buffer* is used when data is provided by the invoker to the object in an operation. In this case, the input buffer is an “in” argument of the operation.

An *output buffer* is used when data is returned from the object to the invoker in an operation. In this case, the output buffer is an “out” argument of the operation.

Chapter 3

Transaction Protocol

All objects described in this document are implemented in processes that are called *servers*. Each object implements a set of methods, its *interface*, which is determined by the *object type*. An user of an object, a *client*, gain access to an object via a *capability*. The capability names the object. To call a method, the client *invokes* the capability, passing the appropriate arguments as invocation parameters. When a client invokes a capability and waits for the reply, it performs a *Remote Procedure Call* (RPC).

The interfaces of the Hurd are described in section 4. This chapter describes the RPC mechanism and the low-level interface of all objects in the Hurd which support the transaction protocol.

3.1 Remote Procedure Calls

Clients perform RPCs to invoke a method of an object, for example to sense or manipulate its state. An RPC consists of an *invocation phase*, followed by a *reply phase*. The following description does not include the technical details of the IPC mechanism, specifically how reply capabilities can be implemented efficiently.

3.1.1 The Client

The invocation phase is initiated by invoking a capability that the client holds for the object. An invocation causes a message to be sent to the server implementing the object. This send operation may block. As long as the send has not been completed, the operation can be aborted by interrupting the invocation; in this case, the operation is not performed. When the message is delivered, the send phase is over.

The client then immediately enters the reply phase to ensure that the server's reply is received. Once the reply message is received, the receive phase is over. The receive phase can not be aborted without potentially losing the reply. Aborting the receive phase does not abort the operation.

3.1.2 The Server

The server receives the client messages, which may contain a reply capability. It then executes the invoked method of the object. After the operation is complete, the server attempts to send a reply message to the client by invoking the reply capability. It does so only once, and without blocking. If the attempt to send the reply message fails, the error is ignored and the reply message is dropped.

3.2 Atomicity

All operations on all objects described in this document are atomic. This means that when an operation is invoked on an object and the server accepts the message, the operation will complete as an atomic unit. If the operation changes the state of an object, the change is committed before the server sends the reply to the client.

Rationale

Atomicity is not a requirement for any of the other mechanisms that are described here. Instead, it is a guarantee made by the Hurd interfaces to improve reliability and reduce the number of potential error causes in a concurrent environment.

3.3 Blocking Operations

All operations on all objects described in this document are *non-blocking*. This means that when an operation is invoked on an object, and the server accepts the message, the server will reply without waiting for events outside of the server to occur. This is a requirement because the transaction is not done inside a session, so a running operation can not be named and thus not be aborted after it has been initiated.

In practice this means that the client will not be blocked indefinitely by waiting for the reply message. In other words: the receive phase of an IPC operation is guaranteed not to be unbounded in time.

The send phase, however, can block the client, even for an indefinite amount of time. This is acceptable, because the client can safely abort a blocked send

operation. Normally this is not the case because the server will do its best to accept and process incoming RPCs quickly. However, the server can make use of a blocking send phase to implement blocking operations in the following way:

1. The server receives an incoming RPC, and notices that the operation can not be performed promptly, but completion logically depends on an unsatisfied condition. The server associates a *retry capability* with the condition, but does not start to serve requests invoked on that capability.
2. The server sends a reply message to the client which contains the retry capability which indicates that the operation would block and needs to be retried on the provided capability.
3. The client repeats the RPC on the retry capability.
4. When the condition is satisfied the server starts to accept requests blocked on the retry capability. These requests will complete promptly as long as the condition is satisfied. When the condition becomes unsatisfied, the server stops serving requests on the retry capability and goes back to step 1.

Because there is a race in the server for serving requests on a retry capability and serving requests on the actual capability, the condition may become unsatisfied after accepting a message from the retry capability. In this case, the client may receive the retry message more than one time. Any retry message received while waiting on the retry capability is called a *spurious wake-up*.

The client may cache the retry capability and reuse it for later invocations of the same operation on the same object, thus avoiding the initial retry in case of blocking. To facilitate this, the server must keep serving requests on the retry capability whenever the associated condition is satisfied.

Rationale

The transaction model is sessionless. We believe that sessions increase code complexity and have a high setup and tear-down time that is only amortized when a lot of operations are done on a single object. Also, sessions require server side state which causes resource accounting problems.

The lack of nameable sessions makes it impossible to safely abort, at the client side, pending operations that are currently blocking, for example when events arrive. To implement abortable blocking operations, the server can push clients back on a blocking send-queue using the retry mechanism described above.

The retry mechanism does not easily support complex server operations which may involve several blocking steps or expensive setup phases. The Hurd does not include such operations.

Chapter 4

Object Reference

4.1 Type Hierarchy

Hurd objects (except for notification objects) share `coyotos.cap` as a common ancestor. This base class supports the operations `destroy`, `reduce` and `getType`.

Only single inheritance is supported.

Rationale

Coyotos uses CapIDL, which only supports single inheritance. The reason is that multiple inheritance does not easily allow to create globally unique message ID without central registration of all participants.

Notification objects are special object types with a restricted interface. A notification object only implements a single method. Invocation of the method returns asynchronously, i.e. there is no receive phase. Notification objects share the method-less base class `notify_t` as a common ancestor. Some notification objects can only be used once. These are called *one-shot notifications* and share the common ancestor `notify_once_t`

4.2 Input/Output Objects

Type hierarchy: `io_t` \rightarrow `coyotos.cap`

Input/Output (I/O) objects provide basic operations for reading and writing binary data. This class supports storage based I/O objects like files as well as stream based I/O objects like pipes or sockets.

```
typedef cap_t io_t;
```

The `io_t` type specifies an object that implements the I/O interface as described in this section.

Objects that do not support reading from and writing to arbitrary offsets are *non-seekable* objects.

<i>io_read</i>	
<code>io_t io</code>	<code>→ error_t result</code>
<code>offset_t offset</code>	<code>buffer_t buffer</code>
<code>size_t amount</code>	<code>cap_t retry</code>

The method *io_read* reads up to *amount* bytes of data from the I/O object *io* starting from offset *offset*. For non-seekable objects, *offset* must be 0.

For non-seekable objects, if no data is currently available (but may become available in the future), and thus the operation would block, a zero-length buffer is returned in *buffer* and a retry capability is returned in *retry*.

If *offset* points past the end of the I/O object, or *amount* is 0, then a zero-length buffer is returned in *buffer* and a void capability is returned in *retry*.

In all other cases, at least 1 and at most *amount* bytes are returned in *buffer* and a void capability is returned in *retry*.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

Rationale

The *io_read* interface supports storage-based objects like files as well as stream-like objects such as pipes and sockets. The offset 0 was chosen for non-seekable objects because for these objects data is always read from the current position in the stream, which is defined as the “beginning” of the I/O object.

Currently, the interface supports well POSIX I/O types like files, pipes and sockets. Hybrid object types can be built as well, but it is unclear if the logical extensions of the interface lead to sane object semantics. Thus, provisionally, blocking is restricted to non-seekable objects, and using non-zero offsets with non-seekable objects is undefined.

To implement a *select*-like functionality, *amount* can be set to 0. This probes if data is immediately available for reading or not. If this method is invoked on the I/O object, then the probe is non-blocking. If it is invoked on the retry capability, the probe is blocking.

io_write

<code>io_t io</code>	<code>→</code>	<code>error_t result</code>
<code>offset_t offset</code>		<code>size_t amount</code>
<code>buffer_t buffer</code>		<code>cap_t retry</code>

The method *io_write* writes the data in *buffer* sequentially to the I/O object *io* starting at offset *offset*. For non-seekable objects, *offset* must be -1 .

This method requires a write capability to the I/O object.

For non-seekable objects, if data can currently not be written (but may be written in the future), and thus the operation would block, the value 0 is returned in *amount* and a retry capability is returned in *retry*.

If *offset* is -1 , then it is set, on the server side, to point to the byte after the last byte of the I/O object before starting the operation.

If *offset* plus the length of the buffer points past the end of the I/O object, the I/O object may be extended.

If no bytes can be written, an error code is returned. Otherwise, the number of bytes written from *buffer* is returned in *amount*, and a void capability is returned in *retry*.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

Rationale

The requirement to write the bytes sequentially is to ensure that no semantic difficulties arise from files with holes. If a write starts in a hole, and extends beyond the end of a hole, allocation of backing store for the hole could fail. In this case, writing the beginning of the buffer would be impossible, but the end of the buffer could be written. To avoid ambiguities (eg. for the condition “if no bytes can be written”), we require that the bytes are written sequentially.

The offset -1 can be used to atomically append data to files.

To implement a *select*-like functionality, the *amount* can be set to 0. This probes if data is immediately available for reading or not. If this method is invoked on the I/O object, then the probe is non-blocking. If it is invoked on the retry capability, the probe is blocking.

io_map

<code>io_t io</code>	<code>→</code>	<code>error_t result</code>
		<code>patt_t patt</code>

The method *io_map* returns a PATT capability *patt* to the I/O object *io*. If the I/O object does not support the *io_map* method, an error value is returned.

If the invoker has read/write access to the I/O object, an opaque PATT capability is returned in *patt*. If the invoker has read-only access to the I/O object, a weak PATT capability is returned.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

io_stat

```
io_t io → error_t result
        struct io_statbuf stat
```

The method *io_stat* returns meta-data about the I/O object *io*.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

struct io_statbuf

Member	Description
oid_t fsid	Alleged file system ID.
oid_t ino	Inode number.
io_type_t type	I/O object type. <i>(Preliminary)</i>
size_t length	Length (0 for non-seekable objects).
size_t size	Allocated storage.
size_t blksize	Optimal block size for I/O.
time_t mtime	Time of last successful write operation.

io_type_t (Preliminary)

Value	Description
HURD_IO_CHR	Character device.
HURD_IO_BLK	Block device.
HURD_IO_REG	Regular file.
HURD_IO_LNK	Symbolic link.
HURD_IO SOCK	Socket.
HURD_IO_FIFO	FIFO.

Rationale

No modifiable meta-data exists for I/O objects, thus a change time is not provided. Access time is omitted because it provides a covert channel between holders of a read-only capability to the same I/O object.

The type part of the mode field in POSIX is omitted, because the *obj_get_alleged_type* method provides superior functionality.

For pipes, the optimal block size may be the pipe buffer size.

The type is an alleged type, which is provided for POSIX compatibility.

io_set_length

```
io_t io          →  error_t result
size_t length
```

The method *io_set_length* sets the length of the I/O object *io* to *length* bytes. This method may not be implemented for non-seekable I/O objects.

This method requires a write capability to the I/O object.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

io_set_mtime

```
io_t io          →  error_t result
time_t mtime
```

The method *io_set_mtime* sets the modification time of the I/O object *io* to *mtime*. This method may not be implemented for non-seekable I/O objects.

This method requires a write capability to the I/O object.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

4.3 Directory Objects

Type hierarchy: `dir_t` → `coyotos.cap`

Directory objects implement a name service: They allow to enter and lookup capabilities under a string identifier, the *name* of the directory entry. If a capability is entered in a directory under a name it is *linked* into the directory. If the capability is removed from the directory it is *unlinked*.

The directory object does not associate any semantics with the name of a directory entry. The only restriction is that it does not contain any bytes that are binary zero (0x00). The recommended practice is that the name is a human-readable identifier encoded in UTF-8 (Normalization Form C).

dir_link

```
dir_t dir          →  error_t result
obj_t object
string_t name
bool_t exclusive
```

The method *dir_link* enters the object *object* into the directory *dir* under the name *name*.

If *exclusive* is true, then the operation fails if there already exists an object under the name *name* in this directory. If *exclusive* is false, any existing object under the same name is replaced.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined

dir_unlink

```
dir_t dir      →  error_t result
string_t name
```

The method *dir_unlink* removes the name *name* from the directory *dir* and drops the associated object capability.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

dir_rename

```
dir_t dir      →  error_t result
string_t name
dir_t new_dir
string_t new_name
bool_t exclusive
```

The method *dir_rename* moves the directory entry under the name *name* in the directory *dir* to the directory entry under the name *new_name* in the directory *new_dir*.

The rename operation may fail if *dir* and *new_dir* are not implemented by the same server. In this case, the error code `EXDEV` is returned.

If *exclusive* is true, then the operation fails if there already exists a directory entry under the name *new_name* in the directory *new_dir*. If *exclusive* is false, any existing directory entry under the same name is replaced.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

Rationale

The *dir_rename* method does the same as the following sequence:

```
error_t
dir_rename_nonatomic (dir, name, new_dir, new_name, exclusive)
{
    error_t err;
    obj_t obj;

    err = lookup (dir, name, &obj);
    if (err)
        return err;

    err = dir_link (new_dir, new_name, obj, exclusive);
    if (err)
        drop (obj);

    dir_unlink (dir, name);
}
```

However, in the *dir_rename* operation the link and the unlink step are performed in one atomic sequence.

The operation *dir_rename_nonatomic* has the following advantages:

- It is implemented in terms of other low-level methods.
- It can work even if *dir* and *new_dir* are implemented by different, non-cooperating servers.
- The important invariant that during the whole operation *new_name* points either to the old object or the new object (with nothing inbetween) is preserved.
- When invoking *dir_rename* The client has no way to know if *dir* and *new_dir* are implemented by the same (or cooperating) servers. It has to provide *new_dir* to the server implementing *dir* to check. In the case where this is *not* the case, the client invoking *dir_rename* provides the server implementing *dir* with an excess of authority, which is a violation of the principle of least authority. The above sequence does not share this problem.

Unfortunately, there is a compelling reason to provide a *dir_rename* operation anyway:

- Some alien file systems (i.e. file systems outside of the persistent core) do not support hard links to directories or any hard links at all, but usually do support the rename operation. These filesystems can in general not successfully emulate the above sequence, because its beginning (the *dir_link* invocation) is ambiguous.

Therefore, the following strategy can be deployed:

1. Attempt to use the *dir_rename_nonatomic* sequence.
2. Attempt to use the *dir_rename* method.
3. Copy the underlying object (and remove the original) instead of linking/renaming the capabilities.

This approach favors security over compatibility. Alternatively, steps 1 and 2 could be swapped.

```

dir_lookup
-----
dir_t dir      →  error_t result
string_t name  obj_t obj
                int magic

```

The method *dir_lookup* looks up the capability associated with the name *name* in the directory *dir*. The name comparison is performed at the byte level.

If the lookup succeeds fully, *obj* contains the resulting capability and *magic* contains 0.

If the lookup completes partially, *obj* contains a new capability on which the lookup has to be retried with a new name, which can be derived from *name* by chopping off the first *magic* bytes.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

dir_get_entries

```
dir_t dir      →  error_t result
int  entry    →  buffer_t buffer
int  nr_entries
```

The method *dir_get_entries* reads the content of the directory *dir* and returns it in *buffer*.

Up to the size of *data* but no more than *nr_entries* records are read, starting from entry *entry*. The first entry is entry 0.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

4.4 Server Control Objects

Type hierarchy: `server_t` → `coyotos.cap`

All Hurd servers provide a control object that allows to alter the server's behaviour, shut it down, and fabricate new objects. Fabrication is specified in server-specific interfaces, while the common interfaces are defined here.

server_go_away

```
server_t server →  error_t result
```

The method *server_go_away* asks the server to cease operation and exit.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

server_get_options

```
server_t server →  error_t result
                    buffer_t buffer
```

The method *server_get_options* retrieves the current options for the server in *buffer*. The options are provided in ARGZ format.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

server_set_options

```
server_t server → error_t result
buffer_t buffer
```

The method *server_set_options* sets the current options for the server, as provided in *buffer*. The options must be provided in ARGZ format.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined.

4.5 Directory Server Control Objects

Type hierarchy: `dirs_t` → `server_t` → `coyotos.cap`

Directory server control objects allow the fabrication of new directory objects.

dirs_mkdir

```
dirs_t dirs → error_t result
dir_t dir
```

The method *dirs_mkdir* creates a new directory object in the server *dirs* and returns a read/write capability to it in *dir*.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined

4.6 File Server Control Objects

Type hierarchy: `files_t` → `server_t` → `coyotos.cap`

File server control objects allow the fabrication of new file objects. Files are I/O objects that provide a persistent storage. Data can be written to the file and read out at any later time, in any order, i.e. files are seekable objects.

files_mkfile

```
files_t files → error_t result
io_t file
```

The method *files_mkfile* creates a new file object in the server *files* and returns a read/write capability to it in *file*.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined

4.7 Pipe Server Control Objects

Type hierarchy: `pipes_t` → `server_t` → `coyotos.cap`

Pipe server control objects allow the fabrication of new pipe objects. Pipes are I/O objects that provide a buffered data stream channel. Pipes are non-seekable objects.

```

pipes.mkpipe
-----
pipes_t pipes  →  error_t result
                  io_t read_end
                  io_t write_end

```

The method *pipes.mkpipe* creates a new pipe object in the server *pipes* and returns a read-end capability in *read_end* and a write-end capability in *write_end*.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined

4.8 Signal Objects

Type hierarchy: *sig_t* → *notify_t*

Signal objects allow to post a signal to a process.

```

sig_post
-----
sig_t signal  →  error_t result

```

The method *sig_post* posts the signal *signal*. Signals do not carry a sender-provided message payload.

If an error occurs, the error code is returned in *result*.

4.9 Process Control Objects

Type hierarchy: *pctl_t* → *coyotos.cap*

Process control objects allow to communicate with a process, i.e. to send signals to it or debug it.

```

pctl_get_sig
-----
pctl_t pctl  →  error_t result
int signo      sig_t signal

```

The method *pctl_get_sig* requests the signal capability for the signal *signo* from the process control object *pctl*.

If an error occurs, the error code is returned in *result* and the values of all other out-parameters are undefined

4.10 Process Exit Objects

Type hierarchy: *pexit_t* → *notify_once_t* → *notify_t*

Process exit objects allow to return a status code from a child process to its parent.

```
          pexit_return
pexit_t pexit → error_t result
int status
```

The method *pexit_return* returns the status code *status* to the parent process providing *pexit*.

If an error occurs, the error code is returned in *result*.

4.11 System Logging

The system log server “syslog” queues system-generated event messages directed at particular users. These events are queued until they are received by the user or until they expire. The user can detect if an event was missed because it expired. Examples of possible events are:

- Log-on/log-off events on a terminal.
- Hot-plug events on hardware in use.
- System software updates.
- Downtime notifications.
- ...

4.11.1 System Log Control Objects

Type hierarchy: `syslog_ctrl_t` → `coyotos.cap`

System log control objects allow to post events to the system log server.

```
          syslog_ctrl_post
syslog_ctrl_t syslog_ctrl → error_t result
syslog_cat_t cat
cap_t ev_cap
user_id_t uid
```

The method *syslog_ctrl_post* posts the event *ev_cap* to the user *uid* in the logging category *cat*.

Rationale

A human-readable string raises internationalization issues and invites broken attempts at parsing its content. In the future, it may be considered to pass binary data along with the capability, if useful. Alternatively, the capability could point to a directory with more information.

The category field is a place holder. It may be subdivided into a category and a sub-category, similar to interface and message IDs.

If an error occurs, the error code is returned in *result*.

4.11.2 System Log Objects

Type hierarchy: `syslog_t` → `coyotos.cap`

The system log service provides a way for system services to deliver an event notification to a system log object. System log objects are semantically bound to a user account, and system services log events to user accounts using system log post object capabilities (FIXME: reference).

```

syslog_read
-----
syslog_t syslog → error_t result
                  syslog_cat_t cat
                  cap_t ev_cap

```

The method *syslog_read* returns the next system log event from *syslog*.

If an error occurs, the error code is returned in *result*.