

# Building a Network File System Protocol for Device Access over High Latency Links

Francisco J. Ballesteros, Enrique Soriano, Spyros Lalas, Gorka Guardiola  
nemo@lsub.org, esoriano@lsub.org, lalis@inf.uth.gr, paurea@lsub.org  
10/5/2007

## ABSTRACT

In the operating system we are currently building, the Octopus, each user has a single PC that is used to run applications. Other, possibly mobile, computers play the role of mere resource providers that can be attached to the PC in an ad-hoc fashion. Independently of their nature, resources appear to be files, and are consequently accessed through the standard file interface. Given that many resources are contributed by machines located at a far distance from the PC, we needed a network file system protocol that operates over high latency links, and which is able to handle synthetic files in a satisfactory way. A further requirement was that legacy applications, designed for the original file system protocol, should continue working as transparently as possible, with minimal or no changes. This paper describes the file protocol we implemented for this purpose, called the Octopus protocol or Op, our experience while developing, deploying and using it, and the lessons we learned. It also includes a quantitative evaluation of the performance enhancement that can be achieved by using it.

## 1. Introduction

Following the ideas of Plan 9 [7] and Plan B [1], we are currently implementing the Octopus, a system whose goal is to provide users with a homogeneous and easily extensible distributed computing environment. We are developing our prototype on top of Inferno [2].

Unlike many other systems, the main idea behind Octopus is to simplify things by *centralizing everything*. To achieve this, each user designates a particular computer in the network as his *personal computer*, referred to as the “PC”, and runs on it all software; both system and applications. Other computers are merely used to provide resources (data, special devices or other services) to the PC. Such machines, known as *terminals*, run the Octopus software in order to export all or some of their resources to the PC.

In Octopus, resources appear to be files. Their interface has the form of a file tree, and they are accessed through the standard file interface. Programs that provide resources are implemented as file servers. Some of them provide access to a real (disk) file system. However, most servers are user-level programs that offer a file-based access interface for a particular device or service. Following this model, an Octopus terminal simply hosts one or more file servers, which are mounted by the PC at runtime in order to make the corresponding resources available to the locally executing user applications.

Communication between the terminals and the PC is usually done across a network

link with bad latency. This is the common case in the EU for connections over the Internet, e.g., when a terminal connects to the PC using an ADSL line or a Wi-Fi hot-spot. Round trip times of 120ms for ICMP are not uncommon. As a consequence, and to support interactive applications, we needed a file system protocol that:

- 1 Works well across network links with bad latency.
- 2 Can be used to access synthesized files (e.g., devices), not just regular disk files.
- 3 Can be transparently introduced without modifying existing applications.

We soon realized that none of the file system protocols used in our environment, including Styx [8], 9P [9], CIFS [5], NFS [13], and WebDAV [15], is particularly appropriate given these requirements. This comes as a big surprise, considering that today there is a real need for accessing devices and other services across the Internet, also in a highly interactive fashion. The fairly extensive research on distributed file systems seems to focus on protocols optimized for poor bandwidth [6]. However, for remote interactive computing, the main problem is more often latency than bandwidth. NFS version 4 is promising in that it introduces compound operations to address latency [13], but like its predecessors, it does a poor job to support the access of synthesized files which are used as an interface for remote devices. It is also utterly complex, which makes it hard to use for devices with scarce memory and processor resources.

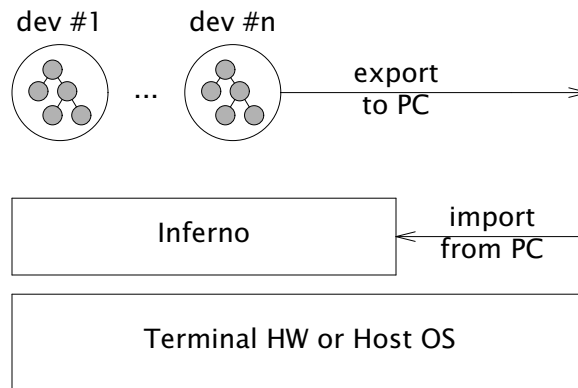
Our starting point is the Inferno OS [2], which uses Styx as its file system protocol for accessing both files and devices. Although Styx works great in local area networks, its design typically results in performing a large number of distinct RPCs, which is quite undesirable for links with bad latency. To repair this weakness, we built a new protocol for the Octopus system, called the Octopus protocol or Op. Op follows the RPC model used by Styx [8] and 9P [9], but it is designed to minimize the number of RPCs needed to perform a series of simple and frequent operations. For example, Op tries to support the retrieval/update of a (small) file using a single RPC. Doing this while achieving backwards compatibility for legacy applications as well as being able to access devices with file-like interfaces has proven to be a tricky job. We have also implemented a client and a server that speak Op. These programs are employed as a bridge to transparently interconnect machines that use Styx as their file system protocol. This means that most of the existing system and application software, which relies on the standard Inferno protocol (Styx), may remain unaware of the existence of Op yet can be used over a wide area network or any link with bad latency.

The main contribution of this paper is not the protocol we designed, but the lessons learned while building, deploying and using it; which required several changes to make it work well. We believe that our experience can be applied to any other protocol for exporting devices through a file interface across networks with bad latency. For example, the same issues would have to be addressed to achieve similar functionality in UNIX over NFS or Windows over CIFS.

The next section describes the environment that we had before developing Op and further illustrates our problem.

## **2. The Octopus and Inferno**

The Octopus software is written in Limbo [3] on top of the Inferno OS [2]. Inferno runs hosted on many different systems as well as on bare hardware. For the underlying platform, Inferno and all the software it runs seem to be just another application. Inferno is efficient (in terms of executing compiled Limbo programs), has a small memory footprint and only few requirements on its own (e.g., does not even need a MMU). Most resources in Inferno are exported by means of virtual file systems and are accessible through the network using the Styx file system protocol [8]. Therefore, Inferno seemed perfect as a basis for letting programs export and access devices and services in a flexible way, according to the vision behind Octopus.



**Figure 1:** *System software at a Terminal*

The prototype implementation of an Octopus terminal (the PC is implemented along the same lines) is illustrated in figure 1. Each terminal features one or more servers that run on top of Inferno. Each server exports a particular resource to the PC. Also, all resources that are known to the PC (imported from terminals) are exported by it and imported into the terminal, making them available to local software.

## 2.1. Styx

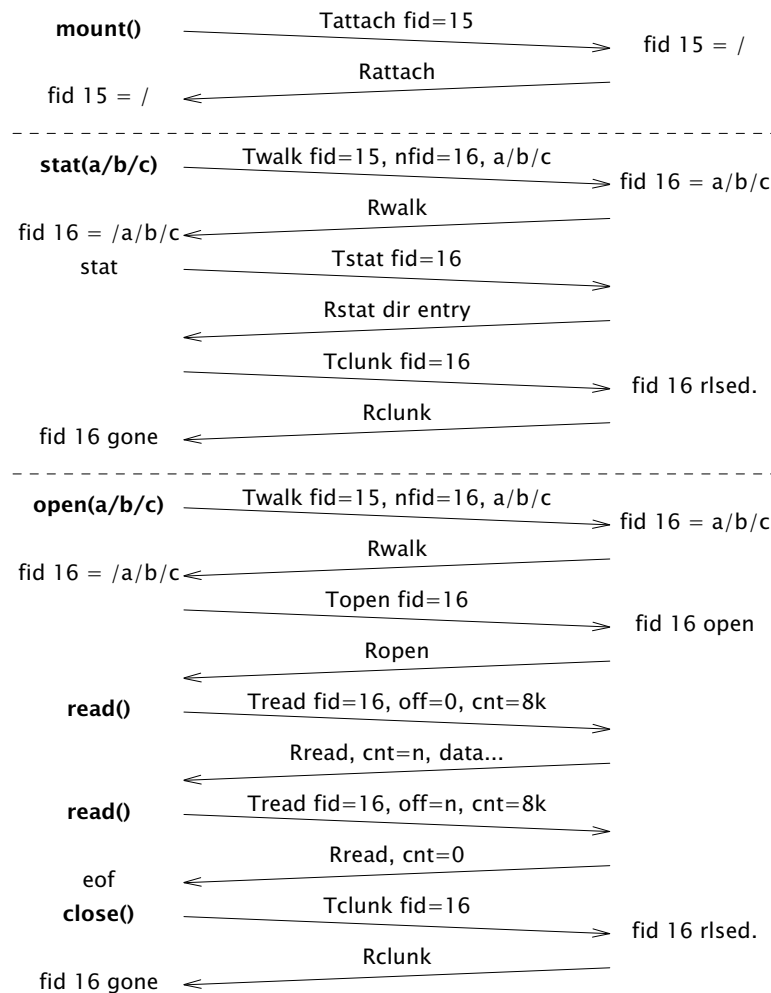
Inferno uses Styx as its file system protocol. It is simple, clean, and effective, and works fine within local area networks. However, things change when latency gets worse, and the protocol becomes inappropriate for interactive usage (the same happens to both NFS and CIFS). In the following, we give a short description of how Styx works and discuss its shortcomings with respect to bad latency.

Styx is a protocol using synchronous RPCs (called *transactions* in the protocol specification). A client sends requests, or *T-messages*, and a server answers by sending replies back, or *R-messages*. In many cases, the Inferno system is the Styx client, speaking Styx on behalf of application processes that use the well-known file interface, i.e., *open*, *read*, *write*, *create*, *remove*, *stat* and *wstat*. On the other hand, there are many Styx servers. Each one provides a particular service using a (synthesized) file-tree as the interface.

Styx includes two important concepts, known respectively as *fids* and *qids*. A *fid* is a file identifier (a number) chosen by the client to refer to a particular file in the server's file tree. A *fid* may be open or not. While open, it may be used to read and write the file it refers to, else it may be used to walk the file hierarchy in the server and, of course, to create, remove, or open a file for further I/O. A *qid* is a file identifier (three numbers actually) chosen by the server to identify a file. Each file is guaranteed to have a unique *qid* (within a server). Besides, a *qid* includes a version number, used by the client to detect file updates, and a type field, used by the client (among other things) to distinguish between files and directories.

A typical Styx conversation is shown in figure 2. Time flows from the top to the bottom. The client depicted on the left performs a sequence of file operations or system calls (shown bold) that result in the Styx conversation shown. For the server, shown on the right, we show the actions resulting from the requests.

The part before the first dashed line introduces the client to the server. An *attach* transaction permits the client to define a *fid* (15 in the figure) that corresponds to the root of the file tree in the server. Most requests carry a *fid* to indicate the file involved in the request. In this case, several of the following requests mention the *fid* 15 just defined.



**Figure 2:** *Styx dialog between a client and a server*

The next part (up to the next dashed line) corresponds to a client accessing the directory entry for file `/a/b/c` in the server. The *walk* transaction proposes a new fid (16 in the figure), which starts as a clone of the fid mentioned (15 in the figure). Furthermore, this request walks that new fid to a path relative to its original position in the server's file tree. In this case, fid 16 points to the file `/a/b/c` within the server once the *walk* transaction completes. At this point, a *stat* request asks the server for metadata (directory entry) for the file identified by the fid 16. Finally, the client issues a *clunk* request when it no longer wants to use a particular fid, to let the server forget about it and release any resource used on its behalf.

The part in the figure below the last dashed line corresponds to a client reading a file. First, a *walk* request defines a new fid (16 again) and points it to the file of interest, `/a/b/c` in this case. Now, the *open* transaction opens the fid for the mode specified in the request (not shown). After receiving the reply, the client issues *read* requests (and the server replies with data) for the offsets and number of bytes desired. Usually, the series of read transactions is completed by a last one, which is used by the server to inform the client that there is no more data to be read past the offset requested; this is the end-of-file indication. At this point, the client probably closes the file, and a *clunk* transaction releases (and closes) the fid.

### 3. Accessing files through poor-latency links

Using Styx means that multiple RPCs are needed to retrieve or update a single file. Indeed, the previously shown dialog is not too different from the one that might be spoken by better known protocols like NFS version 3.

The last two parts of figure 2 are a realistic example corresponding to the execution of `ls` or `lc` to list the contents of a directory. It is not unusual for a client program that is going to read a file or a directory to call *stat* before actually reading the file, for example, to check the file for access or to retrieve some metadata. Executing the command `lc` at a (remotely) mounted directory would make things even worse, because the shell would try to locate the programs `lc`, `ls`, and `mc` in the current directory, before searching for them at `/bin`, adding at least three more RPCs to the list.

An interesting case which is particularly affected by poor latency links is the behavior of programs that simply try to prevent races. Specifically, to reduce the probability of races between opens and creations, the implementation of the *create* system call in Inferno may issue multiple walk requests before finally attempting to create the file (other system kernels acting as clients for FS protocols do the same). This of course adds more RPCs in the style shown in the figure, and the overall time needed to perform the create is greatly increased. This session exemplifies what we just described:

```
; echo hola >/tmp/anewfile
<-12- Twalk tag 4 fid 378 newfid 475 nwname 1 0:anewfile
-12-> Rerror tag 4 ename file not found
<-12- Twalk tag 4 fid 378 newfid 475 nwname 0
-12-> Rwalk tag 4 nwqid 0
<-12- Tcreate tag 4 fid 475 name anewfile perm %M% mode 438
-12-> Rcreate tag 4 qid (0000000000000001 0 ) iounit 0
<-12- Twrite tag 4 fid 475 offset 0 count 5 'hola '
-12-> Rwrite tag 4 count 5
<-12- Tclunk tag 4 fid 475
-12-> Rclunk tag 4
```

It could be argued that latency is not really a problem, because if a client program wants to read a file multiple times, it should keep the file descriptor open, making it unnecessary to issue most of the message exchange shown in the previous page, requiring just the read requests. However, most programs usually open a file, read and/or write it, and then close it before proceeding. In many cases, even the same process would close and then reopen the file and read it again. The repetitive execution of commands is a typical situation.

One way to deal with this problem would be to perform caching in the client. But this may pose severe problems for synthetic files if not done with care. For example, consider caching the `/proc` or `/dev` file systems in UNIX. How can `/dev/cons` be cached? Can it be done in the same way a program binary is cached?

Another possible solution to avoid latency problems would be to send Styx requests concurrently, without waiting for the replies, and then blocking the client until all the replies are received. To achieve this, a client process might spawn multiple child processes, each one using the standard file system calls to issue a request. The net effect would be that Inferno would be retrieving all the files concurrently, multiplexing the Styx communications channel. However, this requires changes in the applications. In fact there are many cases when this is not feasible because the task at hand requires the successful reading and/or writing of one file before proceeding to the next one.

A last resort would be to batch requests, as done by NFS version 4 [13]. However, this poses the additional problem of deciding *what* to batch. In the examples shown above a protocol could batch all the requests to be performed for a single system call, but that would not help much. Applications are programmed assuming that file access is

cheap, and they do not hesitate to issue several system calls just to check out a file for access (as shown) or to close and re-open files. Even though caching might alleviate this problem, we have to remember that this may prevent using the protocol to access devices and other synthesized files.

#### 4. Accessing devices through file system protocols

There are several important issues that arise when a file system protocol is being used to access devices as if they were files. We briefly state the most relevant ones here.

First, the file server is similar to a device driver. Among other things, this makes it necessary for the protocol to inform the server about *when* the client is using a particular resource. For example, a client might open an audio device, and then issue multiple write requests to play audio. The server must know that there is a client playing audio even between requests, when there is no outstanding request from the client. This may also be important in order to be able to inform the server process when an update made by multiple writes has been completed (e.g., to know when a request is complete and has to be processed).

In some cases, devices use file descriptors to multiplex connections using the same file for multiple clients. This is done typically in `clone` device files. A clone file allocates a new resource (usually creating a new file) each time a client opens it. The descriptor obtained from the `open` request is used by the server to identify the client.

Caching and deferred writing must be done with extreme care, if at all. For example, caching a device similar to `/proc` might be unwise because it represents a living entity (at least the process table) and may change even while no external operation is being done to it. Also, writing to a server device file might cause a side effect, like drawing graphics on the screen or reproducing audio. Those effects would be delayed if the client uses delayed writes as part of its caching implementation, and users would notice.

Deferring operations is also an issue because of error reporting. For example, a write may be used to issue a control request to a device (similar to UNIX's `ioctl`s). If the request fails, the server would report the error by replying with an error indication to the write RPC. The client system would report the error to the application at that point. Thus, if multiple write requests are batched or deferred, the client could remain unaware of a failed request, for a period of time, hence might proceed to issue further ones assuming that the request did work.

An extreme case would be to employ session-semantics. In this case, no request would reach the server before the close of the file. In systems like Inferno, that use garbage collection to collect file descriptors (and do not even have a `close` system call!) this would pose severe problems for accessing remote devices. Even in systems like UNIX, reporting errors on close is unwise, because many applications check the error status for `write` calls but forget to check if `close` reported any error.

#### 5. Op

For the Octopus system, we wanted a file protocol that, when feasible, used a single RPC for retrieving (or updating) a file. The intuition behind this is that a protocol along the lines of HTTP, but with all the additions needed for network file-like device access, could suffice for our purposes. Sadly, neither HTTP (nor WebDAV) are reasonable for use as actual network file system protocols because of their lack of support for protection, file metadata, and some properties needed when used to access devices and synthetic files. This section describes Op, the protocol we designed for Octopus, and the rest of the paper discusses the lessons we learned (besides those implicit in what was already said in the previous sections).

In Op, the *server* is a process that provides a hierarchical file system (a file tree) to

be accessed by remote client processes. The server responds to requests to create, remove, retrieve, and update file data and metadata. The client sends *T-messages* carrying requests. The server responds with *R-messages* carrying replies back to the client. The combination of sending (receiving) a request and receiving (sending) a reply is called an RPC, but in some cases, a single request may imply multiple replies. There can be a single client or multiple clients sharing the same connection, but all of the clients operate on behalf of the same user. It is assumed that the communication link preserves the order of messages sent through it.

Op messages are shown in figure 3. They are encoded using the same conventions of Styx [8]. Each message starts with a size field specifying the size of the entire message. Next comes a byte stating the message type. A *tag* field is used to refer to outstanding requests and to match replies to requests. The rest of the message depends on the type. In the figure, the number of bytes in a field is given in square brackets after the field name. The notation *parameter*[*n*] represents a variable-length parameter, encoded as *n*[2] followed by *n* bytes of data. The notation *string*[*s*] is shorthand for *s*[2] followed by *s* bytes of UTF-8 text. Details not described here are the same as in Styx, in particular, file metadata is exactly that used by Styx.

```

size[4] Rerror tag[2] ename[s]
size[4] Tattach tag[2] unname[s] path[s]
size[4] Rattach tag[2]
size[4] Tput tag[2] path[s] fd[2] mode[2] stat[n] offset[8] count[4] data[count]
size[4] Rput tag[2] fd[2] count[4] qid[13] mtime[4]
size[4] Tget tag[2] path[s] fd[2] mode[2] nmsgs[2] offset[8] count[4]
size[4] Rget tag[2] fd[2] mode[2] stat[n] count[4] data[count]
size[4] Tremove tag[2] path[s]
size[4] Rremove tag[2]
size[4] Tflush tag[2] oldtag[2]
size[4] Rflush tag[2]

```

**Figure 3:** Messages in the current version of the Octopus Protocol, Op.

The *attach* request introduces the user to the server. It corresponds to a mount operation. Authentication must take place prior to this transaction. Indeed, authentication is left out of the protocol. Before speaking Op, a client and a server may use the communication channel to mutually authenticate and also to encrypt the channel for further communication. We are using Inferno authentication and encryption mechanisms for that purpose.

Files can be created (and directories) and their contents (and metadata) updated by means of *put* requests. They can be removed by means of *remove* requests. File contents (and their metadata) may be obtained by means of *get* requests. The *flush* request is meant to abort a previous, outstanding, request. It is used to abort ongoing RPCs.

Each T-message has a *tag* field, chosen and used by the client to identify the message. The reply (or replies!) to the message has the same tag. A reply for a T-message is either an appropriate R-message or *Error*, indicating that the request failed. In the latter case, the *ename* field contains a string describing the error. Each request is considered to be atomic with respect to its execution in the server. There is an implementation-specific limit on the amount of data that may be sent in Op in a single request or reply, but note that the get transaction permits the server to send multiple reply messages for a single request (this will be discussed in more detail in the sequel).

### 5.1. File names and descriptors

Most T-messages request that an operation be made for a file. Usually, the file is identified by the *path* field of the T-message. The *path* field contains a string with a file name or path (rooted at the server's root directory). For example, /a/b means the file b inside the directory a inside the root of the server's file tree.

In addition to the *path* field, both Tput and Tget may identify the file using the *fd* field, which contains a small integer that represents a *file descriptor* to the file. This descriptor is to be considered a *cache* identifier for the *path* mentioned in the request. In other words, when a valid descriptor is sent in a Tget (or a Tput) the server ignores the path and uses *fd* to identify the file to be used for the operation. If the *fd* is invalid, the file server uses *path* instead. The special value NOFD (~0) makes this field void and represents a null descriptor.

The role of file descriptors is to let the server know *whether* there are any clients using a *particular* resource, without requiring open or close RPCs. This is critical for exporting devices, in particular since a high-level request may require multiple put or get requests at the level of the protocol, in which case the device has to know when the request has completed (by identifying the last message in the request).

File descriptors are allocated by the server upon request. More specifically, a client may specify in a Tget or Tput request that more requests of the same type will follow, by setting the OMORE bit in the *mode* field of the request. The server then allocates a valid (unique) descriptor and sends it back to the client in the R-message so that it can be used in the subsequent requests for that file. When the client issues the last request (or the server the last reply) the descriptor is deallocated and NOFD is sent as *fd* in the reply. Note that even though the client must issue one last request to cause the descriptor to be deallocated, this can be done once the application is done with the file so that this round-trip-time is never experienced by it (or the user).

This allocation of descriptors by the server in response to *put* or *get* requests permits doing an implicit *open* on a file in the first request of a series, and an implicit deallocation in the last one, without requiring extra RPCs for *open* or *close* operations. Furthermore, because the file path is still sent in requests using descriptors, a server that crashed and restarted may easily recover a descriptor lost. It would allocate a new one for the given path and reply with the new descriptor. The client would use the new one in further requests for the file. That is to say that, as far as the protocol is concerned, it is easy to make servers behave as stateless ones (even though they have state). The loss of the file descriptor means that any ongoing request on that descriptor is finished (be it performed or aborted) but this is of course what happens when a server crashes and recovers. Thus, semantics are reasonable for applications.

### 5.2. The put transaction

The Tput request asks the server to update the file identified by either *fd* or *path* in the message, perhaps also creating the file. We reproduce here the format of the messages for the convenience of the reader.

```
size[4] Tput tag[2] path[s] fd[2] mode[2] stat[n] offset[8] count[4] data[count]
size[4] Rput tag[2] fd[2] count[4] qid[13] mtime[4]
```

The *mode* field carries several bits that determine what has to be put: ODATA, OSTAT, OCREATE, and OMORE. A Tput with the ODATA bit set updates file data in the server, placing in the file at position *offset*, the number, *count*, of bytes sent in *data*. When this bit is not set, *count* is zero and the message does not carry any *data*. But note that it is legal to specify ODATA and use zero as *count* to issue a write of zero bytes (which is sometimes used by some devices as a signaling mechanism).

A Tput with the OSTAT bit set updates file metadata, as indicated by the *stat* field (using the same format used by Styx). Only metadata fields not set to null values are

honored, as in Styx. A *Tput* request without this bit set in the *mode* field does not send the *stat* field through the communication channel.

A *Tput* request with the *OCREATE* bit set creates the file if it does not exist, and truncates it to zero bytes otherwise. The write offset is still obeyed even when *OCREATE* is specified, for messages that also specify *ODATA*. Also, note that using a single request to mean both creation and truncation removes the usual race between open with truncation and creation. Addressing this race in Inferno using Styx required several RPCs just to reduce the race window.

The reply message *Rput*, returns the number of bytes written to the file, and it is considered an error for the user when they are less than the number indicated in the *count* field of the *Tput* request. Although an *Error* message may be returned instead, in case of errors, to report the error and its cause. To help clients caching file contents, an *Rput* reports both a file *qid* and *mtime* back to the client. The *qid* contains a unique number for each file, and a version number that increases for each update to the file. In this case, the *qid* reported corresponds to the file after the *put* request has been processed.

With this transaction, a client may create, write, and adjust permissions (or other metadata) with a single RPC. As an example,

```
Tput tag /a/file NOFD OCREATE|ODATA|OSTAT (nemo,nemo,0664,...) 0 5 hello
Rput tag 5 NOFD qid mtime
```

creates */a/file* in the server, sets the ownership in its directory entry to user *nemo*, group *nemo* and use permissions *0664*, and finally writes 5 data bytes on it. If the file exists already, it is truncated (because of *OCREATE*), otherwise it is created by the RPC.

The *put* transaction permits the creation of directories and modification of the respective metadata as well. For the protocol, directories are files that contain directory entries, in the format of a *stat* field. To create a directory, the *stat* field must have the *DMDIR* bit set in the *mode* field (used to carry file permissions). In this case, *ODATA* is not allowed. Metadata can be set as for files.

### 5.3. The get transaction

The *Tget* message asks the server to retrieve data (file data or directory contents) or metadata for the file (directory) identified by either *fd* or *path*. We reproduce the involved messages here for the convenience of the reader.

```
size[4] Tget tag[2] path[s] fd[2] mode[2] nmsgs[2] offset[8] count[4]
size[4] Rget tag[2] fd[2] mode[2] stat[n] count[4] data[count]
```

The *mode* field in the request, like before, has bits *OSTAT*, *ODATA*, and *OMORE* that can be set independently.

The *OSTAT* bit asks the server to respond with a message including the directory entry for the file, in the *stat* field (it would have the *OSTAT* bit set as well). When *OSTAT* is not set, the *stat* field is not sent through the communications link.

When *ODATA* is set, the server is being asked to reply with at most *nmsgs* messages, each with at most *count* bytes of file data (and the *ODATA* bit set). A zero value for *nmsgs* means that the server may generate "any number" of replies, as needed to retrieve the entire file. Data retrieved starts at *offset* in the file. The reply (each one!) to a *Tget* includes the number of bytes retrieved, reported in *count*, and the actual *data*. All replies include the *count* field, although it might be zero for requests with just the *OSTAT* set in their *mode* field. The end-of-file indication is explicitly signaled by a reply that does not have the *OMORE* bit set. This permits devices to reply with zero byte messages while still having this bit set, to avoid signaling an end-of-file, which can be useful when accessing special devices.

For example, a single RPC is needed to obtain both file metadata and all data, assuming the file is up to *MAXDATA* bytes long:

```
Tget tag /a/file NOFD ODATA|OSTAT 1 0 MAXDATA
Rget tag NOFD ODATA|OSTAT (nemo,nemo,0664,...) 5 hello
```

A single get request may grant the server the right to send multiple replies, to let it stream data to the client (the transport protocol is assumed to deal with congestion and flow control). The series of replies for a single get request completes when one of the following conditions hold, whichever one happens first: (1) there is no more data in the file past the offset used; (2) an error happens; (3) the maximum number, *nmsgs*, of replies have been sent. The descriptor is implicitly deallocated by the server in the first two cases.

For requests targeted at directory files, *nmsgs* is always considered to be zero (no matter its actual value). As a consequence, when reading a directory its entire contents are sent back to the client, irrespectively of the number of replies needed to do that (each reply contains an integral number of directory entries), which simplifies the atomic reading of directories by the server. The key assumption here is that directories are not very large.

## 6. Data I/O and latency

For small files, a single transaction suffices to read/write the entire file, as already illustrated. For large files, multiple communication rounds are needed. However, the latency experienced by the client (and ultimately the user) can be greatly reduced, both for reading and writing.

Reading a large file can be achieved efficiently if the client lets the server send multiple replies for a single request. For example, to obtain the metadata and contents of a file that is up to *N* times the maximum allowed data size:

```
Tget tag /a/file NOFD ODATA|OSTAT N 0 MAXDATA
Rget tag fid ODATA|OSTAT|OMORE (nemo,nemo,0664,...) MAXDATA data
...
Rget tag fid ODATA|OMORE MAXDATA data
...
Rget tag fid ODATA|OMORE MAXDATA data
Rget tag NOFD ODATA k data
```

This scheme saves latency because after the request, the client may receive data continuously, without having to issue any additional request. The total delay experienced by the client is close to what would be needed for a single RPC. Moreover, because the server is actually sending multiple reply messages, instead of a single huge one, the same (shared) communication channel may still be used to multiplex replies for other requests issued by different clients, while a long transfer is in progress.

The protocol can tolerate large latencies also for writing data to the server. In this case, the client may simply issue *all* requests, one after the other, and then wait to receive the corresponding replies. For example:

```
Tput tag1 /a/file NOFD ODATA 0 MAXDATA data
Tput tag2 /a/file NOFD ODATA MAXDATA MAXDATA data
...
Tput tagn /a/file NOFD ODATA (n-1)MAXDATA k data
...
Rput tag1 NOFD MAXDATA qid mtime
Rput tag2 NOFD MAXDATA qid mtime
...
Rput tagn NOFD n qid mtime
```

Again, latency corresponds to that of a single RPC. Note that the requests did not request the allocation of a descriptor, because the client did not want to wait for the server to reply before sending the subsequent requests. Of course, the server might have responded with an error reply to any (or to all!) the requests sent. Error reporting is important for writes that correspond to device control operations, but this is discussed later.

Note that the approach for dealing with latency is not the same for both writing and reading. This asymmetry arises because, when writing, the client usually knows in advance how much data it wants to send. It can therefore simply go ahead and issue all requests, without waiting for a reply (of course, it has to process the replies as they arrive). On the contrary, when reading, the client has no idea about how much data is actually available, at least not before the first *Rget* has been received, so it cannot a priori issue multiple *Tget* requests. Also note that device files usually report a length of zero, so that this information would not be available at all. This problem is nicely solved by letting a client give "credit" to a server for sending up to *nmsgs* replies, permitting it to start sending replies, one after another, without having to wait for further requests. Streaming devices are a particularly interesting case which is trivially supported by leaving the number of replies open; the client can always abort the request by sending a *flush* request in an asynchronous fashion.

## 7. Bridging Styx: Ofs and Oxport

The only two programs currently speaking Op in our system are a server, called *Oxport*, and a client, called *Ofs*. These are used to bridge different islands of Styx speakers (with good internal connectivity) across bad-latency links, in a transparent way. *Oxport* exports the files it sees (i.e., its name-space) to Op clients (instances of the *Ofs*). In turn, *Ofs* implements an Op client that imports a file tree from an Op server (an instance of *Oxport*) and makes it accessible to native clients via the standard Styx file protocol, as a Styx server.

*Ofs* is implemented in Limbo using 2894 lines. *Oxport* is implemented using 533 lines. The difference in complexity is due to the fact that *Ofs* is also a Styx server (besides being an Op client), and because *Ofs* includes the code to maintain a full disk cache in the client. Both programs use a small library that provides some support in the implementation of Op servers/clients, which accounts for (additional) 1003 lines in Limbo.

The application of this scheme in Octopus is illustrated in figure 4. File servers that provide system services speak Styx. They are mounted in the name-space of an *Oxport* process, and are exported via Op to a remote machine. This means that all devices think that everyone is speaking Styx. In the same way, application programs use file system calls (operations) to access files, and Inferno behaves as a Styx client on their behalf. Again, applications think that everyone is speaking Styx. However, part (or all) of the file tree seen by a particular application might come from a *Ofs* process, i.e., ultimately from a remote server.

The implementation of *Oxport* and *Ofs* is not hand-tuned for performance, and is not employing optimizations reported in the literature. For example, we are not (de)compressing data for delivery, and we are not caching raw transmitted data to compress the transmission by sending handles to data already transmitted. All this would improve the effective bandwidth available rather than the perceived latency. File systems addressing bandwidth problems, like LBFS [6], are excellent examples of the techniques mentioned.

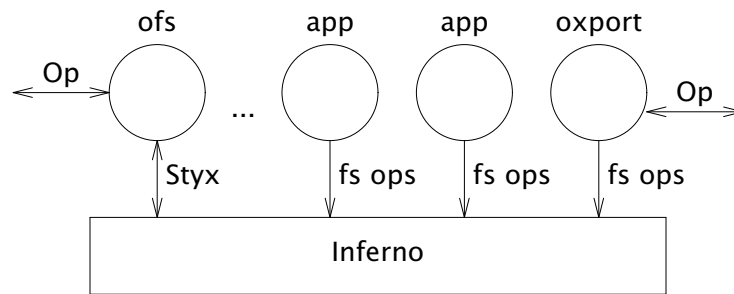


Figure 4: System software running in a Octopus terminal

### 7.1. Styx–Op translation and coherency

In general, the idea is to translate a series of Styx-based RPCs like

*Twalk / Rwalk*  
*Tstat / Rstat*  
*Tclunk / Rclunk*

or, for example,

*Twalk / Rwalk*  
*Topen / Ropen*  
*Tread / Rread*  
 ...  
*Tread / Rread*  
*Tclunk / Rclunk*

or both series, as issued from Styx clients (e.g., the Inferno system kernel), to a single

*Tget / Rget*

RPC between *Ofs* (acting as a Styx server and Op client) and *Oxport* (acting as an Op server and possibly Styx client). Along the same line, a series of

*Twalk / Rerror*  
*Twalk / Rwalk*  
*Tcreate / Rcreate*  
*Twrite / Rwrite*  
 ...  
*Twrite / Rwrite*  
*Tclunk / Rclunk*

would be ideally translated to a single

*Tput / Rput*

RPC between *Ofs* and *Oxport*.

After some contortions, our current translation from Styx (as a server speaking to the Styx client) to Op (as a client for a remote Op server) behaves as follows:

- Each *Twalk* request leads to a *Tget* request to obtain file metadata (plus at most the initial *MAXDATA* bytes from it). This information is cached.
- *Tstat* requests are served from the cached (meta)data if issued shortly after the metadata was retrieved. This introduces a coherency window in our deployment of the protocol in Inferno. This also implies that a *Tattach* request requires a *Tget* to the server (because it would only require a stat on the file).
- A *Tstat* request for a directory, besides fetching its metadata, also results in fetching its contents, via a suitable *Tget* request towards the Op server. It is assumed that the server reads the directory atomically, sending back to the Op client as

many replies as needed to report its entire contents. Directory contents are cached along with the metadata retrieved.

- *Tread* requests are served from the cached data, when available. When reads are made for data beyond an initial cached prefix for the file, we employ further *Tget* requests to gather more data (but do not cache it)<sup>1</sup>.
- *Open* requests are served locally, and not seen by the server. Permission checking is done by our Styx-to-Op client program, to try to report access errors before further put or get requests.
- *Twrite* requests are always sent to the server, as *Tput* requests. In general, it is necessary to deliver write requests to the server synchronously (ie., before responding to the client) to notify the client program of any error related to the write, but this is not always the case in our implementation (as discussed below).
- *Tcreate* requests are handled locally, awaiting for a subsequent write request. For created files, the *Tput* request sent upon a write creates the file, if necessary.
- *Twstat* requests are also deferred until a further write pushes changes to the server.
- *Tclunk* requests push any change made locally (creation and/or metadata) to the server using a *Tput* request. Although we try to make this never happen.

Deferring file creation until the first write means that it could fail, but the client program would not notice that until the write happens; notably, the client may never issue a write! As a precaution, we do check out permissions and other erroneous conditions (in the Op client) to catch obvious error cases locally. Of course, this is not a real fix, but this is the price for combining a *create* and a *write* into a single *put* request.

A further optimization is that *Twrite* requests, which 1) are not at offset zero, and 2) correspond to a full *Twrite* data packet, are sent (as usual) using synchronous *Tput* requests, but the corresponding *Rwrite* reply is sent to the Styx client *without* waiting for the respective *Rput* reply from the Op server. The net effect of this optimization is that errors, if any, are communicated back to the Styx client in a synchronous fashion only for the first and last writes (unless the last write happens to be a full one!). Errors caused by intermediate writes are detected asynchronously, and are passed on to the Styx client as a result of the next write operation issued by it. Note that control operations (for device control files) sent in *Twrite* requests do not usually fill an entire data packet, thus will result to an immediate communication of an error back to the client. For normal files, the only error that might happen given these rules corresponds to full disks, and that would be detected in the last packet (also by other programs sharing the remote disk). In general, we assume that we will never see such an intermediate write failing without this error being detected in the last write, and have placed a diagnostic message in our implementation to notice if that is the case (never seen after months using the protocol daily). In any case, this is the price for delaying some writes.

At some point, we considered having two different modes of operation, write-through and delayed-write, triggered by a command line option. The former could be used to import devices, and the later could be used to import conventional files (stored on disk). However, employing both options at the same time for the same server did not seem a clean thing to do. In the end, we decided to try to employ a single coherency model that works acceptably for all the cases, even if it requires more RPCs than strictly necessary.

Regarding reads, note that stating a directory has the side-effect of retrieving its

---

<sup>1</sup> In our implementation, when the on-disk cache is enabled, this data is cached as well, but we are not considering on-disk caching on this paper. The difference in performance gained by enabling the disk cache on the client is not noticeable for small files.

entire contents. This is a good thing for bad latency links, because the RPC used to check out the directory is also enough to update the client's view for it. Combined with the next optimization, this has a tremendous impact on latency.

In many cases, a single system call in Inferno leads to several different Styx dialogs (perhaps even using different *fids*) similar to the ones shown earlier. For example, trying to create a file named `/a/b/c` would walk to `/a/b`, then try to stat `c`, and then create (or open with truncation) it. Because of the rule that each *Twalk* request checks out with the server the validity of the cached file information, this means that a single `create` may lead to multiple *Op* requests. Our implementation of *Ofs* accepts a parameter defining a *coherency window*, in milliseconds. If cached file information is within that window, no new *Tget* request is issued to validate (or invalidate) the cached entry. Although this does increase the window of opportunity for races, it does not introduce new race conditions on its own. Indeed, a client does not know if after a *Tget* request (but before the next one) the involved file has changed. Such higher-level consistency guarantees have to be provided by employing appropriate application-level protocols anyway.

One surprise for us was that there are many Styx RPCs issued just to discover that a file does not exist. When the file exists, it is promptly used. Else, this fact is checked further times, e.g., during the implementation of the `create` system call. Exploiting the coherency window so that a file that has been recently checked for existence within a given timeframe is considered as not-yet existing, further reduced the RPCs issues between *Ofs* and *Oxport*.

## 7.2. File descriptors

It is important to note that *Op* file descriptors are not the same as Styx *fids*. The former are allocated by the *Op* server whereas the latter are allocated by the Styx client. Of course, when an *Op* server *Op* merely re-exports a conventional file system to an *Op* client, it may decide to map the same *Op* file descriptor numbers to the same file handles. But this is left open for the server to decide, i.e., the *Op* specification does not provide any defined semantics.

The current implementation of *Ofs* tries to map all client processes reading the same file to the same *Op* file descriptor, if available. This is also done for all clients that write the same file. Notably, a file being used both to read and to write would result in using two different *Op* file descriptors, because, at the level of the *Op* protocol, a file is either being updated or retrieved. The benefit of this implementation policy is that *Oxport* most likely ends up allocating a single file handle for servicing all Styx clients that access the same file using the same mode, read or write, respectively. Moreover, it becomes possible for *Ofs* to achieve an efficient sharing of the cached file metadata and prefix for all such clients, which further reduces the number of RPCs performed over the network towards *Oxport*.

What we miss out with this approach is the ability to distinguish between different client processes using the same descriptor. One consequence is that *Oxport* cannot detect whether a particular Styx client has closed the file. The reason is that *Ofs* would release the *Op* file descriptor when one Styx client clunks (releases) a valid corresponding *fid*. Further requests for the same file would reopen the *Op* descriptor if needed. This is so to signal that one request of completed, to the the device collect and process it. Another problem is that we lose the ability to properly support `clone` files. This would require *Ofs* to request a new file descriptor for each different client request, and *Oxport* to actually map each different file descriptor to a different file handle. Instead, as a compromise, we decided to replace clone devices by letting resource allocation be explicitly performed via file creation. This also makes the interface more amenable from the shell. To avoid name collisions between concurrently executing clients, one may randomize the file names being proposed, which is easy to do. For example, a process creating a file that represents a new resource in a device (previously implemented by

opening a clone-file) may use its process id or a random number as part of the (nice) file name.

Styx file servers must behave correctly regarding directory updates. Because a client might be reading a directory while another program might be creating/removing files on it. Servers take a snapshot of directory contents and then serve portions of the snapshot in further directory reads. The snapshot is discarded when the client is done with the fid. In our case, reading directories in a consistent, atomic way is achieved by making it mandatory for directory contents to be sent entirely in a series of Op reply messages to the client. The server discards the directory snapshot after sending the last reply message.

### 7.3. File names and surprises

In Op, we decided to always use absolute paths. For all requests. This avoids the problems of using relative paths (and having to clean them up) in the servers. Moreover, this has an extra benefit. The directory entry for a file (ie., the *stat* field in Op messages) contains the file name in Inferno. However, the name may differ from the one known by the client! That is because of the client's mount table. For example, the name reported by Plan 9's Fossil [10] for the root directory is *active*, although such file is known to clients as */*. To avoid confusion, our Op server always uses the file name found in the request (without the previous path elements) to fill up the *stat.name* field in replies to the client. Thus, Op clients can forget about this particular problem.

## 8. Mistakes and optimizations

The protocol is young, but it has evolved quickly as a result of the mistakes we made and as a consequence of using it "on production", for daily work, from the first day we had a prototype running.

Initially, the protocol did not have file descriptors. We thought that devices could collect resources after some time of client inactivity (similar to using leases). This does not work for files where read requests block potentially for long times as a mean to deliver future events to the client. These kind of files are common in Plan 9 and Inferno, for instance mouse and keyboard device drivers. A file system that provides applications with event channels is another example. Even without considering streaming (blocking) files, the scheme did not work well, because devices need to know when a request made of multiple writes is complete. The alternative was for the device to check out the data for validity, to try to determine by itself when a request was completed. This placed unnecessary load on the device servers.

Another mistake was that, initially, the length of coherency window was fixed. By allowing this to be set when an *Ofs* client is launched, we can establish Op links with almost a negligible (or null) window for devices that we know that change fast, but also permit more lag for remote file trees that do not feature any (such) devices. This flexibility is important, and can be even automated by extending the protocol to let file servers report the suggested coherency policy to the client during the attachment phase.

Not permitting a potentially infinite number of replies for get requests made the client unable to ask a server for a continuous stream of replies. Having a *flush* request, and flow control in the transport, makes such stream a desirable property. We introduced the *nmsgs* parameter in get requests to address this issue. A related mistake was that *nmsgs* was not implied as unlimited (zero) for directories. Thus, our client had to do an initial get just to discover that it is accessing a directory, and then read it all. This significantly hurt performance for bad latency.

On the early stages of design and experimentation we considered accessing big files. However, in this case it is bandwidth the problem and there is not much that we can do (besides applying techniques found in the literature like those of LBFS [6]). For

interactive usage of the system, most of the files we use for our daily work are not larger than 1 Mbyte. So we optimized the protocol thinking on small files. It was a serious mistake not considering the file size distribution (shown later) before deciding how to design the protocol.

Although we thought we were going to need bandwidth optimization techniques soon, that has not been the case. Using remote machines just as terminals means that most of the computations and file accesses are performed within a single (remote) computer, similar to what happens when using the Web. It is the access latency for small files what matters (at least according to our experience).

Many device files seem to have zero-length. In this special case, we do not even cache a prefix for such files, and we ignore the length reported by the file metadata. Any other option can lead to a wrong behavior, e.g., failing to provide up-to-date status information, or reporting an end-of-file for an (endless) stream.

Clone files for devices mean problems for protocols, unless they have fids. But fids mean that multiple clients cannot share data known by the client without asking the server. After trying both with and without fids, in the end we compromised by introducing file descriptors in the protocol as described before.

We have implemented a version of the *Ofs* program that exploits on-disk cache (on the client machine). This is uninteresting otherwise, but one mistake we made in this respect was not placing a limit on the amount of data cached per file. There are files that behave as streams, and which can easily flood the client's cache. Consider files used to access the network, for example. The zero-length hint described above is likely to catch this case, but a limit must be placed for cases when it does not.

## 9. Alternatives discarded

Before building Op, we considered several alternatives, which are briefly discussed here.

Our first thought was to add a *put* and *get* operation to 9P (or Styx). An equivalent (more clean) approach suggested by Sape Mullender and Russ Cox was to accept that a client could send multiple *T-messages* with the same tag to the server. The servers (aware of this protocol change) would perform each one in turn, and abort the entire sequence upon errors.

The problem with this approach is to answer the question: How do we know which requests correspond to a full *get* or *put* operation? Knowing that requires two new system calls, to let the application specify that an entire file is being put or get. Otherwise, a caching model similar to that described here must be employed. Such a change would also have been quite complex. For one thing, the mount table in Inferno (and Plan 9) requires to walk a file before deciding if there is anything mounted on each level. Thus, despite a new *get* or *put* operation, multiple RPCs would be needed. Changing the mount table with a prefix table [14] would fix this problem. But that is a severe change to the system, similar to our previous editions of Plan B.

On the contrary, our approach permits all programs and protocols to run unchanged, while making it possible to install bridges between Styx islands, via *Ofs* and *Oxport* using Op. Although not strictly related to the protocol, we considered initially to run different instances of *Oxport* and *Ofs* for each different device exported from a terminal to the central PC. However, using a single connection permits TCP to adapt nicely to the available bandwidth, without bothering other users of the network, and it is cheaper in any case. Thus, in the Octopus we mount a single Op server for each machine we talk to, and different subtrees from that machine are rebound (linked) later as desired.

Another idea, motivated by file systems like Coda [12], was to employ *session* semantics. In other words, to pull entire files from the server at *open*, and push any

changes back to the server at *close*.

But there are several problems with this regarding how to do it for Styx and Inferno. To begin with, a file (or some information about it) might be needed before trying to open it. For example, during file name resolution, a *Twalk* request needs to know if the named file exists, in which case it also needs to obtain a *qid* for it. Furthermore, there is no *close* system call in Inferno. A file descriptor is closed when no reference points to it. The Inferno garbage collector is used to detect that fact. Although we could still push changes upon a *Tclunk* request (at garbage collection time), there would be no way to report errors back to the application that updated the file. Last but not least, session semantics are quite inappropriate for device files.

## 10. Evaluation

In this section we give some indicative performance results. All experiments were done at the application level, using conventional file operations and commands. Comparison is between a normal Inferno setup using Styx and a customized setup using *Ofs* and *Oxport* via *Op*.

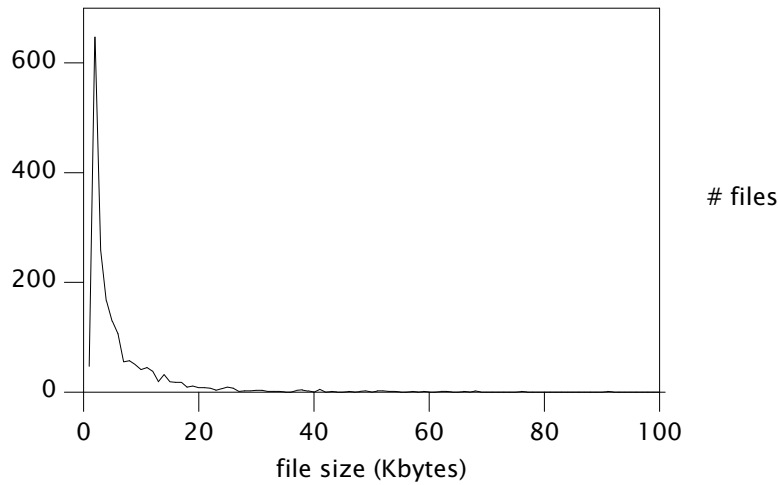
We use three different workloads: compiling the code of our prototype, cleaning up the distribution, and listing directories. These are quite representative of an interactive usage of the system, which is the problem we are addressing. In the first case, many files are being read (module interfaces, libraries, and some binaries), and some files written (and created). In the second case, directory reads and file remove operations dominate. In the last case, most of the operations are reading file metadata. In all cases, we used real Inferno commands and the *time* Inferno tool to measure the time required (by averaging multiple measures). All measures (unless said otherwise) were taken using an Inferno system hosted on MacOS 10.4, using a 2.3GHz Intel Core2 Duo machine. The network connection was either the loopback IP interface or a 1Mbyte ADSL line.

Overall, using *Op* starts paying off when the round trip time (RTT) is above 1ms. For more realistic latencies (for ADSL lines) of 30ms or 50ms of RTT, using *Op* makes the difference between being able to (interactively) use a remote file server or not. We also tried most of the experiments described here with an on-disk cache on the client machine. The effect was not noticeable and we omit the results. This is a result of caching a small prefix of each file as a result of the data retrieved by *Tget* requests, and also of the small file sizes employed by Inferno.

The file size distribution of a complete Inferno (including documents, binaries and sources) is shown in figure 5. Most files are small, although there are a few larger files (up to several megabytes). We expected this, but it was a surprise how small the files required for using Inferno are. Systems using byte code binaries like Inferno tend to exhibit smaller files than systems using actual binaries.

As a concrete example, reading all the source and manual files (one file at a time) from the Octopus (400Kbytes) using *Op* required 63.73 seconds across a real ADSL line with 52ms of RTT according to ICMP. Using Styx required 86.06s. In this case, *Ofs* was configured to maintain strict coherency, by checking out with the server for each *walk* request. That is, the coherency window was set to zero. This is a worst case for our prototype, because multiple different files (each requiring at least one RPC) are being retrieved and also because *Ofs* is not exploiting data already known.

The total number of requests issued via Styx and *Op* (with null coherency window), respectively, is given in Table 1 for the session below:



**Figure 5:** File size distribution in the Inferno OS (also in the Octopus and experiments described).

```
o/ofs -v -m /n/pc tcp!127.0.0.1!16699
% cd /n/pc/usr/octopus/port/ofs
% mk clean
rm -f *.dis *.sbl
% mk
limbo -I/n/pc/usr/octopus/port/module -gw ofs.b
limbo -I/n/pc/usr/octopus/port/module -gw ofsstyx.b
limbo -I/n/pc/usr/octopus/port/module -gw stop.b
limbo -I/n/pc/usr/octopus/port/module -gw mcache.b
% cd
% unmount /n/pc
```

Styx		Op	
Request	#RPCs	Request	#RPCs
<i>walk</i>	102	<i>get</i>	134
<i>open</i>	34		
<i>create</i>	8		
<i>read</i>	52		
<i>write</i>	32	<i>put</i>	32
<i>clunk</i>	66		
<i>stat</i>	0		
<i>remove</i>	0	<i>remove</i>	0
<i>wstat</i>	0		
<i>total</i>	296	<i>total</i>	166

**Table 1:** RPCs for Styx and Op needed to clean up and recompile our source.

Table 2 shows the effect of the (more realistic) usage of *Ofs* with a coherency window of 1s and 2s. This execution was performed on a real ADSL line while the RTT was 85ms. Table 3 shows the reported bit rate achieved for data read by the protocol, and data written by the protocol, for the same setting, according to the *lostats* tool of Inferno. In this case, Op data is for a coherency window of 1s. Note that transfer rates are computed by *lostats* as seen by the client machine (by considering how much data is

sent/received and how much time was required for the Styx RPCs to complete).

Protocol	lc	mk clean	mk
<i>Styx</i>	2.314	30.6	87.5
<i>Op (1s)</i>	0.76	2.93	34.02
<i>Op (2s)</i>	0.142	2.58	30.37

**Table 2:** Times (s) for *lc*, cleaning up, and compiling for *Styx* and *Op*. (85ms RTT).

Protocol	direction	lc	mk clean	mk
<i>Styx</i>	<i>read</i>	97633.19	8677.66	104143.81
<i>Styx</i>	<i>write</i>	na	2928.37	5933.42
<i>Op (1s)</i>	<i>read</i>	4353804.98	50380.33	1163658.94
<i>Op (1s)</i>	<i>write</i>	na	2747.05	16228.07

**Table 3:** Tranfer rate (Kb/s) for *lc*, cleaning up, and compiling for *Styx* and *Op*. (85ms RTT).

The importance of these measures is in that they correspond to a real scenario, not to microbenchmarks. The files involved in the experiment were small files (a few Kbytes). For experiments with big files, bandwidth is the limiting factor, although the results shown here in terms of numbers of requests still apply. In any case, like it happens with the Web, most files are small when using file interfaces to use a machine as a remote terminal for another.

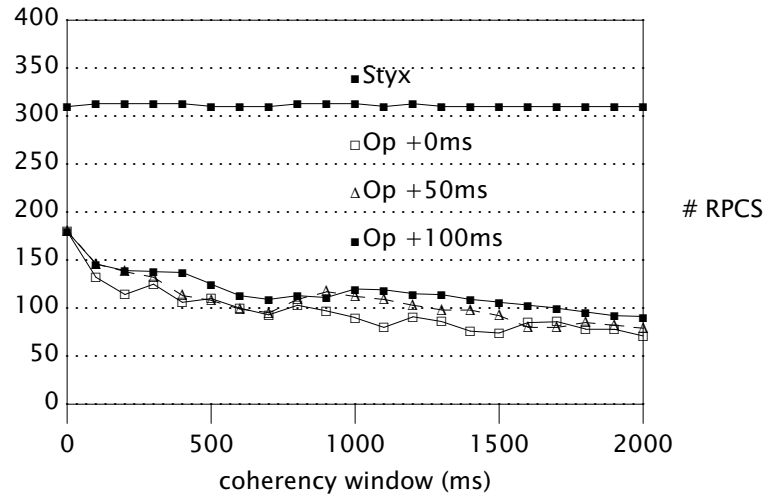
In what follows, we describe some microbenchmarks to show how the protocol and our prototype implementation behaves as a function of the RTT and the coherency window chosen. In all the cases, the measures correspond to listing a directory, and then cleaning up and compiling the entire source code of the Octopus system (as of today). This is similar to the session shown above, which nicely represents an interactive use of the system (which is what we built *Op* for).

To perform these experiments in a controlled way, *Ofs* and *Oxport* where speaking *Op* through the loopback IP interface, using TCP. The latency of the communication link was regulated by introducing an extra delay in *Oxport*. Since *Oxport* attends different requests using different processes, bandwidth is not affected, just latency. The added delay affects the RTT, as witnessed by *Ofs*. The setting used to regulate the latency for *Styx* was via a pipe, not a network interface. Thus, the presented measures favor *Styx*, because the pipe works better and has less overhead than a TCP connection via loopback. The standard Inferno program *Memfs* was instrumented to introduce delays for different requests in the same way described for *Ofs*.

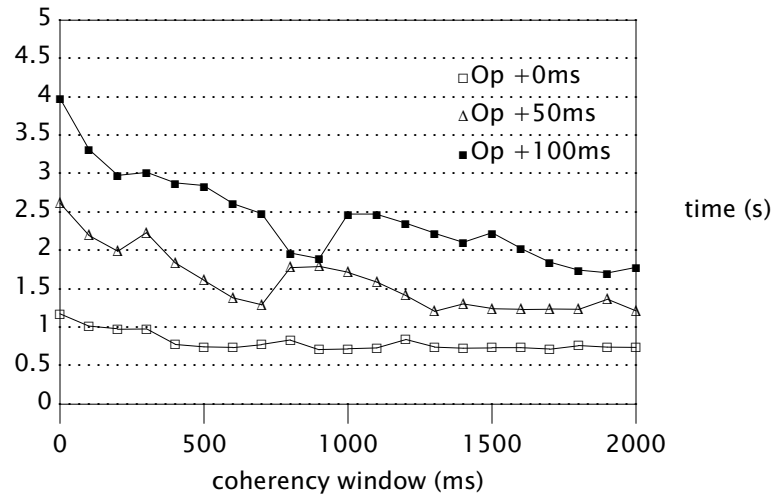
Figure 6 depicts the effect of the coherency window in the number of RPCs required for *Op* for an RTT delay of (almost) zero, 50ms and 100ms. The (fixed) number of RPCs required in *Styx* is shown as a reference. Figure 7 and figure 8 show the time in seconds (for the same experiment) required for running the `mk clean` and `mk` commands, respectively. Again, this is shown for different coherency windows and RTT delays.

The next two figures, 9 and 10, show the same results as figures 7 and 8, but also compare to the corresponding values obtained for *Styx*. It can be seen how even with a null coherency window, avoiding unnecessary RPCs represents the difference between having a system that could be used remotely and having one that cannot. Figures 9 and 10 imply that *Op* is doing a reasonable job trying to avoid extra RPCs, even for relatively small coherency window values.

Figures 11 and 12 report the times for *Styx* and *Op* with a coherency window of 2s, needed to run `mk clean` and `mk`, respectively, as a function of the RTT, starting



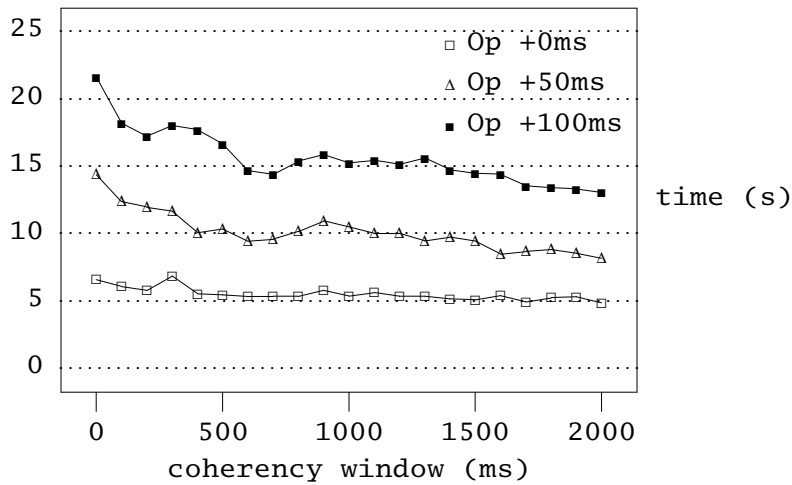
**Figure 6:** Number of RPCs required for Op with different coherency windows.



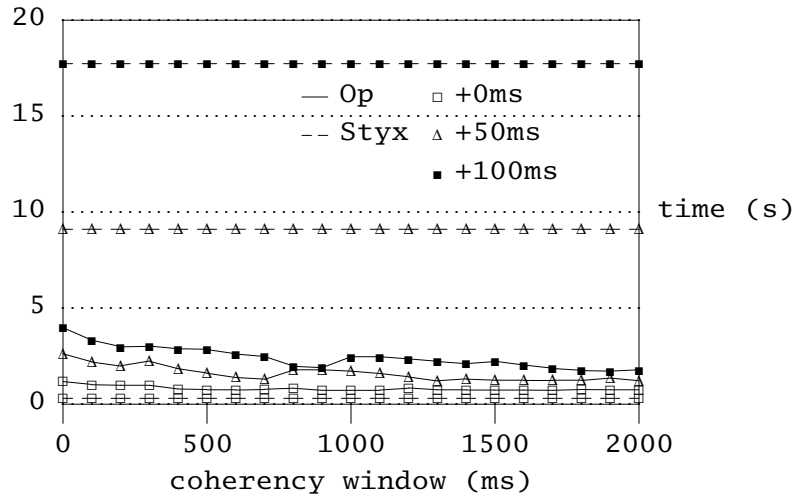
**Figure 7:** Time for a `mk clean` for different coherency windows.

from (almost) zero up to 50ms. It is clear that it already pays off to use Op for RTT times above 1ms. For smaller latencies, the overhead introduced by *Ofs* and *Oxport* becomes quite dominant, making Styx the best choice. The extra RPCs generated by both the Inferno system and the user programs to check if there are files that could be removed means that the effect of Op is even more noticeable when cleaning up than expected.

Because Styx might not be widely known, figures 13, 14, and 15 compare Op to both CIFS and NFS version 3. Experiments were performed using an Inferno hosted on a FreeBSD 6.2 system running on a Pentium 4 machine at 3.2 GHz with 1 GB of memory. The client and the server ran in the same computer and used the loopback interface to communicate. Latency was adjusted by FreeBSD's Dummynet [11]. No bandwidth limitation was set for the experiments. Figures present the average time required (in the same platform) to execute `lc`, `mk clean`, and `mk` in a setting similar to the one used for the measures above. The version of CIFS was Samba 3.0. The NFS version 3 implementation was that supplied with FreeBSD version 6.2. They were tested with the default configuration. NFS was tested over TCP, to be fair. Note that these measures



**Figure 8:** Time for a mk for different coherency windows.



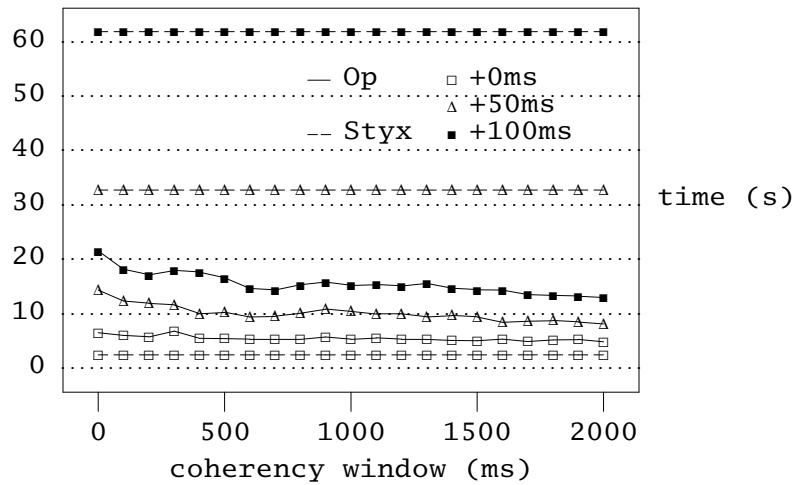
**Figure 9:** Compared time for a mk clean for different coherency windows and latencies.

should not be compared to the ones shown previously (for the comparison of Op vs Styx), because the experiment was different as a result of trying to measure all of Op, CIFS, and NFS using the same platform. In particular the loopback interface used for these experiments is a lot faster than the MacOS loopback interface, used for the previous experiments (to compare loopback measures and a real ADSL link while keeping the underlying platform unchanged).

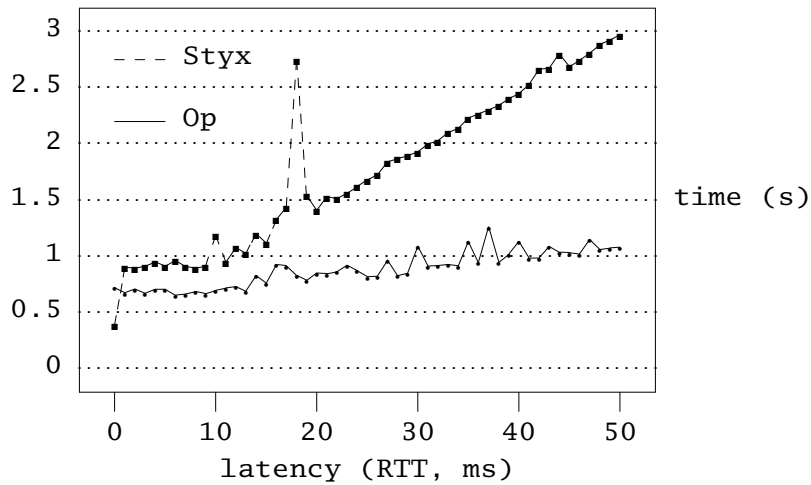
## 11. Related Work

Only the most significant related work is mentioned here, other systems fall in one of the following cases regarding differences to Op and its implementation for Inferno.

LBFS is a distributed file system designed for low bandwidth [6]. It also focuses on interactive usage, but its techniques are complementary to our work. In LBFS the client caches chunks indexed by their SHA1, that alone means that LBFS cannot be used for synthesized files used by devices. Also, to read a file not in the cache, it requires at least two RPCs (twice the RTT for Op). In general, it prefers to issue more RPCs to avoid sending data, and Op prefers sending data instead of requiring more RPCs.



**Figure 10:** Compared time for a mk for different coherency windows and latencies.

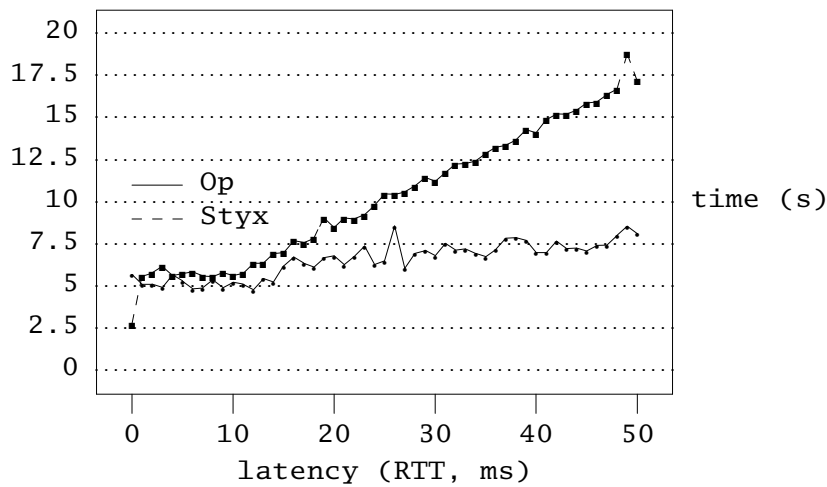


**Figure 11:** Compared time for Styx and Op for a mk clean for different RTTs.

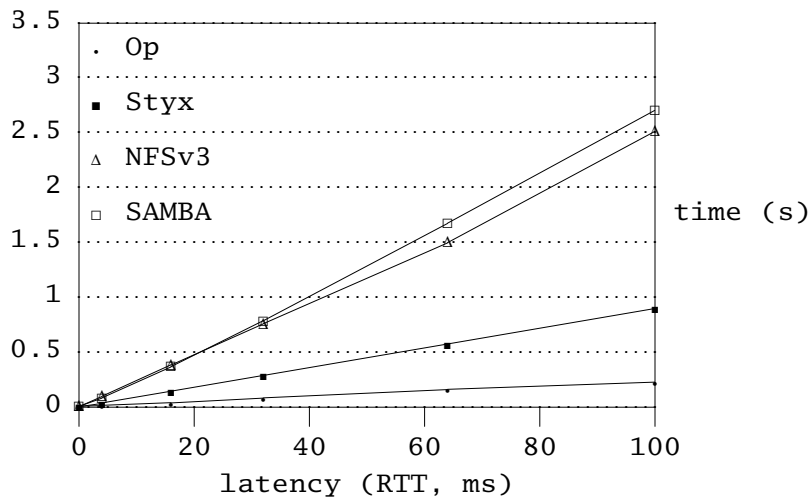
NFS version 4 [13] coalesces operations (by batching them) so that in many cases only an RPC is needed. However, it has not been designed for interactively using remote devices and is much more complex than our protocol. Also, as we have seen in Inferno, real usage of the protocol would mean that the (client) system would not know what to batch (besides batching for each system call). That would mean issuing multiple RPCs in practice. On the contrary, Op tries to compromise to work well for devices while still using a coherency window to be able to issue fewer RPCs.

Coda [12] is a distributed file system that supports disconnected operation by exploiting the client cache and allowing updates while disconnected. Coda and related systems, like Ocean Store [4], focus on mobility rather than on permitting interactive usage of a remote system with bad latency conditions and cannot be used to access remote devices. In general, systems that systematically defer or delay writes cannot be used to access remote devices, because of (lack of) error reporting. This includes systems using session semantics (like Coda).

WebDAV [15] aims at making the Web a large scale distributed file system, but it focuses on collaborative editing of web pages and it is not appropriate to replace a



**Figure 12:** Compared time for Styx and Op for a mk for different RTTs.

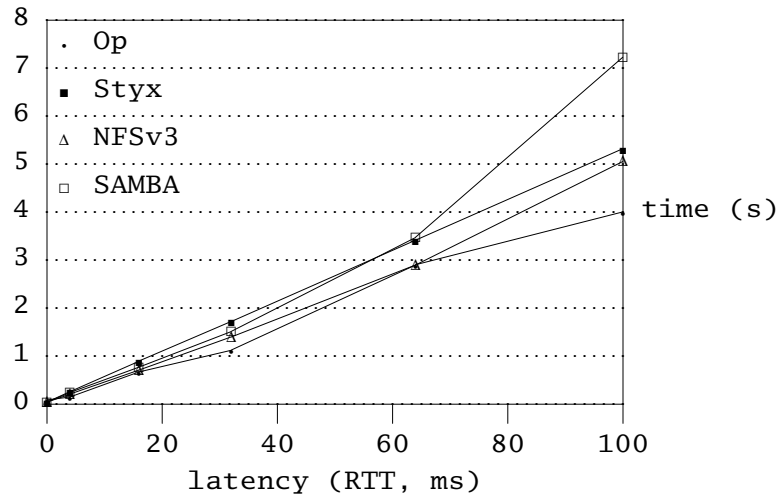


**Figure 13:** Compared time for Op, Styx, CIFS, and NFS (v3), for executing lc.

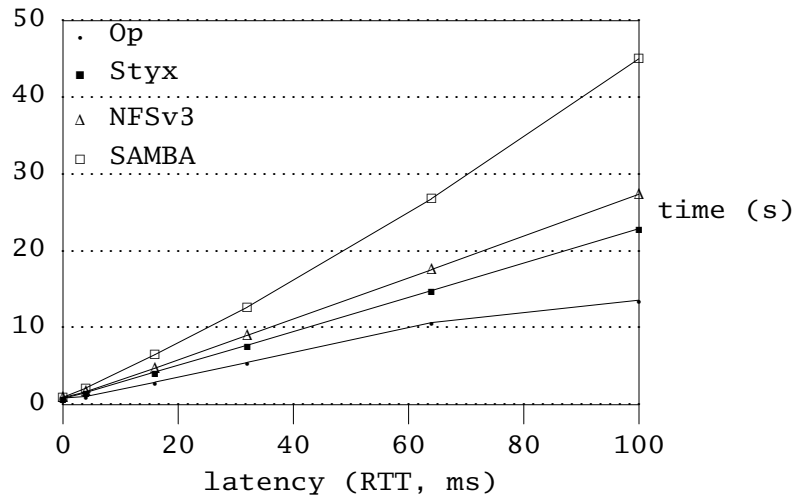
distributed file system. For example, it does not consider file protection nor does it include a standard directory entry (metadata) for files.

## References

1. F. J. Ballesteros, E. Soriano, K. L. Algara and G. Guardiola, Plan B: An Operating System for Ubiquitous Computing Environments, *IEEE PerCom*. Also <http://lsub.org>, 2006.
2. S. Dorward, R. Pike, D. L. Presotto, D. M. Ritchie, H. Trickey and P. Winterbottom, The Inferno Operating System, *Bell Labs Technical Journal* 2, 1 (1997), .
3. B. W. Kernighan, A Descent into Limbo, *Inferno User's Manual* 2(1997), .
4. J. Kubiawicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao, OceanStore: An architecture for Global-Scale Persistent Storage, *ACM SIGPLAN Notices* 35, 11 (2000), .
5. P. Leach and D. Perry, CIFS; A common Internet System, *Microsoft Interactive*



**Figure 14:** Compared time for Op, Styx, CIFS, and NFS (v3), for executing `mk clean`.



**Figure 15:** Compared time for Op, Styx, CIFS, and NFS (v3), for executing `mk`.

Developer, Nov. 1996.

6. A. Muthitacharoen, B. Chen and D. Mazieres, A low bandwidth network file system, *ACM Symp. on Operating System Prin.*, 2001, 174–187.
7. R. Pike, D. Presotto, K. Thompson and H. Trickey, Plan 9 from Bell Labs, *EUUG Newsletter* 10, 3 (Autumn 1990), 2–11.
8. R. Pike and D. M. Ritchie, The Styx Architecture for Distributed Systems, *Bell Labs Technical Journal* 5, 2 (April–June 1999), .
9. D. Presotto and P. Winterbottom, The Organization of Networks in Plan 9, *Plan 9 User's Manual* 2.
10. S. Quinlan, J. McKie and R. Cox, Fossil: An archival file server, *Lucent Technologies Bell Labs. Unpublished technical memorandum. Available at <http://www.cs.bell-labs.com/sys/doc/fossil.pdf>*, 2002.
11. L. Rizzo, Dummynet: A simple approach to the evaluation of network protocols, *ACM SIGCOMM Computer Communication Review* 27, 1 (1997), 31–41.

12. M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel and D. C. Steere, Coda: A Highly Available File System for a Distributed Workstation Environment, *IEEE Transactions on Computers* 39, 4 447–459.
13. S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler and D. Noveck, Network File System (NFS) version 4 Protocol, *Internet RFC3430*, 2003.
14. B. B. Welsh and J. K. Ousterhout, Prefix tables: A Simple Mechanism for Locating Files in a Distributed System, *Proceedings of the 6th ICDCS*, 1986.
15. E. J. Whitehead and Y. Y. Goland, WebDAV: a network protocol for remote collaborative authoring on the Web, *Proc. of the 6th conference on Computer Supported Cooperative Work ECSCW99*, 1999.