Op: Styx batching for High Latency Links

Francisco J. Ballesteros, Gorka Guardiola, Enrique Soriano

Rey Juan Carlos University, Spain
{nemo,paurea,esoriano}@lsub.org

Spyros Lalis

University of Thessaly, Greece
lalis@inf.uth.gr

ABSTRACT

We are designing an operating system based on Inferno, which centralizes everything on a machine we call the PC. Terminals connect to it through the internet to export local devices and viewers and programs wrapped in Styx file servers. The problem this approach faces is that Styx behaves badly for high latency links, no matter what the bandwidth is because of the RPC round-trip time accumulation. We have implemented a new protocol, Op (Octopus Protocol), which reduces drastically the number of RPCs needed for the operations and so behaves well for high latency links. Furthermore, we have implemented a client and a server that can be used to interconnect different Styx islands in a transparent way, keeping all other software unaware of the new protocol. The ideas here could be used, for example, for the Plan 9 sources at Bell Labs, in order to make upgrades much faster across the internet.

1. Introduction

It is well known that Styx and 9P behave badly for high latency links. In this article we focus on Styx because our protocol is implemented around it, but anything said here about Styx also applies to 9P. Even for simple operations, Plan 9 and Inferno issue a great number of RPCs. An RPC issuer waits synchronously for the responses before issuing new RPCs and so, the latency is accumulated, making the file system very unresponsive, especially for interactive use.

We are building a system called Octopus [2]. In Octopus, all the applications run on a central machine, the PC, connected to the internet. In principle, the PC can be any machine (we use Plan B and Inferno combined). Terminals are intended to run on a hosted Inferno instead. The aim is to use any machine nearby, no matter its OS, as a terminal for the octopus. We use Inferno programs to wrap terminal devices and applications installed in the underlying system. They are exported as devices, via file systems which are mounted by the PC to be controlled by the applications running on the PC. These applications keep all the state and the session as the user moves around. Everything exported by the terminal is mounted as a file system through the network, which means that responsiveness of the system heavily depends on the file system responding fast. For high latency networks, using Styx proved unusable for us, so we needed a file system protocol which would behave well under bad latency.

Before anything else, we must make explicit what we mean by “high latency links”. One example of this is the 120ms of RTT for ICMP (using `ip/ping`) we get while at our homes through an ADSL connection to the server in our University. Another example is the 118 ms of RTT (also of ICMP) we get while reaching the Plan 9 sources at Bell Labs from our University.

File system protocols capable of handling such latencies are feasible, as demonstrated constantly by the web. In principle, a single RTT could suffice to fetch a file (if there is no bandwidth problem and the underlying network stack permits it) or to put a file into a remote file server.

Before undertaking the construction of the protocol described here, we considered several alternatives. One was to make changes to Styx following suggestions made at the IWP9 2006. Another one was using approaches similar to LBFS [5]. In the end we had to build our own protocol for reasons we describe later. We did so in a way that permits Inferno systems (hence Plan 9 systems) to bridge across high-latency network links in a (mostly) transparent way. By taking this approach, all programs can be kept without modification, speaking Styx when necessary and using the current standard file system interface.

The resulting implementation can be useful without the rest of the Octopus. For example, it could be used to make it more convenient to browse or work on remote Plan 9/Inferno systems. Indeed, this has been the primary use of the implementation described here, because the Octopus is not yet ready for daily use.

A known drawback of Op which comes as a direct consequence of its design is that files not behaving as proper files may not work. Two examples of this are *OEXCL* files of which the cache is not aware of, and clone files. Making the cache aware of *OEXCL* files is easy enough if one is willing to pay the performance price. Clone files are more complex because of the way the system is implemented as will be explained later.

2. Altering Styx

A proposal has been made to modify Styx to support batching of RPCs. The idea behind this approach is that the client can send RPCs using the same tag without waiting for the answers. The server would then group logically the responses. For example, an error in the first RPC would mean that other requests in the same batch could be ignored. The problem with this approach is that many programs have to be modified. This includes both clients and servers.

A batching client would be needed or the client system would simply issue Styx requests serially, adding up RPC latencies. A batching server is needed to honor the convention that requests with the same tag depend on previous requests within the batch to succeed.

How to implement that client is not even clear. The problem is to determine what and when to batch. For example, two new system calls (or library functions) could be added to `put` and `get` a remote file (as we did on Plan B). However, programs not written to use the new calls would still use series of blocking RPCs.

A better alternative would be to implement a batching file system for the client machine, providing a conventional Styx server, but trying to batch requests. This requires some caching in the client, otherwise there would be no batching in practice; because individual requests from the client system would be forwarded one by one to the server unless they can be satisfied by data from a cache within the client.

Regarding the server side, a similar issue arises. Instead of modifying all the servers, a single batching server can be implemented. Such server would honor the Styx batches and issue individual file system requests to the actual servers.

Figure 1 depicts the scenario described so far. After considering it, the approach to implement the simplest protocol needed to interconnect the batching client and the batching server arises naturally (instead of implementing a more complex Styx). In the next section we describe such protocol.

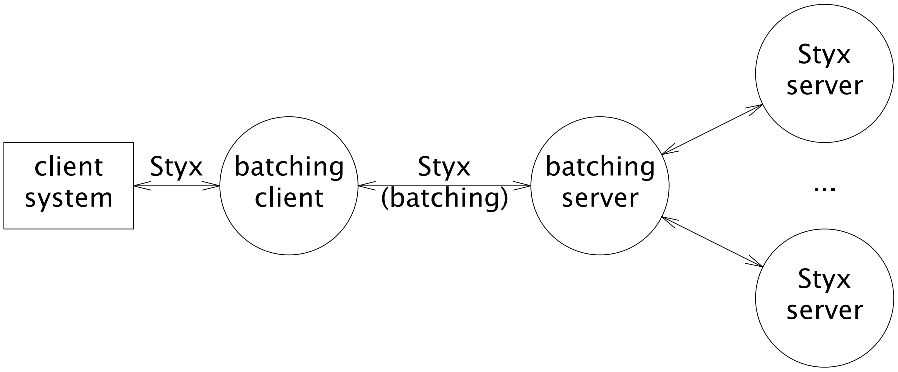

Figure 1: *A discarded alternative: Batching clients and servers for a modified Styx*

3. The Octopus Protocol: Op

Op follows many conventions used in Styx. We will focus here mainly on the differences, although we will also repeat many details taken verbatim from Styx to make the description easier to read. A more detailed description of the protocol and some evaluation measurements are available elsewhere [1]. In Op, the combination of sending (receiving) a request and receiving (sending) a reply is called a *transaction,* 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 2. They are encoded using the same conventions of Styx [6]. 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 (like in Styx, but in Op there can be multiple replies for a single request). Each T-message has a *tag* chosen and used by the client to identify the message. The reply (or replies!) to the message has the same *tag.* 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.

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 *size*[4] `Rerror` *tag*[2] *ename*[s]

size[4] `Tattach` *tag*[2] *uname*[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 2: *Messages in the current version of the Octopus Protocol, Op.*

left out of the protocol as in Styx. 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.

A reply for a T-message is either an appropriate R-message or *Rerror*, 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 later on).

3.1. 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).

In addition to the *path* field, both *Tput* and *Tget* may identify the file using the *fd* field, which contains a small integer representing a *file descriptor* to the file. Note that this *fd* is very different from a *fid.* 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). The *fd* compensates partly for the lack of *fids* in the protocol, though it has to be stated that an fd is a completely different thing than a *fid.*

File descriptors are allocated by the server upon request. This means that the client has to wait one RTT to issue more requests on the same *fd* which would be the case if the *fd* was allocated by the client. The reasons for the server to allocated the *fds* is that the Styx requests which do so can generate an error. This error needs to be conveyed to the application. This cannot be done until the application gets the response for the first request. Considering this, it is easier for the server to be able to crash and recover just by allocating new *fds,* without having to keep track of the ones used by the client.

A client may also specify in a *Tget* or 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 lost descriptor. 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 if they were stateless (even though they do have state). Losing the validity on the server 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.

3.2. 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 *count* bytes from the *data* field in the file at position *offset.* 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 metadada, 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. It is considered an error for the user when the number of bytes written is less than the number indicated in the *count* field of the *Tput* request. Nevertheless, an *Rerror* message may be returned instead, in case of error, 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 reported *qid* 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. 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.

3.3. 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.

4. Implementation

The implementation consists of a program called `ofs` that speaks Styx as a server and Op as a client; and a program called `oxport` that speaks Op as a server to export the name space where it runs. A convenience library to speak Op providing functions to marshal and unmarshal messages is used by both programs. Figure 3 depicts the basic scheme of the system.

Oxport is a single program and it is very simple. It runs on the server and uses file system calls to export the name space. It uses a different process to serve each RPC. Processes are reused when done to avoid the expense of creating them, as with Xfids in acme. A helper process keeps Op fds open and maps them to Inferno fds. Oxport is easy to implement, comprising only 589 lines of code (plus 726 of the Op library). Of those, 105 lines are used to implement *Tput* and 118 to implement *Tget.*

Conversely to Oxport, Ofs is quite complex. The reason for this is that it has to act both as an Op client and as an Styx server. It also has to do some caching. As a consequence, it is six times bigger, 3209 lines long plus the Op library. It also keeps a tree data structure used as cache for remote files; mostly for metadata. Each entry in the tree contains a prefix of the file contents, besides metadata, because a *Tget* in Op retrieves data (along with metadata) that can be used to satisfy further requests.

Optionally, the tree can be instructed to use a directory on the terminal's disk to cache entire files (and not just prefixes). This is mostly useful to avoid bandwidth problems.

The tree cache works as follows. Upon attending a walk to a file, the tree invents intermediate directories needed to reach the file we are walking to. The metadata for the target file is obtained by issuing an Op *Tget* request to the server. For directories, this retrieves the entire directory data besides the metadata and the tree creates additional entries as appropriate. For files, a small data prefix is retrieved along with metadata that is kept cached for future read requests.

Files with a length of zero, as reported by the server, are not cached (only their directory entries are). They are probably stream or device files whose lengths are not updated by the corresponding drivers and Ofs tries not to interfere.

Cached data is only used if the entry is considered valid. An entry is kept valid only for a small period of time, called the coherency window (a user settable parameter). Once the coherency window expires, the server is consulted to check out if the entry is still valid.

This is done also by issuing another *Tget* request, which may also retrieve (an initial prefix for) new file contents.

Regarding writes, cache behavior is different. Creates for directories are performed write-through (synchronous). Removes are done write-through as well. Writes can be either write-through or handled asynchronously. Writes not at offset zero and filling up an entire protocol message are done asynchronously. Other writes are handled synchronously (write-through) to report possible errors to the application. The limit on the maximum number of processes that Ofs may create (a configuration parameter) places a limit on the number of concurrent write requests that may be outstanding at the same time.

Multiple Styx fids are mapped to the same cache entry in the tree. This means that they may share a single Op fd. Fids open for both reading and writing would use two Op fds each; because an Op fd is used either for issuing *Tget* requests, or for issuing *Tput* requests. In any case, a Styx *Tclunk* request is guaranteed to release the associated Op descriptors. This suffices to let the server recognize when a request consisting on multiple reads (or writes) is done.

The mapping between Styx fids and Op fds just described is needed to be able to reuse data kept in the client without having to reach to the server. The drawback is that, as a consequence, clone files will not work under Op.

A clone file is cloned when an open is completed on it. After the open, the only way to distinguish between operations on the original file and on the new file is to use the Styx fid. The absence of Styx fids or anything mapping one to one to them makes clone files unfeasible. Furthermore, any other file multiplexing strategy based on the Styx fid will not work either. This affects programs like the connection server, the registry, and others using clone files.

The application does not have to wait in any case while closing or clunking fids because Op descriptors are released asynchronously. In Inferno this is not really a problem because of the use of the garbage collector to clunk unused fids.

All operations that may be attended from the cached data are handled by a central process devoted to implement the tree. For all other cases, the tree implementation spawns processes to issue concurrent op requests, reusing them in the style of Acme's Xfids. All communication through the Op link is multiplexed by another process that provides a channel based interface for its clients.

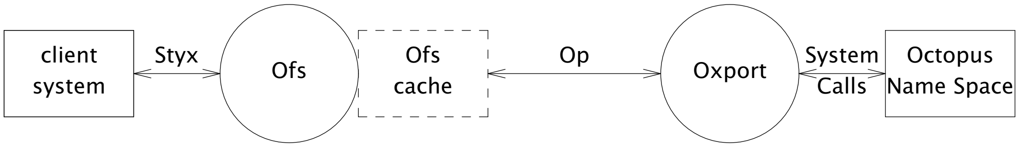

Figure 3: *Basic scheme of the system*

5. Discussion

For most file servers the implementation described works fine and we use it without any problem. There are some problems for some devices. As stated earlier, because of the absence of fids, clone files cannot be exported. Of course they can always be exported using Styx, which makes this problem minor. Note that fids require extra RPCs to maintain them (and thus extra latency); besides making more complex the implementation, as extra state has to be maintained for them. Arguably some of this complication is already in the cache to maintain the state of the Op descriptors.

The cache tries to know the size of a file by using its metadata. This has also resulted to be a tricky issue. As mentioned above, files of length zero are usually files that indeed have data. Some of them are indeed an unlimited source of data. Because of this we had to make the cache ignore data for such files, and also place a limit on the maximum amount of per-file cached data (just for safety).

The protocol is worth its overhead when RTT goes above 1 millisecond. The problems mentioned above make it worth using the protocol for latencies of 50ms and above. The number of RPCs for normal usage of a remote file tree is reduced an order of magnitude. This makes the difference between being able to use a remote file system interactively and not being able to do so.

While using it we learned something about implementing file systems for devices. File sizes should be kept accurate (or the cache may go crazy). Keeping them null to signal unlimited stream files is reasonable. As a related issue, the information conveyed in the Qids should be accurate (also because of caching). For example, if reading a control file reports something else than the last thing written into it, the version must differ from that implied by the write; that makes any cache consider that the file has changed and does not have what was last written on it.

Multiplexing a single file to provide multiple connections by using the fids is a source of problems. For us it seems easier to let a program create a file and use it as its connection than using a clone file. This has the extra benefit of easing direct shell programming using the file system. Using clone devices from the shell is weird and error prone.

Octopus is still work in progress and not completely functional yet. We have been using Op for months to use Inferno as a terminal for a remote Plan 9 system. That has been with RTTs in the range of 50 to 100 milliseconds. Most of the time we have been using the terminal Inferno to execute commands, and the protocol has been used just to make the file system more responsive. We have also tested the protocol with devices, but this is not being used daily.

6. Related work

LBFS [5] focuses on bandwidth problems, rather than latency problems. It breaks the files in chunks and uses hashes to identify the chunks already present in a local cache. In order to do so, it incurs in the extra latency of sending the hashes and waiting for the answers. Therefore, for latency dominated networks there is almost no improvement, specially if most of the files fit in a single packet as is our case.

Cfs(4) is a user-level file server that caches information about remote files onto a local disk. As LBFS, it helps with bandwidth problems, but does not help as much on latency dominated problems. That is because it stats every file before taking it from of the cache to check for its consistency on each open. The cost of a stat is almost the same of the cost of a read over a high latency network. Also, all 9P messages except read, clone, and walk are passed through cfs unchanged to the remote server.

Some efforts have gone into making CIFS work on WANs. These efforts go in three directions. Some of them, like Cisco WAFS [9] (formerly Actona) and Packeteer [10] (formerly Tacitnetworks), do compression and caching. The problem is that they are not capable of exporting synthetic file systems, as we do. Most of the heuristics implemented in Op try to overcome this problem making the cache aware of them.

Riverbed [12] uses a different approach. It changes the TCP/IP stack to decrease the latency and also tries to identify TCP segments that have been already sent. The problem for us in this approach is that there are too many changes at too many levels: it would force us to replace all the protocol stack for all the hosts.

The third approach used by Disksites [11] and Availl [13] is to use some form of asynchronous replication of the file system. This strategy does not work in our target environment. Our users cannot carry the whole file system around. Moreover, this does not work with synthetic file systems.

NFS version 4 [8] tries to batch file systems operations whenever possible, but NFS version 4 is not designed with synthetic file systems in mind, so it is unusable for us. The same also happens to other file systems like CODA [7] or Echo [4].

Rangboom [3] uses 9p to share name spaces between different users across the internet. It is used to share file systems interactively and runs against most of the problems Op deals with. It does metadata caching at the level where the operating system interfaces the file system client (i/o manager), like Op, and sends the clunks asynchronously. Even with this optimizations, some applications run into latency issues.

References

1. F. J. Ballesteros, E. Soriano, G. Guardiola and S. Lalis, Building a Network File System for Device Access over High Latency Links, *Under Review, also available at http://lsub.org/papers*, 2007.
2. F. J. Ballesteros, P. Heras, E. Soriano and S. Lalis, The Octopus: Towards building distributed smart spaces by centralizing everything., *UCAMI*, 2007.
3. G. Collyer, R. Cox, B. Ellis and F. Tavakkolian, Shared Name Spaces, *Iwp9*, 2006.
4. T. Mann, A. D. Birrell, A. Hisgen, C. Jerian and G. Swart, A coherent distributed file cache with directory write-behind, *ACM Transactions on Computer Systems 2*, 12 (May 1994), 123-164.
5. A. Muthitacharoen, B. Chen and D. Mazieres, A low bandwidth network file system, *ACM Symp. on Operating System Prin.*, 2001, 174-187.
6. R. Pike and D. M. Ritchie, The Styx Architecture for Distributed Systems, *Bell Labs Technical Journal 5*, 2 (April-June 1999), .
7. 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.
8. 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.
9. Cisco Systems Inc. - Wide Area File Services Software , *http://www.cisco.com/en/US/products/ps6469/*, 2007.
10. Packeteer Inc. - Response Time Technology, Packeteer White Paper Series, *http://www.packeteer.com*, 2002.
11. Expand Network Inc., *http://www.expand.com/Product*, 2007.

12. Riverbed Technology Inc., *http://www.riverbed.com/technology*, 2007.
13. Availl Inc., *http://www.availl.com/products*, 2007.