The Creepy Πon File System Protocol (call for screams)

Francisco Ballesteros

Enrique Soriano

Gorka Guardiola

Rey Juan Carlos University

ABSTRACT

Creepy is a distributed file system protocol, derived from a not yet implemented version of the ΠP file protocol, as found on a working draft developed jointly by the authors and Bell-Labs. Creepy is meant to become the Πon (Pi-on) file protocol. In the future, both Creepy and ΠP might be unified on a single protocol, or they might not. Creepy supports a distributed file system for the Internet, capable of working well to access devices both on the local and wide area; therefore with caching in mind.

1. Introduction.

We need a file system protocol supporting access to distributed files and devices. For this, 9P [1] suffices. However, we have extra requirements that make 9P not so well suited for our purposes. This is the full list of requirements:

1. The protocol must support access to synthesized files and devices, as we want it to reach sensors, actuators, and software devices in general.
2. The protocol must work well across high-latency, wide-area, network links. Otherwise we would not be able to use it from overseas or from our (poorly connected) homes.
3. The protocol must permit client disconnections, without discarding the client state. Otherwise, suspending clients would be impractical or require additional software.

To work well with devices, the protocol must support file descriptors as found on clients. The equivalent concept in the protocol is the *fid*, as used in 9P. Like in 9P, a *fid* is a small number used as a file identifier. Unlike in 9P, the server assigns *fid* numbers on behalf of the client. Clients must maintain their own data structures to keep track of *fids*, and they do not usually look them up by *fid* number, which means that it is better to let the server assign them; the server can use the *fid* number as an index and save a hash table.

Support for disconnected operation requires the server to be able to keep track of *fids* in use by a client that is disconnected. This is achieved by using ''sessions''. A session is established prior to the Creepy dialog for file access. In some cases creating a new session; in other cases associating to a session from a previous connection. Sessions may be collected by the server (and their state released) after a disconnection, after

some time past the disconnection, or after a server reboot. When to do it is suggested by the client. The server decides.

To reduce latency effects, and use effectively each round trip, protocol requests exchanged between clients and servers are batched. Each batch behaves as an RPC but, instead of using a single request and a single reply, a batch consists on a series of individual operations, each one with its request (and response).

To clarify, a batch, or RPC, is a series of operations, but does not need to be a single message (i.e., a single ''write'' on the channel). Each operation is a separate request; only that it is part of a logical RPC, or batch.

A client sends a series of requests (one per operation) for a single RPC and then waits for a series of response messages from the server. For each operation there is a request message (or ''transaction'' message, or T-message) and a corresponding reply message (or R-message). This is similar to 9P.

Requests to agree on a protocol version, to agree on a session to use, and to flush outstanding requests, are RPCs on their own. But all other requests must be made within a batch of requests.

A batch or RPC has a *begin* request, followed by one or more requests, followed by a final *end* request. This has important implications. A server is free to read from the network all requests in a batch, without starting any of them before reading the end of the batch. Because all such requests are considered a single RPC, with several operations in it, an error in one operation means that the rest of the batch is ignored by the server; the final error response effectively terminates the RPC, when seen by the client.

Batching is needed to achieve a single round trip time in those cases where the client, or an intermediate cache, wants to perform a series of operations and does not need the result of any one before issuing the next one.

A batch is not packaged as a single message, to permit multiplexing of the communications channel. Should a batch be a single message, no requests from other clients might be interleaved with it, and a long batch would effectively block other clients sharing the channel. Packaging each different operation as a different message is also beneficial for the implementation, which can pack, unpack, and handle all messages in the same way.

To help intermediate caches to decide what to batch, when they need to ask the server for data, each batch speaks about a single fid. An intermediate cache may read a series of requests (from a single batch) and, when all of them are known, decide if it makes sense to forward all of them at once to the server. It might decide instead to serve a prefix of the batch from local data, forward the rest to the server, and then reply to the client. Other approaches, mixing requests in a more general way, make it not clear for the cache how long to wait before asking the server: asking too soon leads to multiple round-trips, asking too late leads to extra delays in the service.

Requirements 1 and 2, considered together, imply that the client (or an intermediate cache) must know something about the semantics of each file, to know how to cache it, if at all. There are several types of file to consider:

- Conventional files. No extra requirements on them.
- Device files. They should not be cached at all. All requests on them are to be forwarded to the final server, or the semantics will change and the device interface will break.
- Append only files. These are only appended, which may be exploited by clients and

caches regarding what to cache.

- Timely files. These correspond to audio and video media, where it is usually more important to deliver data timely than it is to retransmit every piece of data.
- Read-only files. Immutable files have nice properties for caches and clients. Those files that are meant to be read only for their whole life, belong to this category. For example, system binaries. Previous versions of mutable files may be considered also immutable.
- Synchronous files. For some files, it is important for clients to see all writes made before (i.e., UNIX semantics). In all other cases, it may suffice to see all writes made prior to the last close of the file (i.e., session semantics). The later is better for caching and works fine in most cases; thus, it is a desirable default. However, the former is sometimes necessary and, therefore, some files may have to be flagged as ''synchronous''.

2. Immutable files, mutable files, and directories

In general, Creepy has immutable files on it. For each file, there is a series of immutable versions perhaps terminating on a mutable version of the file. There are two types of files regarding mutability:

- Files with UNIX semantics, flagged as synchronous.
- Files with session semantics, i.e., not flagged as synchronous.

See before for a description of what that means. A file with session semantics is, in general, a series of immutable versions for it. That is, the last version is usually immutable. A file with UNIX semantics is expected to have a series of immutable versions and, as a last version, a mutable version for the file.

Opening a file for reading (OREAD) yields the last immutable version for files with session semantics. For files with UNIX semantics, the last version (mutable or not) is the one used instead.

Opening a file for writing (OWRITE or ORDWR) requires a mutable version. For UNIX files this yields a *fid* for the last version, which is mutable. For files with session semantics, it depends. The first open for writing on such type of file creates a new mutable version, and associates the *fid* to it. As long as some *fid* is open for writing, new opens for writing find that the version is mutable and associate with it as well. Once the last writer is gone, the mutable version is frozen and made immutable. Following opens for writing would therefore create a new, mutable, version.

For files with UNIX semantics, when a mutable version has been modified and all *fids* are gone, the version might be frozen and made immutable, in which case a new mutable version is added to the chain of versions. This is a description of the behavior and not of the implementation.

This permits files flagged as ''synchronous'' to operate very much like in UNIX, to keep applications working in those few cases where the change in semantics with respect to UNIX and Plan 9 matters. In the default case (session semantics), it avoids races regarding readers interested in the most recent (but consistent) version of the file. As a result, we can now copy new binaries and libraries to their standard location and forget about existing processes paged on demand and also about those calling *exec* or linking against the libraries being replaced.

Directories behave very much like files. Creating a file, removing a file, and updating file metadata can be considered as opening, writing, and closing a directory. Thus, any of these operations leads to a new version of the directory (being the old one immutable from that point in time). Opening a directory for reading yields a *fid* that is kept in the last consistent version of the directory, despite new versions being created. As a result, reading a directory is free from races due to directory updates.

Note that in all other respects, directories are still like files. This implies that directories may be flagged as ''devices'', so that requests for them are always relied verbatim to their servers; they may be flagged as ''append-only'', so that new files may be added, but old files must remain intact; etc.

3. Coherency

For a given file tree, we may consider a **spanning tree** of machines or nodes to govern its data distribution on the Internet. The spaning tree of machines for a file tree has the main fail server on the root of the tree, and intermediate caches on the way to the leaves, which are clients. Depending on the point of view (server or client) the tree may be one or another, but it does not have to involve all the nodes on the system (on the Internet).

In general, **coherency is not guaranteed** between different nodes and clients operate on the newest file version locally known. Another node might know of a more recent version for a file. That is the default, although it is permitted to specify on a per-tree basis (i.e., at mount time) that coherent operation is desired. Either way, the user is notified and asked regarding what to do the very first time that coherency cannot be guaranteed (i.e., upon disconnection). The result would be either to report the error to the application or to perform the operation with local information. Upon reconnection, updates are to be performed on the server, possibly leading to conflicts due to concurrent updates.

Version numbers, and the organization of clients, caches, and servers in a distribution tree (with the server at the root), make it feasible to detect if updates are concurrent or not. When a concurrent update is detected, the data is saved in an alternate file (named to reflect the error), and an error is notified to the user. The file is considered broken. What to do next is up to the user.

Operations made on files (updates, including removal) are sent from a node to the rest of the spanning tree of machines for the file tree involved (both up-stream and down-stream). The aim is to try to make the data available despite network partitions. Forwarding is done after replying to the client causing the request (unless coherent operation is demanded) in the hope that conflicts will not arise. When optimistic operation fails, conflicts are detected and the file is declared as broken (probably on all nodes but the first one reaching the file owner). This is not expected to be a frequent case.

User intervention is required to reconcile broken files by creating a new version. However, it may be reasonable to automatically repair append-only files by applying the operations on all nodes in the order seen by the file owner. Most conflicts are expected on append only files used for system logs and e-mail, and it is clear that these might be automatically reconciled. That is, append only files automatically reconciled after concurrent updates might include the same log entries, or the same e-mails, perhaps in a different order. Note that for this to work, a batch must be applied atomically for append only files. This particular idea is still under consideration and it is not clear yet whether it will be added to the current implementation or not.

To help coherent operation of files, without incurring on extra latency, Creepy nodes speak a file ownership protocol that uses leases to transfer ownership of files (file trees). This protocol is inspired by Sprite, Envoy, and others.

Not all nodes are required to cache everything. This is obvious if you consider client nodes as part of the spanning tree. A data publish/subscribe mechanism is provided de-facto by the leasing protocol. The result is that each node would receive updates for files of interest (regarding caching), but not about others.

3.1. Ownership protocol

This is a protocol that relies on a leasing mechanism to try to ensure coherency to local updates (at each node), when so requested. Initially, the entire file tree belongs to the root of the Creepy spanning tree. Clients accessing subtrees request read-only leases for them (they might affect a single file). Clients modifying subtrees request read-write leases for them. Upon network partitions, leases expire and ownership is released. Some nodes might optimistically continue while disconnected, on their own, knowing that this might lead to future conflicts. The user decides.

The protocol does not introduce further requests. Instead, special files on the file server file tree are used for leasing (similar to authentication files on 9P).

Leases granted by the server define sub-trees so that a few leases may cover the needs for the client. This permits adaptation so that clients obtain leases on entire sub-trees that are in use, and transfers ownership of file trees to those using them. The idea is that the server tries to grant to the client as fewer leases as feasible, but covering as much of the file tree as needed by the client. This may be achieved by granting leases for the least common ancestor covering the leases requested; unless such lease conflicts with other clients. In that case, several leases may be granted instead of one, trying to keep different clients operating on different trees.

4. File system protocol

The file system protocol derives from 9P. Here we focus mostly on the differences. In the description of structures and messages we indicate field names and their encoding using the name of the field, a ''**:**'', and a type expressing the type. Type names include: *u8*, *u16*, *u32*, and *u64* meaning unsigned little-endian integers of 8, 16, 32, and 64 bits. In the same way, *str* represents a string encoded using an *u16* field with the string length followed by that many bytes in UTF-8, followed by a null byte. The name of a previously defined structure or message represents as a type the encoding for its fields. The next section provides an example.

4.1. File identifiers

A file within a given Creepy file tree is identified by a Π*id* (Creepy Πon file id), defined as:

```
Πid: fvers0:u64 fvers:u64;
```

As an example of the notation used, this is the same expressed in C:

```
struct Πid
{
        u64int  fvers0;
        u64int  fvers;
};
```

Fvers is the file version. Versions are meaningful within the context of a server, and correspond to nano-second time-stamps guaranteed to be unique. The first field, *fvers0* reflects the initial time-stamp for the file (equivalent to 9P's *path* field in *Qids* as found on the protocol).

New versions are assigned by the file owner, and may be faked along Lamport's clocks behavior to ensure they behave as logical time.

4.2. Headers

All messages start with a standard header, defined as follows:

```
Πhdr: len:u32 sid:u32 tag:u32;
```

Here, *len* defines the message size, *sid* indicates the session id and *tag* is used to match replies and requests.

In what follows, the header is not shown on messages listed below. Each message consists of the header, followed by a byte encoding the type of message, followed by extra fields depending on the type. Messages as listed below show just the message type name and the list of fields for them.

4.3. Initiation

Before initiating a Creepy dialog, a *proto* request is used to agree on the protocol version and options.

```
ΠTproto: version:str options:str;
ΠRproto: version:str options:str;
```

A Creepy dialog happens within a session, as established by a *session* request. A session may be understood as the state of the server on behalf of a client and is mostly a set of *fids*. In fact, *fids* are only meaningful within the context of a session.

The *session* request defines a client session id for a given user (or associates with a previously defined session). A session is terminated by an *endsession* request.

```
ΠTsession: iosize:u32 uname:str expire:u32;
ΠRsession: iosize:u32;
ΠTendsession:;
ΠRendsession:;
```

Using *Nosid*, i.e. ''~0'', as a *sid* in the Πhdr asks the server to allocate a new session. Using any other *sid* asks the server to associate the dialog to that session. The server picks *sid* values for new sessions and responds to the client indicating such *sid* in the Πhdr. So, *sids* may be used as direct indexes in the server and no hash table is necessary for sessions.

The request determines a maximum I/O size, defined as the maximum size for data in messages exchanged. Thus, different sessions may have different I/O sizes even if they share the same communication channel.

The user name, *uname*, indicated is used to identify the user responsible for requests in the session. Authentication is to be performed after the *session* request using an authentication *fid* to exchange data. By convention, after this request, *fid* 0 is open for reading and writing as an authentication fid. It can be closed later any time but may not be re-open.

The *expire* field in the session request defines a time interval, in seconds, to automatically remove the session (and its associated state). The interval starts after a disconnection from the client; but receiving an *endsession* request immediately deallocates the session, ignoring this interval.

4.4. Interrupts

Any request may be flushed using a *Tflush* request.

```
ΠTflush: oldtag:u32;
ΠRflush:;
```

Semantics are similar to 9P's *Tflush*, but this one flushes an entire batch of requests instead.

4.5. Batches

Only the requests shown so far are sent on their own as single-operation RPCs. All other requests are sent in batches. We refer to an entire batch as an RPC; each one composed of a series of individual operations, or requests.

An RPC is defined by a *begin* operation, followed by at least one other operation, and terminated by an *end* request. All requests in a batch must have the same tag. For example, this would be an RPC:

```
-> Tbegin(0) Tattach(0) Tclone(0) Twalk(0) Topen(0) Tread(0) Tclunk(0) Tend(0)
<- Rbegin(0) Rattach(0) Rclone(0) Rwalk(0) Ropen(0) Rread(0)
        Rread(0)
        Rread(0)
        Rclunk(0) Rend(0)
```

Where tags are indicated after each request (all zero in this example), and *begin* and *end* are as follows:

```
ΠTbegin: fid:u32;
ΠRbegin:;
ΠTend:;
ΠRend:;
```

Because all requests in a group must have the same tag, the same communication channel may interleave requests from different groups as desired. The channel must preserve FIFO delivering for this to work.

Each RPC operates on a single *fid*, as set by *begin*. Initially, when the client does not have any *fid*, it must use *attach* to gain one for the root of the tree; in this case *Nofid* may be used as a value for *fid* in *begin*. When a request changes the *fid*, as *attach* and *clone* do, the rest of the requests in the batch operate on the changed fid.

The RPC is considered the entire series of operations as we explained; as a result, an error reply terminates the entire RPC.

4.6. Navigation

An authenticated user responsible for the session may attach to the root file of a tree with name *aname*, using an *attach* request.

```
ΠTattach: uname:str aname:str;
ΠRattach: fid:u32 Πid;
```

This request must be the first one in its RPC, and the Note that all requests following

Tattach in the RPC will operate on the *fid* it allocates. To permit a user to speak for another, *attach* includes in *uname* a user name. Direct authentication of users (instead of accepting ones to speak for others) can be achieved by using multiple sessions within the same communications channel.

After attaching, a *fid* may be used to walk the file hierarchy, like in 9P version 1. requests.

```
ΠTwalk: name:str;
ΠRwalk:;
```

No file information is returned on its reply; if the file id or any other information is required, a *stat* request may be included in the RPC.

All file trees are expected to have a second file tree containing snapshots for previous versions of files. If the main file tree is named *main*, its dump tree must be named *maindump*. The same happens to all other trees. Therefore, walking through the dump tree grants access to immutable (previous) versions of files in the main tree. By convention, the root directory of the dump tree synthesizes on demand a subdirectory for each time value (expressed in nanoseconds), and it contains the version of the tree as it was on that time. Of course, not all versions may be kept by a server, and such directory might be empty if no previous dumps were taken. That is,

```
/dump/1289383372000000
```

means: ''the state of the file system as it was before 1289383372000000ns after the epoch''. That could be exactly at that time or perhaps one day before.

Using a time-stamp after the time when a file was removed, does not show the file; as expected. If the file is created again later, it will start to show up in the dump after that point in time. All this is to say that removals must work as expected in the dump tree.

All dumps are expected to contain

```
/dump/log
```

reporting, one per line, the list of timestamps preserved in the dump. There is one timestamp per line, perhaps followed by text in free from (usually the date in a more human readable format).

To preserve *fids* despite walks, *clone* allocates a duplicate of an existing *fid*.

```
ΠTclone: ;
ΠRclone: nfid:u32;
```

4.7. Files and metadata

Files (and directories) are created using *create*, similar to 9P. The request includes permissions, an open mode, the name for the file, and a version. As long as time is kept moving forward, any version indicated is considered legal. If *Novers* is supplied as a version, the server assigns a time-stamp.

```
ΠTcreate: name:str perm:u32 mode:u32;
ΠRcreate: Πid;
```

The following bits may be used in *perm* to indicate the type of file, or set file flags; besides the traditional Plan 9 permissions:

```
ΠFdir   = 0x01UL<<24,           /* directory */
ΠFapp   = 0x02UL<<24,           /* append only file|dir */
ΠFdev   = 0x04UL<<24,           /* device */
ΠFro    = 0x08UL<<24,           /* read only immutable file */
ΠFtmp   = 0x10UL<<24,           /* not backed up file */
ΠFunix  = 0x20UL<<24,           /* synchronous writes */
ΠFexcl  = 0x80UL<<24,           /* exclusive use */
```

The counterpart is *remove*, which is similar to the *clunk* request (used to release a *fid*), but also removes the file.

```
ΠTremove:;
ΠRremove:;
ΠTclunk: ;
ΠRclunk: fvers:u64;
```

Attributes are retrieved and updated using *stat* and *wstat* requests, similar to 9P's:

```
ΠTstat:;
ΠRstat: dir:Πdir;
ΠTwstat: dir:Πdir;
ΠRwstat:;
```

The directory entry is given by the following declaration:

```
struct Πdir
{
        Πid     πid;
        u32int  perm;
        u64int  atime;
        u64int  mtime;
        u64int  before;
        u64int  length;
        char*   name;
        char*   uid;
        char*   gid;
        char*   muid;
        u8int   nx;
        char**  xname;
        char**  xval;
};
```

All fields up to and including *muid* are mandatory. Other attributes are expressed as a series of *nx* name/value pairs, kept in arrays *xname* and *xval*. These may be changed only by the file owner. It is important to note *before*. It supplies the time-stamp for the previous version of the file, if any.

4.8. Input/Output

Before performing I/O on a file, it must be open. There is nothing new here.

```
ΠTopen: mode:u32;
ΠRopen: fvers:u64;
```

Once open, the following requests permit reading and writing:

```
ΠTread: offset:u64 count:u32 total:u32;
ΠRread: count:u32 eof:u8 data:u8[count];
ΠTwrite: offset:u64 count:u32 data:u8[count];
ΠRwrite: fvers:u64 count:u32;
```

There are important changes here with respect to 9P. The *read* request includes both a *count* for the number of bytes accepted on each reply and a *total* number of bytes accepted in a series of replies. If both values differ, the server is granted the right to issue a series of reply messages for this operation; still considered part of the single read operation within the RPC.

The reply conveys an *eof* indication to the client, so that reaching eof is indicated even when some bytes are retrieved, and there is no need for the client to issue more read requests to discover the fact.

By convention, reading an append-only file after the eof indication would block the read request until more data is appended. This is similar to the previous version of ΠP. Unlike in that version, there is no need for ‘‘clunk on eof’’. If a client wants to clunk the file after reaching eof, include a clunk request in the RPC after reading.

The reply for write includes the version of the file along with the number of bytes written, for caching.

5. Protocol

```
1    #
2    # cpc(1) definitions for a creepy Πon implementation
3    #

5    Πid: fvers0:u64 fvers:u64;

7    Πdir: πid:Πid perm:u32 atime:u64 mtime:u64 before:u64 length:u64
8         name:str uid:str gid:str muid:str
9         nx:u8 xname:str[nx] xval:str[nx];
```

```
11    ΠTproto: version:str options:str;
12    ΠRproto: version:str options:str;
13    ΠTsession: iosize:u32 uname:str expire:u32;
14    ΠRsession: iosize:u32;
15    ΠTendsession: ;
16    ΠRendsession: ;
17    ΠTflush: oldtag:u32;
18    ΠRflush: ;
19    ΠTerror:; # unused
20    ΠRerror: ename:str;
21    ΠTbegin: fid:u32;
22    ΠRbegin:;
23    ΠTattach: uname:str aname:str;
24    ΠRattach: fid:u32 Πid;
25    ΠTclone: ;
26    ΠRclone: nfid:u32;
27    ΠTwalk: name:str;
28    ΠRwalk:;
29    ΠTopen: mode:u32;
30    ΠRopen: fvers:u64;
31    ΠTcreate: name:str perm:u32 mode:u32;
32    ΠRcreate: Πid;
33    ΠTread: offset:u64 count:u32 total:u32;
34    ΠRread: count:u32 eof:u8 data:u8[count];
35    ΠTwrite: offset:u64 count:u32 data:u8[count];
36    ΠRwrite: fvers:u64 count:u32;
37    ΠTstat:;
38    ΠRstat: dir:Πdir;
39    ΠTwstat: dir:Πdir;
40    ΠRwstat:;
41    ΠTremove:;
42    ΠRremove:;
43    ΠTclunk:;
44    ΠRclunk: fvers:u64;
45    ΠTend:;
46    ΠRend:;
```

```
48    Πhdr: len:u32 sid:u32 tag:u32;
49    Πcall: Πhdr
50         ΠTproto|
51         ΠRproto|
52         ΠTsession|
53         ΠRsession|
54         ΠTendsession|
55         ΠRendsession|
56         ΠTattach|
57         ΠRattach|
58         ΠTflush|
59         ΠRflush|
60         ΠTclone|
61         ΠRclone|
62         ΠTwalk|
63         ΠRwalk|
64         ΠTopen|
65         ΠRopen|
66         ΠTcreate|
67         ΠRcreate|
68         ΠTstat|
69         ΠRstat|
70         ΠTwstat|
71         ΠRwstat|
72         ΠTread|
73         ΠRread|
74         ΠTwrite|
75         ΠRwrite|
76         ΠTremove|
77         ΠRremove|
78         ΠTclunk|
79         ΠRclunk|
80         ΠTbegin|
81         ΠRbegin|
82         ΠTend|
83         ΠRend|
84         ΠRerror;
```

References

1. D. Presotto and P. Winterbottom, The Organization of Networks in Plan 9, *Plan 9 User's Manual 2*.