9p-based filesystems
Implementation Notes
v1.3

Maxim Kharchenko

December 12, 2012

1 Evolution

The view of a filesystem in Erlang on Xen have gone through a few major revisions. A brief overview of these revisions may help to understand the rationale behind the current approach and its future directon.

When a Ling image gets compiled many named blobs of data are baked into it. These blobs may contain Erlang code modules, a boot script, etc. The earliest approach was to represent the blobs as 'files' all belonging to the same 'directory' called _embed. A series of small changes throughout prim_file, erl_prim_loader, and other modules, overlayed _embed over the 'true' filesystem accessed via diod server. It was not possible to emulate directories of embedded files or mount (multiple) subtrees from the diod server. Embedded files had no file descriptors limiting their use to functions, such as file:read_file().

The current approach to 9p-based filesystems in Erlang on Xen is more advanced. The biggest enhancement is the introduction of the mounting server – 9p_mounter. The mounting server made the filesystem interface flexible and uniform. It is discussed extensively later. Another enhancement is introduction of 'synthetic' files not mapped to named data blobs or Linux files. The named data blobs (section 11.1) interface was reworked.

The next (planned) stage of the evolution of 9p-based filesystems includes addition of reestablishing transport connections, multi-mount policies, incomplete operations, auhtentication/authorization. These changes are marked as 'NEW' throughout the document.

2 Protocol

'9p' is a family of protocols. The present

3 Usage

In general, when performance and throughput is an issue, the filesystem interface should be avoided. The primary intention of using and extending the filesystem interface is compatibility with Erlang libraries, meeting expectations of the current application code, gathering and distribution of configuration data. The current implementation of the code_server module relies heavily on the filesystem. The throughput is not usually an issue for code loading.

The initial code loading may affect the startup latency though. The question is addressed by bypassing file operations in a custom version of erl_prim_loader module, which is used by Erlang before the code_server is operational.

The filesystem interface may prove to be convenient for exposing configuration information to other operating systems and tools. In a typical scenario, a bash script may read status information of a certain application by mounting a node filesystem and accessing the corresponding file. The rudimentary control over entities of the application layers may be implemented by writes to synthetic files.

The new clustering layer envisioned for Erlang on Xen relies heavily on the functionality provided by the 9p layer described here.

---

4 9p transports

Both 9p client and server should be able to use multiple protocol to transport 9p messages. The 9p infrastructure emphasizes flexibility over performance, thus TCP/IP transport will suffice in most cases, especially, as public clouds restrict other transports.

For starters the 9p infrastructure will be limited to TCP/IP transport but the transport will be abstracted out to add other transports (SCTP) later. The name of the module implementing the TCP/IP transport for 9p is '9p_tcp'.

The type of the transport is determined by the name of the module that implements it. The behaviour of the transport module is closely follows that of gen_tcp module.

```
TransMod:connect(TransConf) -> {ok,Sock} | {error,Error}
TransMod:connect(TransConf, Timeout) -> {ok,Sock} | {error,Error}
TransMod:send(Sock, Data, TransConf) -> ok | {error,Error}
TransMod:recv(Sock, TransConf) -> ok | {error,Error}
TransMod:recv(Sock, TransConf, Timeout) -> ok | {error,Error}
TransMod:set_maximum_size(Sock, MSize, TransConf) -> ok | {error,Error}
TransMod:activate(Sock, TransConf) -> ok | {error,Error}
TransMod:close(Sock, TransConf) -> ok | {error,Error}
```

TODO: file size?

All entries of the transport module are self-explanatory, except the activate() call. The activate() call switches the socket into active mode and it starts to deliver messages asynchronously to the controlling process. The active mode is facilitated by the non-standard {packet,'9p'} type that understands framing of 9p packets. The standard {packet,4} type cannot be used as the size of 9p messages includes the size field itself. The size field is stripped off for '9p' packets.

5 9p client

9p client makes network resources, data and synthetic files, accessible to the file: layer. 9p client maintains (multiple) connections to 9p servers. Portions of name hierarchies accessible over a given connection are mapped to *mounting points* in the local hierarchy. The mounting is flexible: several remote directories can share a mounting point.

The flexible mounting can recreate, for instance, a directory structure expected by the code_server. Embedded modules of the 'stdlib' application exposed by the local 9p server can be mounted at /erlang/lib. Code of rarely-used applications accessible over the network can be mounted at the same location. Thus, if the root directory is set to /erlang, the code_server will see the familiar list of libraries under ERL_ROOT/lib.

The following two functions add and remove connections from the 9p client:

```
add_connection(Id, TransMod, TransConf, Mounts) -> ok | {error,Error}
remove_connection(Id) -> ok | {error,Error}
```

Id is an arbitrary term. *Mounts* is a list of pair that define the mapping between local and remote paths. The the mapping paths can not have '..' (parent) element. For instance,

```
Mounts = [
       {<<"/erlang/lib">>,<<"/">>},
       {<<"/amqp/q">>,<<"/queue">>}
     ].
```

When a new connection is added to the 9p client, a series of steps happens in sequence. Firstly, a transport link is established to the 9p server. Then a version negotiation happens that insures that the server supports the required version of the protocol (9P2000.L). The maximum message size is negotiated too. The server can not decrease the maximum size requested by the client. The connection is dropped, if it attempts to.

After successful version negotiation, the 9p client attaches to all mounting points listed the add_connection() call. If some attach operations fail, then the corresponding mapping is not added to the mounting map.

The main mounting map is a list of 3-tuples:

```
{Id,ConnPid,Mounts}
```

Id is the identifier of the connection as indicated in the add_connection() call. ConnPid is the identifier of the connection process that handles conversations with a particular 9p server. Mounts is the list of 3-tuples:

```
{Prefix,RootFid,RootQid}
```

Prefix is local path prefix represented as a list of binaries. RootFid and the corresponding RootQid are results of attaching successfully to the root of the server name hierarchy. RootQid is needed to detect situations when the root boundary is crossed while walking to '..'.

Some file operations, such as file:open() return a file descriptor. The file descriptor necessarily contain fids from the lower (9p) layer. Fids are allocated per connection and a file descriptor may refer to multiple fids. A possible internal representation of a file descriptor is

```
[
  {Fid1, ConnPid1, []},        %% ordinary file/directory
```

```
    {Fid2, ConnPid2, [at_root]}, %% at the hierarchy top
    Path                         %% local path - ask '9p_mounter'
  ]
```

The mounting information is encapsulated by the '9p_mounter' process. Each connection is managed by a separate connection process. File operations, as implemented by prim_file module, inquire '9p_mounter' process to retrieve Pids of relevant connections. The '9p_mounter' process acts as a supervisor for connection processes.

In addition to maintaining the mounting information '9p_mounter' process responds to 9p requests that refer to the local portions of mounting points.

A typical scenario implemented at prim_file level involves issuing attach/read/clunk requests to all fids referred by the file descriptor and combining the results.

Composite fids make error handling more complex. An operation on a composite fid may results in dozen of separate conversations over many connections. Some of them may (and will) fail. Thus the result of the operation may be partial and include errors. The file: interface does not provide a facility for returing partial results. The practical approach is to return partial results as a complete set. If there are no usable results at all, the last error should be returned.

A special case when an attempt to walk to '..' is made at the root of the remote directory. Such attempt should be intercepted by matching the qid to the qid of the root directory.

---

6 9p server

The modified version of the kernel application starts a local 9p server. The 9p server exports a limited hierarchy of synthetic directories and files.

The limitation of the hierarchy is twodold. An exported directory always belongs to the first level, e.g. /boot. Exported files are always on the second level, e.g. /boot/app.config. Of course, the exported directory can be mounted at an arbitrary point. Thus these limitations are not carried over to applications consuming the exported tree.

Each exported directory corresponds to a module that implements '9p_export' behaviour (NEW). When a directory is exported the implementation module is accompanied by opaque configuration term that is passed 'as is' to all calls to the module.

9p server follows the convention that paths are represented as lists of binaries. The binaries correspond to path components without slashes. All calls to the module that implements '9p_export' interface use paths relative to the exported directory. Thus the path of `[]` means the exported directory itself and `[Name]` means a file inside the directory.

All '9p_export' modules must export the following functions:

```
list_dir(Conf) -> [binary()]
```

(NEW) returns the list of files (as binaries).

```
make_qid(Path, Conf) -> binary()
```

Returns a 13-byte binary that follows conventions for Qid of 9p protocol [1].

```
exists(Path, Conf) -> true | false
```

Returns `true` if the path exists and `false` otherwise.

```
create(Path, Name, Mode, Conf) -> {ok,Qid} | {error,Error}
```

Create a new synthetic file *Name* in the exported directory. *Path* must be `[]` for the call to succeed. *Mode* values are as specified for 'mode' parameter of 'tlcreate' request [2].

```
remove(Path, Conf) -> ok | {error,Error}
```

(NEW) Removes a synthetic file referenced by *Path*. It is not possible to remove an exported directory using the call.

```
size(Path, Conf) -> {ok,Size} | {error,Error}
```

Returns the size in bytes of the file referenced by *Path*.

```
read(Path, Offset, Count, Conf) -> {ok,Data} | {error,Error}
```

Reads *Count* bytes of the contents of the synthetic file starting at *Offset*.

```
write(Path, Offset, Data, Conf) -> ok | {error,Error}
```

Writes *Data* into the synthetic file referenced by *Path* starting at offset *Offset*.

When started the 9p server has nothing to serve. The filesystem information is added to 9p server on a per-directory basis. The interface is following:

```
'9p_server':publish(Name, Module, ModData)
'9p_server':unpublish(Name)
```

The *Name* is the name of the exported directory, e.g. <<"stdlib">>. The *Module* must implement '9p_export' behaviour. *ModData* will be passed 'as is' to all *Module*: calls.

The initial configuration of the 9p server is stored in the /boot/app.conf file as described in the section Boot (chapter 10).

7 9p sessions

TODO - authentication, Kerberos-like

8 Integration

(OBSOLETE) The approach proposed above should make the integration with standard Erlang libraries easier. The prim_file code is already reworked to use 9p interface. Sporadic hooks that enable '_embed' directory should be removed.

The erl_prim_loader module, which is used early during the boot sequence, should use bypass route to access embedded modules.

The obsolete 9p over raw Ethernet transport should be removed too and replaced with an abstract transport with a working TCP/IP implementation.

9 Future directions

If the approach is successful, more synthetic filesystem can be added to the 9p server in the future. Also, a SCTP transport should be added to take care of the 9p traffic in the private clouds.

The implementation of filesystems as described here should allow running the large Erlang application with ease.

10 Boot

Erlang on Xen uses the modified boot sequence of Erlang/OTP. The additional configuration read from /boot/app.config file. To include the custom configuration file, a '-conf app' option should be added to the command line.

The /boot/app.config adds a single key – 'start_plan9' – to the configuration of the 'kernel' application. The value of the 'start_plan9' item is a 2-tuple `{ServerConf,MounterConf}`. The ServerConf is the configuration for the local 9p server, MounterConf – for the 9p mounter.

10.1 ServerConf

The configuration of the local 9p server can have the following items:

```
{listeners,[TransConf]}
```

Congifures transport endpoints the local 9p server listens on. Example: `{listeners,[{std,'9p_tcp',564}]}`.
TODO: pass endpoint name to 9p_export calls TODO: pass version info to 9p_export calls

```
{exports,[ExportSpec]}
```

Configures additional exported directories. All buckets of data blobs are exported automatically. Example: `{exports,[{<<"reg">>,regproc_exp,[]}]}`.

10.2 MounterConf

TODO

11 Related

11.1 Named data blobs

All named blobs embedded into a Ling image are separated into 'buckets'. The buckets contains data blobs, but not other buckets. The names of buckets and data blobs are atoms.

The following BIFs expose buckets and data blobs to the application:

```
binary:embedded_buckets() -> [atom()].
```

returns the list of all buckets.

```
binary:list_embedded(Bucket) -> [atom()].
```

returns the list of data blobs that belong to the *Bucket*.

```
binary:embedded_size(Bucket, Name) -> integer() | false.
```

returns the size of the data blob identified by *Bucket Name* pair or `false` if the data blob is not found.

```
binary:embedded_part(Bucket, Name, Pos, Len) -> binary() | false.
binary:embedded_part(Bucket, Name, PosLen) -> binary() | false.
```

returns *Len* bytes of the data blob *Bucket Name* starting at the position *Pos*. *Len*, *Pos*, and *PosLen* follow the conventions of the 'binary' module. `false` is returned if the blob is not found.

The build system generates the list of buckets from the contents of the 'import' directory. The directory contain soft links to other directories. The link names become the bucket names and the contents of the referenced directories are embedded as data blobs. .beam files are skipped if there exists a .ling file with the same name.

The 9p server exports all buckets and their contents as synthetic files. The corresponding behaviour is encapsulated in 'embedded_exp' module.

Bibliography

[1] Plan 9 Manual INTRO(5) (http://man.cat-v.org/plan_9/5/intro).

[2] 9p2000.L protocol (http://code.google.com/p/diod/wiki/protocol).