Design Document for Remote File Copy

CS117 Jacob Zimmerman and Tyler Thompson

Introduction

There will be several isolated components, all of which are, to the best of our ability, the bare minimum of cleverness that they need to be.

A major inspiration for this design was a quotation from Joe Armstrong who developed the Erlang BEAM machine:

'It was during this conference that we realised that the work we were doing on Erlang was very different from a lot of mainstream work in telecommunications programming. Our major concern at the time was with detecting and recovering from errors. I remember Mike, Robert and I having great fun asking the same question over and over again: "what happens if it fails?" – the answer we got was almost always a variant on "our model assumes no failures." We seemed to be the only people in the world designing a system that could recover from software failures.'

Rather than build an intricate, robust system to prevent failure, the goal of this system is to have everything be simple enough that failures can be self-healing. Allowing us to prioritize objects that are *stupid* rather than *smart*.

Any *smartness* is an optimization.

Some terminology

- A **message** is a requested action that is conveyed over a network
- A **method** is a function on a C++ object. Though our objects use a *message passing* style, we use the term method to avoid confusion.
- A **id** is a number that we associate with a filename
- An **object** is a C++ object
- A **module** is a set of related functions

The protocol

Packet Structure

```
| HEADER ...                  | DATA ...                                    |
| i32 msg | i32 seq | u32 len | DATA ...                                    |
| SOS     | i32 seq | u32 len | i32 id |                                    |
| ACK     | i32 seq | u32 len | i32 id |                                    |
| PREPARE | i32 seq | u32 len | i32 id | i8[80] filename | u32 nparts      |
| SECTION | i32 seq | u32 len | i32 id | u32 partno | u8[len - 8] data     |
| CHECK   | i32 seq | u32 len | i32 id | i8[80] filename | u8[20] checksum |
| KEEP    | i32 seq | u32 len | i32 id |                                    |
```

```
| DELETE  | i32 seq | u32 len | i32 id |                                  |
```

- `seq` is the sequence number created by the `Messenger`
- `len` is the length of the data
- `id` is unique to each file, not each packet
- `SOS` is always constructed as a response to a previous packet. To construct an SOS we only set a special bitflag, modifying the incoming packet in place. Receiving an SOS triggers the healing mechanism.
- `ACK` is constructed the same as SOS. It notifies its receiver that the requested action with a matching `seqno` was performed.
- `PREPARE` is sent to indicate to the server to get ready for a file separated into nparts, and so the server will associate the `filename` with the `id`.
- `SECTION` is a section of a file, identified by its `partno`
- `CHECK` tells us that an end to end check is necessary, providing a filename in addition to the id, just in case the server doesn't know the association file for the id
- `KEEP` tells the server to save the file matching an `id`
- `DELETE` tells the server to delete the file matching an `id`

It is always assumed that these messages come in any order—valid or not—and that both server or client can correspond an action to any of these.

There are two objects `Packet` and `Message`, which can convert from one another and handle serialization, deserialization and creation of the messages above.

Objects and Modules

The `diskio` Module

There are two utility functions here for remedying the nastyfile hard drive failures `fileToBuffer` and `bufferToFile` which provide reading and writing respectively. These functions use brute force to guarantee a correct fileread.

Given a preset parameter `MAX_DISK_FAILURES`:

To read

- Create a list of checksums `MAX_DISK_FAILURES` in length
- Repeatedly read the file and store the hashcode
- Compare the checksum with previous attempts
- When two checksums match we assume the file contents to be correct.

To write

- Hash the data into a checksum

- Try to write
- Reread what we wrote, using the function above to guarantee safe read then verify the checksum we read matches the checksum from the client

Assumptions

This assumes that the odds of an identical disk error is less than `2 / MAX_RETRY_ATTEMPTS`

The `ClientManager` Object

A `ClientManager` is constructed from a directory and a `NASTYFILE` handler. Upon creation it stores metadata about all the files in the directory that it has to send, giving them each an id.

It stores an association map between an id

- filename
- file contents
- file transfer status `LOCALONLY | EXISTSREMOTE | COMPLETED`
- number of previous SOS failures with this file

It has two methods, `sendFiles` and `endToEndCheck`, both of which take a messenger object to ease communication.

When the server gives an SOS the

The `Messenger` Object

The `Messenger` module implements network communication from the client side.

It has one "real" function, `send`, which:

1. Takes in a list of messages
2. Assign them each a monotonically increasing sequence number
3. Store all the messages in a table indexed by their sequence number.
4. Loop through the table, sending each message
5. Read all messages until a timeout
    - If receives a packet with immature sequence number, drop it
    - Otherwise if it receives an ACK, remove the message from the table with corresponding sequence number
    - If it's is an SOS, abort and return `false`
6. Repeat until there are no unanswered messages left in the table

This guarantees that

- old responses are ignored
- the server definitely knows what client wants it to know

The `FileCache` Object

The `FileCache` is responsible for all the *thinking* that the server will do. This isn't too much thinking at all.

All of its functions return `true`, to signify an expected behaviour, and `false` to signify unexpected behaviour.

It's responses **never** change based on incoming sequence number, if an old message is sent — it **will** be answered. It is the client's responsibility to handle old messages.

The interface provides five methods:

`idempotentCheckFile`

`idempotentSaveFile`

`idempotentDeleteFile`

`idempotentPrepareForFile`

`idempotentStoreFileChunk`

I won't belabor my point by explaining what they do. But these are all the actions a client can tell the server to do.

The only *thinking* the `FileCache` does is manage which operations would be redundant, sometimes based on the sequence number.

However, if something can't be done, the FileCache will notify it's caller by returning `false`.

How, it works is a bit beyond the scope of this design doc, but it's implemented already.

The `Responder` Object

The responder has one function, `bounce`.

`bounce` receives a packet and returns a packet. Always Always.

It does the following

1. Convert packet to a message
2. If received `ACK` or `SOS` (not sure why) returns them
3. Otherwise hand a `Filecache` an idempotent action based on the message
4. If the filecache says it was unexpected, return `SOS` otherwise return `ACK`
5. Ensure that the returned packet has the same sequence number and id as the incoming

There are no other cases, that's it, pretty dumb :)

`Packet` and `Message` Objects

Just some pretty print functionality and accessors.

Internally `Message` uses a discriminated union pattern to store variants, like an SML datatype. It also has some safety checks

`fileclient` and `fileserver` Modules

These have the main function and are responsible for allocating `NASTYFILE` handlers and `NastySocket`s.