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, we use the term method to avoid confusion.
- A **id** is a number that we associate with a filename

The protocol

Packet Structure

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

- `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 Client Module

A client is constructed from a has two methods,