Report 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 designs that are *stupid* rather than *smart*.

In our system *smartness* is an optimization.

Because of this, our system is inherently agnostic to chronology. Files could be tranferred concurrently. In fact **both** the client and server processes can be killed and restarted at any point in a single file transfer, and the protocol is able to heal and succeed.*

We spent a whole week devising this system before writing any code, and once we had all our stupid `C++` bugs sorted, it worked pretty much first time.

*Adding this feature to our initial design took only 4 extra lines of code, the power of *stupidity*.

Successful cases

We think it handles all nastiness, **except** issues with long directory and file names, or directories that don't end in '`/`'—basically we used C style strings.

Some terminology

- **Smart** vs **stupid** is another way of saying selfconscious vs unselfconscious
- A **message** is a requested action that is conveyed over a network
- A **id** is a number that we associate with a filename
- A **module** is a set of related functions

Modules

The `packet` module

This module defines the structs representing packets as defined below, as well as a number of functions used to easily construct those packets. All packets are easily sent over the network because they are implemented as tagged unions which contain no pointers, therefore they are serialized and deserialized easily via C-style casts.

The `files` module

Provides functions to reliably read files from disk and a data structure `files_t` which stores metadata of the files on disk, it doesn't store file contents.

This module uses a nasty file pointer under the hood, but provides the user with reliable reads by repeatedly reading a file, hashing it, and comparing the it to previous hashes until there are a sufficient number of matching hashes.

It uses this functionality to convert a file stream into the all the necessary packets to transfer the file, including correct checksum.

This module **adapts** to the nastiness level by requiring 5 matching reads for `nastiness` 5, 4 for `nastiness` 4, etc...

At a certain point, which is also adaptive to `nastiness`, the read will give up and declare an unfixable disk error, which will try to trigger an SOS.

For write functionality, we again repeatedly write, then read and hash to ensure the write was correct.

It is a sort of personal end to end just for disk IO. Though, it's still separate to the real end to end, and is no more than an optimization instead of resending.

The `messenger` module

Contains functions to reliably send packets over the network or send a collection of files to the server. A `messenger_t` struct must be initialized with a "nasty socket" and the nastiness used for that socket. The messenger functions will adapt to the nastiness level and allow for a greater number of retries under higher nastiness levels. It also contains a monotonically increasing (though not necessarily linear), sequence counter used to order each outgoing packet.

Sending via a `messenger` *guarantees* that the receiver has responded, and is aware of the message.

The module has functions to send either an array of packets or a `files_t` instance containing files to send. The file sending function utilizes the packet sending function, but first splits the files up and sends the whole sequence of packets necessary to fully transfer each file.

As long as the protocol is followed multiple files could be transferred concurrently, though we don't actually do this.

Both packet sending and file sending give a best effort attempt, and fail after a certain number of retries, the number of retries is adaptive to network `nastiness`.

After a file is sent, this module also performs an end-to-end check for that file by hashing it and asking the server to compare the hashes. If the check fails, then the transfer is retried until it succeeds, or retry threshold is hit.

If a message that is sent is responded with via an `SOS`, the transfer of that file restarts.

The `cache` module

This module takes inspiration from walls (which are underappreciated). It provides one real function called `bounce` that takes in a packet, tries to perform the requested action, and returns a new packet `ACK` if the action was successful, and `SOS` otherwise.

This module is meant to be used entirely by the server, and is used to interpret and act on incoming packets. It provides a number of idempotent functions which correspond to the actions requested by various packet types. These functions are designed to be called multiple times safely so that duplicate packets have no adverse effect. This is achieved with a combination of checking sequence numbers and checking file status.

Based on inferring the `cache`'s responses, a client can estimate the current state of the server. The server does not know it's own state, and is therefore unselfconscious and **stupid**.

There are internal optimizations to prevent repeated work that do not violate the idempotence of each action.

The `listen` function

`listen` is the main server function, it is a loop that does the following:

- read a packet
- `bounce` the packet against the cache
- respond with the same packet metadata plus a copy of the highest recorded sequence number that the server has received*

* This is used as part of the self healing for recovering from and old process

The protocol

A big theme of this protocol is to make the server very simple and letting the client handle most of the intelligence. There are 7 packet types which work together to transfer files:

- SOS
- ACK
- PREPARE
- SECTION
- CHECK
- KEEP
- DELETE

1. Prepare for file

When the client wishes to transfer a new file, it sends a PREPARE packet containing the filename, file length, and the number of parts (or "sections") the file will be broken into. Since each packet has a maximum size of around 500 bytes, some of the larger files we tested need to be broken into over 6000 sections. The client then waits for the server to respond with an ACK before moving on to sending sections.

After receiving a PREPARE packet, the server will add a new file entry into its cache with the metadata sent over by the client. The server keeps track of the sequence number the PREPARE message was sent with so that it can filter out packets from older failed transfers. Since the server's actions are idempotent, if it received the same PREPARE packet more than once, it will ACK the duplicates but do nothing.

2. Send sections

Once the client has received an ACK for the PREPARE, it sends a barrage of SECTION packets, each a different bit of the file being sent. It waits to receive ACKs for these, and removes the acknowledged ones from its queue.

When the server receives a SECTION, it ensures that is has a cache entry for the corresponding file. If it does, then it checks whether it has received that section before, and saves it if it's a new section, then responds with an ACK. This ensures that duplicate SECTIONs are still ACK'd but have the same effect, giving the property of idempotence. If the server did not recognize the file that the section was for (i.e. it did not receive a PREPARE), then it sends an SOS message back to the client, indicating that it needs to restart the transfer process.

3. Check

After sending all the sections of a file and receiving ACKs for each, the client sends the hash for the entire file to the server in a CHECK packet. If the client then receives an ACK, then it knows the file is good, and if it receives an SOS then it knows the file transfer did not succeed and it restarts the transfer for that file.

When the server receives a CHECK packet, it checks the following:

1. That it has a file entry for the requested file

2. That is has received all sections for the file
3. That the server-side data hashes to the correct hash

If any of these checks fail, then the server responds with `SOS` which triggers a restart of the transfer. Otherwise, the server saves the file to disk and responds with an `ACK`. If the file was already previously checked, then the server does nothing and responds with the result of that previous check, either a successful `ACK` or a failing `SOS`.

4. Wrap up

Once a file is checked

Packet Structure

i32 type	HEADER			DATA		
SOS	i32 len	i32 seq	i32 id	i32 maxseq		
ACK	i32 len	i32 seq	i32 id	i32 maxseq		
PREPARE	i32 len	i32 seq	i32 id	i8[80] filename	i32 file length	i32 nparts
SECTION	i32 len	i32 seq	i32 id	i32 partno	i32 offset	u8[488] data
CHECK	i32 len	i32 seq	i32 id	u8[20] checksum	i8[80] filename	
KEEP	i32 len	i32 seq	i32 id			
DELETE	i32 len	i32 seq	i32 id			

- `len` is the length of the entire packet in bytes
- `seq` is the sequence number assigned by the `Messenger`, and newer packets must have higher sequence numbers than older packets
- `id` is a unique identifier assigned to each file, not each packet. They are assigned sequentially but it doesn't matter how they're assigned, they only need to be unique
- `SOS` is constructed by the server as a response to a previous packet. SOS packets are constructed by sending the incoming packet back to the client with the type value changed to `SOS`. SOS packets trigger the client to reset whatever process prompted the SOS - this is the self healing mechanism. An `SOS` has a matching `seqno` to the packet that triggered it.
- `ACK` is constructed the same as SOS. It notifies the client that the requested action with a matching `seqno` was performed successfully. The client will continue resending packets until it receives an ACK for them.
- `PREPARE` is sent to indicate to the server to get ready for a file separated into `nparts` sections, totalling `filelength` bytes. The server will associate the `filename` with the `id` and be capable of receiving `SECTION` packets to reconstruct the indicated file.
- `SECTION` is a section of a file, identified by its `partno` and placed in buffer at `offset`. The server expects that it had previously received a `PREPARE`

packet for the indicated `id`.

- `CHECK` tells the server that an end to end check for a specific file is necessary. The server will ensure it received all the file's sections, then hash the file and compare it to `checksum`. Both `filename` and `id` are provided so that the client can also request a `CHECK` for a file that the target directory contained a priori. An ACK is sent if the file is correct and stored in a `.tmp` file. SOS is sent if the file is damaged, triggering a retransmission of that file.
- `KEEP` tells the server to save the `.tmp` file matching an `id` to disk
- `DELETE` tells the server to delete the `.tmp` file matching an `id`

It is always assumed that these messages come in any order and may be duplicated.

The `Packet` data structure is represented as a `struct` and written to be easily serializable.

Gradelogs

TODO