Avi Weinstock
Rachel King
Distributed Systems
2016-11-29

Project 2 Report

Language and Resources

This program uses Rust. Libraries ("crates" in Rust's terminology) and references that we used to build the program are as follows:

- Avi's Project 1 networking code: `https://github.com/aweinstock314/weinsa_westem_distributedsystems_project1`
- The `tokio-core` crate is Rust's main library for asynchronous IO. It builds upon `mio` (a non-abstract wrapper around `epoll(2)` on Linux and `kqueue(2)` on BSD) and `futures` (an abstraction of general asynchronous computations).
- `serde` provides serialization and deserialization. It hooks into the compiler to do static reflection over datastructures, and generates generic code (traits) that serializes/deserializes arbitrary protocols. `serde_json` invokes this generic code to read/write JSON.
- `nom` is a parser combinators library (a way of writing parsers for custom formats, similar in purpose to `flex`/`bison` in C). It is used to define the format for `nodes.txt`.
- `log` is a logging framework that provides the {`warn!`, `info!`, `debug!`, `trace!`} macros, which can be used to print log information of various verbosity levels. `env_logger` is a backend in this framework that reads an environment variable to determine what to print, and prints to standard error with the log level prefixed to the lines.
- `argparse` provides an object-oriented builder for command-line argument parsing that generates a nicely-formated `--help` command.
- `byteorder` provides convenience methods for reading/writing bytes from/to arrays (e.g. `LittleEndian::write_u64(&mut buf, p` instead of a 3-5 line loop)
- `either` provides the `Either` datatype and some convenience functions for working with it (this is in Haskell's standard library).

Program Structure

Six Rust files define the relative functionality for this program:

- **algos.rs** contains functionality for each process to update, maintain, and back up the program's file system. To facilitate this, it also houses message handlers for committed client actions (creating, deleting, or appending files). It also contains a message handler for messages recieved by the process from a connected client.
- **broadcasts.rs** interfaces with the networking code, providing algorithmic oversight to the sending and recieving of messages. It contains structs to manage ZAB and Bully Leader Election respectively. It invokes `algos.rs` message handling on Zookeeper-delivery of a message to edit a process' filesystem.
- **framing_helpers.rs** alkfjsldkjfsdf
- **nodes.txt** lists processes in the system. Each contains a respective PID, IP address, and port numbers for connecting to clients and peer processes, respectively.
- **parsers.rs** contains functionality to parse nodes.txt. This functionality is used in `main.rs`.
- **main.rs** aslkaslkdj

Messages are sent between processes as serialized json data. There are several types of messages that can be passed around between processes and clients. :

- **ServerToClientMessage** is defined in `algos.rs` and contains a string to display on console for human viewing. It is sent from the server to the client on receipt of a client request.
- **ClientToServerMessage** is defined in `algos.rs` and contains actions requested by client, recieved by a process, and forwarded to the program leader. The four types of ClientToServerMessages reflect the four types of client requests: create, delete, append, and read. They each carry the name of the file they act upon, and in the case of append, the data to add to that file. These messages are sent from a process connected to a client to a leader process.

- **SystemRequestMessage** is defined in `algos.rs` and defines actions to perform on the commit/delivery of a client request. Actions are create, delete, and append. These messages are sent from a leader to a process.
- **ZABMessage** is defined in `broadcasts.rs` and defines a set of messages used to manage Zookeeper Atomic Broadcast. Commit and Ack are two messages used to model a Two Phase commit broadcast. Forwarded is a message type used to flag a message from a peer that is passing along a client request. SendAll is a message type used to flag messages to send to every process in the system. Election is a message type used to encapsulate BullyMessages passed between processes during a leader election, as they are handled first using a ZAB message handling function. These messages are sent between follower and leader processes.
- **BullyMessage** is defined in `broadcasts.rs`. They define four types of messages that are passed around to facilitate Bully leader election: Election, Coordinator, Okay, and Tick (which is used to determine time passed during the election.) They are passed between processes as they determine who should be the next defined program leader.

Filesystem

Every processes maintains a logical filesystem in the System struct, as defined in `algos.rs`. Files managed by processes in the program are stored in "files", a HashMap that maps a file's name to its contents. The System struct also handles messages and message logs for a process. When a process is cleared to commit/deliver a requested action, as determined by the ZAB algorithm, that action is forwarded to them in the form of a SystemRequestMessage. The System struct handles that message by writing it to a log vector, writing that vetor entry to a file on disk, and then performing the perscribed action on its logical filesystem. In summary, a process' System determines behavior on commit/delivery of a peer message.

As a side functionality, the System struct also handles messages recieved from clients, forwarding them to the leader by way of a stored broadcast object. This broadcast object compartmentalizes logic for sending, recieving, and delivering messages between processes. In this program, broadcast implements the Zookeeper Atomic Broacast. See Algorithms section for more details. Its construction also handles data recovery on initialization.

Network Configuration

Processes are listed in nodes.txt. Each contains a respective PID, IP address, and port numbers for connecting to clients and peer processes, respectively. The functionality for parsing this data is located in `parsers.rs`, and is run in `main.rs` when the system is initialized.

Initialization

Relevant portions of the system are initialized at startup as follows:

- Main runs:
  - Networking code is initialized, with threads created for processes, etc. (See **Networking** for details).
  - Main parses data from node.txt using `run_parser_on_file`. Processes are stored in a HashMap that maps sockets to PIDs.
    - A ZAB broadcast object is initialized in a `control_thread` (See **Networking**).
    - A System broadcast object is initialized in a `control_thread` (See **Networking**).
- ZAB init:
  - Bookkeeping variables are initialized or set based on arguments passed to the struct. (see **Zookeeper Atomic Broadcast**)
  - A BullyState object is initialized as the leader.
- Bully init:
  - Bookkeeping variables are initialized (see **Bully Algorithm**).

  System init:
  - Bookkeeping variables are initialized (see **FileSystem**).
  - If it doesn't exist already, a log.txt file is created on disk.
  - If log.txt already has data, process fills the logical representation of its filesystem according to its contents. (see **Process Failure & Recovery**).

Process Failure & Recovery

A failed process must be manually restarted in the terminal, as described in the **Instructions** section. On recovery, the process initializes as described in the **Initializaiton** subsection. As part of this initialization process, when it creates a new System object, it checks its local log file. If that file is not empty, it will process the json data within and "resend" each parsed message to itself to add it to its logical filesystem. There is a boolean flag that prevents these "resent" messages from being added to the disk log during this process.

If the leader process fails, followers will detect this via the Tick/Heartbeat system described in **Networking**. Upon recovery, the failed process initiates leader election as descried in the **Bully Algorithm** section.

Algorithm Implementation

Zookeeper Atomic Broadcast

Zookeeper Atomic Broadcast is implemented in `broadcasts.rs` as a set of structures that maintain bookkeeping variables, message types, and message handling for the algorithm. Relatedly, there is also a Zxid struct, which contains a integer variables Epoch and Counter. ZAB structures are as follows:

- **ZabTypes** ZabTypes is an enum that defines a set of message types to be matched against when Zab handles messages. These messages types are articulated in the **Program Structure** section.
- **ZabMessages** ZabMessages is a struct that defines a set of data carried in each message. It passes along the pid of the message sender, the pid of the process the client is connected to, a message type (defined by ZabTypes), and a counter that is a Zxid object.
- **Zab Struct** The Zab struct stores a set of bookkeeping variables for Zookeeper Atomic Broadcast implementation. It stores a ZXID for the process, to be incremented locally and passed along with messages it sends. It has an integer `ack_count`, which the leader uses to keep track of acknowledgements it recieves from followers during its two-phase commit process. It has a ZXID called `next_msg`, which increments to keep track of the next message ID in the system that should be delivered, and a `msg_q` to buffer messages waiting to be delivered, again as in a two-phase commit process. It stores a leader as a BullyState object, which it uses to initialize leader elections and store the leader process' PID. Finally, it stores its own PID, a HashSet of PIDs of other processes in the system, and deliver, which stores the function that gets called when a deliver is invoked.
- **Zab HandleMessage** Zab HandleMessage is an implementation of the generic broadcast HandleMessage function for ZAB. HandleMessage matches against all types of ZabMessages and determines the appropriate behavior on receipt. This includes delivering messages when recieving a commit message from the leader, broadcasting a client request when recieving a forawrded message, and counting ack messages to determine whether or not to send commits to the group.
- **Zab BroadcastAlgorithm** Zab HandleMessage is an implementation of the generic broadcast `set_on_deliver` and `broadcast` functions for ZAB. The `set_on_deliver` function invokes the `deliver` function passed to the struct during **Initialization**. In practice, this function sends a SystemRequestMessage containing the client's request to the process' System struct, which handles the message and performs the given operation accordingly. `Broadcast` forwards the vector of messages to send to the leader, or initiates a leader election if there currently is none.

Zab also contains implementations of a `new` function, which handles its **Initialization**, and an `internal_broadcast` function, which allows a process to send a message to itself. This is used by the leader when it broadcasts a commit message.

In execution, these structures allow the processes in the program to perform Zookkeeper Atomic Broadcast and maintain a consistent filesystem among them. When a client sends a request to a process, that request is forwarded to the leader. The leader then broadcasts it as a SendAll-type message. When a process recieves a SendAll message, it will store it in its `msg_q` and send an Ack to the leader. The leader increments its `ack_count` at each receipt, and when it recieves a majority acknowledgement it broadcasts a Commit message to everyone, incldiuing itself. On receipt of the Commit message, a process invokes its ZAB `deliver` function, and passes the message to its logical filesystem to be processed accordingly.

Bully Algorithm

Leader election protocol is implemented in `broadcasts.rs` as a set of structs that maintain bookkeeping variables, message types, and message handling for the Bully Algorithm.

- **BullyTypes** BullyTypes is an enum that defines a set of message types to be matched against when the BullyState handles messages. These messages types are articulated in the **Program Structure** section.
- **BullyMessages** BullyMessages is a struct that defines a set of data carried in each message. It passes along the pid of the message sender, the pid of the process the client is connected to and a message type (defined by BullyTypes). BullyMessages are encapsulated in Election-type ZabMessages when sent between processes, as they are recieved by the Zab message handler first.
- **BullyState** BullyState stores a set of bookkeeping variables for Bully Leader Election implementation. It stores a process' PID, a collectively recognized `leader_pid`, a counter that tracks Tick messages recieved, a HashSet of its neighbor processes, and booleans for whether or not a given process has recieved Okay or Coordinator messages.

  Leader_pid is an Option object, which allows us to test whether or not there is a collectively recognized leader among processes. If this Option is None, the system is currently holding an election.

  BullyState also has an implementation of HandleMessage, used to process BullyMessages when they're recieved by the process. This is invoked by Zab's HandleMessage when the process recieves an Election ZabMessage.

In execution, these structures allow the processes in the program to perform a Bully Algorithm leader election. When leader election is initialized, processes send election messages to others with lower ZXIDs. The process that initiates the election counts

Tick messages to wait and recieve responses from its neighbors. Upon recieving no responses (`recvd_ok` = false) it sets itself as the leader and broadcasts a Coordinator message. Otherwise, it waits another series of Ticks for a Coordinator from someone else.

ERRATA: This leader election process currently works during initialization of the program, but has bugs that prevent processes from consistently agreeing on an elected leader should that leader crash.

Networking

Historical context

The POSIX standard system calls for networking (e.g. {`socket(2)`, `connect(2)`, `bind(2)`, `listen(2)`, `accept(2)`}) are all blocking by default. That is, while attempting to use a socket, the current thread cannot do anything else. Parallelism (either via threads with `pthread_create(3)` or processes via `fork(2)`) allows a server to use blocking system calls to service multiple clients at once, but incurs the overhead of context switches and costs a nontrivial amount of memory per task. The `select(2)` and `poll(2)` POSIX system calls allow a single thread to check multiple file descriptors for readiness (and hence service multiple clients per thread), but provided an interface that is O(n) in the number of open file descriptors, and cannot be backwards-compatibly changed to something more efficient. `epoll(2)` (Linux-specific) and `kqueue(2)` (BSD-specific) are system calls that provide the same functionality as `select(2)`/`poll(2)`, but are O(n) in the number of active file descriptors (which is optimal).

Rust libraries

The `mio` crate provides a portable wrapper around `epoll(2)`/`kqueue(2)`, but still requires manually keeping track of which file descriptors the program expects to be notified of what types of readiness, and in what mode ("edge-triggered" vs "level-triggered") to be notified. The `futures` crate isn't directly networking related; it provides a concurrency abstraction of the same name. A future of a value is an object that represents an in-progress computation of a value. It provides various methods for composition (e.g. waiting for two computations to complete, applying a callback once a value is complete) that end up building a computation graph, which is executed in a state-machine-like fashion. `tokio-core` bridges `mio` and `futures`: it provides leaf nodes for `futures`'s computation graph that do socket communication via `mio`.

Our organization of these

`server_main` creates two `tokio_core::io::TcpListener` streams (one for clients to connect to, one for peers to connect to). It receives the ports to listen on from `nodes.txt`. Both `TcpListener`s forward the accepted sockets to a `futures::mpsc::Receiver<Control` that has access to all the server state. `ControlMessage`s are a datatype of our invention that is used for communicating within the process. Our `ControlThread` datastructure stores the other algorithmic datastructures, and orchastrates funneling messages through the sockets via `serde`.