Avi Weinstock
Mark Westerhoff
Distributed Systems
10/16/2016

Homework 1 Report

Abstract

This report outlines our design and implementation of Raymond's algorithm for mutual exclusion in a distributed system. While it has been thoroughly tested with EC2 (Amazon Elastic Computing Cloud), the focus here is not on results or performance metrics but instead on the specific design decisions we made in our program.

How to use

There is a README.md that explains how to install and run our program. It outlines how to connect to the CLI and interact with the program, as well as how to format the necessary setup files (one for the graph structure, one for addresses and ports).

Programming Language

For our implementation of Raymond's Algorithm, we decided to use the programming language Rust. It's looks relatively similar to C++, and is a compiled programming language, however it is a blend of functional and procedural style programming and therefore allows more flexibility than C++. It heavily focuses on safety, as it enforces type checking with strict type checking at runtime. Rust also has a very different definition of references, relying heavily on move semantics and scope to ensure that you cannot perform use-after-free issues. Class inheritance does not really exist; instead there are simply traits, which are synonymous to interfaces in other languages like Java. All these features put together allow That being said, our code is heavily commented, so specific details about Rust hopefully should not matter.

The Program

One instance of the program must be booted up for each logical node in the distributed system. There can of course be multiple logical nodes on the same system, but there needs to be an instance of the program for each. Each process is given its own corresponding id, and generates a listing of it's neighbors and creates socket connections based on the configuration files. It boots up a TCP server so the nodes can communicate, and sets up a CLI interface over a socket. Once all this is completed, the program should be up and running and ready to communicate. It supports the functions `create, delete, read, append, ls` on multiple "files" (strings) using Raymond's algorithm.

Raymond's Algorithm

We implemented Raymond's Algorithm exactly the same as presented to us in class. There are internal functions `assignToken` and `sendRequest` to handle moving/locking the resource and sending request messages respectively. There are the four functions from class: `requestToken`, `receiveToken`, `receiveRequest`, and `releaseToken` that should be self explanatory. Instead of calling these directly, we have abstracted our implementation of Raymond's algorithm to match an interface we called `MutexAlgorithm`. `MutexAlgorithm` requires the public functions `request`, `release`, and `handleMessage`. This way, `request` and `release` handle acquiring and releasing the token (i.e.), and `handleMessage` handles receiving the token or a request. Essentially, the public interface just acts as a blocking, asynchronous wrapper around the Raymond's Algorithm functions that do not return until they get the resource.

In terms of handling the state, we decided to keep a global mapping of $res \rightarrow state$, where res is the resource name and $state$ is the set of values the Raymond algorithm needs (`usingResource, holder, reqQ, asked`). This way, we do not need to spawn off an instance of our algorithm for every new resource. Essentially, the state is decoupled from the algorithm itself. Each resource merely adds an entry into the mapping; every time we request a resource res, we pass in the corresponding $state$. This is more realistic, as it does not couple the "file" with the algorithm, providing a more flexible interface. Within $state$, we also added our own pid (so we can do `holder == selfpid`), a generic placeholder for the resource, and a queue of callbacks. The resource field is only ever used when we own the token (i.e. `holder == selfpid`), so it won't be outdated. The callback is the function that is actually using the resource; it is queued to ensure correctness even with multiple requests from the same process.

Create and Delete

To propagate the create and delete messages, each process stores a listing of it's neighbors. Then, when it receives one of the two messages, it can broadcast the message to all its neighbors except the one that it received from. Implementation-wise, we have the process send a create message to itself, much like the broadcast algorithms we discussed in class. Reads cannot occur on a process that has not received the create message yet, simply due to the fact that TCP is inherently FIFO. Delete messages also require the process to not be using the resource, waiting until this holds true, preventing any possibly erroneous behavior.

MPSC

It's worth noting one specific module that we used a module that implemented a Multiple Producer Single Consumer (MPSC) data type. This is actually what we used as our queue of callbacks. This way, each command could produce/add a callback into the queue, and they could only be consumed when it was time to use the resource.

Networking and Technical Details

There are four types of messages sent over the wire: create, delete, grantToken(with the resource), and request. These messages are serialized into JSON and then sent over TCP. Every process has two socket listeners; one for intercommunication between processes, and one for the CLI (on a different port). The intercommunication socket is polled asynchronously to be non-blocking (much like a select loop in typical TCP code), and the CLI is handled in a separate thread. The two threads use mutexed shared memory to avoid data races on the $res \rightarrow state$ mapping.