Avi Weinstock
Mark Westerhoff
Distributed Systems
10/16/2016

Homework 1 Report

Introduction

Abstract This report outlines our design and implementation of Raymond's algorithm for mutual exclusion in a distributed system. While it has been tested thoroughly 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).

Overall Program

Programming Language For our implementation of Raymond's Algorithm, we decided to use the programming language Rust. It's looks relatively similar to C++, as it is a compiled programming language, however it is a blend of functional and procedural style programming and is much different semantically. 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. That being said, our code is heavily commented so don't worry about learning Rust too much.

Program Layout The layout of the program

MPSC It's worth noting one specific feature that we use called `futures::oneshot`. Effectively, they are similar to an asynchronous callback pair for send and receive. They are denoted as a pair of `(Complete, Oneshot)`. When you want to send a message, you call `oneshot`,

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 cl `requestToken`, `receiveToken`, `receiveRequest`, and `releaseToken` that should be self explanatory. Instead of calling these directly, weass 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 an blocking, asynchronous wrapper around the Raymond's

Algorithm functions that don't return until they get the resource (much like a select loop in TCP code).

In terms of handling the state, we decided to keep a 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 resource. Every time we request a resource res, we pass in the corresponding $state$. Within $state$, we also added our own pid (so we can do `holder == selfpid`), a placeholder for the resource (only used when `holder == selfpid`), and a queue of callbacks. 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. Delete messages also require the process to not be using the resource, preventing any possibly erroneous behavior.

Networking