MESSAGING SYSTEMS LAB: DISTRIBUTED CONSENSUS WITH AKKA AND RAFT

Adam Michael
Rose-Hulman Institute of Technology
February 10, 2016

In this lab you will explore the application of messaging systems to the design and construction of fault-tolerant, highly-available, eventually-consistent data systems.

Prerequisites

Before starting this lab you will need to install Scala and SBT. If you prefer, you may use Activator instead of SBT. If you have not used Scala before, it is recommended that you read (or skim) through the documentation and go through some tutorials.

Background

In the world of software systems, availability is a metric of the percentage of time that a service is correctly servicing requests. Maintaining high availability for a system is difficult due to factors like network disconnectivity, server maintenance and hardware failure. One common solution to these problems is architecting the program as a distributed system to eliminate any one point of failure. Ideally, the distributed system should maintain availability even when several of its components are experiencing failures. This can be done by sharing state among the components, so that if any one server is down, its data is available elsewhere.

This model of shared state distributed systems introduces a new set of problems. If you are not careful, some components may have stale data or, worse, incorrect data. This problem is known as consensus. Every request to the distributed service should be agreed upon by all of the components. Even components that are failing at the time of the request should eventually acknowledge the request.

There have been several attempts to algorithmically solve consensus. Most notably, the Paxos algorithm is provably correct and cited by many as the standard way to solve consensus. In industry, most find Paxos too obtuse for practical implementations; Google Chubby and Apache Zookeeper are implementations of the same problem that rely on unproved algorithms.

In this lab we will focus on a new consensus algorithm, Raft. Raft was developed by Diego Ongaro and John Ousterhout at Stanford in 2013 as an attempt to make a simpler, easier to understand consensus algorithm.

Problem

In this lab, you will implement Raft from scratch using the popular Scala actor runtime, Akka. On top of that you will create your own implementation of `scala.collection.mutable.map` that stores its data in a Raft actor system. Finally you will learn how to distribute your Raft actor system across remote hosts using department-provided VMs that Chandan will provision for you.

Part 1 - Understanding Raft

Read and understand at least sections 1, 2, 5 and 8 of the Raft paper. Section 5 contains all of the details of the worker messages. Section 8 contains a description of the client messages.

Part 2 - Understanding Akka

Akka is an implementation of the "Actor pattern". That is, Akka-based software is decoupled into independent Actors that communicate via messages. Under the hood, Akka configures a system of channels to pass messages between actor endpoints, but for our use you can simply treat actors as threads with mailboxes.

Akka is a much larger framework than can be covered in just this lab. Your solution to this lab is not restricted to any part of Akka, however I recommend reading the following docs pages to get started:

- Terminology
- What is an Actor?
- Actor References
- FSM Actors
- Cluster Usage
- Persistence

You will likely need to refer to the Akka documentation repeatedly during this lab.

Part 3 - Raft Actors

Implement the algorithm described in Section 5 of the previously linked Raft paper as an Akka Actor. Some general advice:

- Scala is not Java. Most objects are immutable by default and you should program as such.
- Case classes make fantastic message types.

- The FSM Actor mixin is a convenient way to manage the state of the algorithm through the transitions between follower, candidate and leader roles.
- Don't try to use TypedActors. They're less robust and will make your head hurt.
- The `forMax` syntax of state transitions is convenient for simulating Raft election timeouts.
- Use the Akka cluster seed-nodes to inform each worker of the addresses of the other workers. You do not have to deal with workers joining and leaving the cluster, although it is discussed in Section 7 of the paper if you are curious.
- At the very beginning, workers should wait until all workers are online to save their actor paths.

Part 4 - Distributed Hash Map

Create your own implementation of **`scala.collection.mutable.map`** that manages its own Actor System. Your implementation should at least be generic in its value type. It should use the techniques discussed in Section 8 to communicate with the Raft cluster.

Create a sample application to test adding, removing, getting and iterating over your distributed map.

Part 5 - Remote Actor System

Using the VMs allotted to you, prepare a demo of a 5-node cluster running on 5 VMs and arunning your sample application that queries the cluster from your laptop. Show the working demo to an instructor or TA prior to the due date.