Ad-Hoc Network Simulation Project Report

Benjamin Mayhew

May 9, 2013

1 What I accomplished this semester

1.1 Literature Review

I was first inspired to work on a project related to algorithms on sensor and ad hoc networks when last semester I read a document by Robert Ghrist called Applied Algebraic Topology and Sensor Networks[1]. In it, he describes a method to use the homology of a communications graph to find holes in the geographical coverage of the network. The method requires global knowledge of the graph however.

A separate paper co-authored by Ghrist describes a fully distributed method for a network of nodes to compute its coverage[2]. The algorithm was way too complex to be my first distributed algorithm that I implement, but reading this paper led me to understand that, when a simple computation must be done in a distributed way, the algorithm needed to accomplish this can become vastly more complicated than the birds-eye-view algorithm that accomplishes the same.

Early in the semester I sought out more literature on the topic of distributed algorithms on ad-hoc networks. I obtained a very helpful book on the exact subject[3]. I read through the first several chapters and learned about a few different aspects of this subject area. One chapter focused on describing a formal model of an ad-hoc network. This way of looking at an ad-hoc network proved to be very useful when later in the semester I began coding; because I made sure to adhere to the formal model as much as possible, I am fairly confident that the code that governs node behavior will work as expected when it is finally tested.

Later chapters in the book also described the different ways that one can look at the efficiency of a distributed algorithm. Time complexity is one way, but in an ad-hoc network, it is also useful to analyze how the number of messages passed between nodes, or the total bandwidth used, can be minimized during the calculation of an algorithm.

One chapter talked about one of the most basic distributed algorithms, the *broadcast algorithm.* As a precondition for running this algorithm, it was assumed that nodes in the network have knowledge of a spanning tree of the network. It was then that I decided that what I wanted to code this semester would be an implementation of a distributed algorithm for calculating the spanning tree of a network.

1.2 The algorithm

I searched for papers that described a distributed spanning tree algorithm, and found one that made use of mobile agents that randomly walk throughout the network, marking nodes and constructing spanning subtrees in the process. When two spanning subtree collide, local comparisons are made and an agent representative of the higher-priority subtree is created to annex the lower-priority subtree in a deterministic way. Eventually, all agents have died except for one, and this agent will eventually have traced out the entire spanning tree of the network.

1.3 Coding

1.3.1 Process

I decided to use this opportunity to learn a new programming language, Scala. It was very slow going at first to learn the Scala API and become comfortable with all of the neat functional capabilities of the language, but eventually I got the hang of it and now I'm glad I put in that work because it has become my favorite language, and is way better than Java in my opinion. Earlier in the semester when I first started coding, I see that the style I used was more imperative and closer to how I would program in Java, while now, the style I use is heavily functional and makes use of a lot of aspects of Scala that do not exist in Java.

Early on, I tried to do concurrency using normal Java threads. This turned out to be a nightmare. After a little digging I learned about the actor model of concurrency, and that Scala provides a library to write code using this model. In the actor model, computation is viewed as a process of asynchronous message passing between independent, fully encapsulated *actors*. Not only does the actor model make it much easier to write fault-tolerant concurrent applications, but it's also a perfect conceptual fit for when you are trying to represent independent ad-hoc network nodes (and there are other benefits in memory usage and efficiency, the way this model is implemented in Scala). I also read these two papers[5][6] that are linked to from the Scala API, and they sold me on the superiority of the actor model.

After learning about the actor model, I went back and mostly started coding again from scratch, this time being very meticulous about doing things the 'right' way the first time, since I have learned a lot about the danger of sloppy code from a bad programming experience last summer. I completed the code that runs a single node in the network, and have basically completed a sort of supervisor class that governs an entire network of spanning-tree-computation-nodes.

1.3.2 Program structure

The SpanningTreeNode class represents a single node in the network with the capability of participating in the distributed spanning tree computation.

A SpanningTreeNode may only be interacted with by passing messages to a special ActorRef instance, that serves a 'black box' that hides all of the inner workings of the SpanningTreeNode. Even a direct reference to a SpanningTreeNode itself is forbidden by the program. This is the way that the Akka library (the third-party Scala library for writing code using the actor model) preserves the integrity of the actor model within a concurrent application.

There are three ways to interact with a SpanningTreeNode:

1. **Message the node with the ActorRef representing another SpanningTreeNode**

   This causes the node to add the new ActorRef to its contacts. The node then messages the new contact with its own ActorRef, so the pair forms a bidirectional communication link.

2. **Message the node with a Seed object**

   This causes the node to create a new random walk agent and effectively start the spanning tree computation with a subtree rooted at itself.

3. **Message the node with an InfoRequest object**

   The node replies with a NodeState object representing its state. The NodeState contains fields that describe the subtree that this node belongs to, its parent node, and its children.

The SpanningTreeSupervisor class is also an actor. An instance of this class controls an entire network of SpanningTreeNodes. This class is responsible for being the bridge/translator between the highest-level abstraction of the full simulation layer, and the lower-level workings of the actor model. The SpanningTreeSupervisor is the highest level of the actor hierarchy in this program.

There are two ways to interact with a SpanningTreeSupervisor:

1. **Message the SpanningTreeSupervisor with a Go(Args) object**

   The SpanningTreeSupervisor, informed by the arguments provided with the Go object, sets up a new SpanningTreeNetwork and starts its computation.

   The arguments are of the format numNodes:Int, connections:List[(Int,Int)], seeds:List[Int]

2. **Message the SpanningTreeSupervisor with an InfoRequest object**

   The SpanningTreeSupervisor replies with a list of Option[AbstractNodeState] representing the state of each of the nodes in the network it controls. The Option has value None when the corresponding node fails to reply to the SpanningTreeSupervisor with its node state. An AbstractNodeState is a NodeState 'translated' into the high-level abstraction that the simulation layer understands (i.e., the field *children* is represented by a list of integers, rather than a list of ActorRefs).

The third component of the completed application is the Simulation class. It is not yet implemented. A Simulation's *view* of a network is the most basic way to represent a graph – nodes are simply integer indices, and the entire set of communication links is represented either as a list of (Int,Int) tuples, or as a symmetric $n \times n$ adjacency matrix, where n is the number of nodes in the network. A Simulation never sees the inner workings of the actor model, except in that it instantiates and passes messages with SpanningTreeSupervisors.

So the main function of the Simulation class is to construct the communications graph, and then pass it along with whatever other arguments are needed to a SpanningTreeSupervisor. The Simulation can then send InfoRequests to the SpanningTreeSupervisor to get 'snapshots' at different times of the state of the overall network.

Experimentation with different methods of communications graph construction, and different network sizes, can happen within the Simulations class. A Simulation can be aware of when spanning tree computation completes, and can run tests to time the algorithm. The Simulation can also run multiple networks in parallel and compare results.

2 Future ideas/work

2.1 Short term

Complete the Simulation class and start running some tests and fix whatever errors pop up. Expand the functionality of SpanningTreeSupervisor so that it can set up a middleman node between the endpoints of any communication link. This middleman will make it so that more real-life communications environments can be simulated. For example, the middleman can have a delay before forwarding messages between the two endpoints, or it can have a random chance of dropping any message it handles. Again, these properties of the communication links will be determined in the arguments that Simulation messages to SpanningTreeSupervisor.

2.2 Medium term

Back off from coding for awhile and start to do some reading about *formal verification.* See if it will be feasible to formally verify the SpanningTreeNode code or the distributed algorithm as a whole, as implemented in this program. Also look into doing some statistical tests using the Simulation and get some plots showing how well the algorithm performs for various types of networks under various conditions.

2.3 Long term

The long-term idea of this project is to have this software run on physical nodes in an actual ad-hoc network. The software should work as a 'base layer' that provides useful functionality that more complicated distributed sensor/ad-hoc network applications can be built on top of. For example, probably almost any complex ad-hoc network application will require knowledge of a spanning tree, since this is necessary in order to efficiently broadcast information to the entire network.

The software should be totally adaptable to almost any type of node or communications capability. The exciting thing about the actor model is that this kind of adaptability should actually be pretty simple.

For example, consider this sort of distributed computation: The network consists of nodes that may be hundreds of miles apart, communicating via short-wave radio.

The software that lies on a physical node consists of a SpanningTreeNode contained within an Intermediary. From the perspective of the SpanningTreeNode, its contacts are actually just other SpanningTreeNodes in a normal network. In actuality, these contact ActorRefs contained within the SpanningTreeNode point to special representative actors that the Intermediary also controls.

When the SpanningTreeNode sends a message to one of its contacts, it gets received by the representative actor within the Intermediary. The representative forwards the message to the Intermediary itself, who then serializes it into a byte stream and forwards the bits to the radio transmitter (the bits are encoded in some way that works well for transmitting via radio), where they go out as analog.

Serializable objects are reconstructed from byte streams received through the radio receiver, and passed as messages to the Intermediary. The Intermediary checks if these objects make sense and are messages directed at this node. It discards the trash and forwards the relevant messages to a representative actor of the sender, who then forwards them to the SpanningTreeNode.

The way this system is described, the code for the SpanningTreeNode need not change *at all*. The way that the actor model works from the node's perspective is completely independent of how the communication actually happens, when it is paired with an Intermediary.

In conclusion, given the completed NodeSoftware and Intermediary, the scheme

NodeSoftware $\rightarrow$ Intermediary $\rightarrow$ Transceiver $\rightarrow$ Physical World

should always work, whether the distributed computation happens entirely within a single processor on a laptop computer, across the internet, or via carrier pigeon.

References

[1] Ghrist, Robert. *Applied Algebraic Topology & Sensor Networks.* Notes for AMS Short Couse, 2010.

[2] Dlotko, P. et. al. *Distributed computation of coverage in sensor networks by homological methods*

[3] Wagner, Dorothea; Wattenhofer, Roger (Eds.) *Algorithms for Sensor and Ad-Hoc Networks: Advanced Lectures.* Lecture Notes in Computer Science, Vol. 4621, 2007.

[4] Abbas, Shehla et. al. *Distributed Computation of a Spanning Tree in a Dynamic Graph by Mobile Agents.* IEEE International Conference on Engineering of Intelligent Systems 2006.

[5] Haller, Philipp and Martin Odersky. *Event-Based Programming without Inversion of Control* Proc. JMLC 2006.

[6] Haller, Philipp and Martin Odersky. *Actors that Unify Threads and Events* Proc. COORDINATION 2007.