Distributed Algorithms for Dynamic Networks

February 6, 2014

Welles Robinson

Senior Thesis

Department of Computer Science

Georgetown University

Abstract

Abstract

1 High-Level Description of Goal

The goal of this paper is to simplify the creation of distributed algorithms for dynamic networks by demonstrating that any algorithm that works for the broadcast variant of the synchronous model with a star topology can be made to work for the broadcast variant of the synchronous model with any topology. We will do so by describing a simulation algorithm that, if run on every node in the broadcast model, will match perfectly the output of the nodes of the centrally controlled model.

2 Models

Distributed Model

1. We are considering the broadcast variant of the synchronous model, defined with respect to a connected network topology G=(V,E).
2. The broadcast variant is defined such that a given node has no knowledge of its neighbors but may send a single message per round that all of its neighbors will receive.
3. Nodes in the network have unique identification numbers (UIDs) and they have knowledge of their own UID.

3 Problem Definition

1. We are considering the main execution to be the distributed model as defined in the Models section with n processes.
2. We are also considering a reference execution of the distributed model with a star topology and n+1 processes. The processes in this reference execution will run a given algorithm A and each process will keep an individual communication log, in which it will write its input and output for every round.
3. The main execution will run some algorithm S, which will take as input A, the algorithm run on the reference execution. The processes in the main execution will keep individual communication logs, which do not have the restriction of those in the reference execution that they must contain the given process' input and ouput for every round.
4. The problem is defined as solved if for the communication log of every process in the main execution, there is at least one identical communication log in the leaf processes of the reference execution and vice-versa.

4 Algorithm

1. Run unmodified Leader Election without Network Information
   (a) Assume nodes have UIDs (which is also assumed for the distributed model)
   (b) Each process runs Terminating Synchronous Breadth First Search and the node that manages to terminate elects itself leader and tells the other nodes to terminate
2. The Simulation
   (a) Assume that each and every round of the given algorithm has a finite repetitions of a step, which is comprised of two parts
      i. The star node sends a broadcast message to all the leaf nodes (Broadcast part)
      ii. All leaf nodes send a receive message to the star node in response to the broadcast message (Receive part)
   (b) Each step is simulated by the leader node running a modified instance of terminating synchronous BFS
      i. The search message sent by a parent node to its children nodes is modified to be "search" plus the broadcast message, which is dictated by the output of the input algorithm A1

ii. Upon receiving a search message, a node runs the input algorithm A2 on the broadcast message portion and then creates the receive message portion using the output of A2

iii. The done message sent by a child node to its parent node is modified to be "done" plus both the UID and the receive message of the child node as well as any done messages received by the child node

iv. Upon receiving a search message, a node writes the broadcast message portion to its communication log

v. Upon sending a done message, a node writes the receive message portion to its communication log

vi. The leader simulates sending a search message to itself and then simulates sending a done message back to itself

vii. This algorithm terminates when the leader has received done messages from all of its children and a simulated done message from itself

Static Model - All the nodes turn on at the same time

Simulation Algorithm takes one input, algorithm A, the algorithm to be simulated. Algorithm A can be broken into two distinct algorithms, A1, the algorithm run by the star process, and A2, the algorithm run by the leaf processes.

Member Variables - maxID (UID); parent (UID); totalChildren (int); childCount (int); wait (int); Message has a root (a round, the UID); a id of the sender (UID); a type search, choose, done; a receiver (UID), defaults to NULL;

Dynamic Addition Model - Nodes turn on at various times but they don't turn off The leader elected will be the node with the highest UID out of all of the nodes that turned on at round 1

Variables - maxRoot - (a round, the UID); parent (UID); totalChildren (int); childCount (int); wait (int); Message has a root (a round, the UID); a id of the sender (UID); a type search, choose, done; a receiver (UID);

5 Proofs

Proof. of Leader Election in the Static Model

This algorithm will eventually elect a leader (see Liveness Proof)
This algorithm will never elect multiple leaders (see Safety Proof)

□

Proof. of Safety in the Static Model

Let Umax = the process with the maximum UID
For the breadth-first search instance with source = any node n not equal to Umax, it will send a search message to a node, n1, that is running a BFS instance, bfs1, with source >n

n1 will, by definition of the algorithm, respond with a search message.
Once n receives a search message from n1, it will cease running its current BFS instance and will start running the BFS instance of n1.
As a result, bfs1 will never terminate because n, which was a member of the tree of bfs1, will never respond done to its parent and n will not elect itself leader.

□

Proof. of Liveness in the Static Model
Let Umax = the process with the maximum UID
After one round, every neighbor (defined as every process within broadcast range) of Umax will have received the ID of Umax and will, by definition, set its variable maxID equal to ID of Umax and be running an instance of BFS with source = Umax
After round r, every process within r hops of Umax will be running an instance of BFS with source = Umax where r is the maximum number of hops of any node away from Umax
Now, every node in the network will be running an instance of BFS with source = Umax
Eventually, BFS will terminate and Umax will set its leader variable=true

□

This is my very basic attempt at a proof of the simulation argument Reference Execution Round 1 - leader broadcasts a message m - log (message m sent); a given child node receives m - log (message m received) Round 2 - the child node processes m and in response, broadcasts message n - log (message n sent); the leader receives n - log (message n received) Round 3 - leader processes message n and in response, broadcasts message o - log (message o sent); child node receives o - log (message o received)

Main Execution Round 1 - leader broadcasts a message m - log (message m sent); child node, CN, of leader receives message m - log (message m received) Round 2 - child node, CN, of leader processes message m and in response, rebroadcasts message m; child node of CN receives message m - log (message m received) Round 3 - child node of CN processes message m and in response rebroadcasts message m; children receive message... ... Round N - child node of CN has received messages from all of its children so it broadcasts message n (its message in response to m) - log (message n sent) Round R (not necessarily N+1) - CN receives messages from all of its children so it broadcasts message n (its message in response to m) - log (message n sent); Round Q (not necessarily R+1) - the leader has received message n from all of its children so it broadcasts message o in response - log (message n received, message o sent)

6 Annotated Bibliography

@inproceedingsBrown et al:????, author = Brown, Matthew and Gilbert, Seth and Lynch, Nancy and Newport, Calvin and Nolte, Tina and Spindel, Michael, title = The Virtual Node Layer: A Programming Abstraction for Wireless Sensor Networks, booktitle = Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, year = ????, location = Cambridge, MA, USA, pages = 1–6, numpages = 6, publisher = MIT Computer Science and Artificial Intelligence Lab, address = Cambridge, MA, USA, keywords = Wireless ad hoc networks, network architecture, virtual infrastructure, annote = This paper addresses the problems around reliable coordination in dynamic, wireless networks. The paper proposes creating a static and reliable abstract layer composed of virtual nodes on top of the unpredictable and unreliable client or physical nodes. The architecture of the emulator is well thought-out and perhaps should be emulated. One key weakness of this approach is the requirement that the locations of the virtual nodes be decided before the start of program execution. This weakness results necessarily from the attempt to model an underlying dynamic network with a perfectly static network. ,

@inproceedingsSchneider:1990, author = Schneider, Fred B., title = Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial, booktitle = ACM Computing Surveys, series = Vol. 22, No. 4, year = 1990, location = Ithaca, NY, USA, pages = 299–319, numpages = 20, publisher = ACM, keywords = Algorithms, Design, Reliability, Client-server, distributed services, state machine approach, annote = Schneider lays out the replicated state machine approach to achieving fault-tolerance in a distributed system in this tutorial paper. This paper examines two types of faults: Byzantine failures and Fail-stop failures. The paper lays out the core requirements of reliable replica coordination, agreement and order, and techniques for meeting them including a "logical clock" and a synchronized real-time clock. This paper is a recap of much of my distributed systems class. ,

```
initVariables() ;
for round 1 ... r do
    for each message m in Inbox do
        if m.maxID > maxID then
            updateMaxRoot() ;
        end
        if m.maxID == maxID then
            if m.type == choose AND receiver == myUID then
                childCount++ ;
                totalChildren++ ;
            end
            if m.type == done AND receiver == myUID then
                childCount– ;
                if childCount == 0 then
                    sendDoneMsg( ) ;
                end
            end
        end
        if m.maxID < maxID then
            msg = (type=search, sender=myUID, maxID=maxID) ) ;
            Outbox.enqueue(msg) ;
        end
    end
    if wait != 0 AND childCount == 0 then
        wait– ;
        if wait == 0 then
            sendDoneMsg( ) ;
        end
    end
    for each message m in Outbox do
        broadcast(m);
    end
    myRound++ ;
end
```

Algorithm 1: Simulation Algorithm for Static Model

```
myRound == 0 ;
maxID = myUID ;
message m = (type=search, sender=myUID, maxID=maxID) ;
```

Algorithm 2: initVariables method

```
maxID = m.maxID;
parent = m.sender;
childCount = totalChildren = 0; msg1 = (type=choose, sender=myUID,
maxID=maxID, receiver=m.sender) ;
Outbox.enqueue(msg1) ;
msg2 = (type=search, sender=myUID, maxID=maxID) ;
Outbox.enqueue(msg2) ;
wait = 3 ;
```

Algorithm 3: updateMaxRoot method

```
msg = ( type=done, sender=myUID, maxID=maxID ) ;
Outbox.enqueue( msg ) ;
```

Algorithm 4: sendDoneMsg method

```
for each action in A1 do
    if leader == true then
        message = (type=r.action, sender=myUID) ;
        broadcast ( message ) ;
        msg = ( response to message ) ;
        commLog.write (response to message) ;
    end
    for each round r do
        for each message m in Inbox do
            if m.sender == parent then
                forwardMsgToChildren( ) ;
            end
            if m.receiver == myUID then
                add m to msg ;
                childCount– ;
                if childCount == 0 then
                    Outbox.enqueue(msg) ;
                end
            end
        end
        for each message m in Outbox do
            broadcast(m) ;
        end
    end
end
```

Algorithm 5: Static Simulation Algorithm

```
msg = (response to m, receiver = parent) ;
commLog.write(response to m) ;
if totalChildren == 0 then
    Outbox.enqueue(msg) ;
else
    forwardMsg = (type = m.type, sender=myUID) ;
    Outbox.enqueue(forwardMsg) ;
end
```

Algorithm 6: forwardMsgToChildren

```
initVariables() ;
for round 1...r do
    for each message m in Inbox do
        if m.root > maxRoot then
            updateMaxRoot() ;
        end
        if m.root == maxRoot then
            if m.type == choose AND receiver == myUID then
                childCount++ ;
                totalChildren++ ;
            end
            if m.type == done AND receiver == myUID then
                childCount– ;
                if childCount == 0 then
                    sendDoneMsg() ;
                end
            end
        end
        if m.root < maxRoot then
            msg = (type=search, sender=myUID, root=(r=maxRoot.r+1,
            id=maxRoot.id) ) ;
            Outbox.enqueue(msg) ;
        end
    end
    if wait != 0 AND childCount == 0 then
        wait– ;
        if wait == 0 then
            sendDoneMsg() ;
        end
    end
    for each message m in Outbox do
        broadcast(m);
    end
    myRound++ ;
    maxRoot = (r=maxRoot.r+1, id=maxRoot.id) ;
end
```

Algorithm 7: Simulation Algorithm for the Dynamic Addition Model

```
myRound == 0 ;
maxRoot = (r=myRound, sender=myUID) ;
message m = (type=search, id=myUID, root=maxRoot) ;
```

Algorithm 8: initVariables method for Dynamic Addition Model

```
maxRoot = m.root;
parent = m.sender;
childCount = totalChildren = 0; msg1 = (type=choose, sender=myUID,
root=(r=maxRoot.r+1, id=maxRoot.id), receiver=m.id) ;
Outbox.enqueue(msg1) ;
msg2 = (type=search, sender=myUID, root=(r=maxRoot.r+1,
id=maxRoot.id) ) ;
Outbox.enqueue(msg2) ;
wait = 3 ;
```

Algorithm 9: updateMaxRoot method for Dynamic Addition Model

```
msg = ( type=done, sender=myUID, root=(r=maxRoot.r+1,
id=maxRoot.id), receiver=parent ) ;
Outbox.enqueue( msg ) ;
```

Algorithm 10: sendDoneMsg method for Dynamic Addition Model