Distributed Algorithms for Dynamic Networks

March 20, 2014

Welles Robinson

Senior Thesis

Department of Computer Science

Georgetown University

Abstract

Abstract

1 High-Level Description of Goal

Ignore for now - 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

Synchronous Broadcast Model

The synchronous broadcast model is a variant of the synchronous network model (definition taken from Lynch textbook - how to cite?). The synchronous network model is defined with respect to a directed graph $G = (V, E)$. We define n to denote $|V|$, the number of nodes in the graph. An algorithm is a set of instructions to be followed by the nodes. When we say the network executes an algorithm a, this means each node in the network is running a copy of the a. (For simplicity, we refer to a copy of the algorithm running on node u as simply node u.) In the execution, the nodes proceed in lock-step repeatedly performing the following two steps:

1. Following the algorithm, decide which messages, if any, to send to their neighbors in G.

2. Receive and process all incoming messages from their neighbors.

The combination of these two steps is called a round.

The synchronous broadcast model is different from the synchronous network model in two significant ways. First, the synchronous broadcast model is defined with respect to a connected graph $G = (V, E)$ with bi-directional edges. Second, nodes don't pass individual messages directly to their neighbors. Instead, nodes broadcast one message per round that is sent all neighbors.

Additionally, we assume that nodes have comparable unique identifiers and that nodes are in one of two high-level states, active or deactive. When a node is active, it performs the two steps, sending and receiving messages, that constitute a round. When a node is deactive, it performs neither of the two steps that constitute a round. We say a node is activated when its state changes from deactive to active. When a node is activated, it always begins in an initial state such that it has no knowledge of a global round counter. We say a node is deactivated when its state changes from active to deactive. When a node is deactivated, it resets all local variables such that if it activates, it activates in an initial state.

The only restriction that we place on the activation and deactivation of nodes in G is that the active subset of the graph G must always be connected. We emphasize that other than this minimal restriction the activation and deactivation of nodes are uncontrolled by the algorithm.

3 Problem Definition

The reliable broadcast problem provides messages to arbitrary nodes in the synchronous broadcast model to send to all active nodes in the network. This problem assumes there is an environment at each node u that communicates with u through an interface with three commands, *send*, *receive* and *acknowledge.* We refer to the environment at node u as E_u.

Using the *send* command, E_u can pass a message m to u, which u is expected to send to all other nodes in the network. Once all the other nodes have received m, u is expected to pass a "done" signal to E_u using the *acknowledge* command. We assume E_u will not pass another message to u until it has received a "done" signal from u. When a node u receives a message m from another node, it uses the *receive* command to notify E_u about m.

An algorithm A is said to solve the reliable broadcast problem if it implements the *send*, *receive* and *acknowledge* commands and satisfies the following properties (assume all messages are unique):

1. Liveness Property: If a node u running algorithm A is passed a message by E_u through its *send* command, u will eventually send a "done" signal to E_u using the *acknowledge* command.

2. Safety Property #1: Assume node u is passed a message m by E_u at round r and u sends a "done" signal in some later round r_1. Let $A(r, r_1)$ be the set of nodes that are active in every round in the interval from r to r_1. It must be the case that every node in $A(r, r_1)$, with the exception of u, passes message m to its environment through its *receive* command at some point between rounds r and r_1.

3. Safety Property #2: Assume some node u passes a message m to its environment through its *receive* command in some round r and then passes a different message m_1 in a later round r_1. It follows that no node in the network passed message m_1 to its environment before message m.

4. Safety Property #3: Assume some node u passes a message m to its environment through its *receive* command, the following two conditions must hold:

   (a) u has not previously passed m to its environment

   (b) some node previously received m from its environment through its *send* command

4 Algorithm

1. Run modified Leader Election without Network Information

   Need to write this part up formally but it is identical to the old LE

2. Reliable Message Passing with Simulataneous Activation

   Assume there exists a spanning tree covering the network (this was built in the modified LE step). A node u receives a message m from E_u using the *send* command. u broadcasts message m_1, which contains m as well as instructions that only the parent of u should forward m_1. Parent of u will broadcast m_1 and in this way, m_1 will eventually reach the leader of the tree, u_{max}. m_1 is guaranteed to reach u_{max} because u_{max} is an ancestor of every node in the tree. u_{max} will notify its environment of message m using the *receive* command and then broadcasts m. All the children of u_{max} will do the same and m will eventually be seen by every node in the network. Every node will only pass m to their environment once because a node will only process m when it is sent by its parent and a node can only have one parent. Nodes that don't have any children and receive m will send a message to their parent confirming that they have received message m. When a node receives confirmation messages from all of its children, it will send a message to its parent confirming that it (and all its children) have received message m. When u_{max} receives confirmation messages from all of its children, it sends a message to u telling it that all nodes in the network have received m (talk about optimization later). Once u receives this message, it will notify its environment using its *acknowledge* command.

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 Analysis

Lemma 5.1. *For the given network, a node will eventually set leader to true and no more than one node will have leader equal to true at the beginning of any round r.*

Proof. One node will eventually set leader to true (Lemma 5.4). No more than one node will have leader set to true at any point (Lemma 5.2). □

Definition 1. *Let u_{max} be the ID of the process with the maximum UID in the network.*

Definition 2. *Let BFS instance b_i refer to an instance of the terminating breadth-first search protocol initiated by process with ID i.*

Lemma 5.2. *For every round r, at most one node has leader = true at the beginning of round r.*

Proof. A node with ID i will only set leader = true if the BFS instance b_i terminates. A BFS tree b_i will terminate only if i equals u_{max} (Lemma 5.3). Only the process with ID u_{max}, will set leader = true.

□

Lemma 5.3. *A BFS instance b_i will only terminate if i equals u_{max}.*

Proof. Termination of a BFS instance b_j requires all other processes in the network to send a done message to b_j. Given BFS instance b_j where $j < u_{max}$, there is at least one process, the process with ID u_{max}, that will never reply done to b_j. Therefore, b_j will never terminate. □

Lemma 5.4. *One node will eventually set its variable leader to true.*

Proof. A BFS instance b_i will eventually terminate if every node in the network runs b_i. Every node in the network will eventually run $b_{u_{max}}$ so $b_{u_{max}}$ will eventually terminate and the process with ID u_{max} will set leader = true. □

Definition 3. *For a given node, define r_t as the consecutive rounds of the main execution that directly map to round t of the reference execution.*

Lemma 5.5. *For a given node and any round t in the reference execution, r_t exists.*

Proof. For the star node in the reference execution, the simulation of one round of its execution is defined as beginning when u_{max} broadcasts its message and ending when u_{max} has received messages from all of its children. For any given child node a in the reference execution, the simulation of one round of its execution is defined as between when a broadcasts its message and when a receives the leader's message from its parent. □

Definition 4. *For a given node a in the reference execution, define s_a as the node in the main execution that simulates a.*

Lemma 5.6. *By induction, the main execution correctly simulates the reference execution for any round q.*

Proof. By Induction.

Base Case $q = 0$: Before round 0, the main execution has correctly simulated the reference execution because all the communication logs are empty and therefore equivalent.

Inductive Hypothesis: Suppose the theorem holds for all values of q up to k.

Inductive Step: Let $q = k + 1$. In round k of the reference execution, the star node sends message m to every child node and records 'sent m' in its log. In the main execution, u_{max} broadcasts m and records 'sent m' in its log. In the reference execution, every child node receives m and records 'received m' in its log. In the main execution, the children of u_{max} receive m, record 'received m' in its log, and broadcast m to their children. Eventually, every node in the main execution network receives m and records 'received m' in its log.

In the reference execution, a given child node a sends message m_1 to the star node and records 'sent m_1' in its log. The star node receives m_1 and records 'received m_1' in its log. In the main execution, a eventually broadcasts m_1 to its parent and records sent m_1 in its log. a's parent receives m_1, which is broadcasted up the network until it eventually reaches u_{max}, which records 'received m_1' in its log.

After round $k + 1$, the list of the logs of the reference execution will be equivalent to that of the main execution. So the theorem holds for $q = k + 1$. By the principle of mathematical induction, the theorem holds for all rounds in the execution.

□

```
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