Theory Of Distributed Systems

James Aspnes

March 2, 2024

Contents

1 Model **2**
1.1 Basic message-passing model 2
1.1.1 Formal Details 2
1.2 Asynchronous systems 3
1.3 Synchronous systems 3
1.4 Drawing message-passing executions 3

2 Broadcast and convergecast **3**
2.1 Flooding 3
2.1.1 Basic algorithm 3
2.1.2 Adding parent pointers 5
2.1.3 Identifying children 5
2.1.4 Convergecast 6
2.1.5 Flooding and convergecast together 6

3 Distributed breadth-first search **6**
3.1 Using explicit distances 6
3.2 Using layering 7
3.3 Using local synchronization 8

4 Leader election **9**
4.1 Symmetry 9
4.2 Leader election in rings 10
4.2.1 The Le Lann-Chang-Roberts algorithm 10
4.2.2 The Hirschberg-Sinclair algorithm 10
4.2.3 Peterson's algorithm for the unidirectional ring . . . 11

1 Model

1.1 Basic message-passing model

We have a collection of n **processes** $p_1, \ldots, p_n$, each of which has a **state** consisting of a state from state set Q_i. We think of these processes as nodes in a directed **communication graph** or **network**. The edges in this graph are a collection of point-to-point **channels** or **buffers** b_{ij}, one for each pair of adjacent processes i and j, representing messages that have been sent but that have not yet been delivered.

A **configuration** of the system consists of a vector of states, one for each process and channel. The configuration of the system is updated by an **event**, where

1. zero or more messages in channels b_{ij} are delivered to process p_j, removing them from b_{ij};
2. p_j updates its state in response;
3. zero or more messages are added by p_j to outgoing channels b_{ji}.

An **execution segment** is a sequence of alternating configurations and events $C_0, \phi_1, C_1, \phi_2, \ldots$, where each triple $C_i \phi_{i+1} C_{i+1}$ is consistent with the transition rules for the event ϕ_{i+1} and the last element of the sequence is a configuration. If the first configuration C_0 is an **initial configuration** of the system, we have an **execution**. A **schedule** is an execution with the configurations removed.

1.1.1 Formal Details

Let P be the set of processes, Q the set of process states, and M the set of possible messages.

Each process p_i has a state $\mathsf{state}_i \in Q$. Each channel b_{ij} has a state $\mathsf{buffer}_{ij} \in \mathcal{P}(M)$. We assume each process has a **transition function** $\delta : Q \times \mathcal{P}(M) \rightarrow Q \times \mathcal{P}(P \times M)$ that maps tuples consisting of a state and a set of incoming messages a new state and a set of recipients and messages to be sent. A delivery event $\mathsf{del}(i, A)$ where $A = \{(j_k, m_k)\}$ removes each message m_k from b_{ji}, updates state_i according to $\delta(\mathsf{state}_i, A)$ to the appropriate channels. A computation event $\mathsf{comp}(i)$ does the same thing, except that it applies $\delta(\mathsf{state}_j, \emptyset)$.

1.2 Asynchronous systems

In an **asynchronous** model, only minimal restrictions are placed on when messages are delivered and when local computation occurs. A schedule is **admissible** if

1. there are infinitely many computation steps for each process,
2. every message is eventually delivered

These are **fairness** conditions. Condition (a) assumes that processes do not explicitly terminate.

1.3 Synchronous systems

A **synchronous message-passing** system is exactly like an asynchronous system, except we insist that the schedule consists of alternating phases where

1. every process executes a computation step,
2. all messages are delivered while none are sent

The combination of a computation phase and a delivery phase is called a **round**.

1.4 Drawing message-passing executions

2 Broadcast and convergecast

2.1 Flooding

2.1.1 Basic algorithm

Theorem 2.1. *Every process receives M after at most D time and at most $|E|$ messages, where D is the diameter of the network and E is the set of (directed) edges in the network*

We can optimize the algorithm slightly by not sending M back to the node it came from; this will slightly reduce the message complexity in many cases but makes the proof a sentence or two longer.

initially do
    if *tpid* = *root* **then**
        seen-message ← **true**;
        send M to all neighbors;
    end
    else
        seen-message ← **false**;
    end
upon receiving M **do**
    if *seen-message* = ***false*** **then**
        seen-message ← **true**;
        send M to all neighbors;
    end

Algorithm 1: Basic flooding algorithm

initially do
    if *tpid* = *root* **then**
        parent ← root;
        send M to all neighbors;
    end
    else
        parent ← $\perp$;
    end
upon receiving M *from* p **do**
    if *parent* = $\perp$ **then**
        parent ← p;
        send M to all neighbors;
    end

Algorithm 2: Flooding with parent pointers

2.1.2 Adding parent pointers

Lemma 2.2. *At any time during the execution of Algorithm **??**, the following invariant holds:*

1. *If $u.\mathsf{parent} \neq \perp$ then $u.\mathsf{parent}.\mathsf{parent} \neq \perp$ and following parent pointers gives a path from u to root*
2. *If there is a message M in transit from u to v, then $u.\mathsf{parent} \neq \perp$*

Though we get a spanning tree at the end, we may not get a very good spanning tree.

2.1.3 Identifying children

```
initially do
    nonChildren = ∅;
    if tpid = root then
        parent ← root;
        children ← {root};
        send M to all neighbors;
    end
    else
        parent ← ⊥;
        children ← ∅;
    end
upon receiving M from p do
    if parent = ⊥ then
        parent ← p;
        send ack to p;
        send M to all neighbors;
    end
    else
        send nack to p;
    end
upon receiving ack from p do
    children ← children ∪ {p}
upon receiving nack do
    nonChildren = nonChildren ∪ {p}
```

Algorithm 3: Flooding tracking children

Properties

1. (safety) If $p_j \in p_i.\,\mathsf{children}$, then $p_j.\,\mathsf{parent} = p_i$
2. (safety) If $p_j \in p_i.\,\mathsf{nonChildren}$, then $p_j.\,\mathsf{parent} \notin \{p_i, \bot\}$
3. (liveness) Eventually, every neighbor of p_i appears in $p_i.\,\mathsf{children} \cup p_i.\,\mathsf{nonChildren}$

2.1.4 Convergecast

A **convergecast** is the inverse of broadcast: data is collected from outlying nodes to the root.

```
initially do
    if I am a leaf then
        send input to parent;
    end
upon receiving M from c do
    append (c, M) to buffer;
    if buffer contains messages from all my children then
        v ← f(buffer, input);
        if pid = root then
            return v
        else
            send v to parent;
        end
    end
```

Running time is bounded by the depth of the tree: we can prove by induction that any node at height h (height is length of the longest path from this node to some leaf) sends a message by time h at the latest. Message complexity is exactly $n - 1$, where n is the number of nodes;

2.1.5 Flooding and convergecast together

3 Distributed breadth-first search

3.1 Using explicit distances

The claim is that after at most $O(VE)$ messages and $O(D)$ time, all distance values are equal to the length of the shortest path from the initiator.

initially do
  if *pid* = *initiator* **then**
    distance ← 0;
    send distance to all neighbors
  else
    distance ← ∞;
  end
upon receiving d *from* p **do**
  if $d + 1 <$ *distance* **then**
    distance ← $d + 1$;
    parent ← p;
    send distance to all neighbors;
  end

Algorithm 4: AsynchBFS algorithm

Lemma 3.1. *The variable* $\mathsf{distance}_p$ *is always the length of some path from initiator to* p*, and any message sent by* p *is also the length of some path from* initiator *to* p

Proof. Induction □

A liveness property: $\mathsf{distance}_p = d(\mathsf{initiator}, p)$ no later than time $d(\mathsf{initiator}, p)$

3.2 Using layering

Here we run a sequence of up to $|V|$ instances of the simple algorithm with a distance bound on each: instead of sending out just 0, the initiator sends out $(0, \mathsf{bound})$ where bound is initially 1 and increases at each phase. A process only sends out its improved distance if it is less than bound.

Each phase of the algorithm constructs a partial BFS tree that contains only those nodes within distance bound of the root.

With some effort, it is possible to prove that in a bidirectional network that this approach guarantees that each edge is only probed once with a new distance, and the bound-update and acknowledgment messages contribute at most $|V|$ messages per phase. So we get $O(E + VD)$ total messages. But the time complexity is bad: $O(D^2)$ in the worst case.

TODO: figure out

3.3 Using local synchronization

The reason the layering algorithm takes so long is that at each phase we have to phone all the way back up the tree to the initiator to get permission to go on to the next phase.

We'll require each node at distance d to delay sending out a recruiting message until it has confirmed that none of its neighbors will be sending it a smaller distance. We do this by having two classes of messages:

- $\mathsf{exactly}(d)$: "I know that my distance is d"
- $\mathsf{more\text{-}than}(d)$: "I know that my distance is $> d$"

The rules for sending these messages for a non-initiator are:

1. I can send $\mathsf{exactly}(d)$ as soon as I have received $\mathsf{exactly}(d-1)$ from at least one neighbor and $\mathsf{more\text{-}than}(d-2)$ from all neighbors.

2. I can send $\mathsf{more\text{-}than}(d)$ if $d = 0$ or as soon as I have received $\mathsf{more\text{-}than}(d-1)$ from all neighbors.

The initiator sends $\mathsf{exactly}(0)$ to all neighbors at the start of the protocol.

Proposition 3.2. *Under the assumption that local computation takes zero time and message delivery takes at most 1 time unit, we'll show that if $d(\mathsf{initiator}, p) = d$:*

1. *p sends $\mathsf{more\text{-}than}(d')$ for any $d' < d$ by time d'*

2. *p sends $\mathsf{exactly}(d)$ by time d*

3. *p never sends $\mathsf{more\text{-}than}(d')$ for any $d' \geq d$*

4. *p never sends $\mathsf{exactly}(d')$ for any $d' \neq d$*

Proof. For (3) and (4). The base case is that the initiator never sends any $\mathsf{more\text{-}than}$ messages at all, and any non-initiator never sends $\mathsf{exactly}(0)$. For larger d', observe that if a non-initiator p sends $\mathsf{more\text{-}than}(d')$ for $d' \geq d$, it must first have received $\mathsf{more\text{-}than}(d'-1)$ from all neighbors, including some neighbor p' at distance $d-1$. But the induction hypothesis tells us that p' can't send $\mathsf{more\text{-}than}(d'-1)$ for $d'-1 \geq d-1$. Similarly, to send $\mathsf{exactly}(d')$ for $d' > d$, p must first receive $\mathsf{more\text{-}than}(d'-2)$ from this closer neighbor p', but then $d'-2 > d-2 \geq d-1$ so $\mathsf{more\text{-}than}(d'-2)$ is not sent by p'.

For (1) and (2). The base case is that the initiator sends $\mathsf{exactly}(0)$ to all nodes at time 0, giving (1), and there is no $\mathsf{more\text{-}than}(d')$ with $d' < 0$ for it to send, giving (2).

Message complexity: A node at distance d sends $\mathsf{more\text{-}than}(d')$ for all $0 < d' < d$ and $\mathsf{exactly}(d)$ and no other messages. So we have message complexity bounded by $|E| \cdot D$.

Time complexity: D □

4 Leader election

4.1 Symmetry

A system exhibits **symmetry** if we can permute the nodes without changing the behaviour of the system. More formally, we can define a symmetry as an **equivalence relation** on processes, where we have the additional properties that all processes in the same equivalence class run the same code; and whenever p is equivalent to p', each neighbor q of p is equivalent to a corresponding neighbor q' of p'.

Symmetries are convenient for proving impossibility results, as observed by Angluin. The underlying theme is that without some mechanism for **symmetry breaking**, a message-passing system escape from a symmetric initial configuration. The following lemma holds for **deterministic** systems, basically those in which processes can't flip coins:

Lemma 4.1. *A symmetric deterministic message-passing system that starts in an initial configuration in which equivalent processes have the same state has a synchronous execution in which equivalent processes continue to have the same state.*

Proof. Easy induction on rounds: if in some round p and p' are equivalent and have the same state, and all their neighbors are equivalent and have the same state, then p and p0 receive the same messages from their neighbors and can proceed to the same state (including outgoing messages) in the next round. □

An immediate corollary is that you can't do leader election in an anonymous system with a symmetry that puts each node in a non-trivial equivalence class, because as soon as I stick my hand up to declare I'm the leader, so do all my equivalence-class buddies.

A more direct way to break symmetry is to assume that all processes have identities; now processes can break symmetry by just declaring that the one with the smaller or larger identity wins.

4.2 Leader election in rings

4.2.1 The Le Lann-Chang-Roberts algorithm

This algorithm works in a **unidirectional ring**, where messages can only travel clockwise. Protocol works because whichever process p_{max} holds the

```
initially do
    leader ← 0;
    maxId ← id_i;
    send id_i to clockwise neighbor;
upon receiving j do
    if j = id_i then
        leader ← 1;
    end
    if j > maxId then
        maxId ← j;
        send j to clockwise neighbor;
    end
```

Algorithm 5: LCR leader election

maximum ID id_{max} will

1. refuse to forward any smaller ID
2. eventually have its value forwarded through all of the other processes, causing it to eventually set its leader bit to 1.

4.2.2 The Hirschberg-Sinclair algorithm

Nancy's book is better.

This algorithm improves on Le Lann-Chang-Roberts by reducing the message complexity. The idea is that instead of having each process send a message all the way around a ring, each process will first probe locally to see if it has the largest ID within a short distance. If it wins among its immediate neighbors, it doubles the size of the neighborhood it checks, and continues as long as it has a winning ID. This means that most nodes drop out quickly, giving a total message complexity of $O(nlogn)$. The running time is a constant factor worse than LCR, but still $O(n)$.

4.2.3 Peterson's algorithm for the unidirectional ring

Assume an asynchronous unidirectional ring. It gets $O(n \log n)$ message complexity.

```
Function candidate():
    phase ← 0;
    current ← pid;
    while true do
        send probe(phase, current);
        wait for probe(phase, x);
        id_2 ← x;
        send probe(phase +1/2, id_2);
        wait for probe(phase +1/2, x);
        id_3 ← x;
        if id_2 = current then
            I am the leader;
            return;
        else if id_2 > current ∧ id_2 > id_3 then
            current ← id_2;
            phase ← phase +1;
        else
            switch to relay();
        end
    end
Function relay():
    upon receiving probe(p, i) do
        send probe(p, i);
```

Algorithm 6: Peterson's leader-election algorithm

5 Problems

Link	Problems	
	proof of the complexity	false