Theory Of Distributed Systems

James Aspnes

March 17, 2024

Contents

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** $\delta : Q \times \mathcal{P}(M) \to Q \times \mathcal{P}(P \times M)$ 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)*

```
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

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.

2.1.2 Adding parent pointers

```
initially do
    if tpid = root then
        parent ← root;
        send M to all neighbors;
    end
    else
        parent ← ⊥;
    end
upon receiving M from p do
    if parent = ⊥ then
        parent ← p;
        send M to all neighbors;
    end
```

Algorithm 2: Flooding with parent pointers

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

1. *If* u. parent $\neq \perp$ *then* u. parent . 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. 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 $= \emptyset$;
  if *tpid* $=$ *root* **then**
    parent $\leftarrow$ root;
    children $\leftarrow \{$root$\}$;
    send M to all neighbors;
  end
  else
    parent $\leftarrow \perp$;
    children $\leftarrow \emptyset$;
  end
upon receiving M *from* p **do**
  if *parent* $= \perp$ **then**
    parent $\leftarrow p$;
    send ack to p;
    send M to all neighbors;
  end
  else
    send nack to p;
  end
upon receiving *ack from* p **do**
  children $\leftarrow$ children $\cup \{p\}$
upon receiving *nack* **do**
  nonChildren $=$ nonChildren $\cup \{p\}$

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 $p_j.\,\mathsf{parent} \notin \{p_i, \perp\}$
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.

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 $\mathsf{initiator}$ to p*

```
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

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:

- exactly(d): "I know that my distance is d"
- more-than(d): "I know that my distance is $> d$"

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

1. I can send exactly(d) as soon as I have received exactly$(d-1)$ from at least one neighbor and more-than$(d-2)$ from all neighbors.
2. I can send more-than(d) if $d = 0$ or as soon as I have received more-than$(d-1)$ from all neighbors.

The initiator sends 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 $d(\textsf{initiator}, p) = d$:*

1. *p sends more-than(d') for any $d' < d$ by time d'*
2. *p sends exactly(d) by time d*
3. *p never sends more-than(d') for any $d' \geq d$*
4. *p never sends exactly(d') for any $d' \neq d$*

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

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

Message complexity: A node at distance d sends more-than(d') for all $0 < d' < d$ and 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 maximum ID id_{max} will

1. refuse to forward any smaller ID

```
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

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.

Let's start by describing a version with two-way communication. Start with n candidate leaders. In each of at most lg n asynchronous phases, each candidate probes its nearest surviving neighbors to the left and right; if its ID is larger than the IDs of both neighbors, it survives to the next phase. Non-candidates act as relays passing messages between candidates. As in Hirschberg and Sinclair, the probing operations in each phase take $O(n)$

messages, and at least half of the candidates drop out in each phase. The last surviving candidate wins when it finds that it's its own surviving neighbor.

To make this work in a 1-way ring, we have to simulate 2-way communication by moving the candidates clockwise around the ring to catch up with their unsendable counterclockwise messages. In each phase k, a candicate effectively moves two positions to the right, allowing it to look at the IDs of three phase-k candidates before deciding to continue in phase $k+1$ or not.

```
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

4.3 Leader election in general networks

4.4 Lower bounds

4.4.1 Lower bound on asynchronous message complexity

Here we describe a lower bound for uniform asynchronous leader election in the ring. We assume the system is deterministic.

The proof constructs a bad execution in which n processes sends lots of messages recursively, by first constructing two bad $(n/2)$-process executions and pasting them together in a way that generates many extra messages. If the pasting step produces $\Theta(n)$ additional messages, we get a recurrence $T(n) \geq 2T(n/2) + \Theta(n)$ for the total message traffic, which has solution $T(n) = \Omega(n \log n)$.

We'll assume that all processes are trying to learn the identity of the process with the smallest ID. This is a slightly stronger problem that mere leader election, but it can be solved with at most an additional $2n$ messages once we actually elect a leader. So if we get a lower bound of $f(n)$ messages on this problem, we immediately get a lower bound of $f(n) - 2n$ on leader election.

To construct the bad execution, we consider "open executions" on rings of size n where no message is delivered across some edge (these will be partial executions, because otherwise the guarantee of eventual delivery kicks in). Because no message is delivered across this edge, the processes can't tell if there is really a single edge there or some enormous unexplored fragment of a much larger ring.

Our induction hypothesis will show that a line of $n/2$ processes can be made to send at least $T(n/2)$ messages in an open execution (before seeing any messages across the open edge); we'll then show that a linear number of additional messages can be generated by pasting two such executions together end-to-end, while still getting an open execution with n processes.

For large n, suppose that we have two open executions on $n/2$ processes that each send at least $T(n/2)$ messages. Break the open edges in both executions and replace them with new edges to create a ring of size n; similarly paste the schedule σ_1 and σ_2 of the two executions together to get a combined schedule $\sigma_1\sigma_2$ with at least $2T(n/2)$ messages. Note that in the combined schedule no messages are passed between the two sides, so the processes continue to behave as they did in their separate executions.

Let e and e' be the edges we used to past together the two rings. Extend $\sigma_1\sigma_2$ by the longest possible suffix σ_3 in which no messages are delivered across e and e'. Since σ_3 is as long as possible, after $\sigma_1\sigma_2\sigma_3$, there are no

messages waiting to be delivered across any edge except e and e' and all processes are **quiescent** - they will send no additional messages until they receive one.

We now consider some suffix σ_4 that causes the protocol to finish when appended to $\sigma_1\sigma_2\sigma_3$. While executing σ_4, construct two sets of processes S and S' by the following rules:

1. if a process is not yet in S or S' and receives a message delivered across e, put it in S; similarly if it receives a message delivered across e', put it in S'

2. If a process is not yet in S or S' and receives a message that was sent by a process in S, put it in S; similarly for S'

5 Causal ordering and logical clocks

5.1 Causal ordering

Happens-before relation $\overset{S}{\Rightarrow}$ on a schedule S consists of

1. all pairs (e, e') where e precedes e' in S and e and e' are events of the same process

2. all pairs (e, e') where e is a send event and e' is the receive event for the same message

3. all pairs (e, e') where there exists a third event e'' s.t. $e \overset{S}{\Rightarrow} e''$ and $e'' \overset{S}{\Rightarrow} e'$

A **causal shuffle** S' of S is a permutation of S that is consistent with the happens-before relation on S

Lemma 5.1. *Let S' be a permutation of the events in S. TFAE:*

1. *S' is a causal shuffle of S*

2. *S' is the schedule of an execution fragment of a message-passing system with $S|p = S'|p$ for all p*

5.2 Logical clocks

5.2.1 Lamport clock

Every process maintains a local variable . When a process sends a message or executes an internal step, it sets $\leftarrow + 1$. When a process receives a message with timestamp t, it sets $\leftarrow \max(\ , t) + 1$

6 Problems

Link	Problems	
	proof of the complexity	false