Lecture Notes for CSCI 379: Distributed Computing

Set 1-Introduction, Motivating Example, and Models

Professor Talmage

January 23, 2023

---

1 Introduction

Exercise: Student Introductions (discuss with a neighbor or two, preferably someone you do not know well):

- Name
- A hobby, something you do for fun
- Something you hope to learn about in this class–suggest a topic

Administrivia (Syllabus Review)

- Schedule (MW lecture/F activities/readings sometimes)
- Office Hours
- Disabilities
- Assignment Types:
  - Participation: Attendance, in-class exercises, etc. I want half-baked attempts at in-class exercises. We will discuss why they do not work, which is as important as understanding why correct solutions work.
  - Quizzes(M, retakes): High-level questions to check understanding from class.
  - Homework(W, alternate or less): Practice to ensure that you can use the tools we learn.
  - Projects(F, about 3 total): Putting what you have learned into the real world.
  - Exams(2): Concentrate and evaluate all knowledge.

Goals: By the end of the semester, you should be able to

- reason in a concurrent setting
- recognize and solve classic distributed computing problems
- recognize and apply limitations of distributed computing
- discuss applications of distributed algorithms
- read a research paper at a high-level

2 What is Distributed Computing?

At heart, any computation done by multiple computing entities attempting to work together.

> **Aside: Parallel vs. Distributed Computing:** By this definition, parallel computing is distributed. Similarly, a simple definition of parallel computing would include distributed computing. Why do we discuss them separately? Primarily because the two fields focus on different questions in the same domain.
>
> - Parallel computing focuses on breaking problems into as many parts that can be run at the same time as possible, to reduce overall runtime. Thus, each computing entity is (ideally) working on something different at any given time. The primary measure is computation time.
> - Distributed computing focuses on how to make the different computing entities work together, in the face of the difficulties of real-world systems such as communication delays and variability, node failures, and so on, to solve a single problem. This leaves all computing entities working on the same problem. Primary measures are communication time and complexity, and even whether coordinated action is possible.
> - Others may define this distinction differently, or deny a distinction exists. It is a spectrum, so there are no clear boundaries.

Why do we care about distributed computing?

- Computing problems have been scaling faster than Moore's Law (transistor density doubling every two years). Distributed computing can combine resources to solve larger problems (this is more on the parallel side of things).
- Localized computing: local access is faster than remote, so having a synchronized local copy of data is more efficient for the user.
- Distributed data: replication can tolerate failures, lower requirements at any one site.
- Realistic: Most computation is collaborative in some way. Coordinating that work is critical.

Why is distributed computing hard? (Is it?)

- Concurrency is hard!
- Much harder to reason when multiple things are happening at the same time.
  - "Time" and "happens before" are no longer trivial.
- Different computers work at significantly different rates, so we cannot assume all participants are executing the same part of the algorithm.
- Communication takes time, and is subject to variability and loss.
- Computers may fail, and we would like to still be able to complete our task.

3 Motivating Example: Leader Election

Suppose you have deployed a collection of autonomous vehicles into a disaster zone. They have made a discovery they need to report (such as a survivor you can rescue), but do not have connectivity to report back from their current position. (Hence autonomous vehicles, not remote-controlled.) The drones need to send one member of the group back base to report. Each vehicle has only a partial view of its own position, so it is not obvious which drone it would be "best" to send back. Thus, they just need to decide on a representative, any representative, as sending multiple back would hamper the overall recovery effort. (In general, it might make sense to send multiple back for resilience, but we will stick to one for now.)

Exercise: Try to give an algorithm by which the drones can collaborate to choose one member to send back.

- What must be true of the drones to enable the problem to be solved?
- What assumptions are you making on timing and communication?

Now, in the real-world, the drones may not only be unable to communicate with you, they may be unable to communicate with all the other drones. Assume, for the moment, that the drones are programmed such that each always stays in range of at least one other, in such a way that they are all connected, at least indirectly.

Exercise: Represent the drones as nodes of a graph, with edges between those which can communicate directly. Discuss how they might be able to choose one representative to send back to base.

- Is this problem easier or harder than the last version?
- What are some graph types/shapes that might make the problem easy?
- Are you assuming that drones know the entire graph topology? Is that realistic? How would you maintain that knowledge?

These are examples of the *Leader Election* problem, which is one of the classic problems in distributed computing. We will return to consider some restricted versions of this problem next week.

4 Models of Distributed Computing

We call participants in a distributed computation *processes*, as we typically require that they be able to execute code. In practice, a participant may be a collection of processes running together on a machine, splitting the handling of different aspects of the computation between them. We typically use n to represent the number of processes. (Aside, grammar: process (singular), processes (plural), process' (sing. possessive), processes' (pl. possessive).)

There are at least three primary dimensions in which distributed computing models vary:

	⇐ Easier		Harder ⇒
Communication	Shared Memory	Beeps, etc.	Message Passing
Synchrony	Synchronous	Partially Synchronous	Asynchronous
Failures	None	Crashes	Byzantine (arbitrary)
	⇐ Less Realistic		More Realistic ⇒

4.1 Message Passing

Typically the most realistic, if processes are distributed geographically and not all directly linked. We will spend most of our time here.

- Processes form an undirected graph. Each process can distinguish and communicate with its neighbors in the graph, but does not know any more about the graph. This includes not knowing neighbors' IDs, but using some local numbering scheme to distinguish them.
  - Important graphs to remember: Complete, Ring, Star, Tree
- Each process i has incoming and outgoing *message buffers*, denoted $inbuf_j$ and $outbuf_j$, for each neighbor j (local numbering!).
- An *execution* is a sequence of *configurations* and *events*. A configuration specifies the state of every process, including its buffers.
  - An execution must start from an *initial configuration*. Otherwise, it is an *execution fragment*.
- Each event is one of:
  - *deliver*: Move a message from an $outbuf$ to an $inbuf$ (sending to receiving processes)
  - *computation*: One process' state changes, consumes all messages in its $inbuf$s, may place messages in its $outbuf$s.

  The sequence of events in an execution is called its *schedule*.
- An execution is *admissible* if every process takes an infinite number of computation steps and all sent messages are delivered exactly once.
  - Algorithms can *terminate*, processes just take dummy steps thereafter. The requirement of infinite executions makes formal arguments easier, since we do not have to worry about which processes are still running at any given time.
  - We will add more requirements for executions to be admissible when we consider synchrony and timing properties.
- In a *synchronous* system, an execution is partitioned into *rounds*. A round is a deliver for each message in an $outbuf$, then a computation step for each process.
- To write pseudocode, we will often use an event-driven style. An event handler is considered to be a single computation step, and must not do more work than is allowed in such a step (a constant amount).

Complexity Measures

- Time Complexity:
  - Synchronous systems: Number of rounds before termination.
  - Asynchronous systems: A *timed execution* associates a time with each event. Times are non-negative real numbers, starting with 0, and are non-decreasing. They must also increase without bound in infinite sequences of events, which implies that there can only be a finite number of events before a given finite time.
  - Time complexity is then the time of the last process' termination.
    - Since there is no bound on communication delay, can this be infinite?

* Scale time by the maximum message delay: In a given execution, there is some maximum delay (time of computation event when message is received minus time of computation event when that message was sent) which any message takes. Divide all times in the execution by this value. Execution is equivalent, only timings changed, and max message delay is now 1. Use time of termination in this execution.

  Exercise: Why is the execution equivalent? Why is it impossible for changing timings to change the execution?

- Message Complexity:
  - What is the maximum, across all possible executions of the algorithm, of the number of messages sent? (Typically in terms of number of processes, input size, etc.)
  - What is the maximum size of a message the algorithm may send? (Fairly typical to standardize message size, so this may be expressed as "How large are the messages the algorithm sends?")
  - What is the maximum total amount of data the algorithm sends between processes? (I am unsure off the top of my head if I have ever seen this used, and it can be bounded from the others, but it is something that would be reasonable to ask.)

4.2 Exercises

Broadcast

Exercise: Suppose that one process, p_r, wants to send message M to all processes. Assume that the graph is a rooted tree (Finding a MST of a general graph in a distributed way is an interesting problem in its own right!), with parent-child directionality and p_r is the root. Processes do not know the entire graph, only their own connections.
Give pseudocode for each process to ensure that M reaches all processes in a synchronous, fault-free, message-passing system. What are your algorithm's time and message (count) complexities?

Idea: p_r initially sends M to all children, which forward to their children, etc.

```
   Code for p_r:
1: Initially:
2:     Send M to all children
3:     Terminate

   Code for p_i, i ≠ r:
4: upon receiving message x from parent:
5:     Send x to all children
6:     Terminate
```

Correctness: Inductive proof that after time t, all processes at depth $\leq t$ in the tree have received M.

Complexity: Time complexity is h, the height of the tree (which is at most D, the graph's diameter). Message complexity is $n - 1$, since each process will receive M exactly once. (Alternately, we send over each of the $n - 1$ edges in a tree exactly once.)

Exercise: What if p_r is not the root of the tree? What if the graph was not a tree at all?

4.2.1 Convergecast

Every process p_i starts with numerical data value M_i. Find the largest M_i and send it to p_r. In other words, the problem is solved when p_r (confidently) has the largest value in the system.

> **Exercise:** Give pseudocode to solve Convergecast, with an argument for its correctness and complexity analysis in a synchronous, fault-free, message-passing system.
> How much would your solution change if we wanted the smallest value? The median value? What types of conditions are easy to converge? What are difficult?

Idea: Do the reverse of broadcast: when you have received messages from all of your children, send the maximum value in your subtree to your parent.

Code for p_l, where p_l is a leaf:
Initially:
    Send M_i to parent
    Terminate

Code for p_i, which is not a leaf:
Initially:
    Put M_i in an array $Vals$ of $Deg(p_i)$ values
Upon receiving message x_j from a child:
    Store x_j in $Vals$
    If $Vals$ is full:
        $x = \max(Vals)$
        Send x to parent if parent exists
        Terminate

Analysis is similar to that for broadcast.

4.2.2 Building a Spanning Tree

> **Exercise:** Given an arbitrary network, each process should assign labels (*parent*, *child*, or *cross*) to each incident edge s.t. the parent-child relationships form a spanning tree. Assume node p_r knows it will be root, other processes know they are not the root, but do not know where they are relative to the root. Give pseudocode to solve this problem in a synchronous, fault-free, message-passing system.

Idea: *Flooding*. The root sends a message x to all its neighbors. When another node **first** receives x, it forwards it to all its other neighbors.

- Note that a node could receive x "simultaneously" (in the same round) from two different neighbors. Choose one arbitrarily to be "first".
- Flooding is a general technique for broadcast without a given spanning tree. (What's its complexity?)

To actually construct the spanning tree:

- Each process marks the edge over which it first received x as *parent* and sends a $\langle parent \rangle$ message back across that edge.
- When a process receives a $\langle parent \rangle$ message from a neighbor, it labels the edge to that neighbor *child*.
- When a process receives x after it has set its parent, it responds with an $\langle already \rangle$ message and marks that edge as *cross*

Algorithm 1 Spanning tree algorithm, code for each process p_i

Initially, for every node: $parent = \perp, children = \emptyset, cross = \emptyset$
1: **if** $p_i == p_r$ and $parent == \perp$ **then**
2: Send x to all neighbors
3: $parent = p_i$
4: **end if**
Upon receiving x from neighbor p_j:
5: **if** $parent == \perp$ **then**
6: $parent = p_j$
7: Send $\langle parent \rangle$ to p_j
8: Send x to all neighbors except p_j
9: **else**
10: Send $\langle already \rangle$ to p_j
11: Add p_j to $cross$
12: **end if**
Upon receiving $\langle parent \rangle$ from p_j:
13: Add p_j to $children$
14: If $parent \cup children \cup cross$ contains all neighbors, terminate.
Upon receiving $\langle already \rangle$ from p_j:
15: Add p_j to $other$
16: If $parent \cup children \cup cross$ contains all neighbors, terminate.

Correctness:

- First, we argue that the result spans the graph: Since the graph is connected, there is a path from p_r to each other node p, and the forwarding mechanism will send x along at least one such path.
- Next, we argue that the result is a tree (acyclic): Suppose in contradiction that there is a cycle of parent pointers $p_1, p_2, \ldots, p_k$. Since p_2 is p_1's parent, p_2 first received x in an earlier round than p_1 did. We can proceed with the same argument all around the cycle, which is a contradiction as we then conclude that the round in which p_2 first received x is smaller than that when p_3 did, contradicting the fact that p_3 is p_2's parent. Thus, there cannot be a cycle in the output, so it is a tree.

Complexity

- x is sent at most twice on each edge, so we have message complexity at most $2m$, where m is the number of edges. We can improve this by noting that x is never sent twice across an edge by which a node first received x, so we have message complexity $<= 2m - (n - 1)$.
  - **Exercise:** Can you build a graph that sends x twice on **every** non-parent edge?
- Time complexity:

  Exercise: What kind of tree does this algorithm build?

  - BFS Tree: Can prove inductively that by time t, all nodes at distance $<= t$ are in the tree.
    - **Exercise:** What's the maximum time?
  - The maximum time is the maximum distance from p_r to anther node, which could be the maximum distance between any two nodes in the graph, known as the graph's *diameter*.

4.2.3 DFS Tree

Given a communication network and a designated node p_r in the network, each process should label neighbors as *parent* and *child* s.t. the labels for a DFS tree rooted at p_r.

Exercise: Write pseudocode for every process to build such a tree.

- What events do you need to handle?
- What state does each node need to maintain?

Idea: Only one token active at a time, so equivalent to a sequential algorithm.

Exercise: Write pseudocode to build a DFS tree *without* a specified root.

Idea: Every process tries to build a tree with itself as root. When it meets another tree, the tree whose root has a higher identifier takes over the other.