1 Introduction

This lecture covered administrative details for the course, discussed the difference between parallel and distributed systems, and introduced two models of distributed computation.

2 Course administration

Grades for the course will be determined by:

- 50% final project
- 40% lecture notes
- 10% class participation

The final project will be an in-class presentation of some aspect of theory, though some programming may also be allowed, depending on the topic. Lecture notes are to be produced in \LaTeX, and students will take turns, one per lecture. Notes should be produced quickly, and they will be posted to the Lecture Notes web site.

3 Distributed systems and concurrent algorithms

Distributed computing and parallel computing study different kinds of systems, but they both study concurrent algorithms that run on collections of processors. In parallel computing, processors are tightly coupled, which usually means they move in lockstep, communication between processors is fast, and neither processors nor communication channels suffer from errors. By contrast, in a distributed system, processors are loosely coupled, which often means that different processors take steps at different times, messages placed in communication channels may take a while to reach their destinations, and processors and message channels may fail completely.

A concurrent algorithm is an algorithm that executes on multiple processors that communicate with one another. The notion of “communication” is often trivial in parallel systems, whereas it can be extremely important in distributed systems. Many problems we will encounter are trivial in models where communication speed is fast, and where communication is guaranteed to function correctly at all times.
We can model a problem solved by a concurrent algorithm as inputs to each of the processors, and outputs from each of the processors. One classical problem is consensus, in which processors may start with different values, but must all end with the same value. This model of a problem implies that there are independent inputs at multiple locations, and several processors act simultaneously but independently.

4 Synchrony and asynchrony

In parallel computing, processors act in lockstep. In distributed computing, we consider the entire spectrum from total synchrony to total asynchrony.

**Total Processor Synchrony** Processors act in lockstep. Each processor receives all its messages at the beginning of the round, computes, and sends out all messages at the end of the round.

**Total Processor Asynchrony** There is no restriction on how much time one processor waits with respect to another. Processors can start at different times, and run at different speeds. It is possible for processors to fail.

In addition to a spectrum of processor behavior from synchrony to asynchrony, the behavior of communication channels can be modeled in similar ways.

**Synchronous communication** A simple model. All processors take one step per round, and if a processor sends a message in one round, it is received in the next round.

**Asynchronous communication** Once a message is sent, assuming no failures, then the message is guaranteed to arrive, but with no guarantee on when. Moreover, communication failures may be possible, i.e., channels can drop messages.

Given those notions as extreme cases, we can model the level of failure, and the amount of asynchrony, we are willing to accept. Once we have that model, we can devise algorithms to solve problems in that model, or, alternatively, prove a particular problem cannot be solved there.

Distributed computing is hard to reason about because of the lack of knowledge each processor has of the total system. Often there are an unknown number of participating processors, and sometimes a processor does not even know who its neighbors are. So we must be very clear about what our model assumptions are. In sequential computing, there is a generalized abstraction of computing—the Turing Machine—that can be applied to analyze almost all theoretical problems. Unfortunately, in distributed computing, no universally-accepted abstraction exists, because there are so many different “reasonable” models of distributed systems.
5 Two basic models of distributed computation

Two basic models of distributed computation are the shared memory model and the message passing (network) model. These are the two basic ways processors communicate with one another. Shared memory is a variant of parallel computing, while message passing is a more standard model of separate processors sending and receiving messages.

An important practical paradigm is distributed shared memory (or DSM), which is a network implementation of the shared memory model. DSM allows us to run shared memory algorithms over a network. Shared memory algorithms are often much easier to create than network algorithms, which makes this paradigm practical. However, there is a tradeoff between efficiency and consistency. In general, the more efficient the algorithm, the less memory consistency it can guarantee; and, on the other hand, very strong memory consistency is inefficient to simulate over a network. Memory consistency is a complex subject, but it can be briefly understood as the consistency between writes and reads. Suppose processor \( p \) writes to a memory location, and, later, processor \( q \) reads from that location. A strong memory consistency condition might require that \( q \) read what \( p \) wrote, while a weaker consistency condition might only require that \( q \) either read the value written from \( p \) or a restricted set of other values.

6 Synchronous vs. asynchronous processors

Throughout the course, we will use the terms strong model and weak model. These are not precisely defined, but intuitively, a strong model is a powerful underlying environment, which limits the amount of bad behavior an algorithm can engage in. As a result, you don’t need a very good algorithm to run on a strong model. By contrast, a weak model allows an algorithm to engage in many more different types of behavior. An algorithm that works correctly in a weak model will also work correctly in a strong model. On the other hand, if a problem is impossible in a strong model, then it remains impossible in a weak model. We will look at two extreme cases: a completely synchronous model, and a completely asynchronous model. Both of these models will be message passing (network) models.

6.1 Synchronous model

In the synchronous model, processors take steps in rounds. A given round has three phases, in which each processor does the following:

1. The processor receives all pending messages.
2. The processor computes
3. The processor sends all outgoing messages
6.2 Asynchronous model

The round structure of the asynchronous model is not guaranteed, the way it is in the synchronous model. Rather, we only have a guarantee that certain things will eventually happen. For any processor \( p \) at time step \( t \), eventually \( p \) will take a step. Expressed as a logical formula: 

\[
(\forall t)(\exists t' > t)[p \text{ takes a step at } t'].
\]

Similarly, we don’t know when messages will be delivered, but we know they eventually will be. Any message sent by \( p \) to \( q \) will eventually be received by \( q \).

6.3 Partial synchrony

We briefly note that it is possible to define partially synchronous models that lie in between these two extremes. For example, perhaps the processors are synchronous, and all messages are guaranteed to be delivered within a certain number of rounds. This model is weaker than full synchrony, but much stronger than full asynchrony.

7 Processor failure

We model three different types of processor failure, as follows.

**Crash failure** The processor stops working, and never starts up again. In synchronous systems, this is easy to detect. In asynchronous systems, it can be very hard to detect, because it is not easy to differentiate between a processor that is very slow, and processor that has stopped.

**Omission failure** The processor stops working for one or more rounds, but eventually comes back to life.

**Byzantine (malicious) failure** Processors can behave in an arbitrary fashion. Often, we assume that a processor takes the worst-possible step every time. This is similar to analysis made in the fields of security and cryptography, where the goal is to prevent damage from malicious users. Sometimes it’s useful to consider a processor that takes random steps each round. That models a processor that is “haywire” instead of a processor that has been “hacked.” Random behavior and malicious behavior can be quite different.

8 Formal model of message passing systems

We assume there are \( n \) processors in the system, and each processor knows who its neighbors are. We define the notion of a message buffer somewhat informally by saying the following.

\[
\text{outbuf}_{[p,q]} \quad \text{The sequence (or set) of messages sent by } p \text{ to } q \text{ that have not yet been received by } q.
\]
The sequence (or set) of messages sent by $p$ to $q$ that have been received, but not yet processed, by $q$.

This definition is somewhat informal because we are not rigorously defining what happens when a message is placed into a nonempty buffer. Perhaps we are modeling a FIFO system, in which case each message is placed in a queue when it enters the buffer, to be transferred in turn from the outbuffer to the inbuffer. Or perhaps each message buffer is a set: a message enters the buffer, but when a message is transferred, there is no guarantee that order of entry will be respected. Or perhaps there is even a third situation other than messages being in a sequence or a set. For the rest of the lecture, we will assume that messages in message buffers are in sequences.

**Definition 1** The system configuration is defined by: the state of each processor, the contents of each outbuffer, and the contents of each inbuffer.

**Definition 2** Events are steps that change the system configuration. There are two kinds of events.

**Computation event of $p$** Empty one or more inbuffers of $p$, change the processor state, then place one or more messages into outbuffers.

**Delivery event** A single message moves from an outbuffer to an inbuffer.

Note that the definition of a computation event depends on the amount of processor synchrony in the computation model. In a fully synchronous model, all messages that can be received are received, and all messages that can be sent are sent. In more asynchronous models, this may not be true.

### 8.1 Accessibility and applicability

Recall that an important feature of distributed computing is the lack of complete information each processor has about the system configuration. A processor can only compute using information it can see and deduce. We formalize this idea with the notion of *accessibility*. The only data accessible to the processor are the processor’s internal state and the contents of the processor’s inbuffers.

**Definition 3** An execution is a sequence of alternating configurations and events, starting with an initial configuration.

An example of an initial configuration would be: all inbuffers and outbuffers are empty, and all processors are in some initial state. We might write an execution in the form: $C_0, \phi_1, C_1, \phi_2, \ldots$, where $C_0$ is the initial configuration, $C_1$ is the configuration obtained by applying event $\phi_1$ to configuration $C_0$; and, in general, $C_{i+1}$ is the configuration obtained by applying event $\phi_{i+1}$ to configuration $C_i$. We say that event $\phi_k$ is *applicable* to configuration $C_{k-1}$ if $\phi_k$ can be legally executed by the system when in configuration $C_{k-1}$.
8.2 Formal definition of event applicability

**Definition 4** A delivery event $\phi_k = \text{del}[p,q,m]$ is applicable exactly when: message $m$ is in $\text{outbuf}[p,q]$ in $C_{k-1}$; and $C_{k-1}$ and $C_k$ can differ only in that the message moves from $\text{outbuf}[p,q]$ to $\text{inbuf}[p,q]$. In particular, the states of processors must be the same in both configurations.

Note that in the case of a FIFO model, the delivered message $m$ must be the first message in the outbuffer’s queue.

**Definition 5** A computation event $\phi_k = \text{comp}(p)$ is applicable to configuration $C_{k-1}$ as determined by the states of $p$ and the inbuffers of $p$, and the defined behavior of the algorithm of $p$. $C_{k-1}$ and $C_k$ will differ in three ways: in the state of $p$, $\forall q \in \text{inbuf}[q,p]$ is empty in $C_k$, and, for all $q$, messages are added to $\text{outbuf}[p,q]$ as directed by the algorithm of $p$.

8.3 Commutativity of computation events

If the processors are deterministic, we can always think of our execution as a sequence, even if steps are taking place at the same time. The following sequences below are the “same” because there are no delivery events: $\text{comp}(p) \text{comp}(q) \text{comp}(r)$; and $\text{comp}(q) \text{comp}(p) \text{comp}(r)$. 