NYU Distributed Systems: Modeling Distributed Systems

The goal for this lecture is simple: we want to define a model that allows us to reason about the behavior of the systems we will study through the rest of the class.

A **distributed system** consists of **one-or-more processes** that interact with each other, and implement some functionality. In this class we assume that each process fails independently, that is, the failure of a process p does not imply failure of process q. Achieving independence in practice is challenging (you might want to think about why), and is something one must consider when deploying distributed systems. For most of this class we will assume that *processes* communicate by **sending and receiving** messages over a **network**[1], which the literature refers to as *message passing*. The behavior of a distributed system is dictated by what **protocol** it is running. A protocol is analogous to an algorithm, and some of the papers (and perhaps the class) will use the term algorithm and protocol interchangeably. Given this broad description, our model needs to answer the following questions:

- What can we say about the relative speeds of processes, or the time it takes for the network to deliver messages? Unless otherwise specified, we will be assuming the asynchronous model, which is going to play a crucial part in our analysis throughout the semester.
- How do we describe the behavior of a process? We will be describing processes as I/O automaton, a formal model that was introduced by Lynch and Tuttle in 1987[2], which we explain below.
- What guarantees does the network provide? More precisely, we need to answer questions like: Is a message sent by a process p guaranteed to be eventually delivered? We will use both unreliable and reliable channels throughout the semester.
- What can fail and how? How many processes? When? What does it mean for a process to fail? The answer to this question will depend on the protocol.

The Asynchronous Model

The discussion below is in the context of a distributed system with two processes p_0 and p_1, where process p_0 sends a message m to process p_1. In this context consider two questions:

1. How much time does it take for the message to make its way from p_0 to p_1? Or equivalently, how long before p_1 receives m (assuming we are sure p_1 will eventually receive m)?
2. How much time does it take message m to process p_1 once it is received?

Observe that answering either question requires knowledge about the deployment environment, that is, knowledge about where the distributed system is running:

a. The time it takes for a message to go from one process to another depends on the distance between the two processes (information can go no faster than the speed of light c); messages sent by the other processes; the chance that a message is corrupted in transit; etc.

[1] Processes can also interact with each other using a set of shared registers or shared-memory locations. As we will cover later in class, message passing and shared memory have the same expressiveness, that is, a protocol written for one can be run in the other, albeit doing so affects the protocol's performance and failure guarantees.

[2] Hierarchical correctness proofs for distributed algorithms. Lynch and Tuttle, PODC 1987.

b. Similarly, time taken to process a message once it has been received on what other processes (or programs) are running on the same computer, on background tasks that the operating system or runtime might need to perform, etc.

Furthermore, these factors that are dictated by the deployment environment change over time: operating systems and processes get updated, networks get older, environmental conditions that affect message corruption vary over time, etc.

But when developing and implementing distributed protocols, computer scientists and programmers want to ensure correctness over time and across deployment environments. Therefore, we often build and analyze protocol behavior while making minimal assumptions about message delays and processing times.

This set of minimal assumptions is commonly referred to as the **asynchronous model**, and says that:

- A message sent from a process can take an unbounded amount of time before it is received by another process.
- Any computation (e.g., a function) can take an unbounded amount of time, and the time to compute the same function can differ across processes and over time.
- **Fairness**: An event that is enabled infinitely often will be executed infinitely often.[3]

The first two conditions are simple, we will describe fairness in more detail below but an intuitive explanation is that it limits how bad asynchrony can be. For example, consider the scenario set up at the beginning of this section (where p_0 sends a message m to p_1), extended with the additional assumption that the network is *reliable*, that is with the assumption that the network does not drop messages. Fairness ensures that p_1 must eventually receive the message, that is, if p_0 sends m at time t, there must exit time t' ($t' > t$) when p_1 is received. Without fairness, we need to consider executions (and deployments) where p_1 never receives the message. We will see other uses of the fairness assumption below when we discuss the network model.

Implications: Inability to determine process failure

The asynchronous model's lack of assumptions about the deployment environment means that no protocol can determine if a process has failed without additional assumptions. In particular, processes cannot distinguish between the effects of a slow network or process, and the effect of a failed process.

Concretely, we define that a process p has **failed** at time t if after time t process p does not interact with any other process. In the message passing regime that is our focus, this means that after time t process p does not send any messages, nor process a received message.

Note that in the message passing setting the only way that a process q can determine that the state of another process p ($p \neq q$) is by receiving a message from q. This includes determining whether process p is active (that is, that p has not failed). Consider a case where process q has not received any message from p in the interval $[t, t + \delta]$. Regardless of the value of δ, process q cannot distinguish between cases where p has failed and will never send a message, p is slow and will eventually send a message, and p has already sent a message but the network has not yet delivered it. The inability to distinguish

[3] Really the assumption here is *strong fairness*. In our protocols we often care about liveness, and as we will discuss in a later section we care about turning unreliable channels into reliable channels.

between these three possibilities lies at the core of why protocols in the asynchronous model cannot distinguish between failures and delays.

> You might wonder about what happens in practice? After all, computers (and thus the processes running on them) do fail, and identifying failed processes is important. We generally deal with this by strengthening assumptions: many of the protocols in use make assumptions about the maximum time the network can take to deliver a message. This assumption is often conservative, and protocols are designed to limit negative effects if these assumptions are violated.

Processes and I/O Automata

You have likely encountered several ways to describe computation (in pseudocode, actual code, etc.), analyze computation, and write programs that can be executed by a computer. Why then do we need to discuss or figure out a new (or at least a specific) way to discuss computation in distributed protocols? There are two reasons: (a) the protocols are largely driven by receiving messages or timeouts, and the I/O automata model that we describe below focuses on the effect of these events; and (b) most distributed system implementations use and are described in terms of remote-procedure calls (RPCs, a common abstraction for communication) which necessitate the use of multiple threads. The I/O automata model seeks to hide the effect of concurrency, making analysis easier.

So how do we use/reason about process logic in the I/O automate model? At its core, you specify the behavior of a process by providing event handling logic (or code) that is run in response to the following three types of events:

(a) When a process first starts up (initialization code).
(b) When a process receives a message of type M_t.
(c) When a timeout occurs.

Each event handler can update the process's state, and send 0 or more messages. For convenience, we often treat timeouts as a special type of message.

The model execution model requires that each event handler is executed atomically, that is, it appears as if exactly one event handler is running at a time. Concretely, consider a process specification of the form:

```
on initialize:
  a = 0
  b = 0
on receive increment from c:
  a = a + 1
  b = a * 2
on receive decrement from c:
  a = a - 1
  b = a * 2
on receive current_value from c:
  send(c, {a = a, b = b})
```

This code receives three types of messages, `increment`, `decrement` and `current_value`. On receiving an `increment` or `decrement` message the process update the values of a and b, while it handles a `current_value` message by sending a message to c (the sender of the `current_value`) message the current values of a and b.

Because event handlers are executed atomically, we can assume that whenever the `increment`, `decrement` or `current_value` handlers are run `b = 2*a`, and we do not need to consider cases where two handlers are interleaved.

We use Elixir for labs in this class and the code you write will be very close to an I/O automata specification. For example, the code above in Elixir would read as:

```
def start_process() do
  process_logic(0, 0)
end

defp process_logic(a, b) do
  receive do
    {_c, :increment} ->
      process_logic(a + 1, 2* (a + 1))
    {_c, :decrement} ->
      process_logic(a - 1, 2* (a - 1))
    {c, :current_value} ->
      send(c, %{a=>a, b=>b})
      process_logic(a, b)
  end
end
```

Observe the close correspondence between the pseudocode/protocol description above and the Elixir code.

Network Model

The network in a distributed system connects processes and allows them to interact with each other. However, networks themselves are built using wires and devices (switches, etc.) that can have errors or fail. Furthermore, networks must also deal with situations where they need to carry more messages than they can accommodate. The network model captures what processes can expect from a network regardless of failures, errors or a lack of resources.

We express the network model in terms of the behavior of a channel (or connection) between two processes. We can characterize connections along two dimensions:

Reliability: Does it drop messages

An **unreliable channel** is one that can drop messages. When using an unreliable channel that connects process p to q (neither of which fail), a process q may never receive a message m sent by p.

We require unreliable channels to be **fair** (in the same sense as we assumed for the asynchronous model): if some process p sends m infinitely often to process q over an unreliable channel, then q must receive m infinitely often.

A **reliable channel** is one that does not drop messages: if process p sends message m to process q, then eventually q receives m or q fails (we cannot ensure messages are received by a failed process).

Observe that the fairness assumption allows one to write a distributed protocol that provides reliability (that is, provides the same guarantees as a reliable channel) to processes communicating using an unreliable channel. Indeed, many real networks (including the Internet) provide unreliable channels and programs (or the operating system) use additional protocols for reliability. Unless otherwise stated, we assume unreliable channels.

Ordering

An **ordered channel** guarantees that messages are delivered in the order sent. Put more precisely, an ordered channel ensure that if process q receives message m_0 then m_1 (both sent by process p) then process p must have sent m_0 before m_1. Note, the ordering guarantees only apply to messages sent by a single process: process q might receive m_0 from process p before m_1 from process r even though p sent m_0 before r sent m_1.

On the other hand, an **unordered channel** makes no guarantees about the order in which messages are received even when they are sent by the same process.

Observe that one can produce a protocol that orders messages sent over an unordered channel.

Failure Model

Generally one needs to make assumptions about how many processes can fail and the behavior of failed processes. These assumptions are collectively referred to as the failure model.

The failure model for most protocols specify that no more than f processes (where f is some function of the total number of processes p, e.g., $f = \frac{n}{2} - 1$). Failure models must also specify the behavior of a process after it has failed, including whether it can recover. In this class we will consider three types of failures:

- **Fail-stop**: Most of our analysis will assume fail-stop behavior, where after process p fails it can no longer interact with any other process, that is it can neither send nor receive messages. We assumed this model previously when discussing the implications of the asynchronous model (though note that the inability to distinguish between failures and delays applies to all models). It is important to note that in the fail-stop model once a process has failed it can never recover, that is, once a process fails it will never again send or receive a message.
- **Fail-recover**: The fail-recover model extend the fail-stop model to allow recovery. That is, in the fail-recover model a failed process cannot send or receive any messages. However, if a process p fails at time t, it might recover (and start sending or receiving messages) at some time $t' > t$. Observe that a recovered process might have stale or incorrect state, and protocols that assume the fail-recover model usually include logic to check and update the state of recovered processes.

- **Byzantine failures**: The byzantine failure model, which will be our focus later in the semester does not impose any restrictions on how failed-processes behave. In other words, a failed process can stop sending messages (similar to the fail-stop model), send corrupt messages, or send messages that do not conform to the protocol being executed. This is a general failure model that allows reasoning about protocol behavior in the presence of bugs or malicious actors, and is used when designing protocols whose correctness is critical.