Lecture Notes for CSCI 351: Distributed Computing

Set 6-Simulations

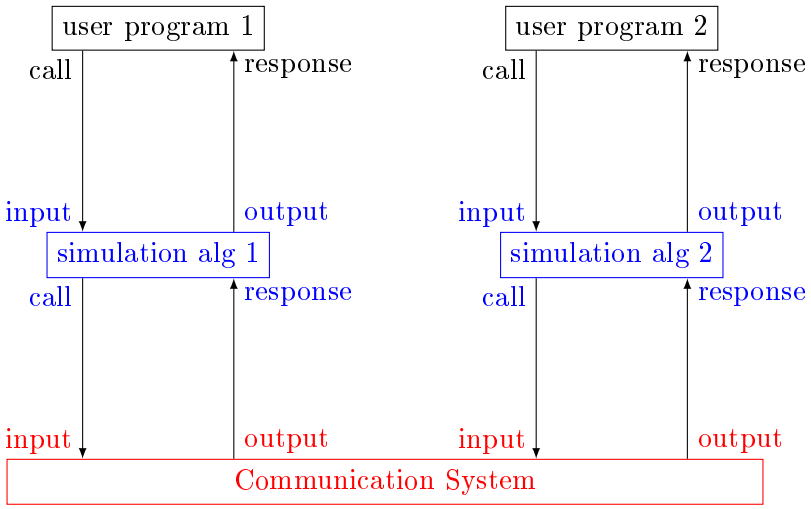
Professor Talmage

March 31, 2023

We have talked some about simulations, in the context of using different ADTs to solve consensus, but have not treated them generally or formally. Simulations are core to much of distributed computing, since we can use them to hide undesired behavior and run algorithms for easy systems on real systems that are more difficult. We will look at providing stronger guarantees, tolerating faults, hiding asynchrony, and simulating shared memory in message passing systems. But first, we need to set up the formalisms we need.

1 Modeling Problems as Simulations

A *simulation* is an abstraction layer, translating a system model or interface and its allowed interactions into a different model or interface.



To specify a simulation, we need to think about each part of a distributed system in terms of its input and output (the top interface in the above figure). This is very similar to how we specify ADTs, and we informally did this when we represented the Consensus problem as an ADT. We were in a sense just representing it by its interface, so that we could simulate its behavior using different ADTs as a lower layer.

Assumptions:

- An object at a higher level can call any function provided by a connected object at the next lower level at any time.
- Assume local computation is instantaneous. This restricts our attention to communication costs. Intuitively, the only delays we care about are between when we ask a lower-level object to do something

and when it sends a response back (though we may sometimes not need to wait for such a response). The communication system may impose some time cost between events.

- Note that the picture is limited, and a lower level object can generate outputs that are not responses to call from the higher level object.
- An object can require a certain behavior from a higher level. For example, a Mutual Exclusion object requires its user to infinitely cycle through Entry, Critical, Exit, Remainder in order. If the higher level does not follow such requirements, the object can behave arbitrarily.

1.1 Example Model and Problem:

Consider a message-passing system model. We can represent this model by an upper interface as follows, then discuss what guarantees that interface provides and build abstractions on top of it.

Definition 1. An asynchronous message passing system provides an interface of two types of events:

- $send_i(S)$, an input event. Process p_i sends the (possibly empty) set of messages S. Each message indicates sender and recipient s.t. S contains at most one message for each process connected to p_i.
  - This is a constraint on the user–you cannot try to send messages to non-adjacent processes.
- $receive_i(R)$, an output event. Process p_i receives the set of messages R, where each message in R specifies p_i as its recipient.

A sequence of these events is allowable if there is a function from the union of all sets R to the union of all sets S. The following conditions must also hold:

- **Integrity:** Every received message maps to a previously sent message with the same content. (No corruption, no phantom messages.)
- **No Duplicates:** Each message sent is received once (function is one-to-one).
- **Liveness:** Every message sent is received (function is onto)

To consider failures of the messaging system, we will weaken one or more of these conditions.

We can similarly specify a problem in the same style. Note that when we say "problem" in this context, we are really just specifying the behavior we want the simulation layer to provide. Solving a problem is implementing an abstraction layer that provides that behavior as its upper interface, connecting its lower interface with the upper interface of whatever model or other abstraction layers we assume are present.

Definition 2. The *broadcast* problem provides the following interface:

- $bc\text{-}send_i(s)$, an input event. p_i sends message s to all processes.
- $bc\text{-}recv_i(r, j)$, an output event. p_i receives message r broadcast by p_j.

Allowable sequences are those where there is a function k from every $bc\text{-}rec_i(m, j)$ to an earlier $send_i(m)$ satisfying

- **Integrity:** k is well-defined.
- **No Duplicates:** For each i, the restriction of k to $bc\text{-}recv_i$ events is a one-to-one function.
- **Liveness:** For each i, the restriction of k to $bc\text{-}recv_i$ events is an onto function.

Exercise: Explain, in English, what the conditions for broadcast mean (well-defined, one-to-one, onto). What types of behavior do they allow or disallow?

Exercise: Give an abstraction-layer style definition of an asynchronous shared memory system. Pick a single ADT and give the specification for that system, then think about how to generalize.

2 Strengthening Broadcast

Exercise: Write an implementation of broadcast as defined above. Think about code structure–what events do we need to handle? What tools are available to us?

Algorithm 1 Basic broadcast implementation, code for p_i

Upon event: *bc-send*(m)
  for $0 \leq j \leq n-1, j \neq i$ **do**
    send$(\langle m, i, j\rangle)$ ▷ Notation: $\langle content, sender, recipient\rangle$
  end for
end Upon event:
Upon event: *receive*$(\langle m, j, i\rangle)$ ▷ Formalism says this could be a set of messages,
  Trigger event *bc-recv*(m, j) ▷ we'll handle one at a time.
end Upon event:

While broadcast is useful, and straightforward to implement, we often want even stronger guarantees. For example, if p_0 and p_1 broadcast messages m_0 and m_1, in what order should other processes receive those messages? If p_2 receives m_0 first, does p_3 also? There are, generally, two categories in which desired broadcast guarantees fall:

- **Ordering:** Do all processes receive all broadcast messages in the same order? Is that order related to the real-time order of *bc-send*s? Do processes at least receive messages from the same sender in send order?
- **Reliability:** If broadcasting processes crash, do other processes receive the same set of messages? In the same order? If a correct process broadcasts, can it be sure that its message is received?

2.1 Ordering

Exercise: Give one or more ordering properties you might like broadcast to guarantee.

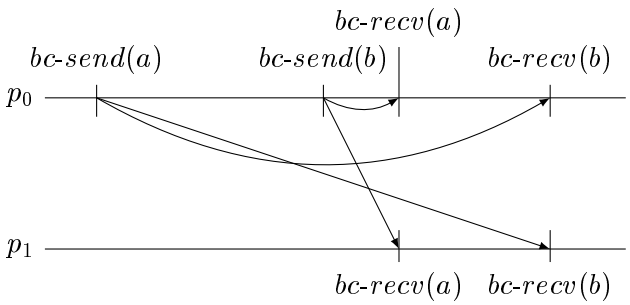
Here are three possible ordering constraints we could define for broadcast (there may be others). We state the claims in the negative, so that we can later weaken the liveness assumption when considering failures.

1. *Single-Source FIFO*: For all messages m, n and all processes p_i, p_j, if p_i broadcasts m, then later broadcasts n, p_j does not *bc-recv* n before m.
2. *Totally Ordered*: For all messages m, n, processes p_i, p_j, if p_i receives m before n, then p_j does not receive n before m.
3. *Causally Ordered*: For all messages m, n, and processes p_i, p_j, if p_j sends n and $m \prec n$, then p_i does not receive n before m.
    - We define the happens-before relation for messages m, n as $m \prec n$ if *bc-rec*$_j(m)$ happens-before (as previously defined) *bc-send*$_j(n)$.

Exercise: Consider the following example execution. Which broadcast ordering constraints does it satisfy? Which does it not satisfy?

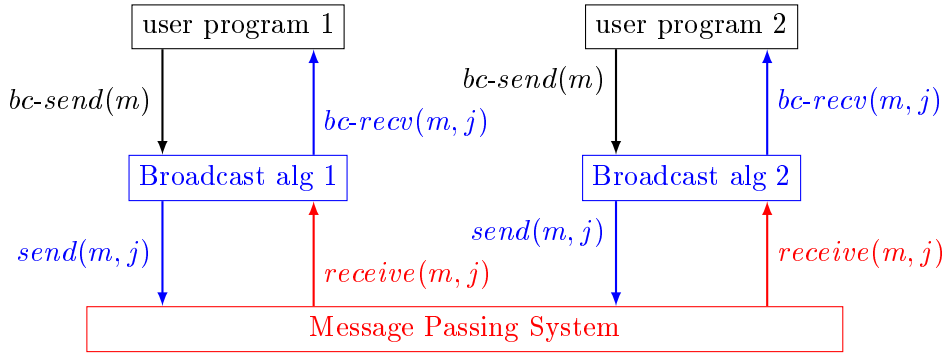
- This execution is Totally-Ordered, since all processes receive broadcast messages in same order.

- This is not SSFIFO, since the receive order doesn't match the sending order of messages from the same process.
- This is causally ordered, vacuously, as neither of the two messages happened before the other.



Exercise: What are the relationships between the three types of broadcast? For every pair of conditions A and B where $A \not\Rightarrow B$, draw executions which satisfy A but not B.

To implement these guarantees, we need to go back and write code for a new broadcast abstraction layer, on top of the simple message-passing layer. That is, we need to write code that handles *bc-send* events from above and *receive* events from below, generating *send* events below and *bc-recv* events above.



Exercise: Implement broadcast providing each of the three guarantees.

Ideas:

- **SSFIFO:** Each process appends a sequence number to its messages. Only *bc-recv* a message when all smaller sequence numbers from that sender have been *bc-recv*'d.
- **Causally Ordered:** Use vector clocks. Cannot deliver a broadcast message until your vector clock is at least as large as the message's, except for the sender's component, where your clock should be one lower than the sender's. Only update the vector clock when you deliver a broadcast message, not when you receive low-level messages.
- **Totally Ordered:** Implement on top of SSFIFO broadcast, making the abstraction stack one layer taller.
  - Use Logical clocks and a modified form of vector clocks.
  - Tag messages with logical clock timestamp (single integer).
  - When you update your logical clock, send a "background" message to update others' clocks.
  - Update component of vector corresponding to p_j to t when confident (from SSFIFO) we cannot receive a message from p_j with timestamp smaller than t.

- *bc-recv* a message when all components of vector clock are greater than t.
- To prove, show that timestamps are all comparable and all processes deliver all messages in timestamp order.

3 Reliability

Another property of a communication system we may wish to hide is failures. As an example, we will consider implementing a failure-tolerant Broadcast primitive. This does not prevent process failures, but should enable processes to broadcast messages with some expectation that it will "work", even if processes crash.

Assume we have an asynchronous, point-to-point (only provides *send* and *receive*) message passing system of n processes, at most f of which may crash. We update the Liveness condition for our underlying system to the following:

- **Nonfaulty Liveness:** Every message sent by a nonfaulty process to another nonfaulty process is eventually received.

> **Exercise:** What condition(s) would you like a fault-tolerant broadcast algorithm to guarantee? State in English and formally.

- **Integrity:** For each process p_i, the restriction of k to $bc\text{-}recv_i$ events is well-defined. (Every message was previously sent.)
- **No Duplicates:** for each p_i, $k|_i$ is one-to-one.
- **Nonfaulty Liveness:** When k's range is restricted to correct processes, for every correct p_i, $k|i$ is onto. That is, every *bc-send* at a correct process is received by all correct processes.
- **Faulty Liveness:** If some correct process *bc-recv*'s a message m, every correct process *bc-recv*'s m. Thus, if a crashing process broadcasts a message, either all correct processes receive it or none do.

> **Exercise:** Write pseudocode for a reliable basic broadcast algorithm.

Idea: When receiving a message for the first time, *bc-send* that message in case the sender didn't manage to send it to all processes, then *bc-recv* it. If any correct process receives it, then it will have forwarded it to all correct processes, so they will all get it.

> **Exercise:** How can you make any of our ordered broadcasts reliable?

We can simply add the repeating mechanism for basic broadcast in most cases. The sneaky one is that we cannot implement totally-ordered reliable broadcast (also called *atomic broadcast*) in an asynchronous, fault-prone system.

> **Exercise:** Why is it impossible to implement totally-ordered reliable broadcast in an asynchronous, fault-prone system? Which previous impossibility result would that circumvent?

We can use totally-ordered reliable broadcast to solve consensus, which in this model would contradict FLP.