Logistics

- Programming assignment 2
  - Let's go over it now to make sure any initial questions get addressed.

- Organizing extra meeting time
  - I'm out for SC08 conference in mid November.
  - I owe you a lecture.

  - The proposal: We meet next week one evening from 6-7:20 pm.
    - **Topic**: Concurrent programming and threading. Useful for programming assignment #2 and the final project.
    - Free food to entice you to come. Pizza?

Multicast

- Multicast is a generic group communication operation.
  - IP multicast is simply one instance of it.
- The basic primitives are:
  - `Multicast(m,g)` where m is a message, and g is a group of participating processes.
  - `Deliver(m)` delivers the message on the process that received it.
- Messages in a multicast system are annotated with the sender and group that they are intended for.

Basic multicast

- This is the easiest multicast scheme.
- For all processes in the group, the sender sends a point-to-point message.
- Delivery is achieved with the basic receive operation.

- Unlike IP multicast, we assume the communication is reliable.
  - One can use IP multicast to implement this if a layer is placed over the UDP messages to do retries, duplicate handling, and order enforcement.
- Not efficient in it's basic form. Bottleneck on the sender side.

Reliable multicast

- In basic multicast, we can see the following occur:
  - Sender starts it's sequence of sends.
  - Part of the way through, it dies.
  - Some set of group members never get the message.

- Reliable multicast adds what is known as an Agreement property.
  - If any member of the group delivers m, then all of the correct (not failed) processes in the group will eventually deliver m.

- We can achieve this on top of basic multicast.

Reliable multicast

- Start: Everyone initializes their received set to empty.
- The originator of the message uses basic multicast to send to the entire group, including itself.

- On the basic deliver occurring on a process q:
  - If the message is not in the received set on q:
    - Add it to the set
    - If q wasn't the originator of the message, basic multicast it to the group.
    - Successfully deliver the message reliably.

Reliable multicast

- Good property: As we can see, processes can die, but if any of them successfully delivered it, then everyone will eventually see it.

- Bad property: The message gets sent by every member of the group to everyone else!
  - Not really efficient.

Ordered multicast

- Ordering requirements on when messages are delivered.
  - **FIFO ordering**: If a correct process says multicast(g,m) and then multicast(g,m'), then every correct process that delivers m' will deliver m before m'.
  - **Causal ordering**: if multicast(g,m) happens before multicast(g,m') then any correct process that delivers m' will deliver m before m'.
    - Note that the happens before relation holds on the group, not just a process.
  - **Total ordering**: If a correct process delivers m before m', then any other correct process that delivers m' will deliver m before m'.

Consequences

- Causal ordering implies FIFO ordering (due to their happens-before relation in a single process).
- FIFO and causal orderings are only partial orders.
- For time reasons, we won't go into these algorithms today.
  - When you read the book though, you should see that use of logical clocks (either Lamport or Vector) makes it possible to negotiate the necessary ordering by attaching stamps to messages as they are moved around and ordered.

Consensus

- Goal: A set of processes agree upon a value after one or more propose one.
- Consensus is hard. In fact, situations can exist in which consensus cannot be achieved.
  - These situations are actually not rare or unexpected.
- Assume a set of processes communicating by messages.
- **Here is the wrinkle**: Consensus must be reachable in the presence of faults.
  - Assume some number of failures.

Consensus problems

- Every process starts undecided.
- Processes propose a value.
- Communication occurs, and a decision value is reached.
- Processes reach the decided state when they receive this final decision message.

Properties

- **Termination**: Eventually every correct process sets its decision variable.
- **Agreement**: All correct processes in the decided state will have the same decision variable value.
- **Integrity**: If a value is agreed upon, then the value was proposed by some process.
- **Validity**: If all processes propose the same value, then every correct process chooses that value.

Simple case

- No failures.
- Everyone multicasts proposed value to everyone else.
- Everyone collects these until all processes see all proposals.
- Some majority function is evaluated to determine the winner, or some special value if no majority winner exists.

Byzantine generals problem

- Proposed by Lamport.
- Three or more generals are to agree to attack or retreat.
- One general, the commander, issues the order.
- The others, are to decide to attack or retreat.
- The complication: one or more generals can be treacherous – faulty.
  - E.g.: Tells one general to attack, another to retreat.
- Differs from “pure” consensus because one special process initiates the orders.

Interactive consistency

- Like consensus, except a vector of values per process is agreed upon.
- E.g.: Let each of a set of processes obtain the same information about their respective states.

Relationships

- These are all related:
  - IC from BG: Run BG N times, once with each process acting as commander. In other words, one BG run per vector entry.
  - C from IC: Run IC, and then compute the majority function on each element of the vector on all processes.
  - BG from C: Commander sends proposed value out and to itself. All processes then run consensus.

- In systems with potential crashes, consensus is equivalent to totally ordered multicast.
  - All processes multicast their value.
  - Each process chooses the first value that it delivers.
  - Total order and reliability of RTO multicast key.

Synchronous consensus

- Assume f failures tolerated.
- Everyone initializes a values array v[1] to their value, and creates an empty prior-step array, v[0] = {}.
- For f+1 rounds:
  - B-Multicast the values that are in the current values are in v[i] and not in v[i-1].
  - Set v[i+1] to v[i]
  - Accumulate up received messages from others, union received value set with v[i+1].
  - Increment i.
- After f+1 rounds, assign the decided upon value on each process to the minimum of v[f+1].

Byzantine generals problem

- What is the main idea behind this problem?
- How to detect a system that is faulty.
  - Either by malfunction of lying.
- Proofs that the book refers to for impossibility of detection for N<=3f, where f is the number of failures and N is the number of participants.

- **Example**: Three generals (X,Y,Z). Commander X tells Y to attack, Z to retreat. Y tells Z attack, Z tells Y retreat. Y and Z can't tell if the commander was lying, or if the other general is lying.
  - With 4 you can tell who was lying assuming only one liar.
- Easier to figure out if signatures used.
  - Z can forward message from X to Y, and Y can tell if Z is honest if the signature is preserved and that X told Z something different than it told Y.

Asynchronous consensus

- Asynchronous systems have no known bounds on times.
  - So, a very long message wait period can look like a failure.
- Methods exist to work around this, through fault detection, fault masking, etc…
- In any case, consensus in an arbitrary asynchronous system is impossible.