Topic 5: Multicast

- Why do you need multicast?
- Explain basic multicast assuming reliable 1:1 communication
- What are the requirements to reliable multicast and how do you implement it over basic multicast and IP-multicast?
- Explain the difference between FIFO, Total and Causal ordering? When is it important?
- Briefly explain the two ideas to implement TO-multicast. What can you say about reliability?

Material: 5ed section 15.4. (lecture 8)

Why Multicast?

Multicast serves the purpose of delivering a message from a single process to all processes belonging to a given group.

Basic Multicast

Two primitives and a guarantee: A correct process will eventually deliver the message, as long as the multicaster does not crash.

- `B-multicast(g, m)`: for each process `p` in `g`, `send(p, m)`
- On `receive(m)` at `p`: `B-deliver(m)` at `p`.

Problems:

- If run concurrently with a large number of receiving processes, liable to suffer from `ack-implosion`, or *acknowledgement-implosion*, where too many processes respond at the same time, causing the process to drop responses and waste more bandwidth trying to re-send the message.
- If the multicaster crashes, then some members of group `g` receive the message while other's don't.

Reliable Multicast

Reliable multicast has the following properties:

- **Integrity**: A correct process `p` delivers a message `m` at most once.
- **Validity**: If a correct process multicasts `m`, then it will eventually deliver `m`.
- **Agreement**: If a correct process delivers message `m`, then all other correct processes in `group(m)` eventually deliver `m`.

Over Basic Multicast

The sender begins by B-multicasting to each member of group `g`, including itself. On delivering `m`, each member which did not send the original message calls `B-multicast` again, multicasting to all members of the group, and then `R-deliver`s the message.

This guarantees that even if the original sender fails, as long as one correct process receives the message, it is forwarded again by each recipient. It should be assumed that duplicates are handled, since this means correct processes will receive the same message more than once. Not very practical, since each process receives each message |`g`| times. Guaranteed to work on even asynchronous systems.

Over IP Multicast

Each process maintains an array of sequence numbers.

1. For each group a process belongs to, it maintains a sequence number of sent messages
2. For each sender in each group it belongs to, it maintains a sequence number of received messages.

For each message sent to the group, the sender adds the value S for the group and a list of acknowledgements, so that each member can verify that their S corresponds with the R provided in the message by the sender. This way, the recipients can see which messages they have not received, and request them from the sender, or from the node which sent them the acknowledgements.

This implementation is not practical, since we cannot assume that the messages continue indefinitely (which is how we could guarantee agreement), or that copies of all messages are retained, so they can be re-sent.

FIFO, Total and Causal Ordering

FIFO Ordering: if a correct process issues `multicast(g, m)` and then `multicast(g, m')`, then every correct process that delivers `m'` will deliver `m` first.

Causal Ordering: If `multicast(g, m)` → `multicast(g, m')`, where → is the happened-before relation incuded only by messages sent between the members of `g`, then any correct process that delivers `m'` will deliver `m` first.

Total Ordering: If a correct process delivers message `m` before it delivers `m'`, then any other correct process that delivers `m'` will deliver `m` first.

Importance

First of all, ordering does not imply reliability. A totally ordered multicast simply states that all processes deliver the messages in the same order. FIFO does not guarantee that the actual order in which different processes issued `multicast(g, m)` will be respected, etc.

Ordering is especially important in applications such as decentralized messaging, where the order in which things are said is relevant. An alternative example is safety-critical systems, or banking, where certain operations only make sense in the context of previous operations.

Consider for example the difference between 1.0 * 10 + 10 vs 1.0 + 10 * 10.

Totally Ordered Multicast

Assign totally ordered identifiers to multicast messages, so each process makes the same ordering decision based on these identifiers.

Use the sequencers from the FIFO-multicast, but make them specific to the group, and not the process. This only works with non-overlapping groups.

The first implementation depends on a sequencer process to assign the sequence. The sequencer assigns a sequence number to each message it receives (handling duplicates), and multicasts an order message to the group once it has B-delivered the message.

The second implementation requires the processes to collectively agree on a sequence number. Each process maintains two sequences, and on receiving a message, they respond with a proposal, which is 1 + the larger number of

1. The largest observed proposal in the group, or
2. Their own largest proposal. The largest number wins, and is assigned to the message id, and multicast. Once messages have been assigned a number, they are moved from the hold-back queue into the delivery queue.

These protocols are reliable if the implementations use R-multicast instead of B-multicast.