Types of Broadcasts (Multicasts)

I Reliable Broadcast

a set of processes P with:

a subset $C \subseteq P$ of correct (fault-free) processes

a subset $P - C \subseteq P$ of incorrect (faulty) processes

processes perform 2 actions within the Reliable Broadcast layer

broadcast(m) - send m to all processes

deliver(m) - deliver m to layer above (could be another broadcast layer or the application)

when messages are received off the network, they are sent to the reliable broadcast layer for processing and potential delivery

Reliable Broadcast is defined by the following properties:

RB1 (Validity): If a correct process broadcasts a message m, then all correct processes eventually deliver m.

RB2 (Agreement): If a correct process delivers a message m, then all other correct processes eventually deliver m.

RB3 (Integrity): For any message m, every correct process delivers m at most once and only if some process broadcast m.

(Note that there are no ordering requirements for Reliable Broadcast.)

II FIFO Broadcast

FIFO Broadcast satisfies Properties RB1-RB3 <u>and</u> the following property:

F1 (FIFO Order): If a correct process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' before it delivers m.

~~Contrasts FIFO channel~~

III Causal Broadcast

CB will <u>strengthen</u> the ordering requirement to account for causality

explain why Causal ⇒ FIFO

for example, if process p broadcasts m_p and q delivers m_p and then broadcasts m_q, then broadcast(m_p) "happened before" broadcast(m_q) and all correct processes should deliver m_p before m_q for a causal ordering

Causal Broadcast satisfies Properties RB1-RB3 <u>and</u> the following property:

C1 (Causal Order): If the broadcast of a message m <u>causally precedes</u> the broadcast of a message m', then no correct process delivers m' before it delivers m.

exactly the same as our def'n of causality, i.e. "happens before" relation

IV Atomic Broadcast

Atomic Broadcast imposes a <u>total order</u> on message delivery by correct processes whereas Causal Broadcast imposes only a partial order (as discussed earlier in semester, causal precedence imposes a partial order), two messages that have no causal relationship (concurrent) can be ordered in either manner by Causal Broadcast

Atomic Broadcast satisfies Properties RB1 - RB3 and the following property:

A1 (Total Order): If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

(all correct processes deliver messages in exactly the same order)

(note that total order can be anything, i.e. does not have to be FIFO nor causal)

V Combination broadcasts

(A) FIFO Atomic Broadcast - satisfies Properties RB1-RB3, F1, and A1

(total order that maintains FIFO order)

(B) Causal Atomic Broadcast - satisfies Properties RB1-RB3, C1, and A1

(total order that maintains causal order - <u>strongest possible ordering requirement</u>)

Simple Broadcast Algorithms

for RB, FB, and CB; assume:

- asynchronous system (unbounded communication delays)
- crash faults (in individual channels or in processes)
- correct processes remain strongly connected at all times

Algorithm Reliable Broadcast (RB)

when a process p wants to reliably broadcast a message m, it calls R-broadcast(m):

```
R-broadcast(m)
{
    tag m with p and seq-num(m)
    send (m, tag) to all processes including p
}
```

globally unique ID for m

locally unique ID for m

when a process p receives a message (m, tag) from the network, it executes Recv-handler (m, tag):

```
Recv-handler(m, tag)
{
    if p has not already executed R-deliver(m, tag)
    {   if (sender(m) ≠ p) send (m, tag) to all processes*
        R-deliver(m, tag)
    }
}
```

Theorem: Algorithm RB satisfies the Reliable Broadcast properties (RB1, RB2, RB3) under the above-stated assumptions.

Proof Sketch: RB1 (Validity): as long as the correct processes remain strongly connected, the rebroadcast operation (* above) ensures that the message will eventually be received by every correct process.

when a correct process receives a message for the 1st time, it delivers the message

∴ all correct processes eventually deliver the message

RB2 (Agreement):

beyond RB1, we need to consider messages sent by incorrect processes for RB2

since we assume crash failures, the only new case is a process that crashes while it is sending a broadcast

Case a: the broadcast message is received and delivered by at least one correct process

since any correct process will forward the message to all other processes before delivering it, this becomes the RB1 case and, by the same reasoning as above, all correct processes will eventually deliver the message and so they agree

Case b: the message is not received and delivered by any correct process

here, since no correct process delivers the message, they again agree

RB3 (Integrity):

a correct process delivers each message at most once because redundant receives are detected and discarded based on the unique message ID

a correct process delivers a message only if it receives the message and, under the stated assumptions, it can only receive a message if some process broadcast it

QED

Note that Algorithm R-Broadcast does not work for less strict fault models, e.g. if messages can be corrupted then the integrity of the unique tags can not be maintained and a correct process can deliver the same message more than once

We will now build FIFO and Causal Broadcast algorithms out of Algorithm R-Broadcast

<u>Algorithm F-Broadcast</u> (FB) (works whether channels are FIFO or not)

When a process p wants to FIFO broadcast a message m, it executes:

```
F_broadcast(m)
{   seq_num ← seq_num + 1
    R_broadcast(m)
}
```

Initialization step on each process:

```
msgQueue ← ∅
next[q] ← 1   for all processes q
seq_num ← 0
```

when a process p executes R-deliver(m, tag), this triggers execution of R-delivery-handler(m, tag):

```
R-delivery-handler(m, tag)
{   q ← sender(m)
    msgQueue ← msgQueue + m
    while (∃ m' ∈ msgQueue : (sender(m') = q and
                              seq-num(m') = next[q]))
    {
        F-deliver(m', tag(m'))
        next[q] ← next[q] + 1
        msgQueue ← msgQueue − m'
    }
```

suppose P3 has next[5] = 16 and P3's message queue contains tags (5,17), (5,18), (5,19), (5,22), (5,23) (and some messages from other processes)

when message 16 from P5 is R-delivered, P3 will FIFO deliver (5,16), (5,17), (5,18), and (5,19) in that order and set next[5] = 20

Algorithm C-Broadcast (CB)

Initialization on each process: prevDelivered ← ϕ

when process p wants to causally broadcast message m, it executes:

```
C-broadcast(m)
{   F-broadcast(<prevDelivered> + m)
    prevDelivered ← φ
}
```

when process p executes F-deliver($\langle m_1, \ldots, m_\ell \rangle$), it also executes F-delivery-handler($\langle m_1, \ldots, m_\ell \rangle$):

```
F-delivery-handler(<m_1, ..., m_ℓ>){
    for i=1 to ℓ
    {
        if p has not already executed C-deliver(m_i)
            then | C-deliver(m_i)
                 | prevDelivered ← prevDelivered + m_i
    }
```

Correctness not as obvious as F-broadcast but idea is that when a process broadcasts a message, it includes the list of all messages that the message depends on, i.e. all messages that the process C-delivered before sending the message

a receiver then makes sure to C-deliver all the "depends on" messages before C-delivering the message being broadcast

Atomic Broadcast can not be achieved in pure asynchronous systems — known result

let us consider Atomic Broadcast in a synchronous system assuming clocks are synchronized

to be specific, assume:

1. at any real time t, clocks on correct processes p and q do not differ by more than ε (must be ensured by a distributed clock synchronization algorithm)
2. clocks on correct processes are monotonically increasing and do not skip any scheduled event times

Algorithm A-Broadcast-Synchronous

when a process p wants to atomically broadcast a message m, it executes:

```
A-broadcast(m)
{
    R-broadcast(m, time-stamp(m))
}
```

local time on p when broadcast is initiated

when a process p executes R-deliver(m, time-stamp(m)), this triggers R-delivery-handler-A(m, time-stamp(m)):

```
R-delivery-handler-A(m, time-stamp(m))
{
    schedule A-deliver(m) at local time time-stamp(m)+Δ
}
```

Δ is the same one every process and must satisfy

$$\Delta \geq \varepsilon + d_{max}$$

d_{max}: max. delay between R-broadcast on sender and R-deliver on any receiver

messages are delivered in time stamp order on every correct process (if 2 messages have identical time stamps, deliver them in order of their sender's process ID)

$\Rightarrow$ all correct processes deliver messages in the same order

Software Replication

(I) Active Replication – a software component is installed on multiple nodes such that, at all times, each correct component (replica) provides an identical service (software version of modular redundancy)

this requires that all correct processes produce the same sequence of outputs (output consistency)

Sufficient conditions for output consistency:

1) input consistency – input messages are delivered to all correct replicas in identical order (atomic broadcast guarantees this)
2) replica determinism – given the same initial states and the same sequences of input messages, two correct replicas' executions will proceed through the same sequences of states (executions will not diverge due to non-deterministic events)

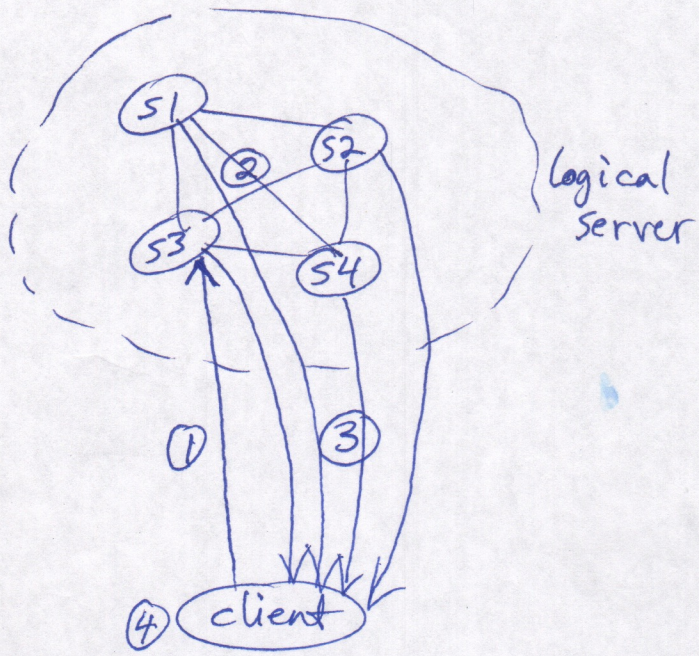
example sources of non-determinism:

- random number generators
- system calls (more generally, any interactions with the external environment, which can be different on different nodes hosting different replicas)
- cache behavior
- thread scheduling
- ...

replica executions must eliminate all non-determinism to guarantee they do not diverge from each other

Output selection –

assume a client-server model



1. client contacts arbitrary server
2. request is atomically broadcast to all servers
   (if initial server is faulty and drops request, client times out and tries another server)
3. each server individually responds to client
4. client chooses correct output

   <u>case 1</u> – crash faults: client accepts the first response it receives (servers may designate only 1 server to respond for efficiency) — discussed on next page

   <u>case 2</u> – Byzantine faults: client waits for $t+1$ matching responses, where t is the max. # of faulty servers (response should include request to prevent initial server from modifying request)

note that, for crash faults, servers could coordinate among themselves to only send 1 response to client

if selected responder fails before responding, client would have to time out and retry request

however, for Byzantine faults, client must be the one to select the correct response because it can not trust any single server

major advantage of active replication – replica failures are masked so that there is no recovery needed upon a failure and clients continue to see same performance;

disadvantage – alot of redundant CPU cycles are used to achieve this

II <u>Passive Replication</u> – a software component is installed on multiple nodes such that one component at a time, referred to as the <u>primary component</u> (replica), processes all input messages and generates all output

can <u>only</u> be used for crash faults, otherwise a failed primary can send incorrect output before it is detected

non-primary replicas are called <u>standby replicas</u>, or <u>backup replicas</u> (approach is sometimes referred to as "<u>primary-backup replication</u>")

backup replicas consist of a copy of the code of the software component and a copy of a valid state of the component from which it can be restarted if the primary fails

the primary is responsible for periodically updating the states on the backups, this is called checkpointing and for maintaining a record of requests it responds to in between checkpoints, this is called logging

backup replicas do nothing except receive state updates and monitor primary to detect failures

~~checkpointing techniques will be discussed later in the semester~~

(discuss how checkpointing + logging + replay is used to recreate exact primary state on backup after a failure) } discussed on next page

failure detection

for crash faults, detection of the primary replica's failure can be done using heartbeats

heartbeats are periodic messages sent by the primary replica to the backups indicating that it is still functioning (sometimes called "I am alive" messages)

depending on checkpoint frequency and how fast failure must be detected, the checkpoint messages could serve as heartbeats

note that in an asynchronous system, heartbeats are imperfect failure detectors

backup replicas set a timer each time they receive a heartbeat message

timer value is max. interarrival time of two heartbeats

if timer expires before next heartbeat arrives, the backup assumes that the primary has failed and initiates a recovery procedure

in an asynchronous system, there is no max. message delay and so any timer value will produce some false detections (this might produce some unwanted recoveries but system will still function)

recovery procedure consists of:

1) leader election - distributed alg. to select one of the backup replicas to be the new primary

2) new primary loading checkpointed state and starting to executing replica code

3) reexecuting any client requests made since last checkpoint, this could produce duplicate responses, which clients simply ignore

4) notifying clients of switchover to new primary or rerouting requests automatically to new primary

note that it is possible to rely on clients for failure detection also but external network delays can interfere

III Semi-Active Replication

in theory, passive replication does not require deterministic replicas (although some checkpointing approaches do require this) (Slember, Narasimhan paper also points out situations where non-determinism can "leak" from passively replicated servers, e.g. w/ multi-tiered replication)

active replication typically performs better because overhead of checkpointing can be quite high, also active replication does not incur the substantial recovery time of passive replication

can we have performance of active replication but loosen determinism requirement?

in semi-active replication, one replica called the <u>leader replica</u> produces all outputs (like in passive replication)

∴ semi-active restricted to crash faults also

however, <u>all</u> replicas receive all input messages

non-leader replicas, referred to as <u>follower replicas</u>, execute all deterministic computations locally

leader replica makes all non-deterministic decisions and broadcasts the results to followers

example of non-determinism:

suppose processes use system call to check local time and condition their execution depending on the time

here, the leader will check its local time at each such case and broadcast the result, all followers will then use the leader's local time as the condition and all replicas will make the same decisions

failure detection and leader selection are identical to passive replication

new leader takes over output generation and handling of non-deterministic events immediately after it is elected

leaders must log what outputs they have generated so new leaders know where to take over the output generation