Recap
- Our recipe for group communication:
  - Group membership
    - We solved this by building a fault-tolerant group membership service
    - Everyone who uses it sees the same group “views” in the same order
    - When it makes a mistake about a failure, we just terminate the unfortunate victim!
  - Fault-tolerant view-synchronous multicast
  - Ordering mechanisms

Ordering: The missing element
- Our fault-tolerant protocol was
  - FIFO ordered: messages from a single sender are delivered in the order they were sent, even if someone crashes
  - View synchronous: everyone receives a given message in the same group view
- This is the protocol we called \textit{fbcast}

But we identified other options
- \textit{cbcast}: If \textit{cbcast}(a) \rightarrow \textit{cbcast}(b), deliver a before b at common destinations
- \textit{abcast}: Even if a and b are concurrent, deliver in some agreed order at common destinations
- \textit{gbcast}: Deliver this message like a new group view: agreed order w.r.t. multicasts of all other flavors

Can we implement them?
- First look at \textit{cbcast}
  - Recall that this property was “like” \textit{fbcast}
  - The issue concerns the meaning of a “single sender”
    - With \textit{fbcast}, a single sender is a single process
    - With \textit{cbcast}, we think about a single causal thread of events that can span many processes
      - For example: p asks q to send a, then asks r to send b. So a \rightarrow b but a happens at q and b happens at r!

Single updater
- If p is the only update source, the need is a bit like the TCP “fifo” ordering
- \textit{fbcast} is a good choice for this case
Causally ordered updates

- Events occur on a "causal thread" but multicasts have different senders

Reminder: Who needs it?

- The issue is that with Web Services and CORBA, you might not even "know" that you are invoking a remote object
- If it does a multicast for you, that event seems like something you did... but may have been issued by some other process
- If we use cbcast, messages will be delivered in the order they were sent

How to implement it?

- Within a single group, the easiest option is to include a vector timestamp in the header of the message
  - Only increment the VT when sending
  - Send these "labeled" messages with fbcast
  - Delay a received message if a causally prior message hasn't been seen yet

Causally ordered updates

- Example: messages from p and s arrive out of order at t

Causally ordered updates

- This works even with multiple causal threads.
  - Concurrent messages might be delivered to different receivers in different orders
  - Example: green 4 and red 1 are concurrent
Causally ordered updates

- Sorting based on vector timestamp
  - In this run, everything can be delivered immediately on arrival

Other uses for cbcast?

- The protocol is very helpful in systems that use locking for synchronization
  - Gaining a lock gives some process mutual exclusion
  - Then it can send updates to the locked variable or replicated data
  - Cbcast will maintain the update order

Cost of cbcast?

- This protocol is very cheap!
  - It requires one phase to get the data from the sender to the receiver
  - Receiver can deliver instantly
    - Same cost as an IP multicast or a set of UDP sends
    - Imposes a small header and a small garbage collection overhead
    - Nobody is likely to notice! And we can often omit or compress the header

Better and better

- Suppose some process sends a bunch of small updates using fbcast or cbcast
  - Pack them into a single bigger message
  - Benefit: message costs are dominated by the system call and almost unrelated to size, at least until we get big enough to require fragmentation!
**Screaming performance!**

- This type of packing can give incredible performance
  - Sender is able to send a small message, then "move on" to the next task (like sending a TCP message without waiting for it to get through)
  - Sender’s "platform" packs them together
  - Receiver unpacks on arrival
- Can send hundreds of thousands of asynchronous updates per second in this mode!

**Snapshots with cbcast**

- Send two rounds of cbcast
  - Round 1: "Start a snapshot"
    - Receivers make a checkpoint
    - And they start recording incoming messages
    - Then say "OK"
  - Round 2: "Done"
    - They send back their checkpoints and logs
- Thought question: *why does this work?*

**What about abcast?**

- Abcast puts messages into a single agreed upon order even if two multicasts are sent concurrently
  - fbcast and cbcast can deliver messages in different orders at different receivers
  - Notice that this disordered delivery wouldn’t matter in the cases we discussed!

**Many options...**

- Literature has at least a dozen abcast protocols, and some are causal too
- Easiest just uses a token
  - To send an abcast, either pass it to the token holder, or request the token
  - Token holder can increment a counter and put it in header of message
    - Only need the counter if token can move...
    - Delay a message until it can be delivered in order

**What about gbcast?**

- This is a very costly protocol
  - Must be ordered wrt all other event types, including fbcast, cbcast, abcast, view changes, other gbcasts
  - Used to change a security key or even modify the protocol stack at runtime
    - Like changing the engines on a jet while it is flying! Not a common event
  - Implement with a fusion of flush protocol and abcast. Requires at least 2 phases

**Life of a multicast**

- The sender sends it...
- The protocol moves it to the right machines, deals with failures, puts it in order, finally delivers it
  - All of this is hidden from the real user
- Now the application “gets” the multicast and could send replies point-to-point
Should we ask for replies?

- **Synchronous versus asynchronous**
  - A “synchronous” operation is RPC-like
  - We need one or more replies from the processes that we invoke
  - An “asynchronous” operation is a multicast with no replies or feedback to the caller
  - I.e. “add flight AF 1981 to the list of active flights in sector D-9”. No reply is needed

Should we ask for replies?

- **Synchronous cases (one or more replies)**
  - won’t batch messages
  - Exception: sender could be multithreaded
  - But this is sort of rare since hackers prefer not to work with concurrent threads unless they really have to
  - Waiting for all replies is worst since slowest receiver limits the whole system
  - So speed is greatly reduced...

Life of a multicast

- **Asynchronous multicast**
  - Pros and cons
  - Asynchronous multicast allows higher speeds
    - The system can batch up multiple messages into one big message, as we saw earlier
    - And the sender won’t be limited by the speed of the network and the receivers
  - This makes asynchronous multicast very popular in real systems
  - But the sender can get “way ahead” and this can cause confusion if it then fails
  - Multicasts still in the channels can be lost

Asynchronous confusion...

- **Insight** is that these red multicasts were unstable
  - If we flush the channels and wait until they have been delivered (become stable), the issue is eliminated
  - Users find this easy to understand because file systems work the same way
    - File I/O is asynchronous through the buffer pool... must use fsync to force writes to disk

Remedies for confusion
Asynchronous confusion...

Flush protocol runs here, pushes data through the channels.

Application invokes flush, but only when it is about to talk to the outside world.

Limits to asynchrony

- At any rate, most systems limit the number of asynchronous multicasts that are running simultaneously
  - Issue is that otherwise, sender can get arbitrarily far ahead of receivers
  - A few messages is one thing... millions is another
  - So most systems allow a few asynchronous messages at a time, but then force new multicasts to wait for some old ones to finish
  - Very similar to TCP window idea

Picking between synchronous and asynchronous multicast

- With synchronous multicast we can "ask" the receivers to do something
  - Please search the telephone book
  - With k members at the time of reception, the group member i searches the i'th part of the book (dividing it into k parts)
  - Each reply has 1/kth of the answer!
  - But we need to wait for the answers
  - This is a shame if we didn't actually need answers

A range of synchrony levels

- A platform usually offers multiple options
  - Wait for k replies, for some specified $k \geq 0$
    - Waiting for no replies: asynchronous
    - Wait for "all" to reply
  - When we say "all":
    - This means "one reply from each member in the view at the time of delivery"
    - If someone gets the message but then fails, obviously, we should stop waiting for a reply....

Recap

- We've got a range of ordered multicast primitives
  - Two (fbcast, cbcast) have low cost
  - Two (abcast, gbcast) are more ordered but more costly
  - And we can use them asynchronously or synchronously
  - Now touch some "esoteric" issues...

Orphaned messages

- With all of these protocols a failure can leave messages "orphaned"
  - E.g. $a \rightarrow b$, but after failure a has been completely lost and someone still has a copy of b (presumably delayed)
  - Similar issue can arise with abcast
  - Modify flush protocol to discard such messages
Dynamic uniformity ("safe")
- Suppose that process p receives message a, delivers is, then fails
  - Application program may have done something, like "issue cash from the machine"
- Now system could "lose" a message after the failure
  - Nobody else will see this message

Dynamic uniformity ("safe")
- We say that a multicast is "safe" if a message delivered to any process will be delivered to all processes (unless they crash first)
- To guarantee this for every multicast is expensive
  - Requires two phase protocol
  - First make sure that everyone has a copy
  - Only then start to deliver copies
- This is quite slow!!!!!

Is this form of safety needed?
- Perhaps not:
  - Many actions only impact the "internal" state of a system
    - Like reports of load, updates to variables employed by algorithm, etc
  - Relatively few multicasts have external visibility
  - We only need dynamic uniformity when something will be visible outside the system

Is this form of safety needed?
- Moreover, can easily hack around issue
  - The same flush primitive we mentioned earlier can solve this problem
  - Just call it when you need to take an external action
  - Seems unnecessary to provide such a costly property for every multicast when there is such a simple alternative

Communication from a client to a group
- Some communication occurs entirely within a group
- But other requests come from outside (from a "client")
- What issues does this raise?

Communication from a client to a group
- It turns out that we can implement client-to-group multicast fairly easily
  - Either hand the request off to a member, who does it for you. Involves a small delay
  - Or cache the membership and label the multicast with the view in which it was sent
    - Some trickiness when view is changing just at this moment... book explains how it can be handled... at worst, client has to retry
    - But multicast goes directly to the members... no delay
Wrapup

- We’ve seen how this stuff works
  - Hopefully, someone else will implement it for you and you’ll use it via a library!
  - Spread and Ensemble are examples
- What are the pros and cons?
  - Pro: a powerful abstraction
  - Con: not trivial to understand or use

Arguments for “platform support”

- ... sometimes, GCS is found in the O/S
  - In IBM Websphere, virtual synchrony is used in a replication package
  - In Microsoft Windows Clusters, group communication is employed within the cluster management technology
- But not often visible to end user
  - Considered a “dangerously powerful tool”

Take-aways?

- We can implement very high performance multicast
  - Virtual synchrony model
  - Incredible asynchronous throughput
  - Ordering matched to the needs of app.
- And many vendors have done so
- But developers aren’t able to access these primitives (for now)