CS514: Intermediate Course in Computer Systems

Lecture 9: Sept. 26, 2003

Overview of reliability techniques

Outline

CS514

- This lecture is designed to help you think about how to approach your project reliability requirements and mechanisms
  - Noting that some of you have “artificial” requirements

What requirements do we care about

CS514

- How much “Fault-tolerance” or “Reliability”
  - Stuff still works when something breaks
- How much Availability
  - How long it takes for stuff to still work after something breaks
- These two will drive our thinking

Other requirements (mentioned in Kens book)

CS514

- Recoverability
  - How a failed component can rejoin the system
  - Really part of “fault tolerance”
- Consistency
  - Coordinating related actions by multiple components
  - Really part of “availability”

And still more…

CS514

- These are things you'd consider for any software development…they only get harder with distributed systems
  - Scalability
  - Security
  - Privacy
  - Correct specification
  - Correct implementation
  - Predictable performance
  - Timeliness

Replication, replication, and replication

CS514

- Fundamentally there is only one way to get fault tolerance and availability
  - Replicate
- But many ways to replicate
  - Mainly what this lecture overviews
- For performance reasons, may also require replication when using parallel or distributed techniques

A theoretical diversion

CS514

- Engineering issues are messy
- Yet, we want to think precisely about fault tolerance
- Therefore, theory folks derive simplistic models for failure
- These are good to understand for "how to think" about fault tolerance
  - But don't apply much to the real world

Theoretical failures

CS514

- Halting
  - A process just stops without doing anything wrong
  - Dynamic memory is lost, stable memory is not (and is correct up to point of failure)
- Fail-stop
  - A halting process where the failure is reliably reported to all other processes
- Byzantine
  - A process may exhibit any behavior, including malicious

Theoretical system models

CS514

- Asynchronous model
  - No clock synchronization, messages and processes may take arbitrarily long
- Synchronous model
  - Clocks are synchronized, message and process times are bounded
  - All messages and processing can happen in well-defined “rounds”

Upper and lower bounds on solvability and impossibility

CS514

- Byzantine failures and asynchronous systems are unrealistically hostile
  - If problem is solvable here, probably you can do at least as well in the real world
- Fail-stop failures and synchronous systems are unrealistically benign
  - If problem is impossible here, then you are unlikely to have any luck in the real world

What do your failures look like?

CS514

- In designing your reliability solution, think hard about what kinds of failures you may have
- Design rule of thumb:
  - First eliminate or simplify specific point failures where possible/cost effective
  - Only after this should you design system-level features that can tolerate the remaining types of failures

Example I: Process failures

CS514

- Real software failures are not quite byzantine, but they are very unpredictable
- Use monitoring procedures that make them look like fail-stop
  - Don't ask a process if it is alive, rather ask it to perform a predictable task
    - Write/read a database entry
    - GET a web page
  - If task not performed right, or with too much delay, declare the process dead

Example I: Process failures

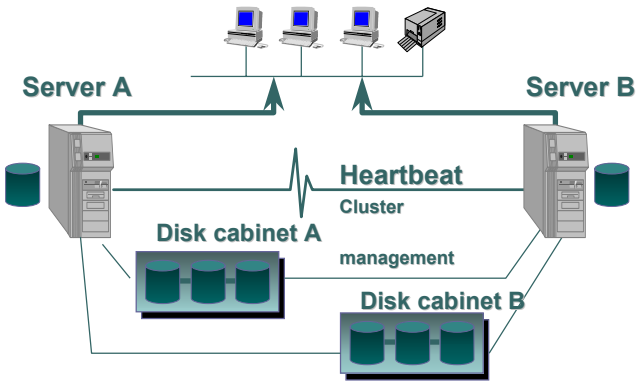
CS514

- Keep your monitoring process very simple, and test the hell out of it
  - I.e., eliminate possibility of a monitoring process failure
  - (Ultimately the turtle has to stand on something!)
- In other words, don't over-design your monitoring task
  - (but don't under design it either)

Example II: MSCS Cluster

CS514

Why is the server replicated, but not the disk???



Example II: MSCS Cluster

CS514

- Disk operation is relatively simple
  - Scan, read, write, …
  - Failures can be well-defined and detected (parity errors etc.)
  - You can build a (pretty) reliable disk
- Windows software on the other hand…

Example II: MSCS Cluster

CS514

- Microsoft followed the rule of thumb:
- Eliminate specific point failures where possible
  - The disk
- Design a fault-tolerant system where not possible
  - Server failover

Questions to ask about replication

CS514

- Do you even need replication?
- If multiple active replicas, do they all have to be perfectly identical?
- If backup (non-active) replica, does failover need to be 100% accurate?
- How many replicas do you need?
- Where do they need to be?
- How much non-availability can you tolerate during failover?

Do you even need replication?

CS514

- Consider this scenario:
  - Company has 10 national offices
  - Each office needs to query and update the customer database frequently, reliably, and with little delay
- Solution:
  - Replicate the database at all 10 sites?

Do you even need replication?

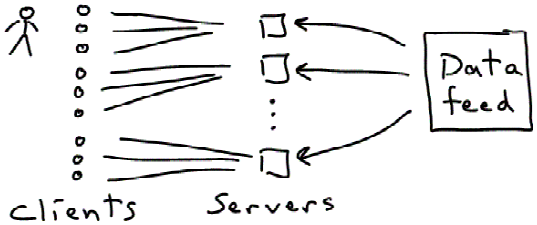
CS514

- Digging deeper:
  - Each site typically queries/updates for local customers (not those at other sites)
  - Queries about other site customers turn out not to be time critical
- Better solution:
  - Partition database into 10 different local partitions
  - Use redundant networking (e.g. multihoming) to maximize access to other partitions

Do active replicas need to be perfectly identical?

CS514

- If Google, they don't
  - There is no "right" answer
- If financial data feeds to stock brokers, they had better be identical quickly and reliably
- Other applications may fall in-between
  - Eventual consistency, probabilistic reliability
  - Epidemic (gossip) approach



Does failover have to be perfect (no lost data)?

CS514

- Amazon (lots of small transactions, at least used to be!), but huge volume
- Large volume makes it hard to have perfect failover
- Fortunately, occasional errors not that big a deal
  - Customer will complain, just take customer's word
    - If no history of complaints
    - Or if time of error coincides with known failure
  - Alternatively keep a local log on disk and consult only to resolve complaints

Does failover have to be perfect (no lost data)?

CS514

- Million dollar bank transfers, on the other hand…
  - Fortunately volume tends to be smaller
  - Can afford a higher per-transaction cost to pay for perfect failover
- Again, other applications may fall in between these extremes

How many replicas do you need?

CS514

- Try to have as few as possible, but no fewer!
- Answer depends on a lot of factors
  - Reliability of components, cost of downtime, restoration speed, …
- But usually the answer will be one (if not zero)
  - If the answer is not one, try to make it one!

Where do the replicas need to be?

CS514

- Same room
  - Survive equipment failure, but not a fire
- Across the street
  - Survive fire, but not an earthquake
- In another city . . .
  - But network latency or bandwidth may kill you . . .
  - You might need this for performance reasons even if not fault-tolerance reasons

Where do the replicas need to be?

CS514

- At Tahoe, I saw major wireless carriers with only a single data center
  - Replication within the data center, but no geographically distant replica
  - They wanted to, but just didn't have the budget, and weren't sure they could do it
- One option:
  - Two "partitioned" data centers, with capability to fail-over within hours
    - Fail-over managed by humans

What are your availability requirements?

CS514

- Can you wait:
  - One minute?
  - Ten minutes?
  - One hour?

  for the backup to take over operations…
- True continuous availability scenarios are relatively rare, but certainly exist
  - Air traffic control
  - Weapons systems on a battleship
  - Online games

A strong but typical fault-tolerant situation

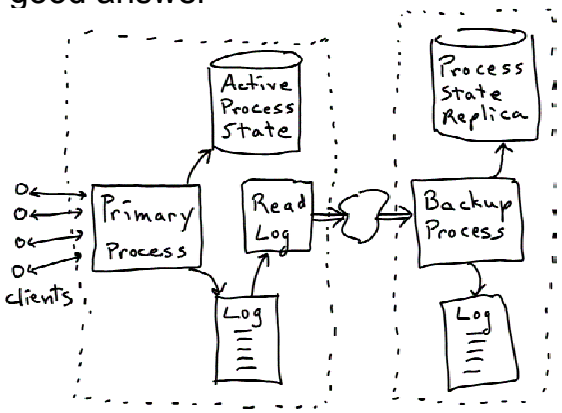
CS514

- One replica is adequate
  - Physically separated
    - In another building across the street
  - Strong attention to redundancy at all levels
    - Redundant network, power, air conditioning, etc.
- Performance can be handled with a single server
  - Albeit maybe very high-end
  - So hot-standby model ok
- Non-availability on the order of a few minutes
- Occasional error acceptable, as long as it can be recovered later

A strong but typical fault-tolerant situation

CS514

- Asynchronous log-based replication is a good answer



Basic procedure

CS514

1. Primary process reliably writes to log
   - The log is the authoritative state!
2. Primary process updates active process state
3. Read log process ships latest log updates to backup process
   - With some delay
4. Backup process reliably writes to log
5. Backup process updates process state replica

Asynch versus sync

CS514

- Procedure is asynchronous . . . primary "commits" to its log and process state without an ack from the backup
  - Primary and backup logs will be out-of-sync
  - Even 100ms out of sync may mean a thousand or more transactions
- But….synchronous performance would suck by comparison

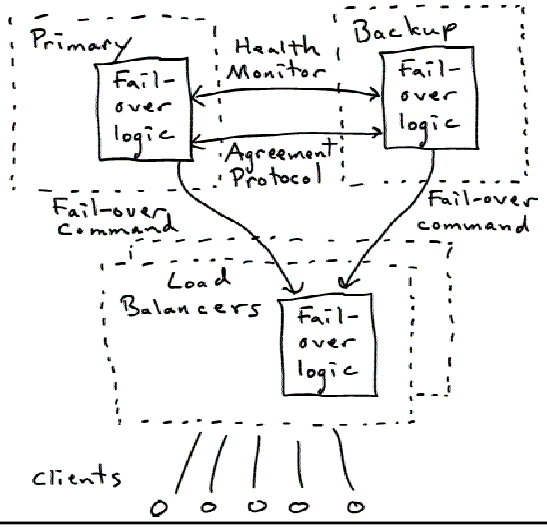
Issues in asynchronous log-based replication

CS514

- How to decide when to fail-over
  - And if/when to switch back to primary after it is healthy
- How to redirect clients after fail-over
- How to avoid both servers being active
  - How to merge process state if both server can be active!
- How to sync up logs and process state after a partition or failed server resumes
  - Including gaps in the log
- How to roll-back to stable process state (if necessary) after failover

Simple fail-over protocol and client redirection

CS514



Simple fail-over protocol and client redirection

CS514

- Involves primary, backup, and load-balancer
- If crash, backup (or primary) can direct load balancer to switch when it is ready to take over
  - I.e. after roll-back is finished
- If primary is up, it directs timing of fail-over
  - I.e., after process state is synchronized

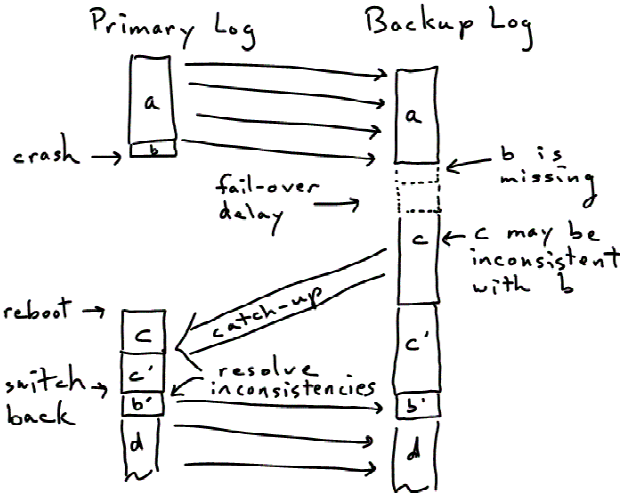
Avoiding two active servers

CS514

- Would only happen if servers can both talk to the load balancers, but not each other
  - Can design local network so that this is virtually impossible
- *Harder to do across the Internet*
  - Though in this case, load balancing might be DNS, which would itself be replicated and might get confused!

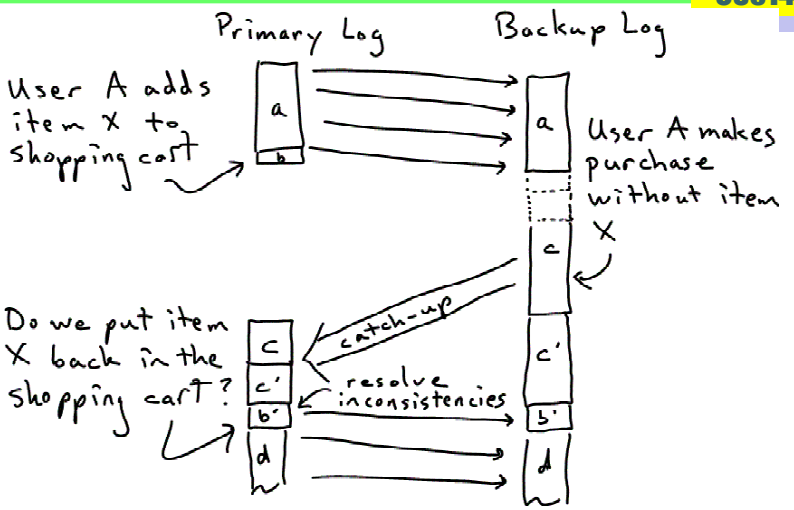
Synching up logs

CS514



Issues with gaps in log

CS514



Rolling back to stable process state

CS514

- Transactions consist of “tentative” reads and writes followed by a commit or an abort
  - Two-phase commit (2PC)
- Log may contain reads and writes, but not the commit (or abort)
- After failover, backup may abort the transaction
  - Unlikely you’d want to try to complete the transaction at that point

Rolling back to stable process state

CS514

- There may be thousands of uncompleted transactions
- Rollback process may take minutes as a result

Revisiting our typical fault-tolerant situation

CS514

- One nearby replica, single primary server, brief non-availability and occasional error ok
- If some of these assumptions are too weak, then things get more interesting!
  - Multiple replicas
  - Multiple active servers
  - Geographically distant servers
  - Continuous or near-continuous availability
  - No errors

Stronger requirements

CS514

- Stronger requirements takes us into the realm of *group communications systems*:
  - Membership tracking
  - Multicast with various ordering and delivery guarantees
  - Virtual synchrony

What are Group Communications Systems?

CS514

- Simply put:
  - *A group of processes, all of which can communicate with all others*
- But:
  - In such a way that all members will respond identically (and correctly) to external systems
  - And in a way that is (to the extent possible) transparent to external systems
  - (oh, and with good performance…)

How can multiple processes act identically?

CS514

- They have exactly the same state at the same time…
  - This is the crux of the problem
- But, who are *they*?
  - Membership changes over time
- And when is "*the same time*"?
  - System clocks are inaccurate and not synchronized
- And how can any process be sure the others have the *same state*?

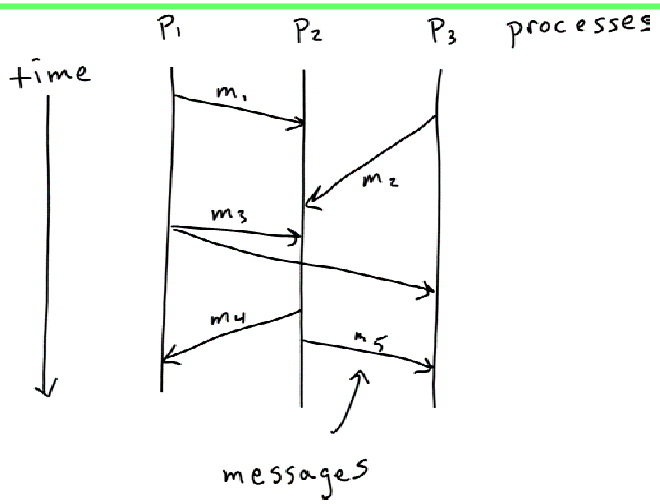
These are hard questions

CS514

- Even though a relatively small number of real world applications require these kinds of guarantees . . .
- These applications are important
  - Military, transportation, energy . . .
- And anyway, if you can understand this stuff, the simpler problems will be that much easier

First, lets consider Time . . .

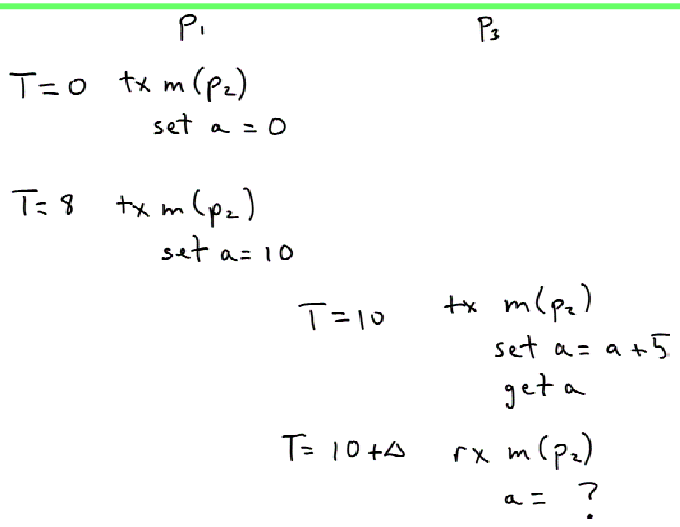
CS514



Picture with 3 processes

The log files of p1 and p3

CS514



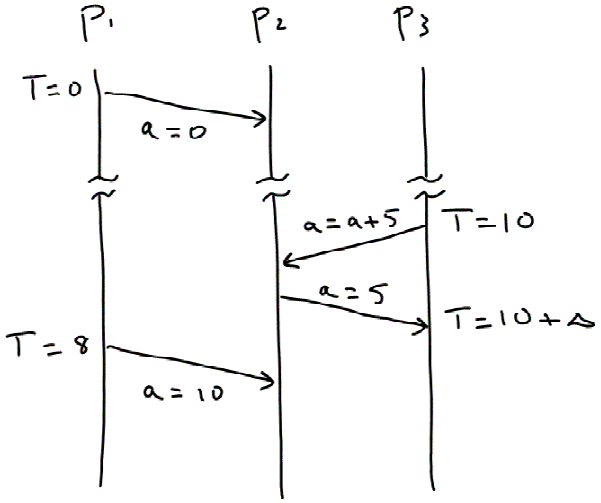
What is the value of a?

CS514

- We don't know!

Because the clocks are unsynchronized…

CS514



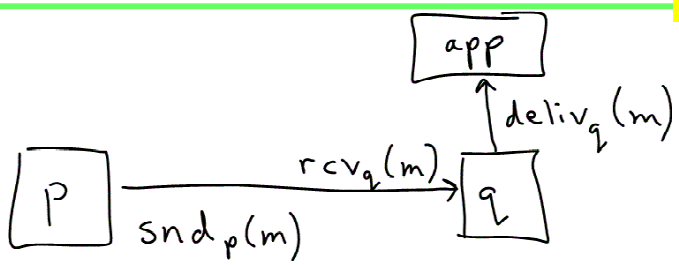
Group Communications Systems provide ordering

CS514

- Through concept of causality
  - *a* causes *b*
- Lamport's notations:
  - $snd_p(m)$, $rcv_p(m)$, $deliv_p(m)$
  - Three message events
    - Process *p* sends message *m*
    - Process *p* receives message *m* (by lower layer)
    - Process *p* delivers message *m* (from lower layer to application)

Some messages are obviously ordered

CS514

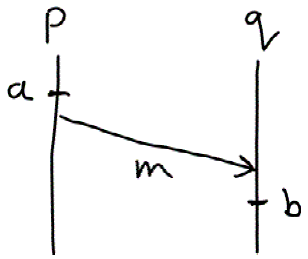


- $snd_p(m)$ happens before $rcv_q(m)$
- Denote this as:
  - $snd_p(m) <_m rcv_q(m)$

And this defines a partial ordering of events

CS514

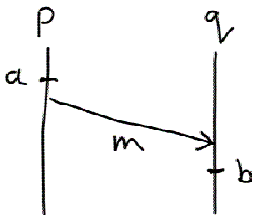
- Event *a* at process *p* happened before $snd_p(m)$
- $rcv_q(m)$ happened before event *b* at process *q*
- Therefore, event *a* happened before event *b*
- Denote as: $a \rightarrow b$
  - Concurrent means neither $a \rightarrow b$ nor $b \rightarrow a$



$a \rightarrow b$ doesn't mean "*a* caused *b*"

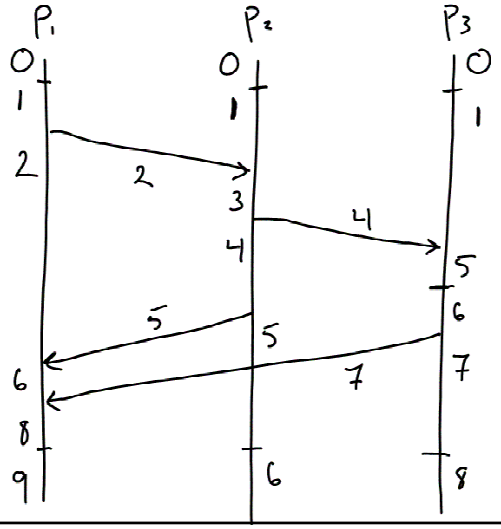
CS514

- For all we know, *m* is just a keep alive ping unrelated to events *a* and *b*
- But we can say that *a* has *potential causality* for *b*
- We can build *logical clocks* using this notion



A simple logical clock

CS514



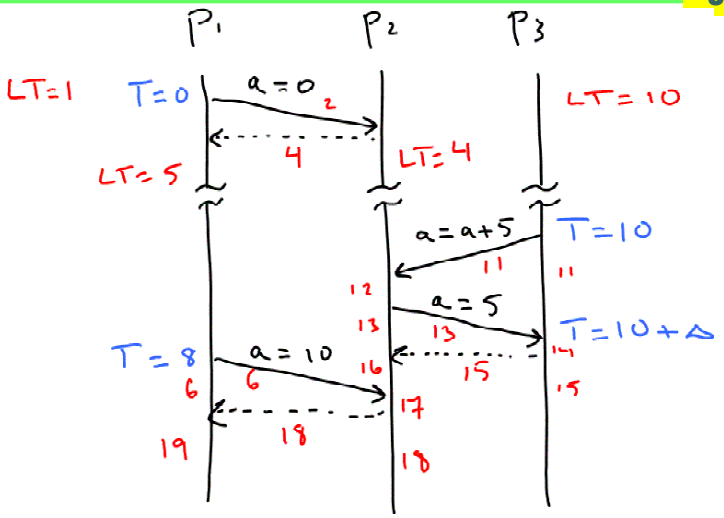
A simple logical clock

CS514

- Each process and message has a logical time (LT_p and LT_m)
- Set LT_m of transmitted messages to that of process LT_p
- If received message LT_m > process LT_p, set process LT_p to LT_m +1

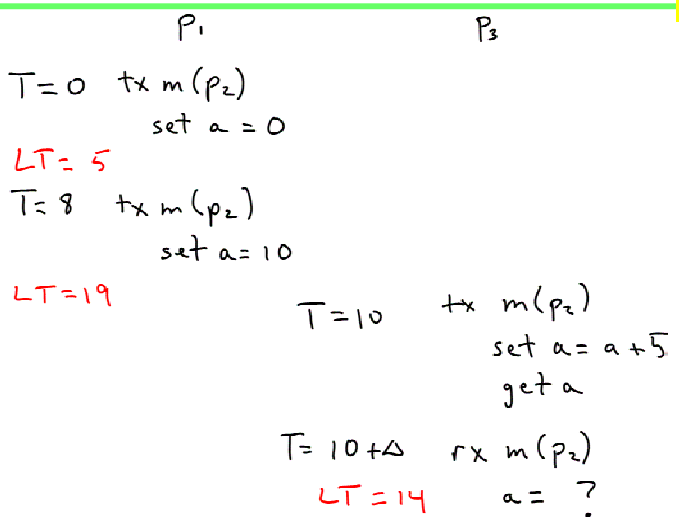
Our simple example revisited with logical clocks

CS514



And their corresponding logs

CS514



Next, let's consider Multicast

CS514

- Multicast: one system transmits the same message to multiple other systems
- Clearly needed to provide consistent state among a group of communicating systems
- There are lots of flavors of multicast

What about IP multicast?

CS514

- IP multicast model:
  - Individual members join a multicast group through interaction with local router
  - Members don't know who is in the group
  - Any member can transmit IP packets to the group
  - Delivery is unreliable: any member may or may not receive a multicast
  - Delivery is unordered
- Essentially an extension of LAN multicast

IP Multicast clearly inadequate for our purpose

CS514

- We need reliable delivery
- We need ordered delivery
  - "Go up, turn right" different from "turn right, go up"
- Ok, so why not add sequence numbers to packets, and receivers send acks to sender?

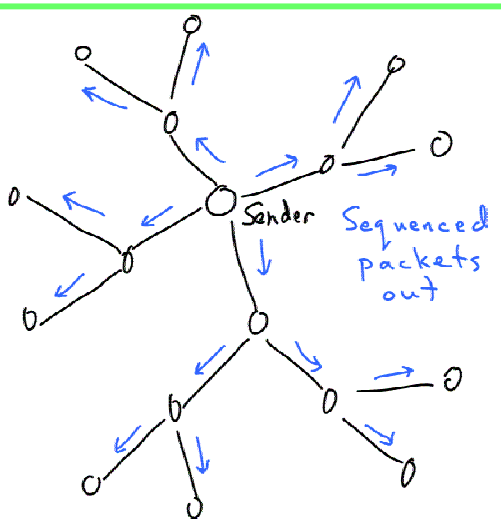
Reliable IP multicast

CS514

- Multicast version of TCP
- Sequenced IP packets traverse outgoing multicast tree to receivers
- ACKs (or NAKs) of IP packets "implode" uptree from receivers towards sender
  - May be aggregated at intermediate points
  - Retransmissions may come from intermediate points
  - Lots of clever schemes have been designed

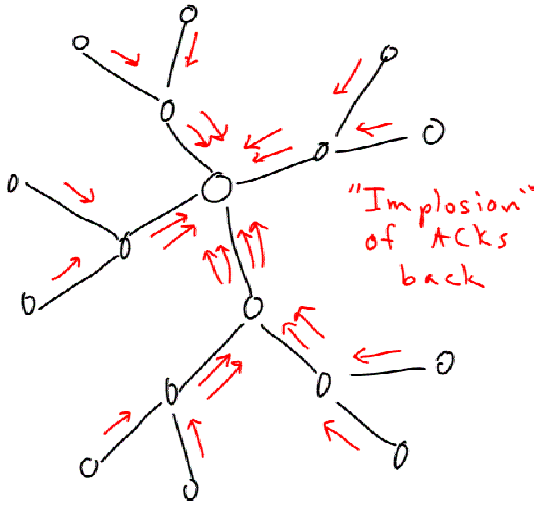
Reliable IP multicast example 1/4

CS514



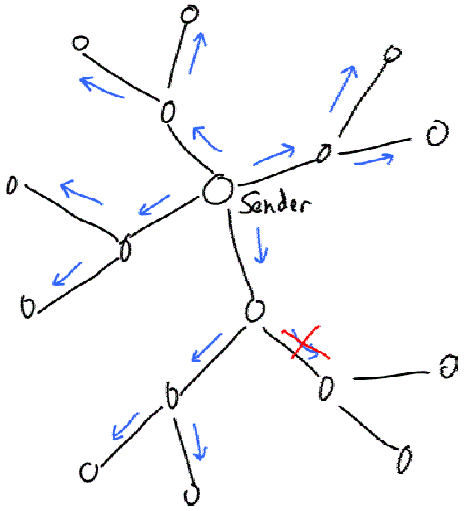
Reliable IP multicast example 2/4

CS514



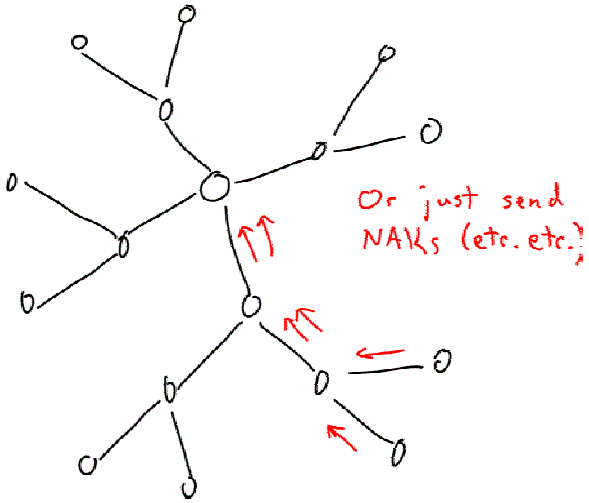
Reliable IP multicast example 3/4

CS514



Reliable IP multicast example 4/4

CS514



Ultimately Reliable IP Multicast wrong model

CS514

- TCP: If sender dies, nothing can be done
  - "Reliable" not defined for case where sender dies
- Reliable multicast: Same thing!
- If group member X says a plane is entering an air space, other members better learn that even if X crashes immediately after transmission!

We need stronger "reliability" models

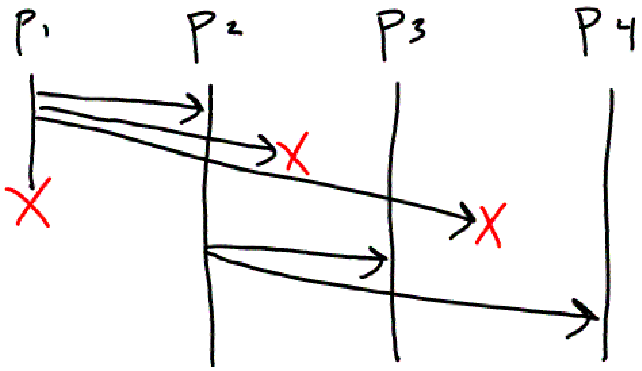
CS514

- Dynamically Uniform:
  - If any member delivers message (even if it crashes immediately afterwards), all alive members will deliver it to app
- Failure-atomic:
  - If any alive member delivers a message, all alive members will deliver it
  - But, even if no alive member delivers it, some crashed members may still have delivered it

Failure-Atomic

CS514

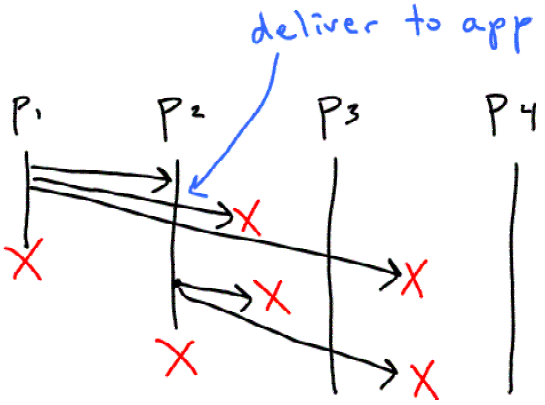
- If any alive member delivers a message, all alive members will deliver it



Failure-Atomic

CS514

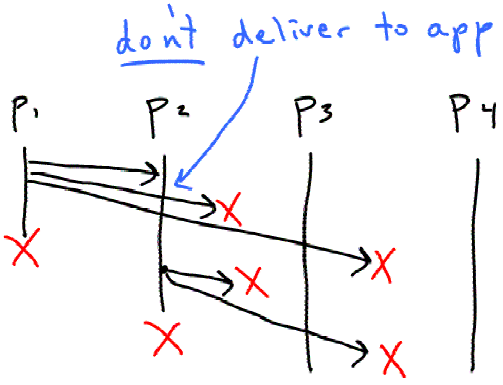
- But, even if no alive member delivers it, some crashed members may still have delivered it



Dynamically Uniform

CS514

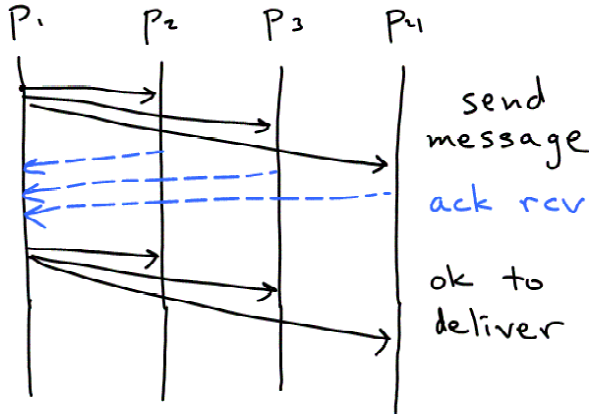
- If any member delivers message (even if it crashes immediately afterwards), all alive members will deliver it to app



Dynamically Uniform

CS514

- Implemented as a 2-phase process
  - So substantially slower than failure-atomic



And richer ordering models

CS514

- fbcast: fifo ordered by sender
- cbcast: causally ordered, but concurrent message order not defined
  - Different members may see concurrent messages in different order
- abcast: totally ordered (but not causally ordered)
  - Used for membership services
- cabcast: totally and causally ordered

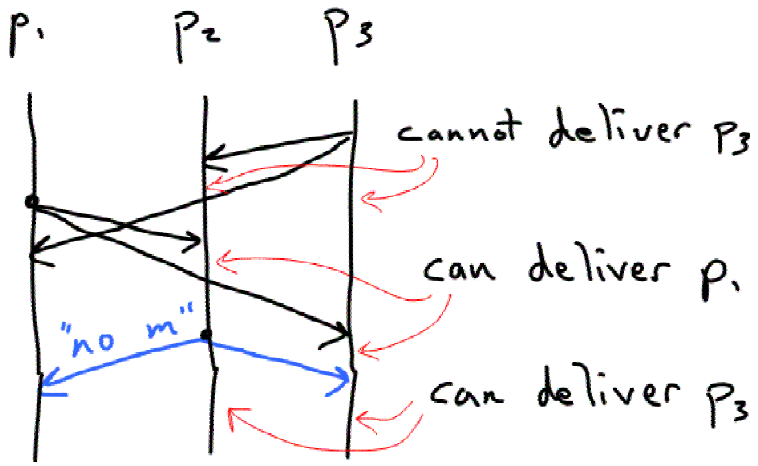
Total ordering? (aka "agreed ordering")

CS514

- Define a lexicographic ordering
  - i.e. {LT, process ID}
- Collect all messages at a given LT
  - This means that each process must explicitly state that they won't send any more messages for a given LT!
- Order by process ID and deliver

Total ordering is slow

CS514



Next, lets consider Membership…

CS514

- Core to any group communications system is the notion of membership
- Reason is obvious:
  - If all members of a group communications system don't agree on the membership, they cannot possibly insure consistent state among members

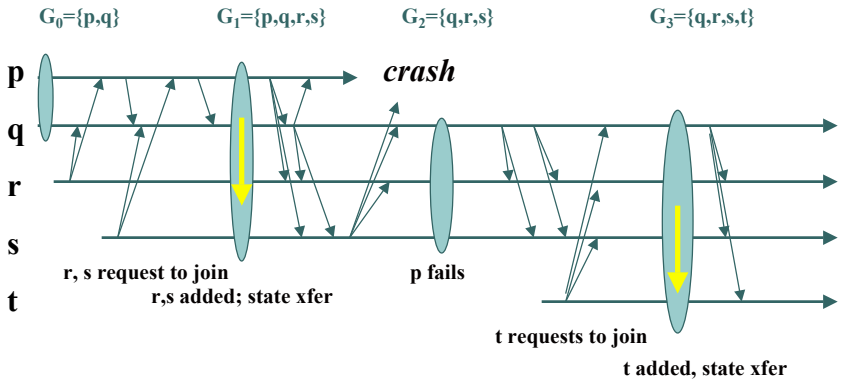
Membership "views"

CS514

- Members may join and leave a group
  - Leaving may be announced or abrupt (a crash)
- The list of members at a given time is known as the "view"
- Group multicast is always with respect to the current view
  - Before a new view is established, multicasts in the current view are completed

A series of views

CS514



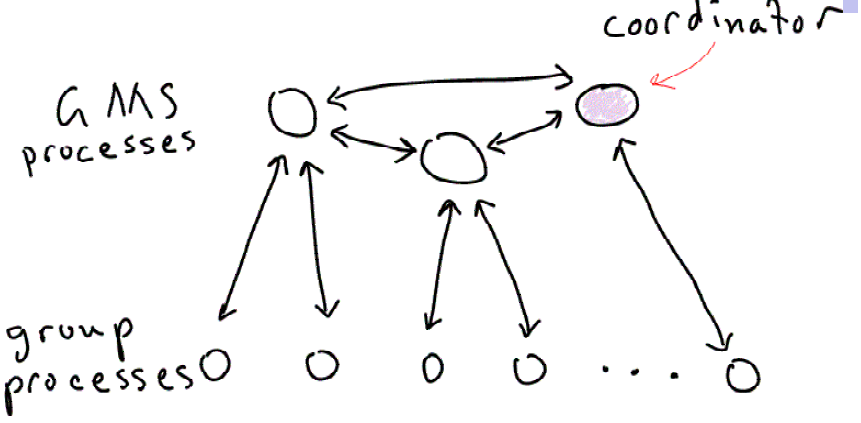
Group Membership System (GMS)

CS514

- Views are managed by a GMS
- A GMS is a small number of processes that:
  - Monitor processes in the group
  - Force a faulty process out of the group
  - Notify the group of new views
- The GMS will dynamically select a coordinator among themselves to be the arbiter

Group Membership System (GMS)

CS514



Group Communications Systems

CS514

- Spread (www.spread.org)
  - John Hopkins
  - I recommend you use this
  - Programming APIs for C/C++, C#, Java, Perl, Python, and Ruby
- Ensemble, Horus, Isis
  - Cornell (Ken Birman)
- Others
  - Relacs, Transis, Totem, Phoenix, xAMP

Spread

CS514

- Allows processes to organize into groups (join and leave)
- Manages multicast communications in the group
- Manages group membership
- LAN and WAN operation
  - (you'll just deal with LAN in the project)

Spread API

CS514

- Get group membership (callback)
- Join and leave group
  - First join creates group, last leave destroys group
- Send to group
  - Unreliable, reliable, and SAFE
    - SAFE: Not delivered until all alive processes have received it. Uses Agreed ordering
  - FIFO, Causal, and Agreed

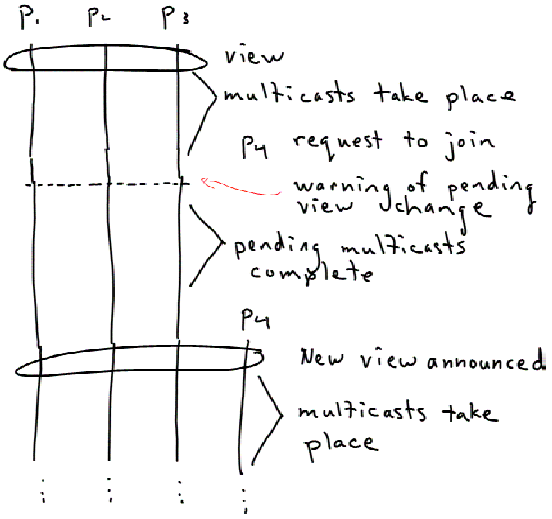
Spread API: Receive Events

CS514

- Receive message
- Receive membership change
  - If add, application can
- Receive pending membership change "warning" (trans_memb)
  - Indication that all subsequent messages, up to next membership change, are completions of current multicasts

Spread View Change

CS514



Summary

CS514

- Eliminate failure in individual components where possible
- Use single-primary, hot-standby approach where possible
  - Avoid too much synchronization if possible
- Use group communications concepts otherwise