CDS 270-2: Lecture 1-3
Message Transfer Architectures

Richard M. Murray

31 March 2006

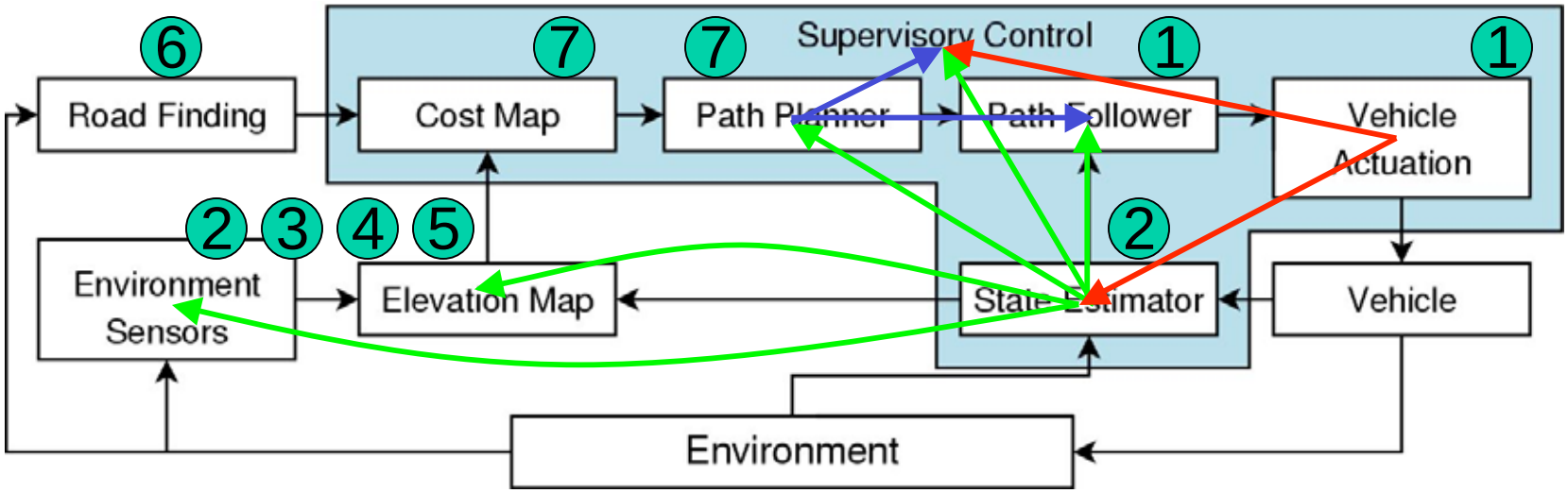
Goals:

- Discuss choices in NCS message delivery
- Describe *Spread*, a group communications toolkit
- Discuss event ordering in distributed systems

Reading:

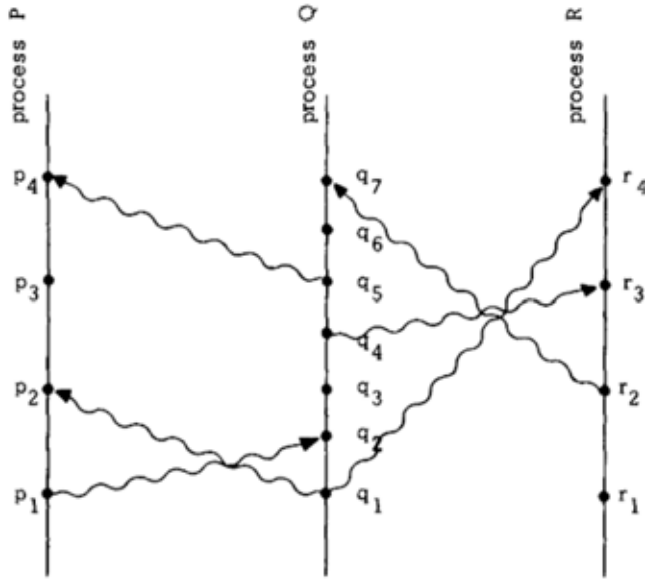
- "Time, Clocks, and the Ordering of Evvents in a Distributed System", L. Lamport. *Comm. ACM*, 1978
- "A User's Guide to Spread", Jonathan R. Stanton, 2002.

Communication Management: Spread



Message type	Bytes/Freq	Recv	Comments
Vehicle State	~250 B @ 40 Hz	15	Pos, vel, acc; highest update rate
Actuator State	~220 B @ 30 Hz	3?	Actuators + OBD II information
Elevation Map	4 MB @ 10 Hz	0	Not transmitted
Cost Map	4 MB @ 10 Hz	3	Deltas transmitted
Trajectory	??? @ 5 Hz	2	Variable size (I think)
Cameras	640x480 @ 30 Hz	5	Firewire (~20 MB/s per camera)

Causality in Distributed Communications (Lamport, '78)



Partial ordering: $a \rightarrow b$

- If *a* and *b* are events in the same process, then $a \rightarrow b$
- If *a* is the sending of a message by one process and *b* is the receipt of the same message by another process, then $a \rightarrow b$
- If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$
- $a \rightarrow b$ means "*a* can causally effect *b*"

Logical Clocks

- Let $C_i\langle a \rangle$ be a clock for process P_i that assigns a number to an event
- Define $C\langle b \rangle = C_j\langle b \rangle$ if b is an event in process P_j
- *Clock condition*: for any two events *a*, *b*: if $a \rightarrow b$ then $C\langle a \rangle < C\langle b \rangle$

Remarks

- Events are *partially* ordered: can compare some events but not all events
- Example: $p_1 \rightarrow q_3$ but p_3 and q_3 are no related
- Clocks are not unique (can choose any set of integers with appropriate relations)

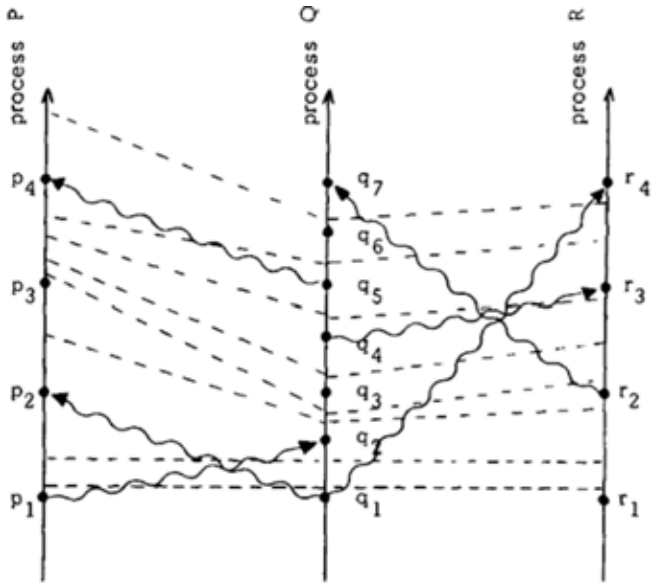
Implementing a Clock

Conditions for a clock

- C1: If *a, b* are events in process P_i and *a* comes before *b*, then $C_i\langle a\rangle < C_i\langle b\rangle$
- C2: If event *a* is the sending of a message by process P_i and event *b* is the receipt of that message by process P_j, then $C_i\langle a\rangle < C_j\langle b\rangle$

Space-Time Diagram

- Add ticks for every count in each process
- Draw "tick lines" between equally numbered ticks
- C1 ⇒ tick line between two events
- C2 ⇒ every msg must cross tick line



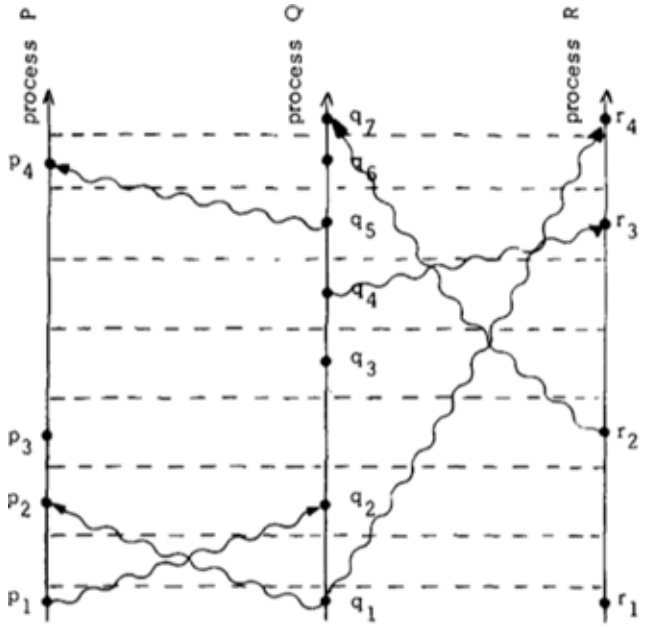
Remarks

- Events can shift around between tick lines without changing logical clocks ⇒ logical time is different than physical time

Constructing Clocks

Implementation rule

- IR1: Each process P_i increments C_i between any two successive events
- IR2:
  - (a) If event *a* is the sending a message *m* by process P_i, then the message *m* contains a timestamp $T_m = C_i\langle a\rangle$
  - (b) Upon receiving a message *m*, process P_j sets C_j greater than or equal to its present value and greater than T_m



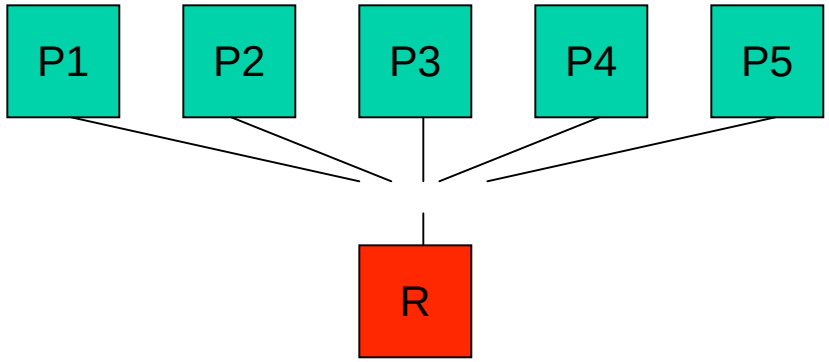
Remarks:

- Gives an easy algorithm for constructing a clock
- Note that $C\langle a\rangle < C\langle b\rangle$ does *not* imply $a \rightarrow b$. Still only a partial order (can only compare certain elements)

Total order

- Order events according to logical clocks
- Break ties using process number
- Allows any two events to be compared
- Total ordering is not unique (depends on choice of clocks)

Example: Resource allocation



Problem description

- Fixed processes P_i sharing resource R
- Once a process grabs a resource, it must release it before it is use again
- Requests granted in order they were requested
- Every request is eventually granted

- Solve in *distributed* way; processes agree on who goes next
- Problem is non-trivial, even with central scheduling (see Lamport paper)

Algorithm

1. P_l sends message T_m:P_i *request* to every other process and puts message on its queue
2. P_j queues all requests and sends timestamped acknowledgement to sender
3. Process P_i uses resource when
   - T_m:P_i *request* is ordered before any other request in queue (according to total order)
   - P_i has been received ack from everyone with timestamp > T_m

- P_i removes T_m:P_i *request* message from queue and sends T_m:P_i *release* message to everyone
- When P_j receives a T_m:P_i *release* message, it removes message from its queue

Group Messaging Systems

Group

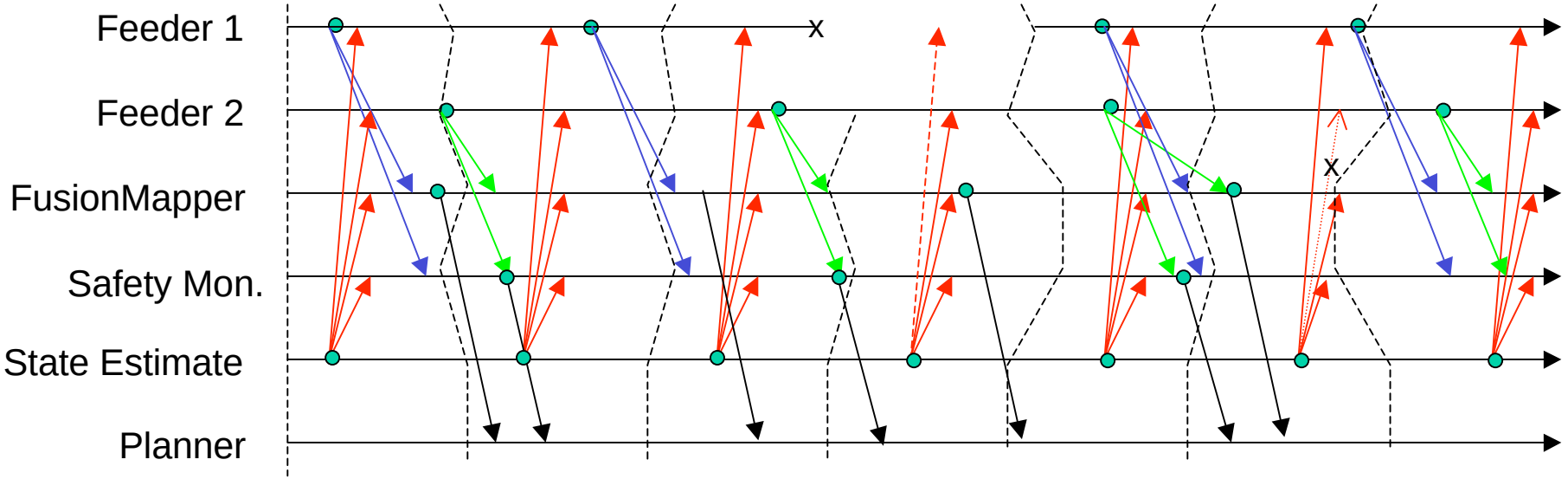
- Collections of processes that can send messages back and forth to everyone
- Messaging system has to keep track of people joining and leaving groups
- Goal: deliver packets reliably and *causally*

Ex: Alice NCS group message types

- Modules receive certain message types

Issues

- Need to track membership over time
- Need to provide different levels of reliability (at the group level)
- Need to provide different levels of ordering (or causality)
- Also need to keep track of the fact that time may be different on different computers (no global clock)



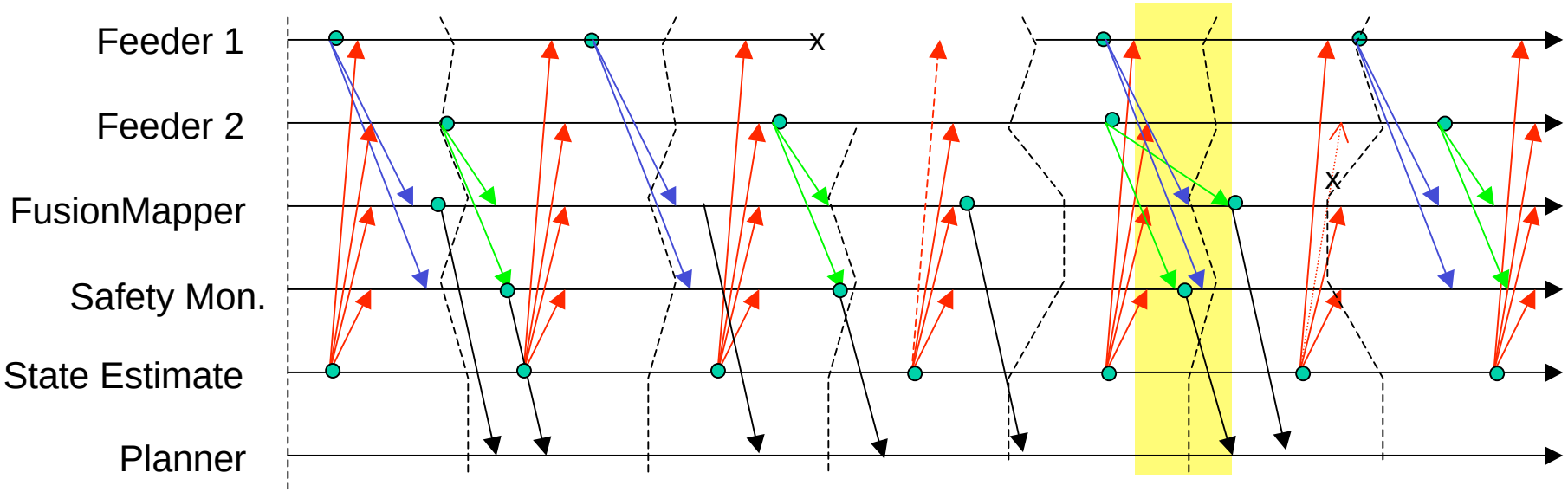
Message Ordering ("Virtual Synchrony")

Ordering

- *None* - No ordering guarantee.
- *Fifo by Sender*- All messages sent by this sender are delivered in FIFO order.
- *Causal* - All messages sent by all senders are delivered in Lamport causal order.
- *Total Order* - All messages sent by all senders are delivered in the exact same order to all recipients

Remarks

- Imposing causality increases message overhead; need to make sure that everyone has the message
- Things get interesting with multiple groups - everyone in same collection of groups should receive all messages in same order
- HW: figure out an example where causal and total order are different



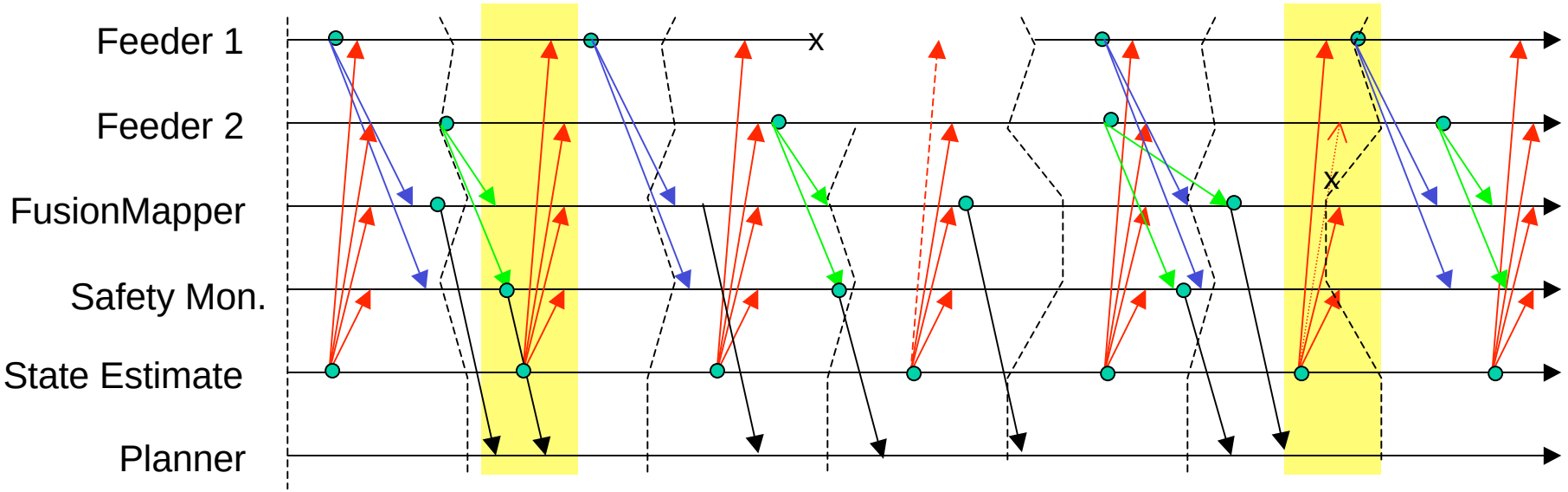
Message Reliability ("Extended Virtual Synchrony")

Reliability

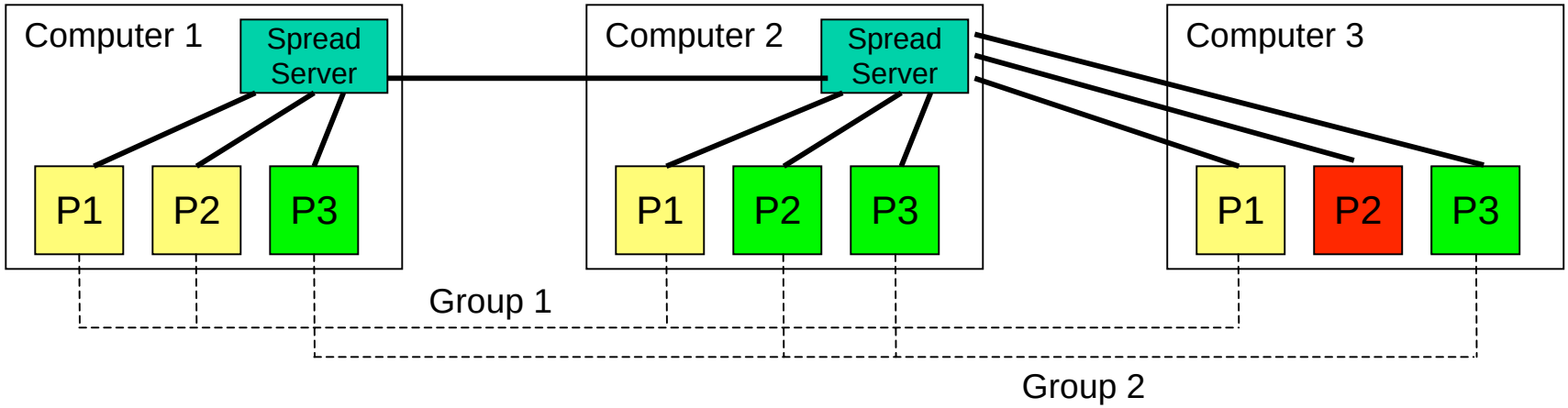
- *Unreliable* - Message may be dropped or lost and will not be recovered.
- *Reliable* - Message will be reliably delivered to all recipients who are in group to which message was sent.
- *Safe* - The message will ONLY be delivered to a recipient if everyone currently in the group definitely has the message

Remarks

- Key issue is keeping track of reliability in *groups*. Reliable messages should be received by everyone (eventually).
- Requires agreement algorithm across computers (who has what)
- HW: find an example where reliable messages are not safe.



Spread Toolkit (Stanton '02)



Spread Functions

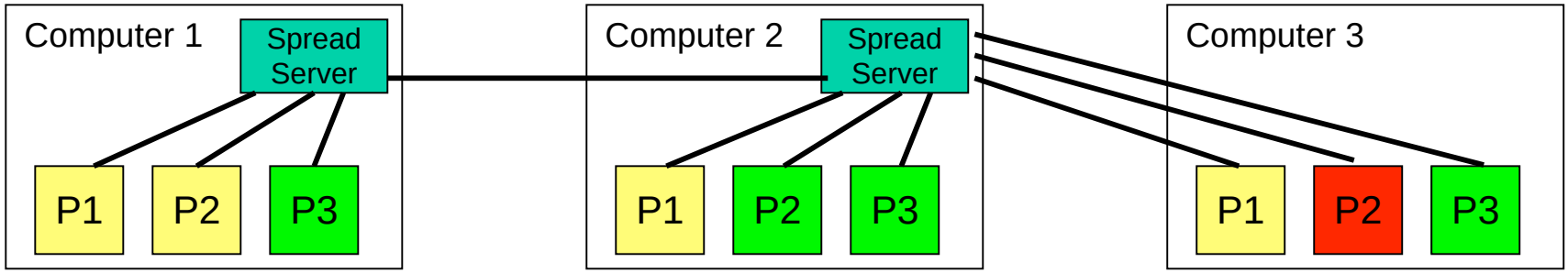
- SP_connect: establish a connection with the spread daemon
- SP_disconnect: terminate connection
- SP_join(mbox. group): join a group
- SP_leave(mbox. group): leave a group
- SP_multicast(…, group, message): send a message to everyone in group
- SP_multigroup_multicast(…, groups, message): send message to multiple groups all at once

Message types

- Unreliable - no order, unreliable
- Reliable - no order, reliable
- FIFO - FIFO by sender, reliable
- Causal - Causal (Lamport), reliable
- Agreed - Totally ordered, reliable
- Safe - Totally ordered, safe

- Note: each message has a type; these can be mixed within groups

Features of Spread



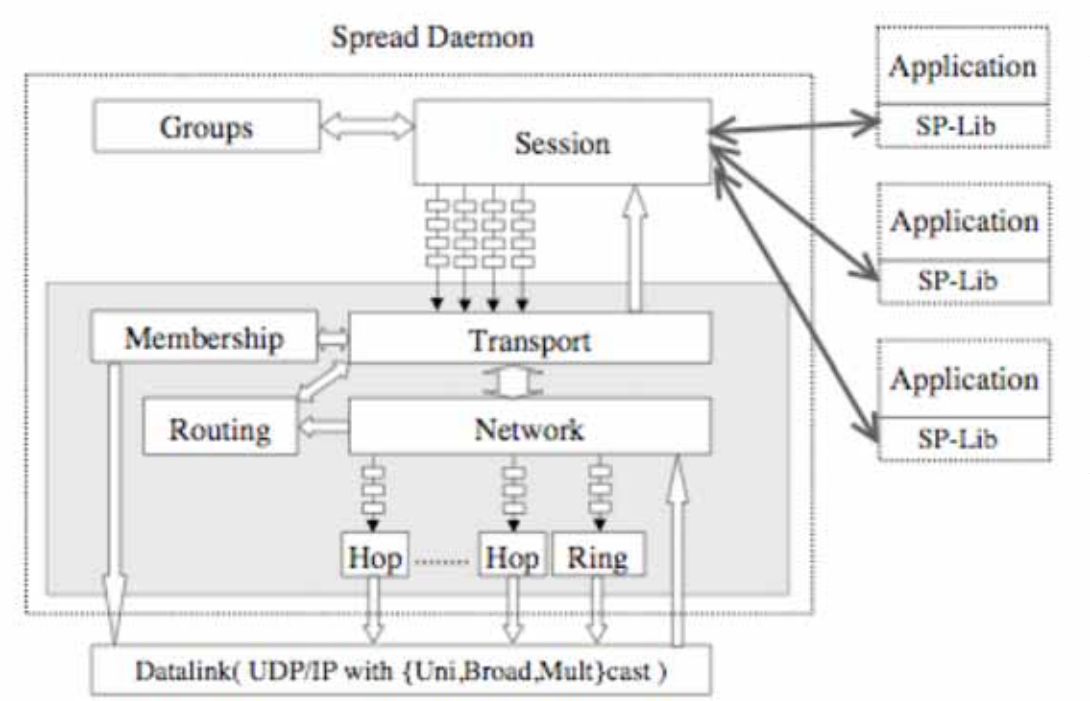
Features of Spread

- Number and location of servers are configurable
- Retransmits are optimized in multi-hop environments
- Guarantees are provided at servers; assumes that inter-process comms on single computer is reliable
- Data is combined in packets when possible to increase efficiency
  - Gives a correlated channel model when data is lost
- No hardwired addresses (exc servers)

Project ideas

- How can we model a spread-based communications network from the point of view of estimation and control
- Is it better to have one server or multiple servers? What are the latency tradeoffs?
  - Alice originally used one server per computer
  - Eventually moved to a single server (not sure why)

How Spread Works (Amir and Stanton, '98)



Spread daemon

- Implements group communi-cations protocols
- Uses UDP to talk between spread hosts
- Two protocols: hop and ring

Applications

- Think client library
- Uses TCP to talk to server

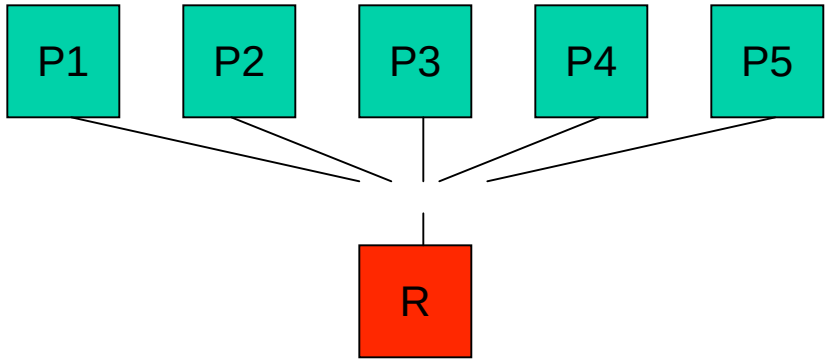
Hop protocol

- UDP-based protocol between *sites*
  - Sites connected by slower links
- Provide low latency communcations
- Packet loss handled on hop-by-hop basis (instead of end to end)

Ring Protocol

- Used for communications between multiple servers at same *site*
  - Assumes dedicated (switched) links
- Token ring based protocol: pass control from one server to the next

Example: Resource allocation



Problem description

- Fixed processes P_i sharing resource R
- Once a process grabs a resource, it must release it before it is use again
- Requests granted in order they were requested
- Every request is eventually granted

- Solve in *distributed* way; processes agree on who goes next
- Problem is non-trivial, even with central scheduling (see Lamport paper)

Solution using Spread

- Assume totally ordered, reliable messages ("agreed" message type)
- All processes and resource in single spread group

Algorithm

1. P_i sends multcast message to group requesting resource
2. P_j queues all requests and sends ack
3. Process P_i uses resource when
   - P_i request is at top of queue
   - Ack has been received from everyone
- P_i sends release message when done
- P_j dequeues release when message received

- Note: spread provides single order

Summary: Message Transfer Architectures