Elections and Group Communication

Web Applications and Services

Spring Term

Naercio Magaia

UNIVERSITY
OF SUSSEX

Contents

UNIVERSITY OF SUSSEX

Assumptions

- Processes are connected by reliable (not necessarily FIFO - TCP) channels among each other, although the network may be unreliable
  - In synchronous systems, there is an upper bound for the message transmission, processing delay, and clock drift
- Processes fail independently
  - A process failure is not a threat to other processes or communication among them
- Network can be partitioned

Assumptions

- Reliability assumption
  - **eventually** any failed link or router will be repaired or circumvented
  - but processes may not all be able to communicate at the same time
- Processes only fail by crashing
  - Unless explicitly stated – arbitrary (or Byzantine) failures
- A correct process is the one that does not crash at any point during execution
- There is no correct process which then failed

Failure Detector

- A service that processes queries about the state of a process
- Unreliable
  - may produce one of the *unsuspected* or *suspected* values when given the identity of a process (both hints)
- Reliable
  - may produce one of the *unsuspected* or f*ailed* values when given the identity of a process (*unsuspected* is a hint)
    - a crashed process stays crashed

UNIVERSITY OF SUSSEX

Failure Detector

- Implemented by sending heartbeats to the detector and timeouts
  - Trade-offs when selecting timeout values
  - What timeout value to select in a synchronous distributed system?

Elections

- Is a helper algorithm to bootstrap other algorithms
- If the leader crashes or retires, a new election is started
- A process can start a single election but all processes in a system can start an election at the same time
- A process can be a <u>participant</u> (i.e., engaged in some election) or a <u>non-participant</u>
- The elected process has the ‘largest’ identifier

UNIVERSITY OF SUSSEX

Elections

- Each process has a variable $elected_i$, which can be undefined or contain the leader process
- Requirements
  - Safety: during an election a participant process' $elected_i$ value is either undefined or P, where P is the elected leader process
  - Liveness: eventually all processes' $elected_i$ value is defined
- Evaluation
  - Bandwidth: total number of sent messages to elect leader

UNIVERSITY OF SUSSEX

Ring-based Algorithm

- Processes arranged in a ring
  - communicate with next process in the ring
- Assumptions
  - no failures occur
  - The system is asynchronous
  - The elect process (i.e., coordinator) has the largest ‘identifier’
- Initially, every process is marked as a non-participant in an election
- Any process can begin an election
  - marks itself as a participant

UNIVERSITY OF SUSSEX

Ring-based Algorithm

- Sends a message to the next process (with its identifier in it)
- When an **election message** is received, process identifiers are compared

if $id_{message} > id_{process}$ **then**

forward message to the next process

else if $id_{message} < id_{process}$ **then**

if receiver is not a participant,

substitute id and forward

else

discard message

UNIVERSITY OF SUSSEX

Ring-based Algorithm

- *On forwarding, mark itself as participant*
- If the received identifier is that of the receiver itself, then this process's identifier is the greatest
  - it becomes the leader
- The leader marks itself as a *non-participant* once more and sends an *elected message* to its neighbour, announcing its election and *id*

Ring-based Algorithm

- When a process receives an elected message
  - it marks itself as a non-participant, sets its variable $elected_i$ to the *id* in the message
  - If not the coordinator, then forward elected message
- Note: duplicate messages from multiple elections are discarded as soon as possible
- Bandwidth: what’s the worst and best case for a single election?

The Bully Algorithm

- Processes can crash during election (fail - stop)
- The system is synchronous (i.e., timeouts to detect a process failure)
- Each process knows which processes have higher identifiers (IDs) and it can communicate with all these processes
- Election message
  - sent to announce an election

The Bully Algorithm

- Answer message
  - sent in response to an election message
- Coordinator message
  - sent to announce the elected process
- Election begins when one or more processes identify a failed leader

UNIVERSITY
OF SUSSEX

The Bully Algorithm

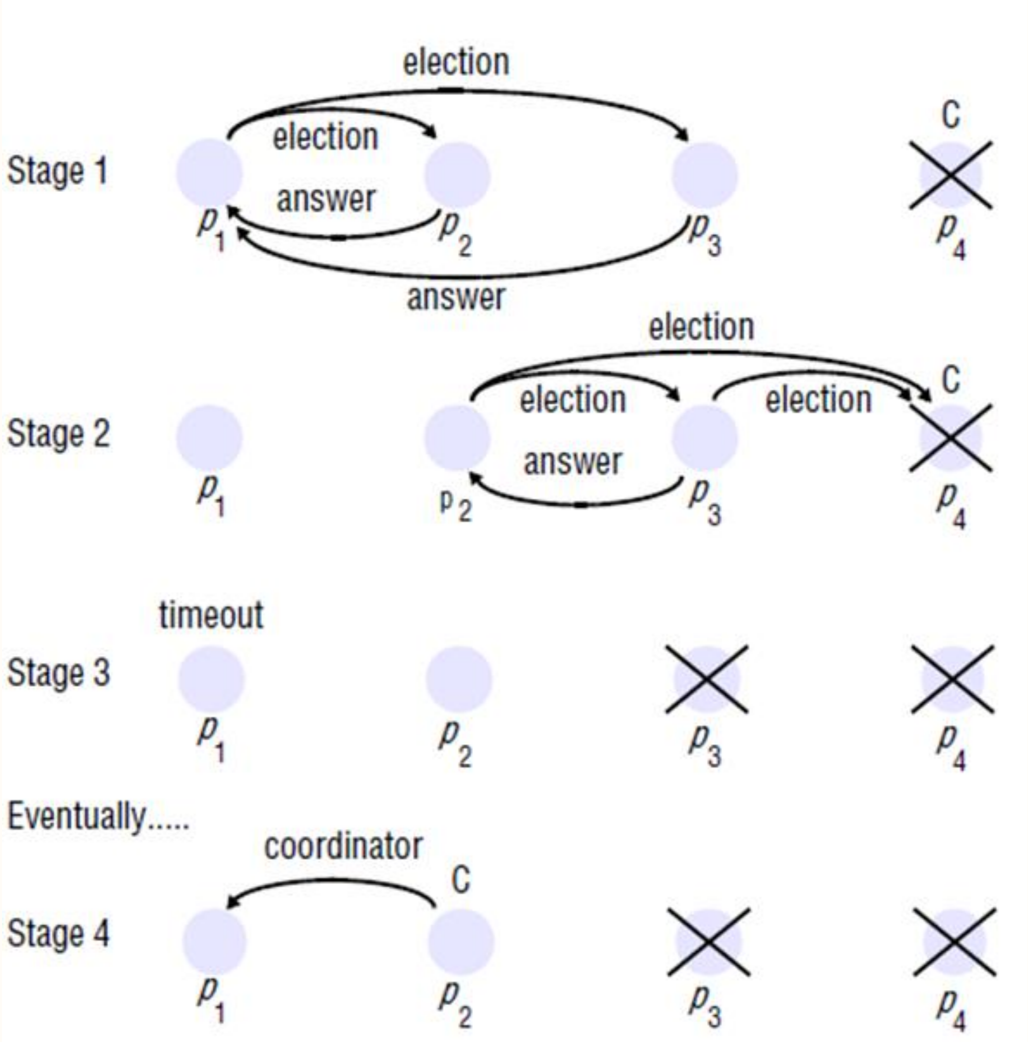
- The process with the <u>highest</u> ID can bully all other processes
- A process with a lower identifier can begin an election by sending an election message to processes with higher identifiers and waits for an answer
- If none arrives within time T (i.e., *synchronous system*), the process bullies the ones with lower IDs and announces itself as the leader

The Bully Algorithm

- Otherwise, the process waits a further period for a *coordinator message* to arrive from the new coordinator
- If none arrives (i.e., a process crashed), it begins another election
- If a process receives a *coordinator message*, it sets its variable *elected*$_i$ to the ID of the coordinator
- If a process receives an election message, it sends back an answer message and begins another election
  - unless it has begun one already

The Bully Algorithm

- Bandwidth
  - Best case
    - the process with the second-highest identifier notices the coordinator's failure (*N – 2* coordinator messages)
  - Worst case
    - *O* (N^2) messages in the worst case
    - the process with the lowest identifier detects the coordinator's failure (*N – 1* processes begin elections)



UNIVERSITY OF SUSSEX

Group Communication

- Collection of processes can communicate reliably over one-to-one channels and may fail only by crashing (fail-stop)
- Processes are members of groups – destinations of messages sent with the multicast operation – a process belongs to a single group
- ***multicast(g, m)***: sends message *m* to all members of group *g*
- ***deliver(m)***: delivers message *m* sent by multicast to the process

Group Communication

- A multicast message can be queued before the *deliver(m)* operation is called
- Every message *m* carries the unique identifier of the sender and the unique destination group identifier

UNIVERSITY OF SUSSEX

Basic Multicast

- Primitive multicast service
  - a correct process will eventually deliver the message as long as the sender does not crash
- Implemented by multi-unicasting the message using reliable channels
- *B-multicast(g, m)*: *send(m)* to each process belonging to *g* separately
- On *receive(m)* at *p*: *B-deliver(m)* at *p*
- Feedback implosion by acknowledgments when calling *send(m)*

Reliable Multicast

- Integrity
  - a correct process *p* delivers a message *m* at most once
- Validity
  - if a correct process multicasts message *m*, then it will eventually deliver *m*
- Agreement
  - **i**f a <u>correct</u> process delivers message *m*, then <u>all</u> other correct processes in *g* will <u>eventually</u> deliver *m* (all-or-nothing)

Reliable Multicast

On initialization
Received := {};

For process p to R-multicast message m to group g
B-multicast(*g*, *m*); // $p \in g$ is included as a destination

On B-deliver(*m*) *at process q with* $g = group(m)$
if ($m \notin Received$)
then
$Received := Received \cup \{m\}$;
if ($q \neq p$) *then B-multicast*(*g*, *m*); *end if*
R-deliver m;
end if

- Inefficient algorithm
- each message is sent **|*g*|** times to each process!

Reliable multicast over IP multicast

- IP multicast, piggybacked acknowledgements (in other messages) and negative acknowledgements
- Processes do not send separate acknowledgement messages
- Processes send a separate response message only when they detect that they have missed a message (negative ack)

UNIVERSITY OF SUSSEX

Reliable multicast over IP multicast

- Each process keeps
  - a sequence number (initially 0) for *g*
  - ack per process: the latest message sequence number sent by others
- *R-multicast(g, m)*
  - IP-multicast message to *g*
  - include sequence number and acks from all processes
- Piggybacked acks enable recipients to learn about messages that they have not received

UNIVERSITY OF SUSSEX

Reliable multicast over IP multicast

- A process *R-delivers* a message destined for *g* carrying the sequence number *S* from *p* if and only if the stored ack for that process refers to $S – 1$ (ack is incremented)
- Already delivered messages are discarded (*ack* > *S*)
- The message is stored in a hold-back queue
  - if the stored ack for that process refers to a value S' < S – 1
  - or an acknowledgement in the message refers to a message from some other process that is not yet received

UNIVERSITY
OF SUSSEX

Reliable multicast over IP multicast

- The process then sends a negative ack requesting the lost messages (there can be duplicates)
- When gaps are filled, it *R-delivers(m)* (removing it from queue)
- Assumption for agreement: messages flow indefinitely!!

Uniform agreement

- Agreement
  - if a correct process delivers message *m*, then all correct processes in *g* will eventually deliver *m*
- Uniform agreement
  - if a process, whether it is correct or fails, delivers message *m*, then all correct processes in *g* will eventually deliver *m*
- R-multicast meets the uniform agreement condition
- Uniform agreement is important to keep state consistent! (e.g., replication)

UNIVERSITY OF SUSSEX

Ordered Multicast

- Some applications require ordering guarantees in group communication
- 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* before *m'*
- Causal ordering
  - If *multicast(g, m)* → *multicast(g, m')*, where → is the *happened-before* relation induced by messages sent between members of *g*, then any correct process that delivers *m'* will deliver *m* before *m'*

UNIVERSITY OF SUSSEX

Ordered Multicast

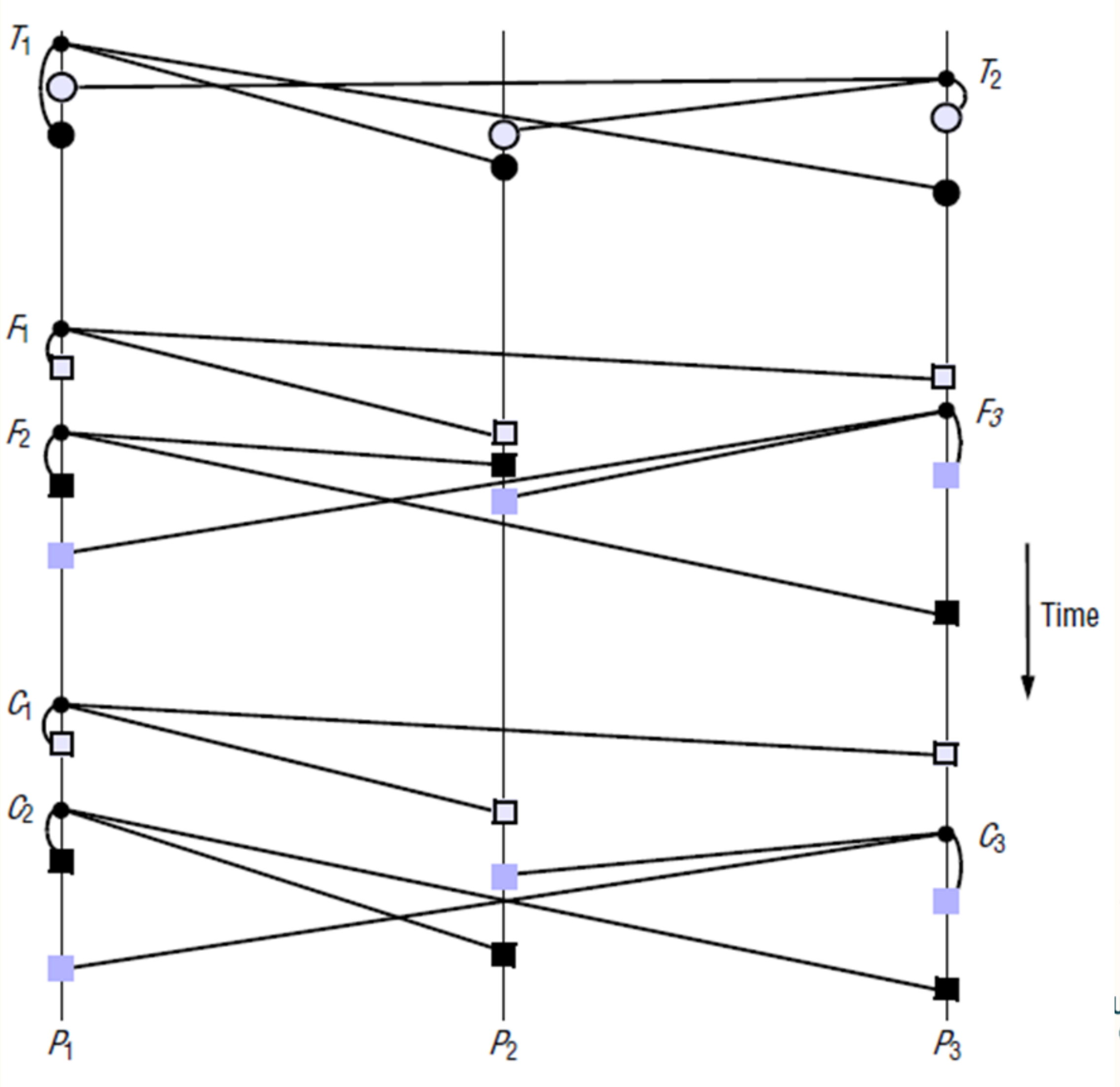
- Total ordering
  - If a correct process delivers message *m* before *m'*, then any other correct process that delivers *m'* will deliver *m* before *m'*
- Causal ordering implies FIFO ordering!
  - Any two multicasts by the same process are related by *happened-before*
- FIFO and Causal orderings are partial orderings
  - Not all messages are sent by the same process and some multicasts are concurrent (not ordered by *happened-before*)

UNIVERSITY OF SUSSEX

Ordered Multicast

- Total ordering is not necessarily a FIFO or causal ordering
- Ordering the delivery of multicast messages can be expensive in terms of delivery latency and bandwidth consumption

Ordered Multicast



- Notice
  - the consistente ordering of totally ordered messages T1 and T2
  - the FIFO-related messages F1 and F2 and
  - the casually related messages C1 and C3 and
  - The otherwise arbitrary delivery ordering of messages
- Total Ordering doesn't say anything about how messages are ordered

UNIVERSITY OF SUSSEX

Distributed Bulleting Board

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

- Posts as they appear to one of the users
- Reliable multicast is required if every user is to receive every posting
- At minimum, FIFO ordering is desirable
  - <u>every</u> posting from a given user will be received <u>in the same order</u>

UNIVERSITY OF SUSSEX

Distributed Bulleting Board

Bulletin board: *os.interesting*		
Item	From	Subject
23	A.Hanlon	Mach
24	G.Joseph	Microkernels
25	A.Hanlon	Re: Microkernels
26	T.L'Heureux	RPC performance
27	M.Walker	Re: Mach
end		

- Messages 25 and 27 appear after 24 and 23, respectively (causal ordering)
- The item numbering is the same for all users (Total ordering) – then users can discuss about items *X*

FIFO Ordering

- Sequence numbers just like in R-multicast over IP-multicast

Causal Ordering

- *happened-before* relationship **only** as established by multicast messages
- Each process maintains a *vector clock*
  - entries count the number of multicast messages that *happened-before* the next multicast message
- CO-multicast
  - increase the process entry by 1 and *B-multicast(g,<m,T>)*

Causal Ordering

- On B-deliver: place message in a hold-back queue until
  - all previous messages from the same process have been delivered
  - all previous messages that the sending process has delivered, are delivered (causally preceded – look at vector timestamps)

Total Ordering (Sequencer)

- A special process is the **sequencer** (can be elected)
- TO-multicast(g, m): a unique identifier *i* is attached and then ***B-multicast(g and sequencer, <m, i>)***
- The *sequencer(g)* maintains a group-specific sequence number to assign increasing and consecutive sequence numbers to the messages that it *B-delivers*

Total Ordering (Sequencer)

- It announces the sequence numbers by *B-multicasting* **order** messages to *g*
- A message will sit in a process' hold-back queue until it can be *TO-delivered* according to its sequence number

Total Ordering (Collective Agreement)

- TO-multicast: *B-multicasts(g, m)*, *m* carries a unique identifier *i*
- On *B-deliver(m)*
  - each process *p* sends to the sender a proposed sequence number, which is the *max(lastProposed, lastAgreed) + 1, include process ID to break ties.*
  - provisionally assigns the proposed value to the message in a hold-back queue
- Sender *B-multicasts(g, agreed_sequence_number) : maximum proposed*

UNIVERSITY OF SUSSEX

Total Ordering (Collective Agreement)

- On *B-deliver(agreed_sequence_number)*
  - Each process assigns the agreed value and reorders the message in the queue if the value differs from the proposed one (updates the *lastAgreed* value)
  - If message is at the head of the queue, then TO-deliver(m)
- Not causally or FIFO ordered delivery

Next Lecture ...

- ✓ Introduction
- ✓ HTTP, Caching, and CDNs
- ✓ Views
- ✓ Templates
- ✓ Forms
- ✓ Models
- ✓ Security
- ✓ Transactions
- ✓ Remote Procedure Call
- ✓ Web Services
- ✓ Time
- ✓ Elections and Group Communication
- ➢ **Coordination and Agreement**

UNIVERSITY OF SUSSEX