The UNIVERSITY *of* EDINBURGH
SCHOOL *of* INFORMATICS

CS4/MSc

Distributed Systems

Björn Franke

`bfranke@inf.ed.ac.uk`

Room 2414

(Lecture 11: Failure Detection and Leader Election,
10th November 2005)

Failure Detection

- A **failure detector** is a service that processes queries about whether a particular process has failed (crashed). It is often implemented as a set of objects, one local to each process, which run a failure-detection algorithm.
- Most detectors are **unreliable** and only able to make two judgements on processes:
  - **Unsuspected** and **Suspected**
- These judgements may not accurately reflect the state of a process. **Unsuspected** means that the failure detector does not believe that the process has failed, e.g. because it has just received a message from it. However a failure could have occurred after the message was sent.
- Similarly a judgement of **suspected** indicates that the failure detector has reason believe that the process has failed, e.g. because an expected message did not arrive. However, congestion could be delaying the message.

Reliable Failure Detectors

Reliable failure detectors are accurate at detecting process failures and make judgements of **Unsuspected** or **Failed**. As previously a judgement of unsuspected does not mean that the process has necessarily not failed but a judgement of **failed** means that the detector has determined that the process has crashed.

A **synchronous** distributed system is one in which bounds are defined:

- the time to execute each step of a process has known lower and upper bounds;
- each message transmitted over a channel is received within a known bounded time;
- each process has a local clock whose drift rate from real time has a known bound.

Reliable failure detectors can most easily be implemented on synchronous systems.

Leader Elections

- A **leader election** is defined as getting a set of processes to agree on a **unique** leader. The leader (coordinator) typically knows all the other processes (participants) in the group.
- A leader election may be initiated when an existing leader is detected or suspected to have failed.
- The election of a leader must be announced to all processes in the group.
- Leader elections can be based on
  - a global priority – extrema finding
  - individual preference – a voting scheme
- The main concern is usually the fast and successful completion of the election process.

Safety and Liveness Properties

We assume that each process P_i $i = 1, \ldots, N$, has a variable $elected_i$ which contains the process identifier of the coordinator process. A special value $\perp$ is used to mark when this variable is undefined, i.e. when there is an election in progress.

The key properties of a useful leader election algorithm are:

Safety: A participating process P_i has variable $elected_i = \perp$ or $elected_i = P$ and P is chosen as the non-crashed process at the end of the run with largest identifier.

Liveness: All processes P_i participate and eventually $elected_i \neq \perp$ or crash.

As with mutual exclusion algorithms, performance is judged on the basis of the number of messages necessary to complete the algorithm, which is proportional to the total network bandwidth used.

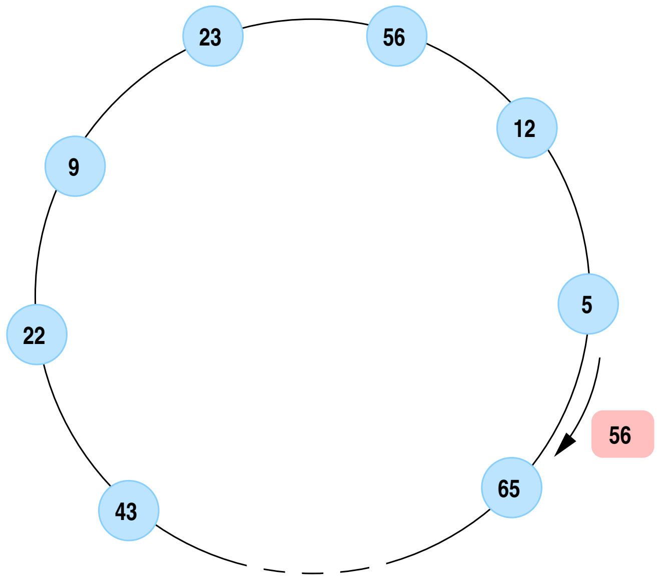
Ring-based Election Algorithm (1)

- Processes are assumed to be arranged in a logical ring: each process P_i has a communication channel to the next process in the ring $P_{(i-1)modN}$. Each process has an identifier.
- The values of the identifiers must be unique and totally ordered.
- Identifiers may be dynamic, for example, reflecting the current computational load each process P_i might have $\langle 1/load_i, i\rangle$ as its identifier.
- All messages are sent clockwise round the ring and communication is reliable.
- The objective is to elect the process with the largest identifier as the **coordinator**.
- Any process can initiate the election by placing its identifier in an **election** message and sending it to its neighbour. It will also change its state from **non-participant** to **participant**.

Ring-based Election Algorithm (2)

- On receiving an **election** message a process compares the identifier it contains with its own identifier
  - if the message identifier is greater than its own and its state is **participant**, it simply passes the message on.
  - if the message identifier is greater and the process state is **non-participant**, it changes its state to **participant** and forwards the message on unchanged.
  - if the process identifier is greater (it must be non-participant) it changes its state to **participant** and forwards a new election message with its own identifier inserted.
- On receiving an **election** message with its own identifier in a process has won the election. It sets it state to **non-participant** and sends an **elected** message containing its own identifier.
- On receiving an **elected** message each process set its state to **non-participant** and sets its variable recording the elected coordinator.

Ring-based Election Algorithm Example



Both safety and liveness properties are satisfied. In the worst case (the process with the highest identifier is the last to be consulted) $3N - 1$ messages need to be sent.

Bully Algorithm (1)

- The bully algorithm is more robust than the ring-based algorithm because it can continue even if a process crashes during the election. In order to detect such failures the system must be assumed to be synchronous: **timeouts** are used.
- We assume that there is a set of processes $P_1, P_2, \ldots, P_N$, each of which has knowledge of the complete set and each of which can communicate with each other process directly.
- Moreover the set is ordered by the identifiers of the processes.
- The algorithm uses three types of messages:

  election messages used to announce an election;

  answer messages sent in reply to election messages;

  coordinator messages sent when the election is complete to announce the election result—the new coordinator.

- An election can be instigated by any process which detects (by message timeout) that the current coordinator has failed.

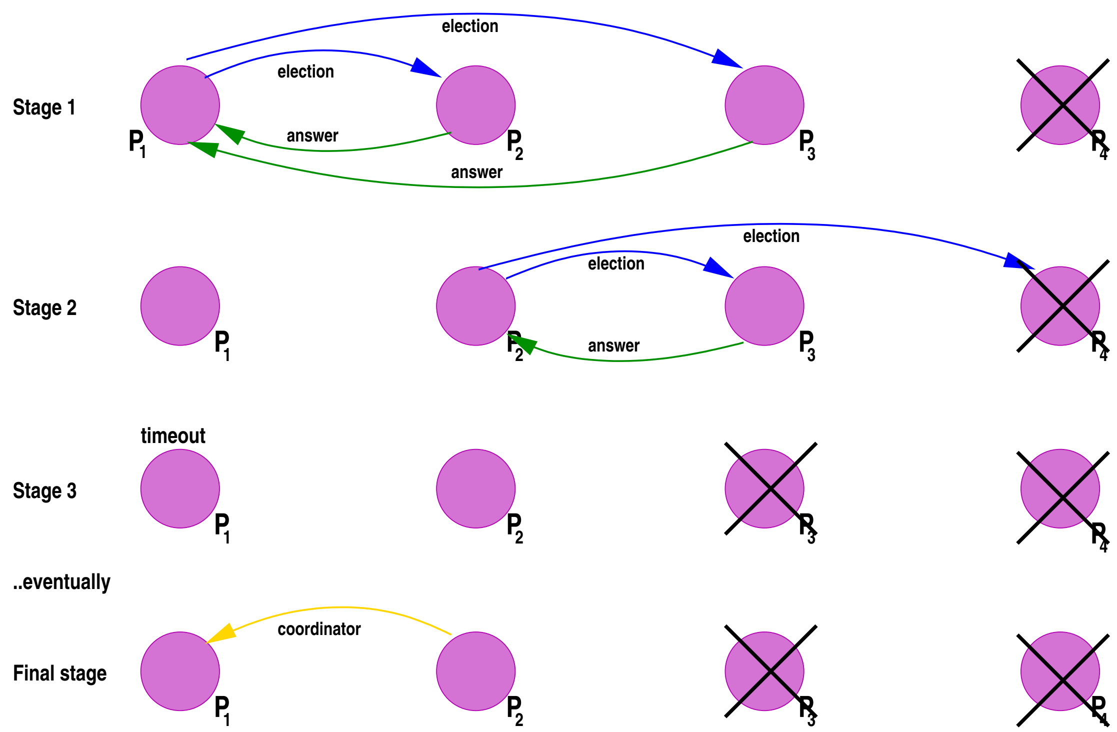
Bully Algorithm (2)

- If the process detecting the failure has the highest remaining identifier it can simply send a **coordinator** message, announcing itself as the new coordinator, circumventing the election.
- If a lower process detects the failure it announces the need for an election by sending an **election** message to each process with a higher identifier. It sets a **timeout** T and waits for responses to these messages.
- If no response arrives within time T the process assumes that all higher processes have failed and takes on the role of coordinator itself. It sends a **coordinator** message to all lower processes announcing itself as the new coordinator.
- If it receives an **answer** message from a higher process it then sets a second **timeout** T' and waits for a coordinator message.
- If no coordinator message arrives it assumes that there has been a subsequent failure and initiates another election by sending an **election** message to each higher process.

Bully Algorithm (3)

- If a process receives an **election** message, it sends back an **answer** message, and begins another election (sending **election** messages to all higher processes) if it has not already started one. It is possible for several elections to be started at once because several processes detect the failure of the current coordinator.
- If a process P_i receives a **coordinator** message, it sets its variable $elected_i$ to the identifier of the coordinator contained within the message.
- The algorithm is intended to work in a dynamically changing group of processes. As well as crashing, processes may also be restarted. When a process is inserted into the group to replace a failed process it starts an election.
  - If it has the highest identifier it can simply announce itself as coordinator even though there is a coordinator currently working.

Bully Algorithm Example



Bully Algorithm Characteristics

- If we assume that message delivery is reliable and the failure detector is reliable, the liveness condition is satisfied for the algorithm and all process will eventually reach agreement on the coordinator or will have crashed.
- If failed processes are not restarted the safety property is also satisfied. However if a crashed process can be replaced by another with the same identifier then the safety property is not met.
- If the replacement process has the highest identifier but joins the group, sending a **coordinator** message, just as another process is sending its **coordinator** message, the messages may arrive at different orders with different members of the group. This could lead to subsets of processes having different records of who is the current coordinator.
- In the best case the number of messages is just $N - 2$ coordinator messages; in the worst case $O(N^2)$ messages are required.

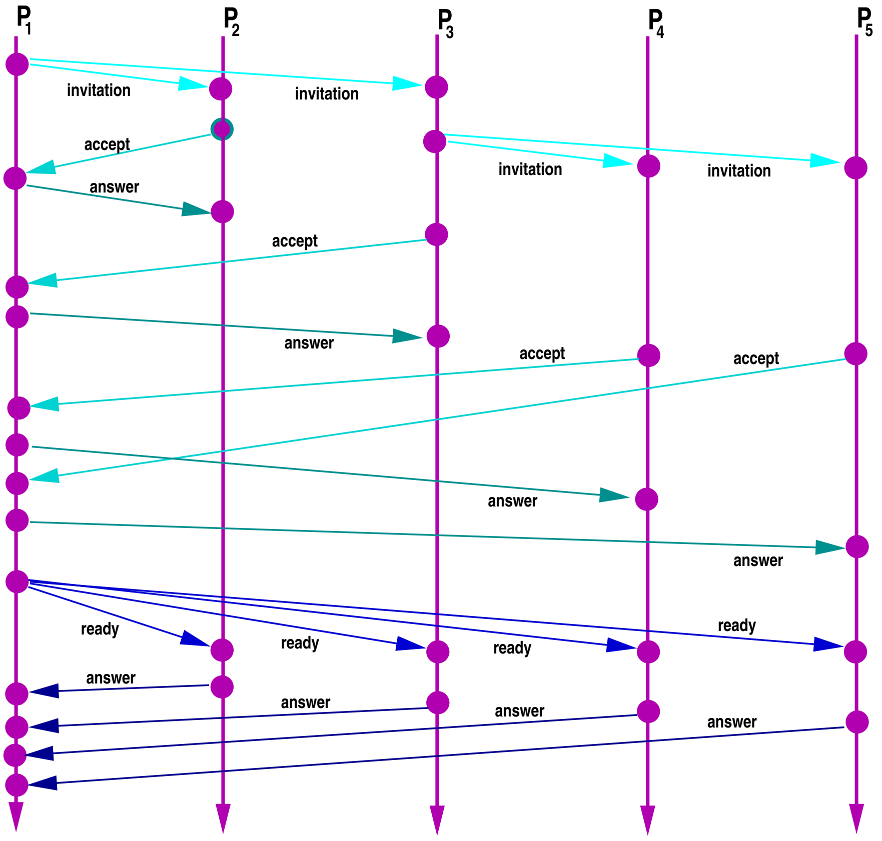
Forming groups around coordinators

- Both the ring-based algorithm and the bully algorithm aim to appoint a coordinator within a group, and although the membership of the group may be dynamic, the existence of the group is not.
- If we no longer assume that message delivery is reliable and allow that communication infrastructure can fail as well as processes, this approach is not so applicable.
- For example a network may become partitioned if a router fails, making communication not possible between two subsets of the processes within a group.
- In this scenario it no longer makes sense to think of a single global coordinator. Instead it is more appropriate to think in terms of a coordinator for each subgroup.
- The invitation algorithm is designed to work in this type of situation, and can be used in asynchronous systems.

The Invitation Algorithm

- Each process can form a singleton group with itself as coordinator. Each group has a group identifier known to all its members.
- When a coordinator finds it can communicate with other processes outside its group, it seeks a merger with the other group.
- To avoid a livelock situation on detecting another group, each coordinator will wait for a period based on its own group identifier before issuing the **invitation** to initiate the merger.
- When a coordinator receives an invitation to merge it forwards it to all members of its own group. Any process receiving an invitation accepts by sending an **accept** message to the potential coordinator which acknowledges with an **answer** message.
- The coordinator which initiated the merger becomes the coordinator for the enlarged group, and confirms the new group by sending a **ready** message to each member of the new group. These processes respond with an **answer** message.

Invitation Algorithm Example



The Invitation Algorithm (2)

- Each group has an identifier which is unique and any change in the structure of the group results in a new identifier being assigned. Thus coordination is always with respect to the current members of the group only.
- Unlike the bully algorithm, no logical structure is imposed on the processes in the invitation algorithm. This may mean that it takes a long time for groups to merge.
- The invitation algorithm does not make assumptions about bounded response time and can work correctly in the presence of timing failures.
- Correctness is defined to depend on a group of processes agreeing to group membership and then agreeing on a value. This is a weaker condition than in the bully algorithm where the requirement is that all functioning processes agree to a value.