Distributed Systems — Coordination and Agreement

Allan Clark

School of Informatics
University of Edinburgh

http://www.inf.ed.ac.uk/teaching/courses/ds
Autumn Term 2012

Coordination and Agreement

Overview

- In this part of the course we will examine how distributed processes can agree on particular values
- It is generally important that the processes within a distributed system have some sort of agreement
- Agreement may be as simple as the goal of the distributed system
  - Has the general task been aborted?
  - Should the main aim be changed?
- This is made more complicated than it sounds, since all the processes must, not only agree, but be confident that their peers agree.
- We will look at:
  - mutual exclusion to coordinate access to shared resources
  - The conditions necessary in general to guarantee that a global consensus is reached
  - Perhaps more importantly the conditions which prevent this

Coordination and Agreement

No Fixed Master

- We will also look at dynamic agreement of a master or leader process i.e. an election. Generally after the current master has failed.
- We saw in the Time and Global State section that some algorithms required a global master/nominee, but there was no requirement for that master/nominee process to be fixed
- With a fixed master process agreement is made much simpler
- However it then introduces a single point of failure
- So here we are generally assuming no fixed master process

Coordination and Agreement

Synchronous vs Asynchronous

- Again with the synchronous and asynchronous
- It is an important distinction here, synchronous systems allow us to determine important bounds on message transmission delays
- This allows us to use timeouts to detect message failure in a way that cannot be done for asynchronous systems.

Coping with Failures

- In this part we will consider the presence of failures, recall from our Fundamentals part three decreasingly benign failure models:
  1. Assume no failures occur
  2. Assume omission failures may occur; both process and message delivery omission failures.
  3. Assume that arbitrary failures may occur both at a process or through message corruption whilst in transit.

A Brief Aside

Failure Detectors

- Here I am talking about the detection of a crashed process
- Not one that has started responding erroneously
- Detecting such failures is a major obstracle in designing algorithms which can cope with them
- A failure detector is a process which responds to requests querying whether a particular process has failed or not
- The key point is that a failure detector is not necessarily accurate.
- One can implement a “reliable failure detector”
- One which responds with: “Unsuspected” or “Failed”

Failure Detectors

Unreliable Failure Detectors

- An “unreliable failure detector” will respond with either: “Suspected” or “Unsuspected”
- Such a failure detector is termed an “unreliable failure detector”

A simple algorithm

- If we assume that all messages are delivered within some bound, say D seconds.
- Then we can implement a simple failure detector as:
- Every process p sends a “p is still alive” message to all failure detector processes, periodically, once every T seconds
- If a failure detector process does not receive a message from process q within $T + D$ seconds of the previous one then it marks q as “Suspected”

Failure Detectors

Reliable and Unreliable

- If we choose our bound D too high then often a failed process will be marked as “Unsuspected”
- A synchronous system has a known bound on the message delivery time and the clock drift and hence can implement a reliable failure detector
- An asynchronous system could give one of three answers: “Unsuspected”, “Suspected” or “Failed” choosing two different values of D
- In fact we could instead respond to queries about process p with the probability that p has failed, if we have a known distribution of message transmission times
- e.g., if you know that 90% of messages arrive within 2 seconds and it has been two seconds since your last expected message you can conclude there is a:

Failure Detectors

Reliable and Unreliable

- NOT a 90% chance that the process p has failed.
- We do not know how long the previous message was delayed
- Even if so, Bayes theorem tells that, in order to calculate the probability that p has failed given that we have not received a message we would also require the probability that p fails within the given time increment without prior knowledge.
- Bayes: $P(a|b) = \frac{P(b|a) \times P(a)}{P(b)}$
- here $a = p$ has failed and $b =$ the message has failed to be delivered
- Further the question arises what would the process receiving that probability information do with it?
- 1. if $(p > 90)$ ...
  2. else ...

Coordination and Agreement

Mutual Exclusion

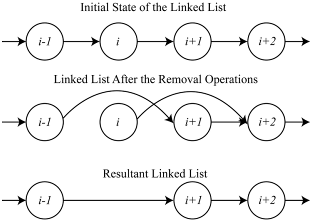
- Ensuring mutual exclusion to shared resources is a common task
- For example, processes A and B both wish to add a value to a shared variable ‘a’.
- To do so they must store the temporary result of the current value for the shared variable ‘a’ and the value to be added.

Time	Process A	Process B	
1	t = a + 10		A stores temporary
2		t’ = a + 20	B stores temporary
3		a = t’	(a now equals 25)
4	a = t		(a now equal 15)

- The intended increment for a is 30 but B’s increment is nullified

Coordination and Agreement

Mutual Exclusion


Initial State of the Linked List
i-1
i
i+1
i+2
Linked List After the Removal Operations
i-1
i
i+1
i+2
Resultant Linked List
i-1
i+1
i+2

```
new-next = i.next
```

```
new-next = (i+1).next

i.next = new-next
```

```
(i-1).next = new-next
```

Shamelessly stolen from Wikipedia

- A higher-level example is the concurrent editing of a file on a shared directory
- Another good reason for using a source code control system

Coordination and Agreement

Distributed Mutual Exclusion

- On a local system mutual exclusion is usually a service offered by the operating system's kernel.
- But for a distributed system we require a solution that operates only via message passing
- In some cases the server that provides access to the shared resource can also be used to ensure mutual exclusion
- But here we will consider the case that this is for some reason inappropriate, the resource itself may be distributed for example

Distributed Mutual Exclusion

Generic Algorithms for Mutual Exclusion

- We will look at the following algorithms which provide mutual exclusion to a shared resource:
    1. The central-server algorithm
    2. The ring-based algorithm
    3. Ricart and Agrawala — based on multicast and logical clocks
    4. Maekawas voting algorithm
- We will compare these algorithms with respect to:
    1. Their ability to satisfy three desired properties
    2. Their performance characteristics
    3. How fault tolerant they are

Generic Algorithms for Mutual Exclusion

Assumptions and Scenario

- Before we can describe these algorithms we must make explicit our assumptions and the task that we wish to achieve
- Assumptions:
  1. The system is asynchronous
  2. Processes do not fail
  3. Message delivery is reliable: all messages are eventually delivered exactly once.
- Scenario:
  - Assume that the application performs the following sequence:
    1. Request access to shared resource, blocking if necessary
    2. Use the shared resource exclusively — called the *critical section*
    3. Relinquish the shared resource
- Requirements:
  1. Safety: At most one process may execute the critical section at any one time
  2. Liveness: Requests to enter and exit the critical section eventually succeed.

Generic Algorithms for Mutual Exclusion

Assumptions and Scenario

- The *Liveness* property assures that we are free from both deadlock and starvation — starvation is the indefinite postponement of the request to enter the critical section from a given process
- Freedom from starvation is referred to as a "*fairness*" property
- Another fairness property is the order in which processes are granted access to the critical section
- Given that we cannot ascertain which event of a set occured first we instead appeal to the "*happened-before*" logical ordering of events
- We define the *Fairness* property as: If e_1 and e_2 are requests to enter the critical section and $e_1 \longrightarrow e_2$, then the requests should be granted in that order.
- Note: our assumption of request-enter-exit means that process will not request a second access until after the first is granted

Generic Algorithms for Mutual Exclusion

Assumptions and Scenario

- Here we assume that when a process requests entry to the critical section, then until the access is granted it is blocked only from entering the critical section
- In particular it may do other useful work and send/receive messages
- If we were to assume that a process is blocked entirely then the *Fairness* property is trivially satisfied

Generic Algorithms for Mutual Exclusion

Assumptions and Scenario

- Here we are considering mutual exclusion of a single critical section
- We assume that if there are multiple resources then either:
    - Access to a single critical section suffices for all the shared resources, meaning that one process may be blocked from using one resource because another process is currently using a different resource or
    - A process cannot request access to more than one critical section concurrently or
    - Deadlock arising from two (or more) processes holding each of a set of mutually desired resources is avoided/detected using some other means
    - We also assume that a process granted access to the critical section will eventually relinquish that access

Generic Algorithms for Mutual Exclusion

Desirable Properties — Recap

- We wish our mutual exclusion algorithms to have the three properties:
  1. Safety — No two processes have concurrent access to the critical section
  2. Liveness — All requests to enter/exit the critical section eventually succeed.
  3. Fairness — Requests are granted in the logical order in which they were submitted

Distributed Mutual Exclusion Algorithms

Central Server Algorithm

- The simplest way to ensure mutual exclusion is through the use of a centralised server
- This is analogous to the operating system acting as an arbiter
- There is a conceptual token, processes must be in possesion of the token in order to execute the critical section
- The centralised server maintains ownership of the token
- To request the token; a process sends a request to the server
    - If the server currently has the token it immediately responds with a message, granting the token to the requesting process
    - When the process completes the critical section it sends a message back to the server, relinquishing the token
    - If the server doesn't have the token, some other process is "currently" in the critical section
    - In this case the server queues the incoming request for the token and responds only when the token is returned by the process directly ahead of the requesting process in the queue (which may be the process currently using the token)

Distributed Mutual Exclusion Algorithms

Central Server Algorithm

- Given our assumptions that no failures occur it is straight forward to see that the central server algorithm satisfies the Safety and Liveness properties
- The Fairness property though is not

Distributed Mutual Exclusion Algorithms

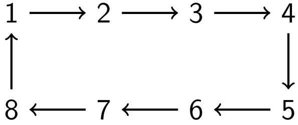
Central Server Algorithm

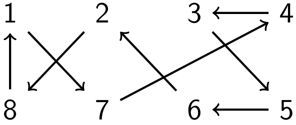
- Given our assumptions that no failures occur it is straight forward to see that the central server algorithm satisfies the Safety and Liveness properties
- The Fairness property though is not
- Consider two processes P_1 and P_2 and the following sequence of events:
  1. P_1 sends a request r_1 to enter the critical section
  2. P_1 then sends a message m to process P_2
  3. P_2 receives message m and then
  4. P_2 sends a request r_2 to enter the critical section
  5. The server process receives request r_2
  6. The server process grants entry to the critical section to process P_2
  7. The server process receives request r_1 and queues it
- Despite $r_1 \rightarrow r_2$ the r_2 request was granted first.

Distributed Mutual Exclusion Algorithms

Ring-based Algorithm

- A simple way to arrange for mutual exclusion without the need for a master process, is to arrange the processes in a logical ring.
- The ring may of course bear little resemblance to the physical network or even the direct links between processes.





Distributed Mutual Exclusion Algorithms

Ring-based Algorithm

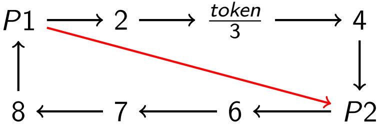
- The token passes around the ring continuously.
- When a process receives the token from its neighbour:
    - If it does not require access to the critical section it immediately forwards on the token to the next neighbour in the ring
    - If it requires access to the critical section, the process:
        1. retains the token
        2. performs the critical section and then:
        3. to relinquish access to the critical section
        4. forwards the token on to the next neighbour in the ring

Distributed Mutual Exclusion Algorithms

Ring-based Algorithm

- Once again it is straight forward to determine that this algorithm satisfies the Safety and Liveness properties.
- However once again we fail to satisfy the Fairness property

Ring-based Algorithm



- Recall that processes may send messages to one another independently of the token
- Suppose again we have two processes P_1 and P_2 consider the following events
  1. Process P_1 wishes to enter the critical section but must wait for the token to reach it.
  2. Process P_1 sends a message m to process P_2.
  3. The token is currently between process P_1 and P_2 within the ring, but the message m reaches process P_2 before the token.
  4. Process P_2 after receiving message m wishes to enter the critical section
  5. The token reaches process P_2 which uses it to enter the critical section before process P_1

Distributed Mutual Exclusion Algorithms

Multicast and Logical Clocks

- Ricart and Agrawala developed an algorithm for mutual exclusion based upon mulitcast and logical clocks
- The idea is that a process which requires access to the critical section first broadcasts this request to all processes within the group
- It may then only actually enter the critical section once each of the other processes have granted their approval
- Of course the other processes do not just grant their approval indiscriminantly
- Instead their approval is based upon whether or not they consider their own request to have been made first

Distributed Mutual Exclusion Algorithms

Multicast and Logical Clocks

- Each process maintains its own Lamport clock
- Recall that Lamport clocks provide a partial ordering of events but that this can be made a total ordering by considering the process identifier of the process observing the event
- Requests to enter the critical section are multicast to the group of processes and have the form $\{T, p_i\}$
- T is the Lamport time stamp of the request and p_i is the process identifier
- This provides us with a total ordering of the sending of a request message $\{T_1, p_i\} < \{T_2, p_j\}$ if:
  - $T_1 < T_2$ or
  - $T_1 = T_2$ and $p_i < p_j$

Multicast and Logical Clocks

Requesting Entry

- Each process retains a variable indicating its state, it can be:
  1. “Released” — Not in or requiring entry to the critical section
  2. “Wanted” — Requiring entry to the critical section
  3. “Held” — Acquired entry to the critical section and has not yet relinquished that access.
- When a process requires entry to the critical section it updates its state to “Wanted” and multicasts a request to enter the critical section to all other processes. It stores the request message $\{T_i, p_i\}$
- Only once it has received a “permission granted” message from all other processes does it change its state to “Held” and use the critical section

Multicast and Logical Clocks

Responding to requests

- Upon receiving such a request a process:
  - Currently in the "Released" state can immediately respond with a permission granted message
  - A process currently in the "Held" state:
    1. Queues the request and continues to use the critical section
    2. Once finished using the critical section responds to all such queued requests with a permission granted message
    3. changes its state back to "Released"
  - A process currently in the "Wanted" state:
    1. Compares the incoming request message $\{T_j, p_j\}$ with its own stored request message $\{T_i, p_i\}$ which it broadcasted
    2. If $\{T_i, p_i\} < \{T_j, p_j\}$ then the incoming request is queued as if the current process was already in the "Held" state
    3. If $\{T_i, p_i\} > \{T_j, p_j\}$ then the incoming request is responded to with a permission granted message as if the current process was in the "Released" state

Multicast and Logical Clocks

Safety, Liveness and Fairness

- Safety — If two or more processes request entry concurrently then whichever request bares the lowest (totally ordered) timestamp will be the first process to enter the critical section
- All others will not receive a permission granted message from (at least) that process until it has exited the critical section
- Liveness — Since the request message timestamps are a total ordering, and all requests are either responded to immediately or queued and eventually responded to, all requests to enter the critical section are eventually granted
- Fairness — Since lamport clocks assure us that $e_1 \rightarrow e_2$ implies $L(e_1) < L(e_2)$:
- for any two requests r_1, r_2 if $r_1 \rightarrow r_2$ then the timestamp for r_1 will be less than the timestamp for r_2
- Hence the process that multicast r_1 will not respond to r_2 until after it has used the critical section
- Therefore this algorithm satisfies all three desired properties

Distributed Mutual Exclusion Algorithms

Maekawas voting algorithm

- Maekawa's voting algorithm improves upon the multicast/logical clock algorithm with the observation that not all the peers of a process need grant it access
- A process only requires permission from a subset of all the peers, provided that the subsets associated with any pair of processes overlap
- The main idea is that processes vote for which of a group of processes vying for the critical section can be given access
- The processes that are within the intersection of two competing processes can ensure that the Safety property is observed

Distributed Mutual Exclusion Algorithms

Maekawas voting algorithm

- Each process p_i is associated with a *voting set* V_i of processes
- The set V_i for the process p_i is chosen such that:
  1. $p_i \in V_i$ — A process is in its own voting set
  2. $V_i \cap V_j \neq \{\}$ — There is at least one process in the overlap between any two voting sets
  3. $|V_i| = |V_j|$ — All voting sets are the same size
  4. Each process p_i is contained within M voting sets

Distributed Mutual Exclusion Algorithms

Maekawas voting algorithm

- The main idea in contrast to the previous algorithm is that each process may only grant access to one process at a time
- A process which has already granted access to another process cannot do the same for a subsequent request. In this sense it has already voted
- Those subsequent requests are queued
- Once a process has used the critical section it sends a release message to its voting set
- Once a process in the voting set has received a release message it may once again vote, and does so immediately for the head of the queue of requests if there is one

Maekawas voting algorithm

The state of a process

- As before each process maintains a state variable which can be one of the following:
  1. "Released" — Does not have access to the critical section and does not require it
  2. "Wanted" — Does not have access to the critical section but does require it
  3. "Held" — Currently has access to the critical section
- In addition each process maintains a boolean variable indicating whether or not the process has "voted"
- Of course voting is not a one-time action. This variable really indicates whether some process within the voting set has access to the critical section and has yet to release it
- To begin with, these variables are set to "Released" and False respectively

Maekawas voting algorithm

Requesting Permission

- To request permission to access the critical section a process p_i:
  1. Updates its state variable to “Wanted”
  2. Multicasts a request to all processes in the associated voting set V_i
  3. When the process has received a “permission granted” response from all processes in the voting set V_i: update state to “Held” and use the critical section
  4. Once the process is finished using the critical section, it updates its state again to “Released” and multicasts a “release” message to all members of its voting set V_i

Maekawas voting algorithm

Granting Permission/Voting

- When a process p_j receives a request message from a process p_i:
  - If its state variable is “Held” or its voted variable is True:
    1. Queue the request from p_i without replying
  - otherwise:
    1. send a “permission granted” message to p_i
    2. set the voted variable to True
- When a process p_j receives a “release” message:
  - If there are no queued requests:
    1. set the voted variable to False
  - otherwise:
    1. Remove the head of the queue, p_q:
    2. send a “permission granted” message to p_q
    3. The voted variable remains as True

Maekawas voting algorithm

Deadlock

- The algorithm as described does not respect the Liveness property
- Consider three processes p_1, p_2 and p_3
- Their voting sets: $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$ and $V_3 = \{p_3, p_1\}$
- Suppose that all three processes concurrently request permission to access the critical section
- All three processes immediately respond to their own requests
- All three processes have their "voted" variables set to True
- Hence, p_1 queues the subsequently received request from p_3
- Likewise, p_2 queues the subsequently received request from p_1
- Finally, p_3 queues the subsequently received request from p_2
- $\ddot{\frown}$

Maekawas voting algorithm

Safety, Liveness and Fairness

- Safety — Safety is achieved by ensuring that the intersection between any two voting sets is non-empty.
  - A process can only vote (or grant permission) once between each successive “release” message
  - But for any two processes to have concurrent access to the critical section, the non-empty intersection between their voting sets would have to have voted for both processes
- Liveness — As described the protocol does not respect the Liveness property
  - It can however be adapted to use Lamport clocks similar to the previous algorithm
- Fairness — Similarly the Lamport clocks extension to the algorithm allows it to satisfy the Fairness property

Mutual Exclusion Algorithms

Performance Evaluation

- We have four algorithms: central server, ring based, Ricart and Agrawala's and Maekawa's voting algorithm
- We have three logical properties with which to compare them, we can also compare them with respect to performance:
- For performance we are interested in:
  1. The number of messages sent in order to *enter* and *exit* the critical section
  2. The *client delay* incurred at each *entry* and *exit* operation
  3. The *synchronisation delay*, this is delay between one process exiting the critical section and a waiting process entering
- Note: which of these is (more) important depends upon the application domain, and in particular how often critical section access is required

Mutual Exclusion Performance Evaluation

Central Server Algorithm

- Entering the critical section:
  - requires two messages, the request and the reply — even when no other process currently occupies it
  - The client-delay is the time taken for this round-trip
- Exiting the critical section:
  - requires only the sending of the "release" message
  - Incurs no delay for the client, assuming asynchronous message passing.
- The synchronisation-delay is also a round-trip time, the time taken for the "release" message to be sent from client to server and the time taken for the server to send the "grant" message to the next process in the queue.

Mutual Exclusion Performance Evaluation

Ring-based Algorithm

- Entering the critical section:
  - Requires between 0 and N messages
  - Delay, these messages are serialised so the delay is between 0 and N
- Exiting the critical section:
  - Simply requires that the holding process sends the token forward through the ring
- The synchronisation-delay is between 1 and N-1 messages

Mutual Exclusion Performance Evaluation

Ricart and Agrawala

- Entering the critical section:
  - This requires 2(N - 1) messages, assuming that multicast is implemented simply as duplicated message, it requires N-1 requests and N-1 replies.
  - Bandwidth-wise this may be bad, but since these messages are sent and received concurrently the time taken is comparable to the round-trip time of the previous two algorithms
- Exiting the critical section:
  - Zero if no other process has requested entry
  - Must send up to N-1 responses to queued requests, but again if this is asynchronous there is no waiting for a reply
- The synchronisation-delay is only one message, the holder simply responds to the queued request

Mutual Exclusion Performance Evaluation

Maekawa's Voting algorithm

- Entering the critical section:
  - This requires $2 \times \sqrt{N}$ messages
  - As before though, the delay is comparable to a round-trip time
- Exiting the critical section:
  - This requires $\sqrt{N}$ messages
  - The delay though is comparable to a single message
  - The total for entry/exit is thus $3 \times \sqrt{N}$ which compares favourably to Ricart and Agrawala's total of $2(N-1)$ where $N > 4$.
- The synchronisation-delay is a round-trip time as it requires the holding process to multi-cast to its voting set the "release" message and then intersecting processes must send a permission granted message to the requesting process

Mutual Exclusion Performance Evaluation

Further Considerations

- The ring-based algorithm continuously consumes bandwidth as the token is passed around the ring even when no process requires entry
- Ricart and Agrawala — the process that last used the critical section can simply re-use it if no other requests have been received in the meantime

Mutual Exclusion Algorithms

Fault Tolerance

- None of the algorithms described above tolerate loss of messages

Mutual Exclusion Algorithms

Fault Tolerance

- None of the algorithms described above tolerate loss of messages
- The token based algorithms lose the token if such a message is lost meaning no further accesses will be possible

Mutual Exclusion Algorithms

Fault Tolerance

- None of the algorithms described above tolerate loss of messages
- The token based algorithms lose the token if such a message is lost meaning no further accesses will be possible
- Ricart and Agrawala's method will mean that the requesting process will indefinitely wait for (N - 1) "permission granted" messages that will never come because one or more of them have been lost

Mutual Exclusion Algorithms

Fault Tolerance

- None of the algorithms described above tolerate loss of messages
- The token based algorithms lose the token if such a message is lost meaning no further accesses will be possible
- Ricart and Agrawala's method will mean that the requesting process will indefinitely wait for (N - 1) "permission granted" messages that will never come because one or more of them have been lost
- Maekawa's algorithm cannot tolerate message loss without it affecting the system, but parts of the system may be able to proceed unhindered

Fault Tolerance

Process Crashes

- What happens when a process crashes?
    1. Central server, provided the process which crashes is not the central server, does not hold the token and has not requested the token, everything else may proceed unhindered

Fault Tolerance

Process Crashes

- What happens when a process crashes?
  1. Central server, provided the process which crashes is not the central server, does not hold the token and has not requested the token, everything else may proceed unhindered
  2. Ring-based algorithm — complete meltdown, but we may get through up to N-1 critical section accesses in the meantime

Fault Tolerance

Process Crashes

- What happens when a process crashes?
  1. Central server, provided the process which crashes is not the central server, does not hold the token and has not requested the token, everything else may proceed unhindered
  2. Ring-based algorithm — complete meltdown, but we may get through up to N-1 critical section accesses in the meantime
  3. Ricart and Agrawala — complete meltdown, we might get through additional critical section accesses if the failed process has already responded to them. But no subsequent requests will be granted

Fault Tolerance

Process Crashes

- What happens when a process crashes?
  1. Central server, provided the process which crashes is not the central server, does not hold the token and has not requested the token, everything else may proceed unhindered
  2. Ring-based algorithm — complete meltdown, but we may get through up to N-1 critical section accesses in the meantime
  3. Ricart and Agrawala — complete meltdown, we might get through additional critical section accesses if the failed process has already responded to them. But no subsequent requests will be granted
  4. Maekawa's voting algorithm — This can tolerate some process crashes, provided the crashed process is not within the voting set of a process requesting critical section access

Mutual Exclusion Algorithms

Fault Tolerance

- All of these algorithms may be adapted to recover from process failures
- Given a failure detector(s)
- Note, however, that this problem is non-trivial
- In particular because for all of these algorithms a failed process looks much like one which is currently using the critical section
- The key point is that the failure may occur *at any point*
- A synchronous system may be sure that a process has failed and take appropriate action
- An asynchronous system cannot be sure and hence may steal the token from a process currently using the critical section
  - Thus violating the *Safety* property

Mutual Exclusion Fault Tolerance

Considerations

- Central server
  - care must be taken to decide whether the server or the failed process held the token at the time of the failure
  - If the server itself fails a new one must be elected, and any queued requests must be re-made.
- Ring-based algorithm
  - The ring can generally be easily fixed to circumvent the failed process
  - The failed process may have held or blocked the progress of the token
- Ricart and Agrawala
  - Each requesting process should record *which* processes have granted permission rather than simply how many
  - The failed process can simply be removed from the list of those required
- Maekawa's voting algorithm
  - Trickier, the failed process may have been in the intersection between two voting sets

Coordination and Agreement

Elections

- Several algorithms which we have visited until now required a master or nominee process, including:
    1. Berkley algorithm for clock synchronisation
    2. Distributed Debugging
    3. The central server algorithm for mutual exclusion
- Even other algorithms may need a nominee to actually report the results of the algorithm
- For example Chandy and Lamport's snap shot algorithm described how to record the local state at each process in such a way that a consistent global state could be assembled from the local states recorded at different times
- To actually be useful these local states must be gathered together, a simple way to do this is for each local process to send their locally recorded state to a nominee process

Elections

No Fixed Master/Nominee

- A simple way to provide a master process, is to simply name one
- However if the named process fails there should be a recovery plan
- A recovery plan requires that we dynamically decide who should become the new master/nominee
- Even with a fixed order this is non-trivial, in particular as all participants must agree that the current master as failed
- A more dynamic election process can allow for greater flexibility of a running system

Elections

Assumptions and Scenario

- We will assume that any of the N processes may call for an election of a nominee process at any time
- We will assume that no process calls more than one such election concurrently
- But that all N processes may separately call for an election concurrently

Elections

Requirements

- We require that the result of the election should be unique
- (no hung-parliaments or coalitions)
- Even if multiple processes call for an election concurrently
- We will say that the elected process should be the best choice:
  - For our purposes we will have a simple identifier for each process, and the process with the highest identifier should "win" the election
  - In reality the identifier could be any useful property, such as available bandwidth
  - The identifiers should be unique and consist of a total ordering
  - In practice this can be done much like equal Lamport time stamps can be given an artificial ordering using a process identifier/address
  - However care would have to be taken in the case that several properties were used together such as uptime, available bandwidth and geographical location

Elections

Assumptions and Scenario

- Each process at any point in time is either a *participant* or a *non-participant* corresponding to whether the process itself believes it is participating in an election
- Each process p_i has a variable $elected_i$ which contains the identifier of the elected process
- When the process p_i first becomes a participant, the $elected_i$ variable is set to the special value $\perp$
- This means that the process does not yet know the result of the election

Elections

Requirements

- Safety A participant process p_i has $elected_i = \perp$ or $elected_i = P$, where P is chosen as the non-crashed process at the end of the run with the largest identifier
- Liveness All processes participate and eventually either crash or have $elected_i \neq \perp$
- Note that there may be some process p_j which is not yet a participant which has $elected_j = Q$ for some process which is not the eventual winner of the election
- An additional property then could be specified as, no two processes concurrently have $elected_i$ set to two different processes
    - Either one may be set to a process and the other to $\perp$
    - But if they are both set to a process it should be the same one
    - We'll call this property Total Safety

Elections

Election/Nominee Algorithms

- We will look at two distributed election algorithms
    1. A ring-based election algorithm similar to the ring-based mutual-exclusion algorithm
    2. The bully election algorithm
- We will evaluate these algorithms with respect to their performance characteristics, in particular:
    - The total number of messages sent during an election — this is a measure of the bandwidth used
    - The turn-around time, measured by the number of serialised messages sent:
        - Recall Ricart and Agrawala's algorithm for mutual exclusion that required $2(N-1)$ messsages to enter the critical section, but that that time only amounted to a turn-around time, since the only serialisation was that each response message followed a request message.

Elections

Ring-based Election Algorithm

- As with the ring-based mutual exclusion algorithm the ring-based election algorithm requires that the processes are arranged within a logical ring
- Once again this ring is logical and may bear no resemblance to any physical or geographical structure
- As before all messages are sent clockwise around the ring
- We will assume that there are no failures after the algorithm is initiated
- It may have been initiated because of an earlier process failure, but we assume that the ring has been reconstructed following any such loss
- It is also possible that the election is merely due to high computational load on the currently elected process

Ring-based Election Algorithm

Initiating an election

- Initially all processes are marked as “non-participant”
- Any process may begin an election at any time
- To do so, a process p_i:
  1. marks itself as a “participant”
  2. sets the $elected_i$ variable to $\perp$
  3. Creates an election message and places its own identifier within the election message
  4. Sends the election message to its nearest clockwise neighbour in the ring

Ring-based Election Algorithm

Receiving an election message

- When a process p_i receives an election message:
  1. Compares the identifier in the election message with its own
  2. if its own identifier is the lower:
     - It marks itself as a participant
     - sets its $elected_i$ variable to $\perp$
     - forwards the message on to the next clockwise peer in the ring
  3. if its own identifier is higher:
     - It marks itself as a participant
     - sets its $elected_i$ variable to $\perp$
     - Substitutes its own identifier into the election message and forwards it on to the next clockwise peer in the ring

Ring-based Election Algorithm

Receiving an election message

- When a process p_i receives an election message:
  1. Compares the identifier in the election message with its own
  2. if its own identifier is the lower:
     - It marks itself as a participant
     - sets its $elected_i$ variable to $\perp$
     - forwards the message on to the next clockwise peer in the ring
  3. if its own identifier is higher:
     - It marks itself as a participant
     - sets its $elected_i$ variable to $\perp$
     - Substitutes its own identifier into the election message and forwards it on to the next clockwise peer in the ring
  4. if its own identifier is in the received election message:
     - Then it has won the election
     - It marks itself as non-participant
     - sets its $elected_i$ variable to its own identifier
     - and sends an "elected" message with its own identifier to the next clockwise peer in the ring

Ring-based Election Algorithm

Receiving an elected message

- When a process p_i receives an elected message:
  1. marks itself as a non-particpant
  2. sets its $elected_i$ variable to the identifier contained within the elected message
  3. if it is not the winner of the election:
     - forward the *elected* message on to the next clockwise peer in the ring
  4. otherwise The election is over and all peers should have their $elected_i$ variable set to the identifier of the agreed upon elected process

Ring-based Election Algorithm

Required Properties

- Safety:
  - A process must receive its own identifier back before sending an *elected* message
  - Therefore the *election* message containing that identifier must have travelled the entire ring
  - And must therefore have been compared with all process identifiers
  - Since no process updates its $elected_i$ variable until it wins the election or receives an *elected* message no participating process will have its $elected_i$ variable set to anything other than $\perp$
- Liveness:
  - Since there are no failures the liveness property follows from the guaranteed traversals of the ring.

Ring-based Election Algorithm

Performance

- If only a single process starts the election
- Once the process with the highest identifier sends its *election* message (either initiating or because it received one), then the election will consume two full traversals of the ring.
- In the best case, the process with the highest identifier initiated the election, it will take $2 \times N$ messages
- The worst case is when the process with the highest identifier is the nearest anti-clockwise peer from the initiating process In which case it is $(N - 1) + 2 \times N$ messages
- Or $3N - 1$ messages
- The turn-around time is also $3N - 1$ since all the messages are serialised

Elections

The Bully Election Algorithm

- Developed to allow processes to fail/crash during an election
- Important since the current nominee crashing is a common cause for initiating an election
- Big assumption, we assume that all processes know ahead of time, all processes with higher process identifiers
- This can therefore not be used *alone* to elect based on some dynamic property
- There are three kinds of messages in the Bully algorithm
  1. *election* — sent to announce an election
  2. *answer* — sent in response to an election message
  3. *coordinator* — sent to announce the identity of the elected process

The Bully Election Algorithm

Failure Detector

- We are assuming a synchronous system here and so we can build a reliable failure detector
- We assume that message delivery times are bound by T_{trans}
- Further that message processing time is bound by $T_{process}$
- Hence a failure detector can send a process $p_{suspect}$ a message and expect a response within time $T = 2 \times T_{trans} + T_{process}$
- If a response does not occur within that time, the local failure detector can report that the process $p_{suspect}$ has failed

The Bully Election Algorithm

A simple election

- If the process with the highest identifier is still available
- It knows that it is the process with the highest identifier
- It can therefore elect itself by simply sending a *coordinator* message

The Bully Election Algorithm

A simple election

- If the process with the highest identifier is still available
- It knows that it is the process with the highest identifier
- It can therefore elect itself by simply sending a *coordinator* message
- You may wonder why it would ever need to do this
- Imagine a process which can be initiated by any process, but requires some coordinator
    - For example global garbage collection
    - For which we run a global snapshot algorithm
    - And then require a coordinator to:
        1. collect the global state
        2. figure out which objects may be deleted
        3. alert the processes which own those objects to delete them
- The initiator process cannot be sure that the previous coordinator has not failed since the previous run.
- Hence an election is run each time

The Bully Election Algorithm

An actual election

- A process which does not have the highest identifier:
- Begins an election by sending an *election* message to all processes with a higher identifier
- It then awaits the *answer* message from at least one of those processes
- If none arrive within our time bound $T = 2 \times T_{trans} + T_{process}$
  - Our initiator process assumes itself to be the process with the highest identifier who is still alive
  - And therefore sends a *coordinator* message indicating itself to be the newly elected coordinator
- otherwise The process assumes that a *coordinator* message will follow. It may set a timeout for this *coordinator* message to arrive.
- If the timeout is reached before the *coordinator* message arrives the process can begin a new election

The Bully Election Algorithm

Receiving Messages

- *coordinator* If a process receives a *coordinator* message it sets the $elected_i$ variable to the named winner
- *election* If a process receives an *election* message it sends back an *answer* message and begins another election (unless it has already begun one).

The Bully Election Algorithm

Starting a process

- When a process fails a new process may be started to replace it
- When a new process is started it calls for a new election
- If it is the process with the highest identifier this will be a simple election in which it simply sends a *coordinator* message to elect itself
- This is the origin of the name: Bully

The Bully Election Algorithm

Properties

- The *Liveness* property is satisfied.
  - Some processes may only participate in the sense that they receive a *coordinator* message
  - But all non-crashed processes will have set $elected_i$ to something other than $\perp$.
- The *Safety* property is also satisfied if we assume that any process which has crashed, either before or during the election, is not replaced with another process with the same identifier during the election.
- *Total Safety* is not satisfied

The Bully Election Algorithm

Properties

- Unfortunately the *Safety* property is not met if processes may be replaced during a run of the election
  - One process, say p_1, with the highest identifier may be started just as another process p_2 has determined that it is currently the process with the highest identifier
  - In this case both these processes p_1 and p_2 will concurrently send *coordinator* messages announcing themselves as the new coordinator
  - Since there is no guarantee as to the delivery order of messages two other processes may receive these in a different order
  - such that say: p_3 believes the coordinator is p_2 whilst p_4 believes the coordinator is p_1.
- Of course things can also go wrong if the assumption of a synchronous system is incorrect

The Bully Election Algorithm

Performance Evaluation

- In the best case the process with the current highest identifier calls the election
  - It requires (N - 1) *coordinator* messages
  - These are concurrent though so the turnaround time is 1 message
- In the worst case though we require $\mathcal{O}(N^2)$ messages
  - This is the case if the process with the lowest identifier calls for the election
  - In this case $N - 1$ processes all begin elections with processes with higher identifiers
- The turn around time is best if the process with the highest identifier is still alive. In which case it is comparable to a round-trip time.
- Otherwise the turn around time depends on the time bounds for message delivery and processing

Election Algorithms Comparision

Ring-based vs Bully

	Ring Based	Bully
Asynchronous	Yes	No
Allows processes to crash	No	Yes
Satisfies Safety	Yes	Yes/No
Dynamic process identifiers	Yes	No
Dynamic configuration of processes	Maybe	Maybe
Best case performance	$2 \times N$	$N - 1$
Worst case performance	$3 \times N - 1$	$\mathcal{O}(N^2)$

Global Agreement

MultiCast

- Previously we encountered group multicast
- IP multicast and Xcast both delivered “Maybe” semantics
- That is, perhaps some of the recipients of a multicast message receive it and perhaps not
- Here we look at ways in which we can ensure that all members of a group have received a message
- And also that multiples of such messages are received in the correct order
- This is a form of global consensus

Global Agreement

Assumptions and Scenario

- We will assume a known group of individual processes
- Communication between processes is
  - message based
  - one-to-one
  - reliable
- Processes may fail, but only by crashing
  - That is, we suffer from process omission errors but not process arbitrary errors
- Our goal is to implement a $multicast(g, m)$ operation
- Where m is a message and g is the group of processes which should receive the message m

Global Agreement

deliver and receive

- We will use the operation *deliver*(m)
- This delivers the multicast message m to the application layer of the calling process
- This is to distinguish it from the *receive* operation
- In order to implement some failure semantics not all multicast messages received at process p are delivered to the application layer

Global Agreement

Reliable Multicast

- Reliable multicast, with respect to a multicast operation *multicast*(g, m), has three properties:
    1. Integrity — A correct process $p \in g$ delivers a message m at most once and m was multicast by some correct process
    2. Validity — If a correct process multicasts message m then some correct process in g will eventually deliver m
    3. Agreement — If a correct process delivers m then all other correct processes in group g will deliver m
- Validity and Agreement together give the property that if a correct process which multicasts a message it will eventually be delivered at all correct processes

Global Agreement

Basic Multicast

- Suppose we have a reliable one-to-one $send(p, m)$ operation
- We can implement a *Basic Multicast*: $Bmulticast(g, m)$ with a corresponding *Bdeliver* operation as:
  1. $Bmulticast(g, m) =$ for each process p in g:
     - $send(p, m)$
  2. On $receive(m)$: $Bdeliver(m)$
- This works because we can be sure that all messages will eventually receive the multicast message since $send(p, m)$ is reliable
- It does however depend upon the multicasting process not crashing
- Therefore *Bmulticast* does not have the *Agreement* property

Global Agreement

Reliable Multicast

- We will now implement reliable multicast on top of basic multicast
- This is a good example of protocol layering
- We will implement the operations:
- $Rmulticast(g, m)$ and $Rdeliver(m)$
- which are analogous to their $Bmulticast(g, m)$ and $Bdeliver(m)$ counterparts but have additionally the Agreement property

Global Agreement

Reliable Multicast — Using Basic Multicast

- On initialisation: $\mathit{Received} = \{\}$
- Process p to $\mathit{Rmulticast}(g, m)$:
  - $\mathit{Bmulticast}(g \cup p, m)$
- On $\mathit{Bdeliver}(m)$ at process q:
  - If $m \notin \mathit{Received}$
    - $\mathit{Received} = \mathit{Received} \cup \{m\}$
    - If $p \neq q$: $\mathit{Bmulticast}(g, m)$
    - $\mathit{Rdeliver}(m)$

Global Agreement

Reliable Multicast

- Note that we insist that the sending process is in the receiving group, hence:
- *Validity* — is satisfied since the sending process p will deliver to itself
- *Integrity* — is guaranteed because of the integrity of the underlying *Bmulticast* operation in addition to the rule that m is only added to *Received* at most once
- *Agreement* — follows from the fact that every correct process that $Bdelivers(m)$ then performs a $Bmulticast(g, m)$ before it $Rdelivers(m)$.
- However it is somewhat inefficient since each message is sent to each process $\mid g \mid$ times.

Global Agreement

Reliable Multicast Over IP

- So far our multicast (and indeed most of our algorithms) have been described in a vacuum devoid of other communication
- In a real system of course there is other communication going on
- So a reasonable method of implementing reliable multicast is to piggy-back acknowledgements on the back of other messages
- Additionally the concept of a “negative acknowledgement” is used
- A negative acknowledgement is a response indicating that we believe a message has been missed/dropped

Global Agreement

Reliable Multicast
IP

- We assume that groups are closed — not something assumed for the previous algorithm
- When a process p performs an $Rmulticast(g, m)$ it includes in the message:
  - a sequence number S_g^p
  - acknowledgements of the form $\{q, R_g^q\}$
- An acknowledgement $\{q, R_g^q\}$ included in message from process p indicates the latest message multicast from process q that p has delivered.
- So each process p maintains a sequence number R_g^q for every other process q in the group g indicating the messages received from q
- Having performed the multicast of a message with an S_g^p value and any acknowledgements attached, process p then increments its own stored value of S_g^p
- In other words: S_g^p is a sequence number

Global Agreement

- The sequence numbers S_g^p attached to each multicast message, allows the recipients to learn about messages which they have missed
- A process q can $Rdeliver(m)$ only if the sequence number $S_g^p = R_g^p + 1$.
- Immediately following $Rdeliver(m)$ the value R_g^p is incremented
- If an arriving message has a number $S \leq R_g^p$ then process q knows that it has already performed *Rdeliver* on that message and can safely discard it
- If $S > R_g^p$ then the receiving process q knows that it has missed some message from p destined for the group g
- In this case the receiving process q puts the message in a *hold-back* queue and sends a negative acknowledgement to the sending process p requesting the missing message(s)

Global Agreement

Properties

- The hold-back queue is not strictly necessary but it simplifies things since then a simple number can represent all messages that have been delivered
- We assume that IP-multicast can detect message corruption (for which it uses checksums)
- Integrity is therefore satisfied since we can detect duplicates and delete them without delivery
- Validity property holds again because the sending process is in the group and so at least that will deliver the message
- Agreement only holds if messages amongst the group are sent indefinitely and if sent messages are retained (for re-sending) until all groups have acknowledged receipt of it
- Therefore as it stands *Agreement* does not formally hold, though in practice the simple protocol can be modified to give acceptable guarantees of *Agreement*

Global Agreement

Uniform Agreement

- Our Agreement property specifies that if any correct process delivers a message *m* then *all* correct processes deliver the message *m*
- It says nothing about what happens to a failed process
- We can strengthen the condition to Uniform Agreement
- Uniform Agreement states that if a process, whether it then fails or not, delivers a message *m*, then all correct processes also deliver *m*.
- A moment's reflection shows how useful this is, if a process could take some action that put it in an inconsistent state and then fail, recovery would be difficult
- For example applying an update that not all other processes receive

Global Agreement

Ordering

- There are several different ordering schemes for multicast
- The three main distinctions are:
  1. FIFO — If a correct process performs $mulitcast(g, m)$ and then $multicast(g, m')$ then every correct process which delivers m' will deliver m before m'
  2. Causal — If $mulitcast(g, m) \rightarrow multicast(g, m')$ then every process which delivers m' delivers m before m'
  3. Total — If a correct process delivers m before it delivers m' then every correct process which delivers m' delivers m before m'

Global Agreement

Ordering

- There are several different ordering schemes for multicast
- The three main distinctions are:
  1. FIFO — If a correct process performs *mulitcast*(g, m) and then *multicast*(g, m') then every correct process which delivers m' will deliver m before m'
  2. Causal — If *mulitcast*$(g, m) \rightarrow$ *multicast*(g, m') then every process which delivers m' delivers m before m'
  3. Total — If a correct process delivers m before it delivers m' then every correct process which delivers m' delivers m before m'
- Note that Causal ordering implies FIFO ordering
- None of these require or imply *reliable* multicast

Global Agreement

Total Ordering

- As we saw Causal ordering implies FIFO ordering
- But Total ordering is an orthogonal requirement
- Total ordering only requires an ordering on the delivery order, but that ordering says nothing of the order in which messages were sent
- Hence Total ordering can be combined with FIFO and Causal ordering
- FIFO-Total ordering or Causal-Total ordering

Multicast Ordering

Implementing FIFO Ordering

- Our previous algorithm for reliable multicasting
- More generally sequence numbers are used to ensure FIFO ordering

Multicast Ordering

Implementing Causal Ordering

- To implement Causal ordering on top of Basic Multicast (*bmulticast*)
- Each process maintains a vector clock
- To send a Causal Ordered multicast a process first uses a *bmulticast*
- When a process p_i performs a $bdeliver(m)$ that was multicast by a process p_j it places it in the holding queue until:
    - It has delivered any earlier message sent by p_j
    - and
    - It has delivered any message that had been delivered at p_j before p_j multicast m
- Both of these conditions can be determined by examining the vector timestamps

Global Agreement

Implementing Total Ordering

- There are two techniques to implementing Total Ordering:
  1. Using a sequencer process
  2. Using *bmulticast* to illicit proposed sequence numbers from all receivers

Implementing Total Ordering

Using a sequencer

- Using a sequencer process is straight forward
- To total-ordered multicast a message *m* a process *p* first sends the message to the sequencer
- The sequencer can determine message sequence numbers based purely on the order in which they arrive at the sequencer
    - Though it could also use process sequence numbers or Lamport timestamps should we wish to, for example, provide FIFO-Total or Causal-Total ordering
- Once determined, the sequencer can either *bmulticast* the message itself
- Or, to reduce the load on the sequencer, it may just respond to process *p* with the sequence number which then itself performs the *bmulticast*

Implementing Total Ordering

Using Collective Agreement

- To total-order multicast a message, the process *p* first performs a *bmulticast* to the group
- Each process then responds with a proposal for the agreed sequence number
    - And puts the message in its hold-back queue with the suggested sequence number provisionally in place
- Once the process *p* receives all such responses it selects the largest proposed sequence number and replies to each process (or uses *bmulticast*) with the agreed upon value
- Each receiving process then uses this agreed sequence number to deliver (that is TO-deliver) the message at the correct point

Ordered Multicast

Overlapping Groups

- So far we have been happy to assume that each receiving process belongs to exactly one multicast group
- Or that for overlapping groups the order is unimportant
- For some applications this is insufficient and our orderings can be updated to account for overlapping groups

Ordered Multicast

Overlapping Groups

- Global FIFO Ordering If a correct process issues $multicast(g, m)$ and then $multicast(g', m')$ then every correct process in $g \cap g'$ that delivers m' delivers m before m'
- Global Causal Ordering If $multicast(g, m) \rightarrow multicast(g', m')$ then every correct process in $g \cap g'$ that delivers m' delivers m before m'
- Pairwise Total Ordering If a correct process delivers message m sent to g before it delivers m' sent to g' then every correct process in $g \cap g'$ which delivers m' delivers m before m'

Ordered Multicast

Overlapping Groups

- Global FIFO Ordering If a correct process issues $multicast(g, m)$ and then $multicast(g', m')$ then every correct process in $g \cap g'$ that delivers m' delivers m before m'
- Global Causal Ordering If $multicast(g, m) \rightarrow multicast(g', m')$ then every correct process in $g \cap g'$ that delivers m' delivers m before m'
- Pairwise Total Ordering If a correct process delivers message m sent to g before it delivers m' sent to g' then every correct process in $g \cap g'$ which delivers m' delivers m before m'
- A simple, but inefficient way, to do this is force all multicasts to be to the group $g \cup g'$, receiving processes then simply ignore the multicast messages not intended for them.
- e.g. process $p \in g - g'$ ignore multicast messages sent to g'

Summary

Further Thoughts

- These algorithms to perform mutual exclusion, nominee election and agreed multicast suffer many drawbacks
- Many are subject to some assumptions which may be unreasonable
- Particularly when the network used is not a Local Area Network
- These problems can be, and are, overcome
- But for each individual application the designer should consider whether the assumptions are a problem
- It may be that coming up with a solution which is less optimal but does not rely on, say, a reliable communication network, may be the best approach
- For example, Routing Information Protocol

Consensus

Three Kinds

- The problems of mutual exclusion, electing a nominee and multicast are all instances of the more general problem of consensus.
- Consensus problems more generally then are described as one of three kinds:
    1. Consensus
    2. Byzantine Generals
    3. Interactive Consensus

Global Agreement

Consensus

- A set of processes $\{p_1, p_2, \ldots p_n\}$ each begins in the *undecided* state
- Each proposes a single value v_i
- The processes then communicate, exchanging values
- To conclude, each process must set their decision variable d_i to one value and thus enter the *decided* state
- Three desired properties:
  - Termination: each process sets its $decision_i$ variable
  - Agreement: If p_i and p_j are correct processes and have both entered the *decided* state, then $d_i = d_j$
  - Integrity: If the correct processes all proposed the same value v, then any correct process p_i in the *decided* state has $d_i = v$

Global Agreement

Byzantine Generals

- Imagine three or more generals are to decide whether or not to attack
- We assume that there is a commander who issues the order
- The others must decide whether or not to attack
- Either the lieutenants or the commander can be faulty and thus send incorrect values
- Three desired properties:
  - Termination: each process sets its *decision*$_i$ variable
  - Agreement: If p_i and p_j are correct processes and have both entered the *decided* state, then $d_i = d_j$
  - Integrity: If the commander is correct then all correct processes decide on the value proposed by the commander
- When the commander is correct, *Integrity* implies *Agreement*, but the commander may not be correct

Global Agreement

Interactive Consensus

- Each process proposes its own value and the goal is for each process to agree on a vector of values
- Similar to consensus other than that each process contributes only a part of the final answer which we call the *decision vector*
- Three desired properties:
  - Termination: each process sets its $decision_i$ variable
  - Agreement: The final decision vector of all processes is the same
  - Integrity: If p_i is correct and proposes v_i then all correct processes decide on v_i as the ith component of the decision vector

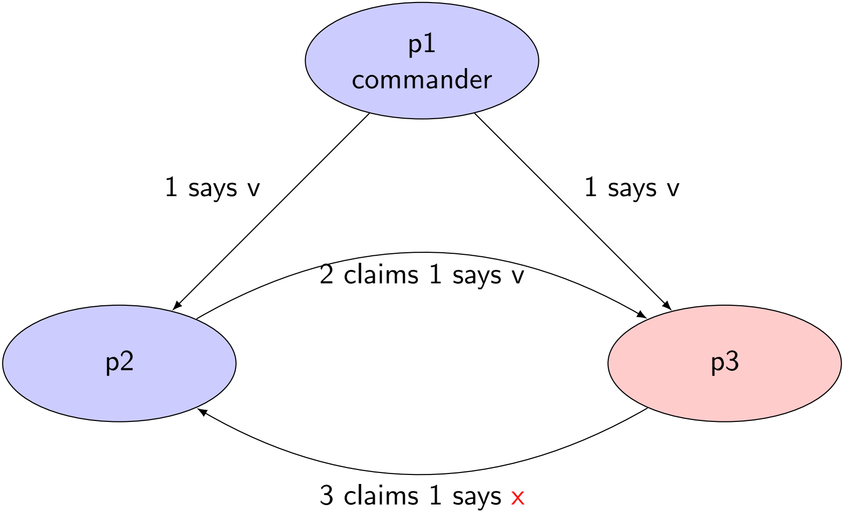
Global Agreement

Relating the three

- Assuming we had a solution to any of the three problems we could construct a solution to the other two
- For example, if we have a solution to Interactive Consensus, then we have a solution to Consensus, all we require is some way consistent function for choosing a single component of the decision vector
  - We might choose a majority function, maximum, minimum or some other function depending on the application
  - It only requires that the function is context independent
- If we have a solution to the Byzantine Generals then we can construct a solution to Interactive Consensus
  - To do so we simply run the Byzantine Generals solution N times, once for each process
- The point is not necessarily that this would be the way to implement such as solution (it may not be efficient)
  - However if we can determine an impossibility result for one of these problems we know that we also have the same result for the others

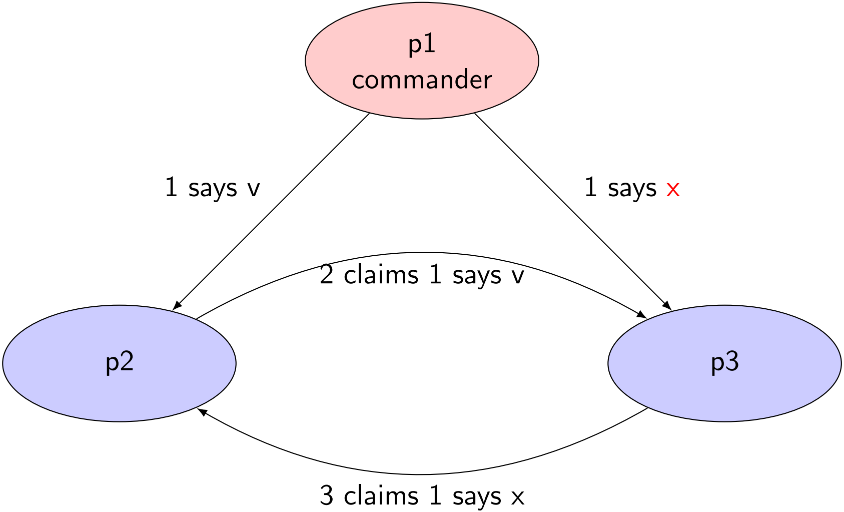
Global Agreement

Byzantine Generals in a Synchronous System



Global Agreement

Byzantine Generals in a Synchronous System



Global Agreement

Impossible

- Recall:
  - Agreement: If p_i and p_j are correct processes and have both entered the *decided* state, then $d_i = d_j$
  - Integrity: If the commander is correct then all correct processes decide on the value proposed by the commander
- In both scenarios, process p_2 receives different values from the commander p_1 and the other process p_3
- It can therefore know that one process is faulty but cannot know which one
- By the *Integrity* property then it is bound to choose the value given by the commander
- By symmetry the process p_3 is in the same situation when the commander is faulty.
- Hence when the commander is faulty there is no way to satisfy the *Agreement* property, so no solution exists for three processes

Global Agreement

$N \leq 3 \times f$

- In the above case we had three processes and at most one incorrect process, hence $N = 3$ and $f = 1$
- It has been shown, by Pease *et al* that more generally no solution can exist whenever $N \leq 3 \times f$
- However there can exist a solution whenever $N > 3 \times f$
- Such algorithms consist of *rounds* of messages
- It is known that such algorithms require at least $f + 1$ message rounds
- The complexity and cost of such algorithms suggest that they are only applicable where the threat is great
- That means either the threat of an incorrect or malicious process is great
- and/or the cost of failing due to inability to reach consensus is large

Global Agreement

Consensus in an Asynchronous System

- Fisher *et al* have shown that it is impossible to design an algorithm which is guaranteed to reach consensus in an asynchronous system, under the following condition:

Global Agreement

Consensus in an Asynchronous System

- Fisher *et al* have shown that it is impossible to design an algorithm which is guaranteed to reach consensus in an asynchronous system, under the following condition:
  - We allow a single process crash failure
- Even if we have 1000s of processes, and the failure is a crash rather than an arbitrary failure of just a single process, any consensus algorithm is not guaranteed to reach consensus
- Clearly this is a pretty benign set of circumstances
- We therefore know that there is no solution in an asynchronous system to either:
  1. Byzantine generals (and hence consensus or interactive consensus)
  2. Totally order and reliable multicast

Consensus in an Asynchronous System

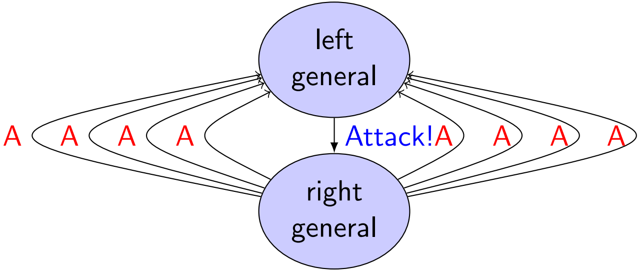
So what to do?

- The important word in the previous impossibility result is: guarantee
- There is no algorithm which is guaranteed to reach consensus
- Consensus has been reached in asynchronous systems for years
- Some techniques for getting around the impossibility result:
  - Masking process failures, for example using persistant storage such that a crashed process can be replaced by one in effectively the same state
    - Thus meaning some operations appear to take a long time, but all operations do eventually complete
  - Employ failure detectors:
    - Although in an asynchronous system we cannot achieve a reliable failure detector
    - We can use one which is “perfect by design”
    - Once a process is deemed to have failed, any subsequent messages that it does send (showing that it had not failed) are ignored
    - To do this the other processes must agree that a given process has failed

Consensus in an Asynchronous System

Back to the pair of attacking generals

A = Acknowledge Attack!



- If the probability of any one message being dropped is 0.5
- Then the probability that two acknowledgements fail to be returned is 0.25
- For 3 it is 0.125 etc, for 8 it is $\frac{1}{256} = 0.0039$
- In reality we have to consider the probability that the message is not dropped but not received by some time out value t
- This complicates the calculation but not the general idea

Coordination and Agreement

Summary

- We looked at the problem of Mutual Exclusion in a distributed system
  - Giving four algorithms:
    1. Central server algorithm
    2. Ring-based algorithm
    3. Ricart and Agrawala's algorithm
    4. Maekawa's voting algorithm
  - Each had different characteristics for:
    1. Performance, in terms of bandwidth and time
    2. Guarantees, largely the difficulty of providing the *Fairness* property
    3. Tolerance to process crashes
- We then looked at two algorithms for electing a master or nominee process
- Then we looked at providing multicast with a variety of guarantees in terms of delivery and delivery order

Coordination and Agreement

Summary

- We then noted that these were all specialised versions of the more general case of obtaining consensus
- We defined three general cases for consensus which could be used for the above three problems
- We noted that a synchronous system can make some guarantee about reaching consensus in the existance of a limited number of process failures
- But that even a single process failure limits our ability to guarantee reaching consensus in an asynchronous system
- In reality we live with this impossibility and try to figure out ways to minimise the damage

Any Questions

Any Questions?