Embedded Systems Engineering

Algorithms for Distributed Systems

- Ordering of Events
- Implementing Global Time
- Stable Storage
- Reaching Agreement in the presence of Faulty Processes

Michael Brockway

Ordering of Events

- Uniprocessor, tightly coupled systems have common memory, clock; but
- In a distributed system, delays in transit of a message between processors might mean
  - Given a sequence of events, two processors might not observe identically the same sequence
  - If event A *causes* event B, all observers (processes) should *see* A occurring before B -- this might not happen in a distributed system.
- Processes in a distributed system need to co-ordinate and synchronise their activities in response to activities as they occur
  - To be sure we are avoiding deadlock between processes sharing resources, we need to know process X released resource R before requesting resource S.
- So we need an algorithm to place a causal order on events, even when they occur on different processes at different places on the network. The following algorithm is due to Lamport (1978: see references).

Ordering of Events

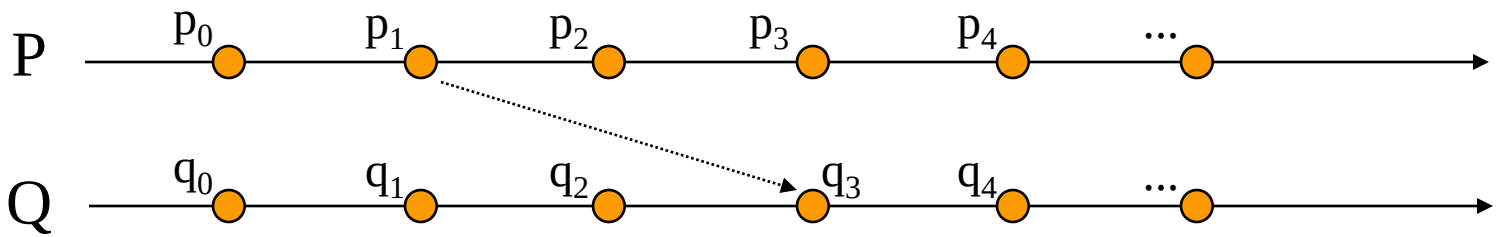
- Consider a process P consisting of events (actions) p_0, p_1, p_2, p_3, p_4,... p_n in sequence on a processor.
- So p_0 must have happened before p_1, which must have happened before p_2, etc.
- Write this as

$$p_0 \rightarrow p_1 \rightarrow p_2 \rightarrow p_3 \rightarrow p_4 \rightarrow \ldots \rightarrow p_n$$

- Similarly another process Q:

$$q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4 \rightarrow \ldots \rightarrow q_n$$

- These processes might be distributed across two different processors, and there might be no communication between them. Then is impossible to say whether $p_0 \rightarrow q_o$ or $q_0 \rightarrow p_0$ -- which happened before the other -- and so on.
- But suppose event p_1 is the sending of a certain message through the network and q_3 is the receipt of this message:



Ordering of Events

- A message must be received after it is sent, so $p_1 \rightarrow q_3$
- It follows ($\rightarrow$ is transitive) that $p_1 \rightarrow q_4$, $p_1 \rightarrow q_5$ etc. Action p_1 can have a causal effect on q_3 and later Q event/actions
- But we still do not know whether p_0 or q_0 happened first.
  - These are *concurrent* events
  - There is no causal ordering
  - In the absence of communication between the tasks, a process *doesn't care* which happened first.
- We can impose an ordering on such a pair of events arbitrarily; but it is important that all processes which might make decisions based on this order assume the *same* order.
- To order events totally in a distributed system we *time-stamp* each event.
  - May be a *logical* rather than a *physical* time stamp -- a sequence number
  - Each processor keeps a "logical clock" -- variable which is incremented each time an event occurs, and whose new value time-stamps the event.
  - Thus $p_j \rightarrow p_k$ iff timeStamp(p_j) < timeStamp(p_k)

Ordering of Events

- It is possible for the logical clocks on two processors to get out of synchronisation.
  - P's logical clock at p_1 could be > Q's logical clock at q_3, giving an anomaly of time stamps.
  - We can resolve this by requiring a message to carry the time-stamp of its sending event, and requiring the receiving process to update its logical clock at the receiving event
    - to this time stamp + (at least) 1 in the case of asynchronous messages,
    - to this time stamp in the case of synchronous message passing

- This algorithm ensures all events can be ordered by time stamp and this ordering can be deemed to be "causal"
  - Where timeStamp(p) < timeStamp(q) we can, for practical purposes, assume p "caused" q
  - Where two events happened to have equal time stamps, we can resolve the ordering by some arbitrary condition, such as the numeric process Ids.

Implementing Global Time

- In some applications (eg hard real-time) it may be appropriate for all processors to *physically* time-stamp events.
- Usually, the nodes in a distributed system each have their own clock and the problem is to keep them synchronised.
  - Two quartz crystal clocks will drift by around a second per 6 days -- 1 millisecond (ms) in 8 minutes, or 2 microseconds (μs) per second
  - To synchronise a clock that is out of sync, you may not jump it forward or back -- time must be *monotonic*.
  - Rather, a fast clock must be slowed down, eg by leaving out every n^{th} tick, until it is compensated.
  - a slow clock must be speeded up, eg by inserting some extra ticks, for a few ticks, until it has caught up.
- A clock coordination algorithm should provide a bound, Δ, on the maximum difference between any two clocks; so event p can be presumed to precede event q iff $time(p) + \Delta < time(q)$.

- Question: How to achieve synchronisation between nodes?

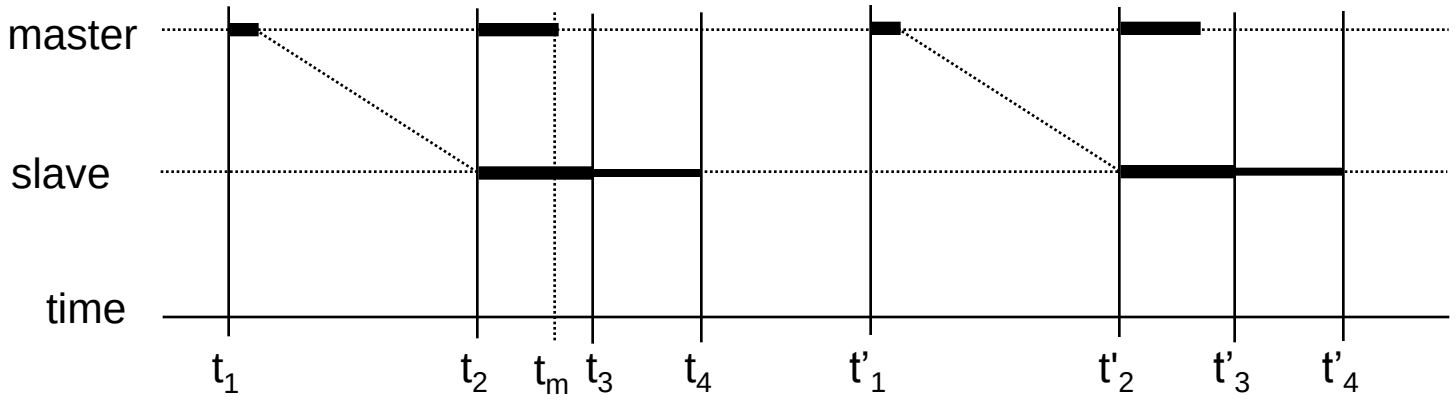
Implementing Global Time

- How to achieve synchronisation between nodes?
- One way:
  - A “master time server” process keeps time using an accurate “master” clock
  - On request (message) by another process, the master supplies the time according to its clock
  - There are sources of error in a distributed system -
    - The time taken by the message to the master and its reply
    - Non-determinism in the responsiveness of the client once it receives the message.
  - the master must not be pre-empted between reading its clock and sending the message
  - the master process and the messages can be given high priority to minimise these errors
- Another approach
  - The client sends the server S a message containing its own time reading
  - On receipt, S reads its own clock and sends back a correction factor
    $\delta = t(client) - t(S) + [min\ message\ com\ delay]$
  - The client advances its clock if $\delta < 0$ or retards it if $\delta > 0$.

Implementing Global Time

- An interesting example applicable to CAN-bus based systems has been Proposed by Martin Gergeleit and Hermann Streich (see references)
- Their scheme can be used on any network in which
  - A successfully sent frame arrives at all nodes with a fixed and known delay which can be approximated as 0, some other constant, or a function of the receiving node.
  - The delay from transmission & reception of a frame to the interrupt service routine which time-stamps it is known to high accuracy -- may be a function of the receiving node.
  - There is a guaranteed bound t_{max} on the time between two valid synchronisation messages.
- Their protocol works as follows.
  - The master process periodically broadcasts a time stamp message
  - The value if this time stamp is time on its master clock at the moment of indication of successful transmission of the last synchronisation message
    - Presumed synchronous with reception of the message on the slave nodes

Implementing Global Time



- Two rounds of synchronisation are shown
  - At t_1, t'_1 etc the master prepares its time stamp message.
  - At t_2, t'_2 etc the slave recognises the message
  - At t_3, t'_3 the slave has read the message; it takes a *local* time stamp at t_3, t'_3
  - The data in the message is the master's time stamp for the same "instant" in the *previous* synchronisation round. For instance, the time stamp the master sends in round t'_1 to t'_4 is the time on its clock corresponding to t_3 (t_m in the diagram) The slave has saved its time stamp t_3 during t'_3 to t'_4 can work out the difference $[t_3 - t_m]$ and use it to correct its clock (t'_4).
  - The local (slave) time stamp at t'_3 will be carried forward to the next round, t''_1 to t''_4 (not shown)

Implementing Global Time

- The differences $t_3 - t_2$ (and $t'_3 - t'_2$, etc) and $t_m - t_2$ (and corresponding numbers for in other rounds) are fixed and known
- $t_{max} = t'_2 - t_2$, the maximum time between synchronisation rounds is given
- The slave's clock is synchronised at t_4, t'_4, ...
- The protocol guarantees that at all times, time on the slave clock differs from time on the master clock by less than $2\ t_{max}\ \Delta_s$ where Δ_s is the drift of the slave clock
  - Can be made as small as you like by having synchronisation rounds sufficiently often

- This is the ideal protocol!
  - In reality there are inaccuracies
    - δ_m in our knowledge of $t_m - t_2$
    - δ_s in our knowledge of $t_3 - t_2$
  - The bound the master - slave difference is in practice $2\ t\ \Delta_s + \delta_m + \delta_s$.

- In testing the protocol, these inaccuracies turn out to be significant

Implementing Global Time

- A single “master” clock process as in these examples gives a single point of failure

- Some schemes are “decentralised”
  - All nodes broadcast their “time”
  - A consensus is taken
  - There is an *agreement* policy
    - There is agreement only if the skew between non-faulty clocks is with a bound
  - There is an *accuracy* policy
    - The condition is satisfies only if all non-faulty clocks have bounded drift with respect to real time.

Implementing Stable Storage

- We want storage which will survive a processor crash
- Writes to disk are not atomic
  - a crash could occur part way though
  - the stored data are now in an inconsistent state
- A common way of dealing with this is to store everything twice, perhaps on separate disks (or other non-volatile media) on different processors
- Assume each disk unit can indicate whether the last write completed successfully
  - CRC, etc
- Normal operation:
  - Write the first copy repeatedly until successful completion
  - Then write the second copy
- Recovering after a crash:
  - Try to read the first copy
  - Try to read the second copy
  - If both are readable and agree then assume the crash did not corrupt the data
  - If one copy is readable and one unreadable, copy the good copy to the bad
  - If both are readable but different then copy first copy to the second.

Reaching Agreement in the presence of Faulty Processes

- The algorithm above presumes the processor stopped immediately on crashing; but some spurious state transitions or messages or other actions could have occurred.
- We can still end up with inconsistent data.
- We cannot guarantee fault tolerance using just a finite amount of hardware; but assume a *bounded* number of failures. How can a group of processes executing on different (but communicating) processors reach a consensus in the presence of a *bounded* number of faulty processes?

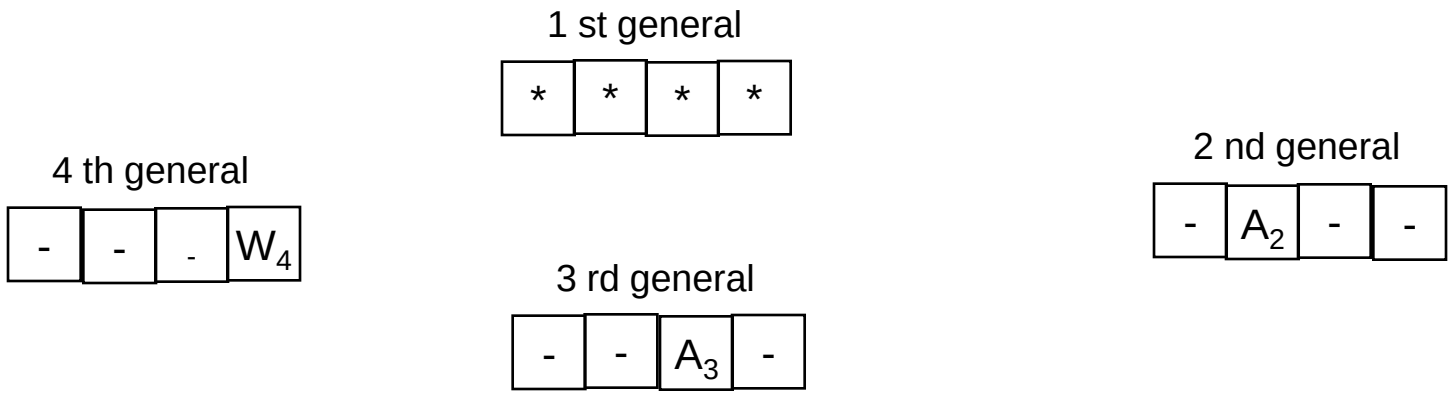
- The Byzantine Generals Problem (Lamport and others, 1982)
  - Several divisions of the Byzantine army, each commanded by a general, are surrounding an enemy camp. The generals can communicate by messengers and must come to agreement about whether to attack the camp. Some of the generals are traitors and may communicate false information. How can it be ensured that all the loyal generals obtain the same information?
  - In general, 3 m + 1 generals can cope with m traitors by using m + 1 rounds of message exchanges.

Reaching Agreement in the presence of Faulty Processes

- The Byzantine Generals Problem - example with 4 generals and 1 traitor
  - Assume every message sent is delivered correctly
  - Assume a receiver of a message knows who sent it.
  - Assume the absence of a message is detectable.
  - Each message contains one of: **attack** (A), **retreat** (R) or **wait** (W)
- Each general G_n maintains a array $I_n[\ ]$ of information received from the other generals
  - $I_n[k]$ = information G_n received from G_k
- Initially
  - $I_n[n] = O_n$ = the information observed by G_n (= A_n or R_n or W_n)
  - $I_n[k]$ = null for $k \neq n$
- Each general sends every other general a message containing their observation.
  - A loyal general G_n will send the correct observation O_n
  - A traitor may send false information, and may send different information to different generals.

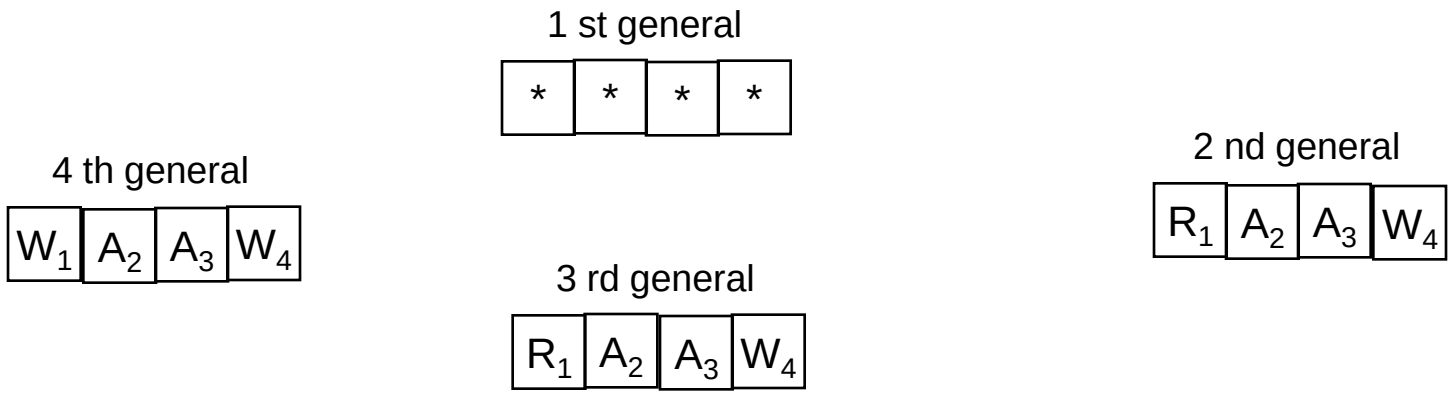
Reaching Agreement in the presence of Faulty Processes

- On receiving these observations, each general updates his/her array and sends on to each of the other three generals the observations received from the remaining *two*. (Ie, s/he does not send another general the data just received from him/her.)
  - A loyal general does this accurately
  - A traitor may may report falsely on what he/she received, or not report at all
- After this exchange of messages, each general can construct an array from the majority value of the three values received from the other three generals' reported observations.
  - If no majority exists, assume no observations have been made.
- For example suppose G_1 is a traitor, G_2 says attack, G_3 says attack, G_4 says wait. Here is the initial state of the arrays:



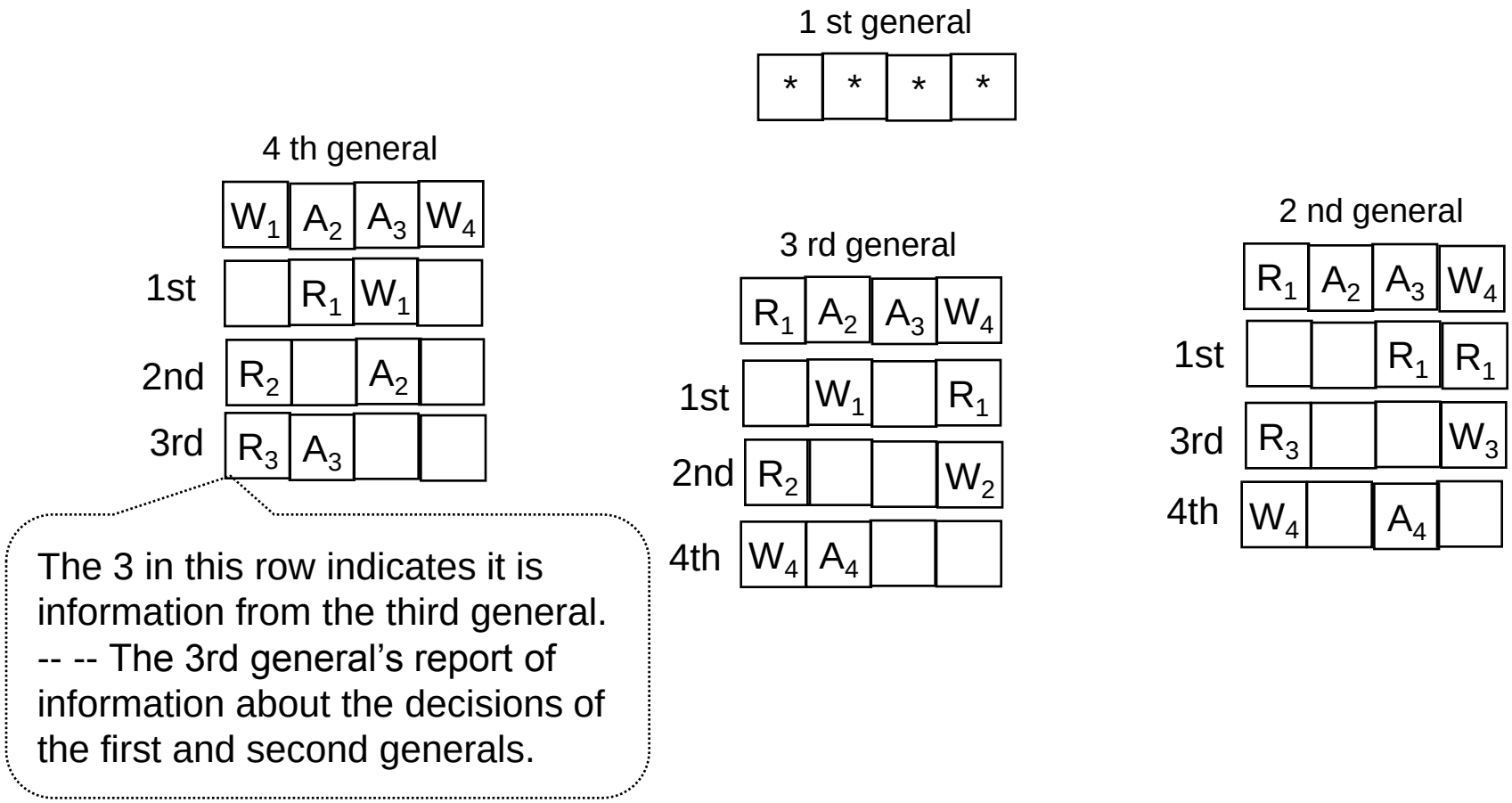
Reaching Agreement in the presence of Faulty Processes

- The generals each report their observation to the others. The traitor, realising the camp is vulnerable, sends “retreat” and “wait” messages to the others at random.
- So after the first round, each general has updated their array with information from the other three:



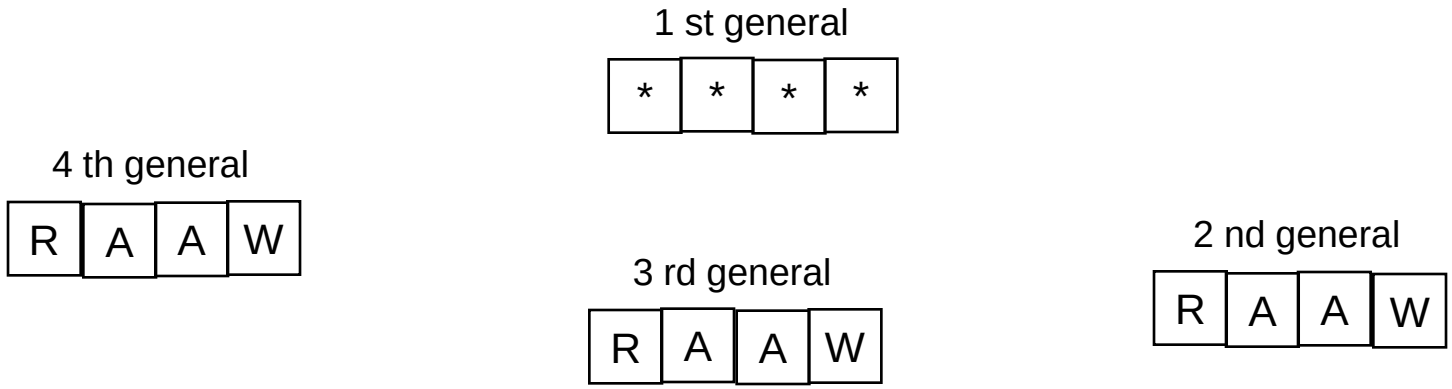
Reaching Agreement in the presence of Faulty Processes

- Now each general sends each other general the information s/he received from remaining two. Again, the traitor G_1 sends R or W at random.



Reaching Agreement in the presence of Faulty Processes

- Now each general has a clear majority view in each column:



- Note that the three loyal generals have come to a consistent view -- a consensus

Reaching Agreement in the presence of Faulty Processes

- If it is possible to restrict the actions of traitor further (a stricter failure model), the number of generals (processors) required to tolerate *m* traitors (failures) can be reduced.
  - Eg if a traitor is unable to modify a loyal general's observation when passing it on (eg must passed a signed copy) the only 2 *m* + 1 generals (processors) are required.
- Using solutions of the Byzantine generals problem and its generalisations, one can construct a fail-silent processor by having internally replicated processors carry out a Byzantine agreement. If the non-faulty processors detect a disagreement they stop execution
- See Schneider (1984)

References

- Alan Burns and Andy Wellings, *Real-Time Systems and Programming Languages*, Addison-Wesley (3rd ed, 2001)

- Lamport, L (1978) *Time, clocks and the ordering of event sin a distributed system*, CACM 21(7) 558-565

- Gergeleit, M & Streich, H, *Implementing a Distributed High-Resolution Real-Time Clock using the CAN-Bus*, `http://ais.gmd.de/RS/Papers/CAN-clock/CAN-clock.html`

- Lamport, L , Shostak, R & Pease, M (1982). *The Byzantine Generals Problem* Transactions on Programming Languages & Systems, 4(3) 382-401

- Scheider, F (1984). Byzantine Generals in Action: Implementing fail-stop processors, Transactions on Computer Systems, 2(2) 145-154