Backward error recovery

- Interested in implementing servers that tolerate machine crashes
  - processor service with partial amnesia crash failures (contents of primary memory are lost)
- *Backward error recovery* refers to the situation where we revert (roll-back) to a previous (saved) state on failure and continue from that point
- Simplest strategy is to periodically *checkpoint* the server's state to disk and restart the server from its last saved checkpoint

For example:

- A server which generates unique identifiers (UIds)
  - assume that a new UId is obtained by incrementing a counter

Checkpointing 1/4

Time

Server	Checkpoint
return 0	
return 1	
return 2	
checkpoint	
return 3	
return 4	last UId was 2
crash	
recover	
return 3!!!!!	

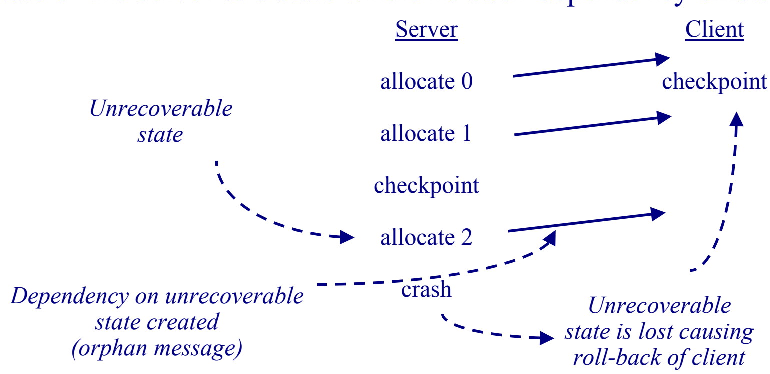
Multiple clients might be allocated the same *unique* identifier - WHY!?

Checkpointing 2/4

- Basic problem is that changes made to the server between the last checkpoint and the crash are *unrecoverable*
  - → i.e. its as if UIds 3 and 4 were never allocated - history has been undone!
- If we consider the server as a state machine, then the states in which UIds 3 and 4 were allocated have been *lost*
- This is not a problem if the lost states were not visible outside of the server
- However in the example the clients who were originally allocated UIds 3 and 4 have observed these lost states
- In particular, these clients *depend* on the lost states
  - → c.f. isolation property of transactions (dirty read)

Checkpointing 3/4

- One solution is to roll-back any client that has a dependency on a lost state of the server to a state where no such dependency exists



- Of course any process that has a dependency on an unrecoverable state of the client must also roll back - cascading roll-backs (*the domino effect*)

Checkpointing 4/4

- Possible solution to this problem is to *prevent* dependencies on unrecoverable states from arising
- Could checkpoint before replying to each request
  - → synchronous disk write - must make sure that the block is flushed to disk before replying
  - → may be practical for this example but not if the state to be checkpointed is larger
- Better solution is to keep a *log* of all requests received since last checkpoint and replay these in order after restoring checkpoint
  - → again synchronous write - but only of request message

Using checkpoints and message logs 1/2

Time ↓

Server	Checkpoint	Message Log
return 0		<req>
return 1		<req,req>
return 2		<req,req,req>
checkpoint	<UId = 2>	<>
return 3		<req>
return 4		<req,req>
crash		
recover		
replay		
replay		
return 5		<req,req,req>

Using checkpoints and message logs 2/2

- Each request must be logged before result is returned
  - in case crash between returning result and writing log record
  - still pay one disk write per request
- Requires that server execution is *deterministic*
  - so replaying the messages in the log will yield the same state after recovery as before failure
  - needs care if server is multi-threaded!
- Any other inputs to the server must be logged too
  - e.g. terminal input
- Sending of (duplicate) reply messages must be suppressed during replay
  - unless clients can detect duplicates
- Keep reply messages in case clients retransmit requests
  - e.g. crash after logging message but before replying

Sender-based message logging 1/6

See [Johnson and Zwaenepoel 1987]

- A cheaper alternative to logging requests on disk
- Supports recovery from a *single* failure at a time
- Processes must be deterministic
- Basic idea is to keep the message log in the sender's volatile memory
- On recovery each process restores its state from its checkpoint and asks its correspondents to resend their messages to it
- Messages must be replayed in the same order as originally processed
- Each message carries a Send Sequence Number (SSN) for duplicate suppression

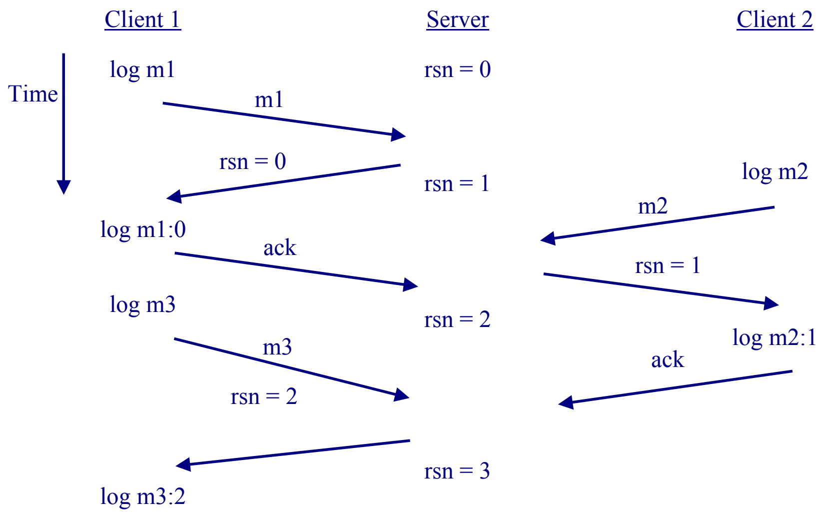
Sender-based message logging 2/6

- To solve the ordering problem each receiver assigns a Receive Sequence Number (RSN) to each message that it receives
- Messages are replayed in RSN order
- RSNs are returned to senders and must be added to the corresponding messages in the log
- A message for which no RSN has yet been logged is known as a *partially logged* message

Sender-based message logging 3/6

- The state resulting from processing a message is unrecoverable while the message is partially logged
- The receiver may not send further messages or perform I/O while there are any partially logged messages
- Sender must ack each RSN
- During recovery partially logged messages may be replayed in any order after all fully logged messages
  - → e.g. if the server fails without returning an RSN for some message

Sender-based message logging 4/6



Sender-based message logging 5/6

During recovery:

- process is restored from checkpoint (includes its highest used SSN/RSN)
- process broadcasts to solicit messages to it with RSNs greater than the checkpoint RSN (or no RSN)
- fully-logged messages are processed in RSN order
- if recovering process (re)sends any messages these will have the same SSN as before and are added to its local log
- suppression of these messages is the responsibility of their receivers
- the receiver of such a duplicate must return its RSN or an indication that the message need not be logged (if it has checkpointed)
- partially logged and new messages now processed in any order

Sender-based message logging 6/6

- Checkpoint includes state of process, its current message log and its highest used SSN and RSN
- Once a process checkpoints, log records for messages sent to it with RSNs lower than the checkpoint RSN can be discarded

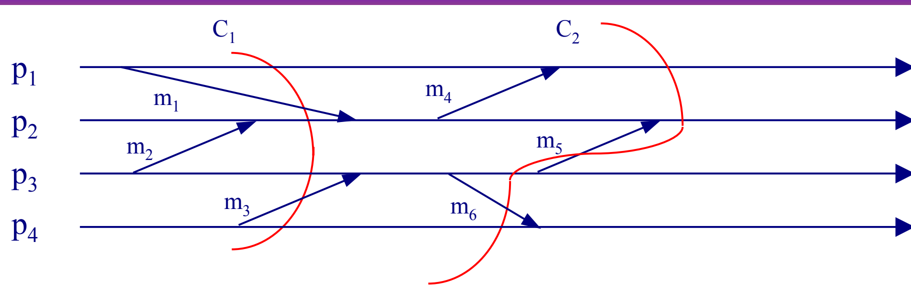
Optimistic approaches to checkpointing

- Both the basic and sender-based checkpointing/message logging schemes *prevent* dependencies on unrecoverable states from arising
- These protocols are *pessimistic* in that they assume that failures will occur and prevent communication/output that could lead to inconsistency
- Possible to have an *optimistic* protocol which will allow communication between processes to proceed even while some messages are not logged
- Such protocols are optimistic in that they assume that logging will normally complete without failure
- Recovery in this case may involve rolling back multiple processes
- May support *non-deterministic* processes
- Still necessary to delay output from unrecoverable states

Distributed checkpointing 1/3

- Yet another approach is to take a *distributed checkpoint* covering a set of communicating processes rather than a single process
- Such a checkpoint must capture a *consistent* state of the processes
- The major criterion for consistency is that after recovery no process should be in state where it has received an *orphan* message
- Since there is no message log, such protocols must handle message loss due to processor crashes
- Like optimistic approaches, distributed checkpointing may support *non-deterministic* processes
- Must not allow output from unrecoverable states

Distributed checkpointing 2/3



- Note that C_1 represents a consistent checkpoint while C_2 does not
- In C_2, p_2 has received an orphan message (m_5) from p_3
- Also note that rolling back to C_1 may cause message loss (m_1, m_3)

Distributed checkpointing 3/3

- Design algorithms for taking a consistent checkpoint of the state of a set of communicating processes and for rolling back to the checkpoint state after the failure of one of the processes
- For extra brownie points, ensure that your algorithms only require that the minimum number of processes checkpoint/rollback as required
- Hints:
  - Taking a checkpoint may be initiated by any process
  - Your algorithm may give rise to lost messages

Taking a distributed checkpoint 1/9

See [Koo and Toueg 1987]

- Assume that we can identify a set of mutually consistent local states of the required processes
- Failures during checkpointing might prevent one or more processes from checkpointing their local states
- So, we need to ensure that all required processes checkpoint or none do
  - taking a checkpoint should be an atomic (all-or-nothing) operation
  - even in the presence of failures
- Algorithm requires two rounds of communication initiated by process that decides to take the checkpoint - *the initiator*
  - essentially 2PC

Taking a distributed checkpoint 2/9

Round one:

- Initiator
  - takes a *tentative* checkpoint
  - requests all required processes to make tentative checkpoints
- Other processes
  - may/may not take tentative checkpoints
  - inform initiator of their action
  - may *not* send any further messages until algorithm is complete

Round two:

- If initiator establishes that all processes have taken tentative checkpoints
  - it makes its checkpoint *permanent*
  - requests others to do likewise
- Other processes comply with initiator's request

Taking a distributed checkpoint 3/9

Why no orphan messages?

- Suppose p_1 sends m_1 to p_2
- For m_1 to be an orphan
  - reception of m_1 would have to occur before p_2 takes its checkpoint
  - sending of m_1 would have to occur after checkpoint of p_1
- No sending allowed after taking checkpoint until all other processes have checkpointed!

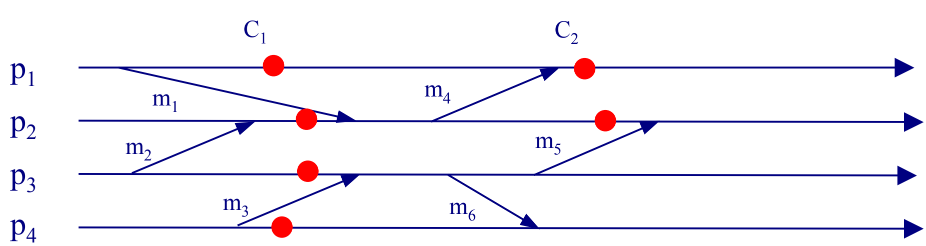
Taking a distributed checkpoint 4/9

What about failure?

- If process is unable to reply in round one, its reply is assumed to be negative and no checkpoint is taken
- Once process has replied in round one, it must follow initiator's decision
- If it fails it may need to contact initiator to determine outcome
- If the initiator fails before making checkpoint permanent, no checkpoint is taken
- If the initiator fails after making checkpoint permanent, all processes will make checkpoints permanent

Taking a distributed checkpoint 5/9

- Note that it may not be necessary for every process to checkpoint



- If a checkpoint was taken at C_1 and p_1 decides to initiate a checkpoint after receiving m_4, only p_1 and p_2 need to take new local checkpoints to have a consistent distributed checkpoint C_2

Taking a distributed checkpoint 6/9

- So, every message carries a send sequence number (SSN)
- Each process r, keeps track of the SSN of the last message it received from every other process since its last checkpoint

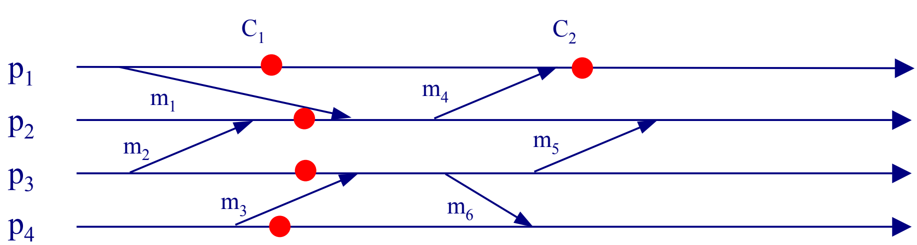
  $last\_recd_r(s)$ for all $s <> r$
- Each process s, keeps track of the SSN of the first message that it sent to every other process since its last checkpoint

  $first\_sent_s(r)$ for all $r <> s$
- When p_i asks p_j to checkpoint, it includes $last\_recd_{pi}(p_j)$ in the request
- p_j needs to checkpoint only if

  $last\_recd_{pi}(p_j) > first\_sent_{pj}(p_i)$
- Each process p_i keeps a list of the processes from which it has received messages since its last checkpoint - $cohorts_{pi}$

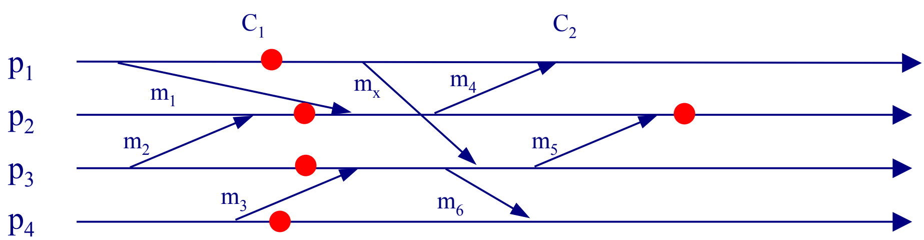
Taking a distributed checkpoint 7/9

- Checkpoint request is only sent to cohorts of initiator
- May be propagated to *their* cohorts transitively



Taking a distributed checkpoint 8/9

- Say p_2 initiates a checkpoint after receiving m_5 in the following scenario:



Taking a distributed checkpoint 9/9

- On recovery of a failed process, could rollback every process in the system to its last permanent checkpoint
  - → as an atomic action requiring two rounds of communication
- If the failed process p_i hasn't communicated to p_j it may not be necessary to rollback p_j
- Each process s, keeps track of the SSN of the last message that it sent to every other process before it took its last permanent checkpoint

$$last\_sent_s(r) \text{ for all } r <> s$$

- When p_i asks p_j to rollback, it includes $last\_sent_{pi}(p_j)$ in the request
- p_j needs to rollback only if

$$last\_recd_{pj}(p_i) > last\_sent_{pi}(p_j)$$

References

- [Johnson and Zwaenepoel 1987] David Johnson and Willy Zwaenepoel, *Sender-Based Message Logging*, Proceedings of the 17th IEEE International Symposium on Fault-Tolerant Computing Systems, pages 14-19, 1987.

- [Koo and Toueg 1987] Richard Koo and Sam Toueg, *Checkpointing and Rollback Recovery for Distributed Systems*, IEEE Transactions on Software Engineering, SE-13(1):23-31, January 1987.

See also:

- [Jalote 1994] Pankaj Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994, ISBN 0-13-301367-7, Chapter 5.

Process replication

- Another approach to providing servers that can tolerate machine crashes is to replicate server processes

Basically two approaches:

- Passive/loosely-synchronised/primary-backup replication
  - primary process handles each request and replies to clients while passive/backup process collects recovery information
  - backup takes over on failure
  - can have multiple backups
- Active/closely-synchronised replication
  - multiple processes handle each request in parallel

Passive replication

- Each server has a *primary* that accepts and responds to requests and...
- ...a *backup* that takes over when the primary crashes
- An alternative to storing checkpoints/message logs on disk
- Backup has checkpoint and message log
- Trade-off cost of logging against cost of sending requests to both primary and backup processes

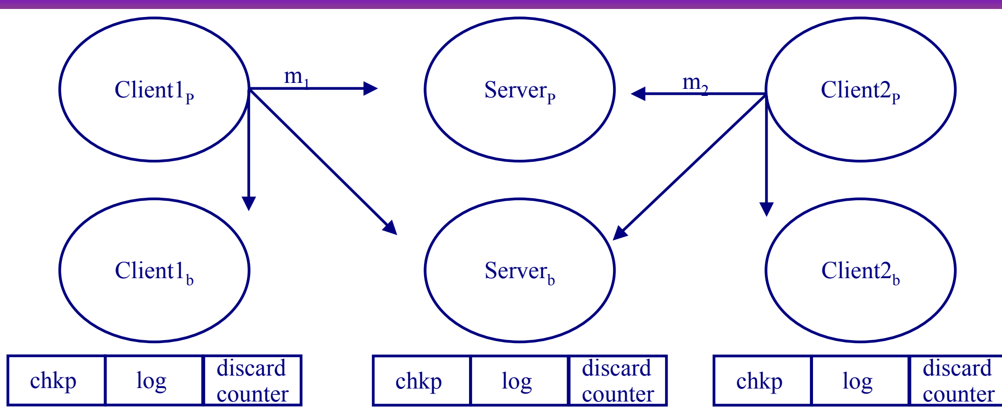
TARGON/32 1/4

- TARGON/32 is a fault-tolerant version of UNIX [Borg et al. 1989]
- Goals:
  - recover from any single crash failure
    - partial amnesia crashes as usual
  - complete transparency (programs do not have to be modified)
  - all processors available to do useful work in the absence of failure
  - tradeoff low overhead during normal execution vs. longer recovery time
- Requirements:
  - processes must be deterministic
  - crashed processes checkpoint must be available to backup
  - all messages since checkpoint must be available to backup and must be replayed in the same order

TARGON/32 2/4

- To have messages replayed in the *same order*, messages from all senders must be *received* by both the primary and backup in the *same order*
- Messages to a server are *multicast* using a *totally-ordered, atomic multicast* to the primary and the backup (and the sender's backup)
  - → *atomic* means that multicasts are received by all destinations or none
  - → *totally-ordered* means that multicasts are received in the same order at all overlapping destinations
- Messages are copied to the sender's backup so that it knows which messages were sent by primary and can suppress them during recovery

TARGON/32 3/4



Messages m_1 and m_2 are received in the same order at {Server$_p$,Server$_b$}
Log contains messages received by the primary
Discard counter counts messages sent by the primary

TARGON/32 4/4

- Multicast implemented using hardware support
- Checkpoint taken after n messages processed or specified time elapses
- Checkpointing mechanism uses an external pager
  - → primary flushes dirty pages (and other state) during checkpoint
  - → backup faults most recent checkpoint during recovery
- Replication is optional and a number of different strategies are supported
  - → "quarter-backs" - backed up until a crash occurs
  - → "half-backs" - new backup created when primary machine rebooted
  - → "full-backs" - new backup created asap after crash (not implemented)
- Details are complex (especially supporting UNIX semantics)
  - → e.g. signal handling gives rise to non-determinism - checkpoint before handling signal
  - → also must be careful about changes to the file system

Active replication

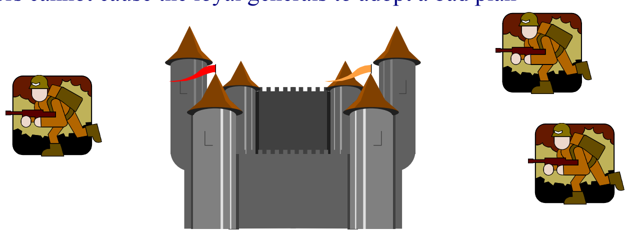
- Requests processed by multiple processes in parallel
- Each request is sent to all replicas
- Requests must be processed in the same order at each process
- Client can collect one or more replies as required

Notes:

- Additional load on system due to replicated processing
- Minimise recovery time after failure - no rollback/recovery/replay
  - → suitable for applications with guaranteed response time requirements
- Potentially mask response/arbitrary as well as crash failures by collecting and comparing replies (voting)

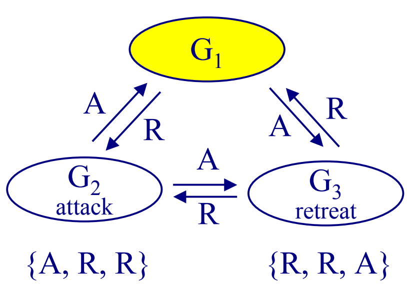
Tolerating Byzantine failures 1/14 (The Byzantine Generals Problem)

- Three generals need to agree whether or not to attack an enemy city
- They communicate by exchanging messages (reliably)
- Some of the generals may be traitors
- The traitors may lie but can’t impersonate others
- Want to ensure that:
  - → all loyal generals decide upon the same plan of action
  - → traitors cannot cause the loyal generals to adopt a bad plan



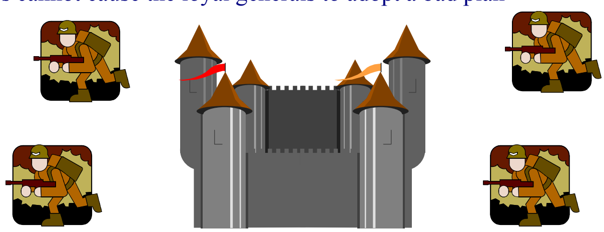
The Byzantine Generals Problem 2/14

- There is no solution!
  - → One traitor can subvert the efforts of two loyal generals to reach agreement



The Byzantine Generals Problem 3/14

- *Four* generals need to agree whether or not to attack an enemy city
- They communicate by exchanging messages (reliably)
- Some of the generals may be traitors
- The traitors may lie but can't impersonate others
- Want to ensure that:
  - → all loyal generals decide upon the same plan of action
  - → traitors cannot cause the loyal generals to adopt a bad plan



The Byzantine Generals Problem 4/14

See [Lamport et al. 1982]

- Given *g* generals of which at most *t* are traitors, it can be shown that consensus is possible iff

$$g \geq 3t + 1$$

- No solution for $g \leq 3$
- $g = 4$ generals can tolerate $t = 1$ traitor
- Put another way, to tolerate m faults, we need at least 3m + 1 processes!
  - → if faults are inconsistent and arbitrary

The Byzantine Generals Problem 5/14

- Let v_i be the value for general i and assuming that all generals use the same method for making a decision based on the information they have

- Every loyal general must obtain the same information $v_1, \ldots v_n$
- If the i^{th} general is loyal, then the value that he sends must be used by every loyal general as the value of v_i

Put another way:

- Any two loyal generals use the same value for v_i
- If the i^{th} general is loyal, then the value that he sends must be used by every loyal general as the value of v_i

The Byzantine Generals Problem 6/14

A general must send an order to his n-1 lieutenants such that:

- IC1: All loyal lieutenants obey the same order
- IC2: If the general is loyal, then every loyal lieutenant obeys the order that he sends

- Known as the *interactive consistency conditions*

- No lieutenant can trust an order received from any other participant
- Orders received by other lieutenants are needed to verify the original order
- Lieutenants must *exchange* the orders that they receive

The Byzantine Generals Problem 7/14

- The *oral message* algorithm depends on the number of traitors
  - OM(t)
  - assume that a loyal lieutenant follows the algorithm correctly

Assumptions:

- A1: Every message that is sent is delivered correctly
- A2: The receiver of a message knows who sent it
- A3: The absence of a message can be detected

- Algorithm proceeds in *rounds* requiring a *synchronous* system where message delays and differences in relative speed of processors are bounded

The Byzantine Generals Problem 8/14

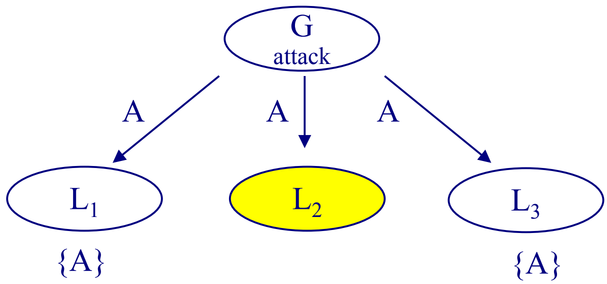
Algorithm OM(0)

1. The general sends his value to every lieutenant
2. Each lieutenant uses the value he receives from the general or uses the default value if he receives no value

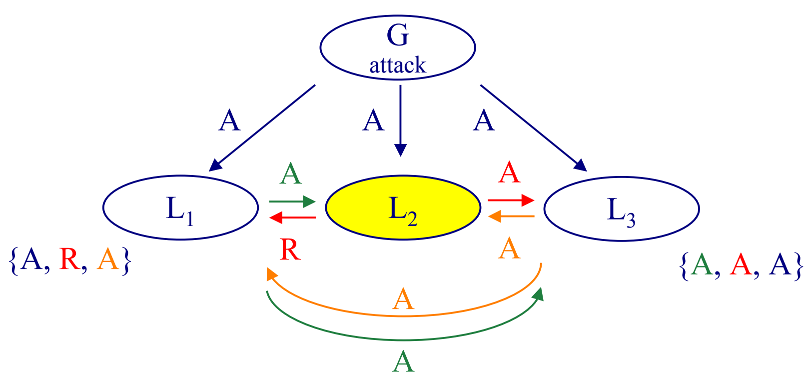
Algorithm OM(t), $t > 0$

1. The general sends his value to every lieutenant
2. Let v_i be the value that lieutenant i receives from the general or else be the default value. Lieutenant i acts as the general in algorithm OM(m-1) to send the value to each of the n-2 other lieutenants
3. For each i and each $j \neq i$, let v_j be the value that lieutenant i received from lieutenant j in step 2 or else the default value. Lieutenant i uses the value *majority*$(v_1,\ldots,v_n)$

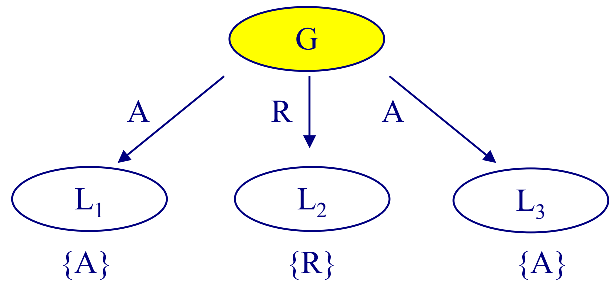
The Byzantine Generals Problem 9/14



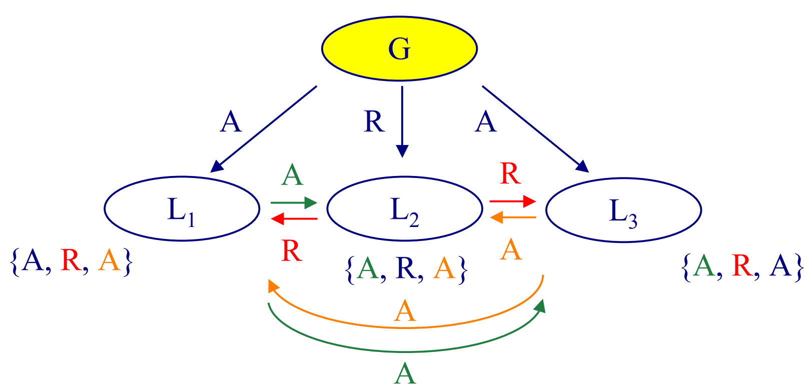
The Byzantine Generals Problem 10/14



The Byzantine Generals Problem 11/14



The Byzantine Generals Problem 12/14



The Byzantine Generals Problem 13/14

- OM(m) requires m+1 rounds of message exchange
  - No algorithm can reach agreement in less than m+1 rounds
- $O(n^m)$ messages
- Use OM(m) to distribute the opinion of each general to other generals in consensus problem

The Byzantine Generals Problem 14/14

- The situation is improved if messages can be *signed*
- General might still be a traitor
- A traitor can't lie about the contents of a message that it received but only fail to pass it on
- Tolerate an *arbitrary number of traitors* but still need m+1 rounds of message exchange

References

- [Borg et al. 1989] Anita Borg et al., *Fault Tolerance Under UNIX*, ACM Transactions on Computer Systems, 7(1):1-24, February 1989.

- [Lamport et al. 1982] Leslie Lamport, Robert Shostak, and Marshall Pease, *The Byzantine Generals Problem*, ACM Transactions on Programming Languages and Systems, 4(3):382-401, July 1982.

See also:

- [Jalote 1994] Pankaj Jalote, *Fault Tolerance in Distributed Systems*, Prentice Hall, 1994, ISBN 0-13-301367-7, Chapter 3.