Lecture 18: Distributed Agreement

CSC 469H1F / CSC 2208H1F
Fall 2007
Angela Demke Brown

A rose by any other name…

- Distributed Consensus has many names (depending on the assumptions and application)
  - Reliable multicast
  - Byzantine Generals Problem
  - Interactive agreement
  - Atomic broadcast

"This has resulted in a voluminous literature which, unfortunately, is not distinguished for its coherence. The differences in notation and the haphazard nature of the assumptions obfuscates the close relationship among these problems"

– Hadzilacos & Toueg, Distributed Systems.

Outline

- Distributed Algorithms
- Distributed Agreement
- Castro's BFT Library

Distributed Algorithms

- System model from last lecture.
- Distributed system is composed of *n* processes
- A process executes a sequence of events
  - Local computation
  - Sending a message *m*
  - Receiving a message *m*
- A distributed algorithm is an algorithm that runs on more than one process.

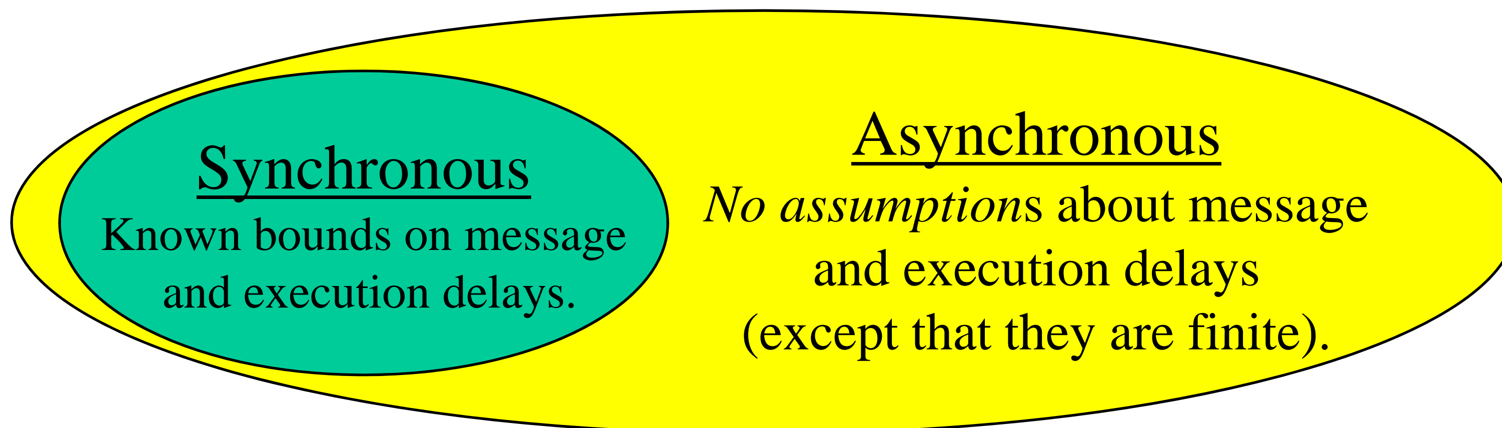
Properties of Distributed Algorithms

- Safety
  - Means that some particular "bad" thing never happens.
- Liveness
  - Indicates that some particular "good" thing will (eventually) happen.

- Timing/failure assumptions affect how we reason about these properties and what we can prove

Timing Model

- Specifies assumptions regarding *delays* between
  - execution steps of a *correct* process
  - send and receipt of a message sent between *correct* processes
- Many gradations. Two of interest are:



  - *Partial synchrony* is more realistic in distrib. system

Synchronous timing assumption

- Processes share a clock
- Timestamps mean something between processes
- Communication can be guaranteed to occur in some number of clock cycles

Asynchronous timing assumption

- Processes operate asynchronously from one another.
- No claims can be made about whether another process is running slowly or has failed.
- There is no time bound on how long it takes for a message to be delivered.

Partial synchrony assumption

- "Timing-based distributed algorithms"
- Processes have some information about time
  - Clocks that are synchronized within some bound
  - Approximate bounds on message-deliver time
  - Use of timeouts

Failure Model

- A process that behaves according to its I/O specification throughout its execution is called correct
- A process that deviates from its specification is faulty
- Many gradations of faulty. Two of interest are:

Fail-Stop failures
A faulty process halts execution prematurely.

Byzantine failures
No assumption about behavior of a faulty process.

Errors as failure assumptions

- Specific types of errors are listed as failure assumptions
  - Communication link may lose messages
  - Link may duplicate messages
  - Link may reorder messages
  - Process may die and be restarted

Fail-Stop failure

- A failure results in the process, *p*, stopping
  - Also referred to as *crash failure*
  - *p* works correctly until the point of failure
- *p* does not send any more messages
- *p* does not perform actions when messages are sent to it
- Other processes can detect that *p* has failed

UofT

Fault/failure detectors

- A perfect failure detector
  - No false positives (only reports actual failures).
  - Eventually reports failures to all processes.
- Heartbeat protocols
  - Assumes partially synchronous environment
  - Processes send "I'm Alive" messages to all other processes regularly
  - If process *i* does not hear from process *j* in some time $T = T_{delivery} + T_{heartbeat}$ then it determines that *j* has failed
  - Depends on $T_{delivery}$ being known and accurate

Other Failure Models

- We can classify some of the likely failure modes that lie between crash and Byzantine
  - Omission failure
    - Process fails to send messages, to receive incoming messages, or to handle incoming messages
  - Timing failure
    - process's response lies outside specified time interval
  - Response failure
    - Value of response is incorrect

Byzantine failure

- Process *p* fails in an arbitrary manner.
- *p* is modeled as a malevolent entity
  - Can send the messages and perform the actions that will have the worst impact on other processes
  - Can collaborate with other "failed" processes
- Common constraints on Byzantine assumption
  - Incomplete knowledge of global state
  - Limited ability to coordinate with other Byzantine processes
  - Restricted to polynomial computation (i.e., assume $P \neq NP$...)

Setup of Distributed Consensus

- N processes have to agree on a single value.
  - Example applications of consensus:
    - Performing a commit in a replicated/distributed database.
    - Collecting multiple sensor readings and deciding on an action
- Each process begins with a value
- Each process can irrevocably *decide* on a value
- Up to $f < n$ processes may be faulty
  - How do you reach consensus if no failures?

Properties of Distributed Consensus

- *Agreement*
  - If *any correct* process believes that *V* is the consensus value, then *all correct* processes believe *V* is the consensus value.
- *Validity*
  - If *V* is the consensus value, then some process proposed *V*.
- *Termination*
  - Each process decides some value *V*.
- *Agreement* and *Validity* are **Safety** Properties
- *Termination* is a **Liveness** property.

Synchronous Fail-stop Consensus

- FloodSet algorithm run at each process *i*
  - Remember, we want to tolerate up to *f* failures

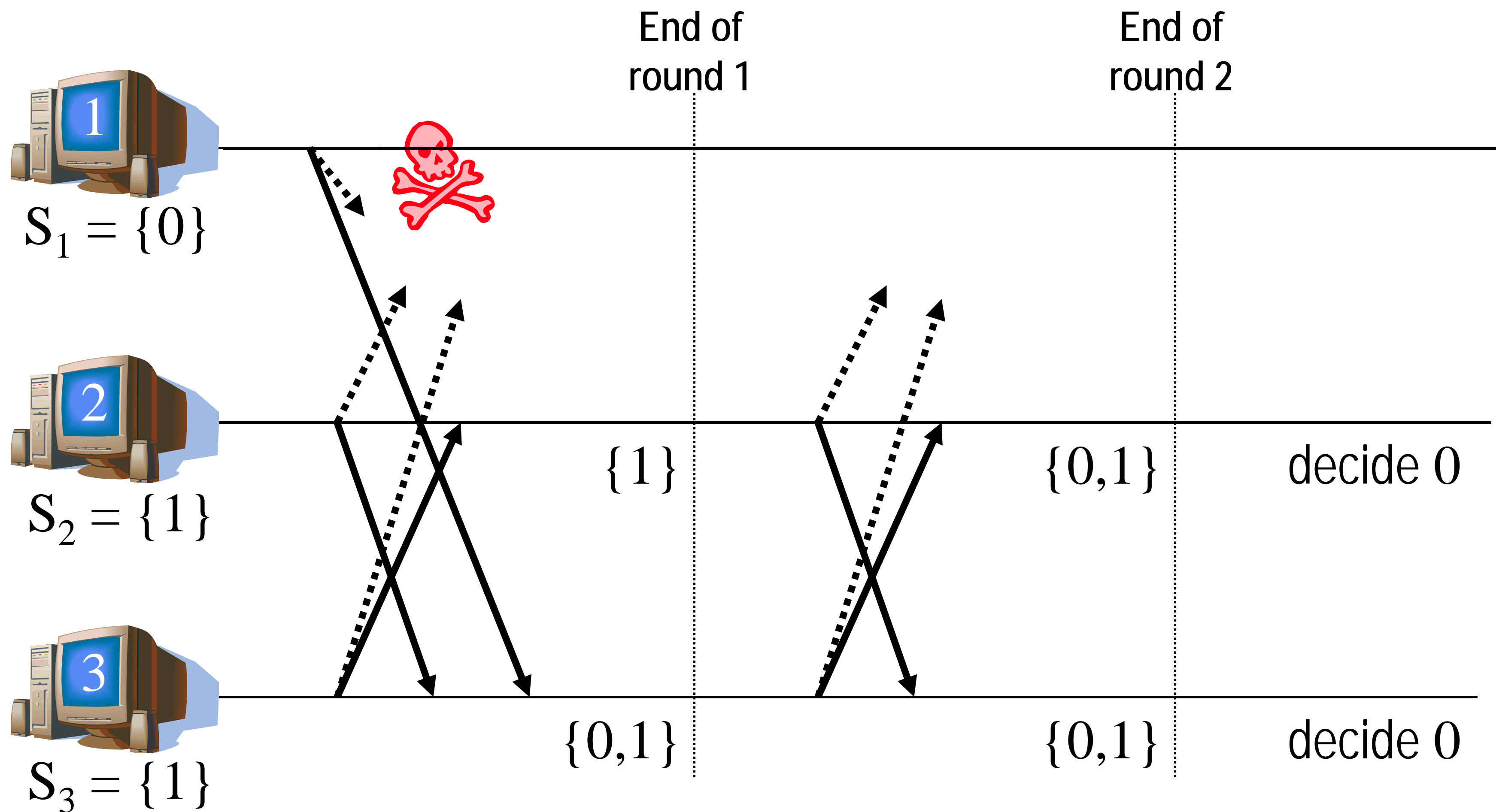
```
Si ← {initial value}
for k = 1 to f+1
    send Si to all processes
    receive Sj from all j != i
        Si ← Si ∪ Sj  (for all j)
end for
Decide(Si)
```

- S is a set of values
- Decide(x) can be various functions
  - E.g. min(x), max(x), majority(x), or some default
- Assumes nodes are connected and links do not fail

Analysis of FloodSet

- Requires *f+1* rounds because process can fail at any time, in particular, during send
- *Agreement*: Since at most *f* failures, then after *f+1* rounds all correct processes will evaluate `Decide`(S_i) the same.
- *Validity*: `Decide` results in a proposed value (or default value)
- *Termination*: After *f+1* rounds the algorithm completes

Example with $f = 1$, Decide() = min()



Synchronous/Byzantine Consensus

- Faulty processes can behave arbitrarily
  - May actively try to trick other processes
- Algorithm described by Lamport, Shostak, & Pease in terms of Byzantine generals agreeing whether to attack or retreat. Simple requirements:
  - All loyal generals decide on the same plan of action
    - Implies that all loyal generals obtain the same information
  - A small number of traitors cannot cause the loyal generals to adopt a bad plan
  - Decide() in this case is a majority vote, default action is "Retreat"

Byzantine Generals

- Use v(i) to denote value sent by *i^th^* general
- traitor could send different values to different generals, so can't use v(i) obtained from i directly. New conditions:
  - Any two loyal generals use the same value v(i), regardless of whether i is loyal or not
  - If the ith general is loyal, then the value that she sends must be used by every loyal general as the value of v(i).
- Re-phrase original problem as *reliable broadcast:*
  - General must send an order ("Use v as my value") to lieutenants
  - Each process takes a turn as General, sending its value to the others as lieutenants
  - After all values are reliably exchanged, Decide()

Synchronous Byzantine Model

Theorem: There is no algorithm to solve consensus if only oral messages are used, unless *more than two thirds* of the generals are loyal.

- In other words, impossible if $n \leq 3f$ for *n* processes, *f* of which are faulty
- *Oral messages* are under control of the sender
  - sender can alter a message that it received before forwarding it
- Let's look at examples for special case of n=3, f=1

Case 1

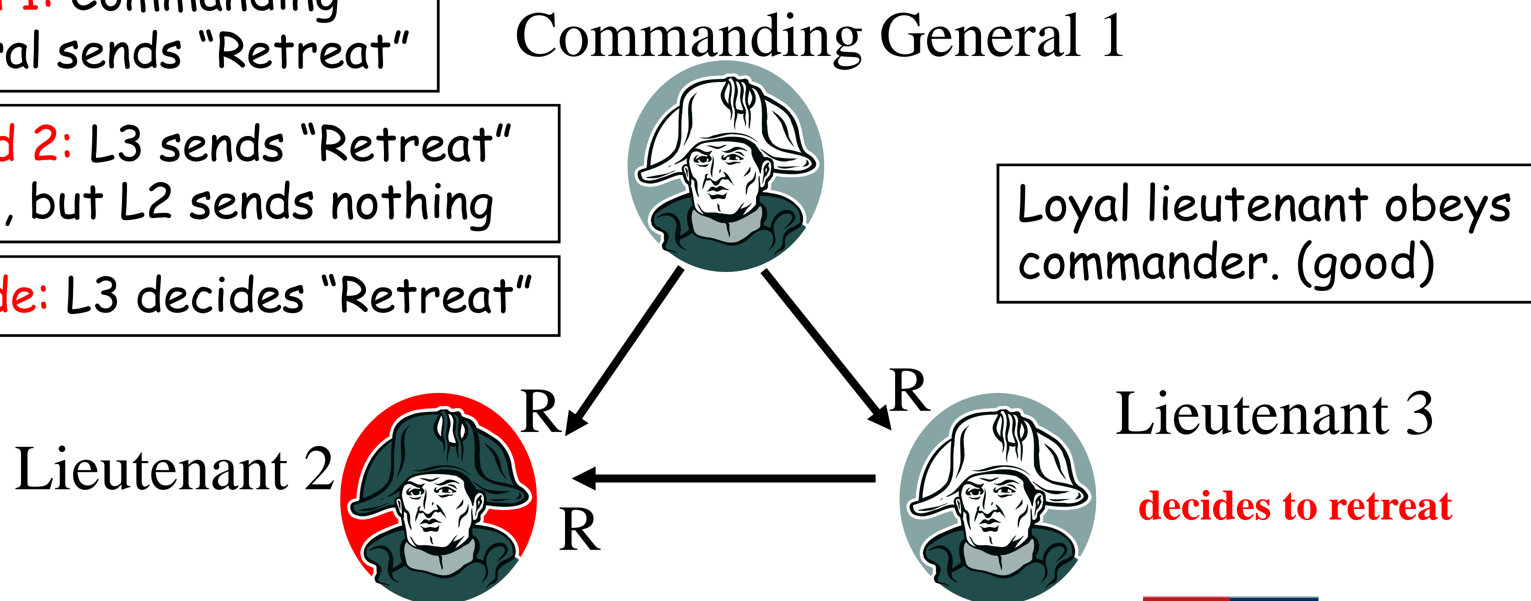
- Traitor lieutenant tries to foil consensus by refusing to participate

"white hats" == loyal or "good guys"
"black hats" == traitor or "bad guys"

Round 1: Commanding General sends "Retreat"

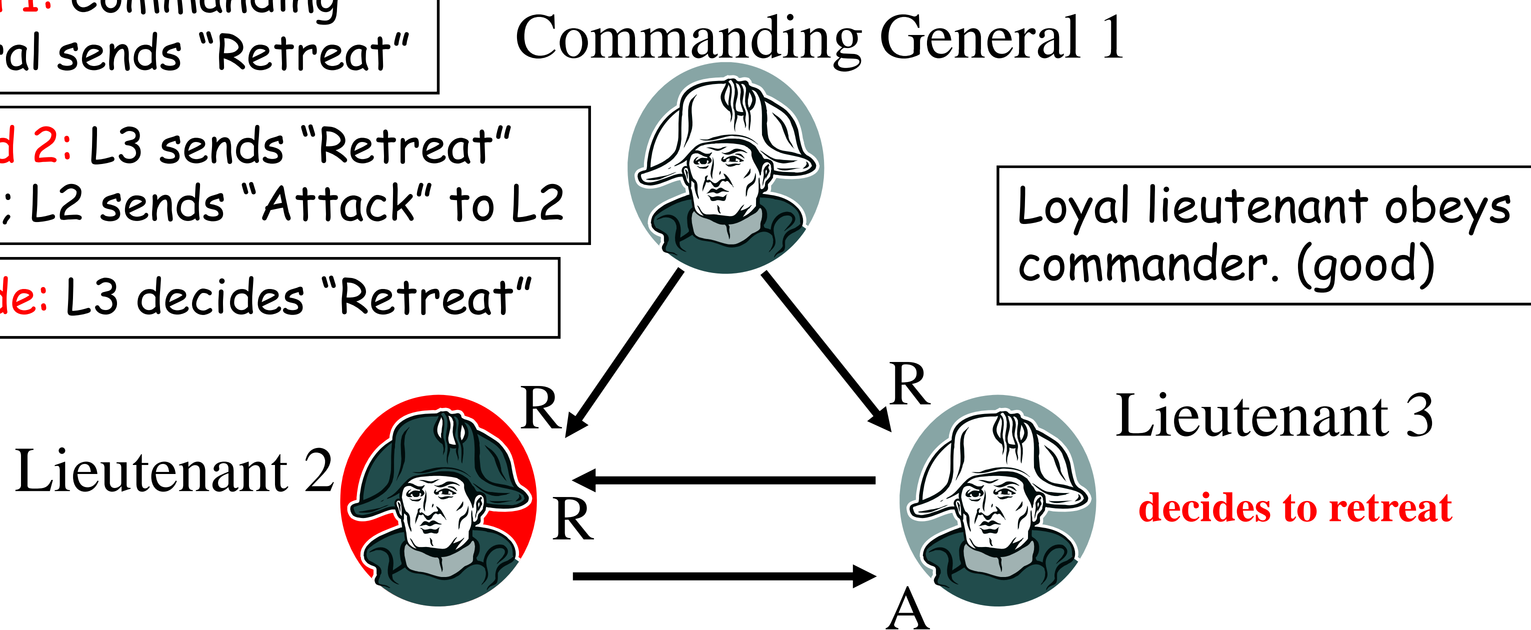
Round 2: L3 sends "Retreat" to L2, but L2 sends nothing

Decide: L3 decides "Retreat"



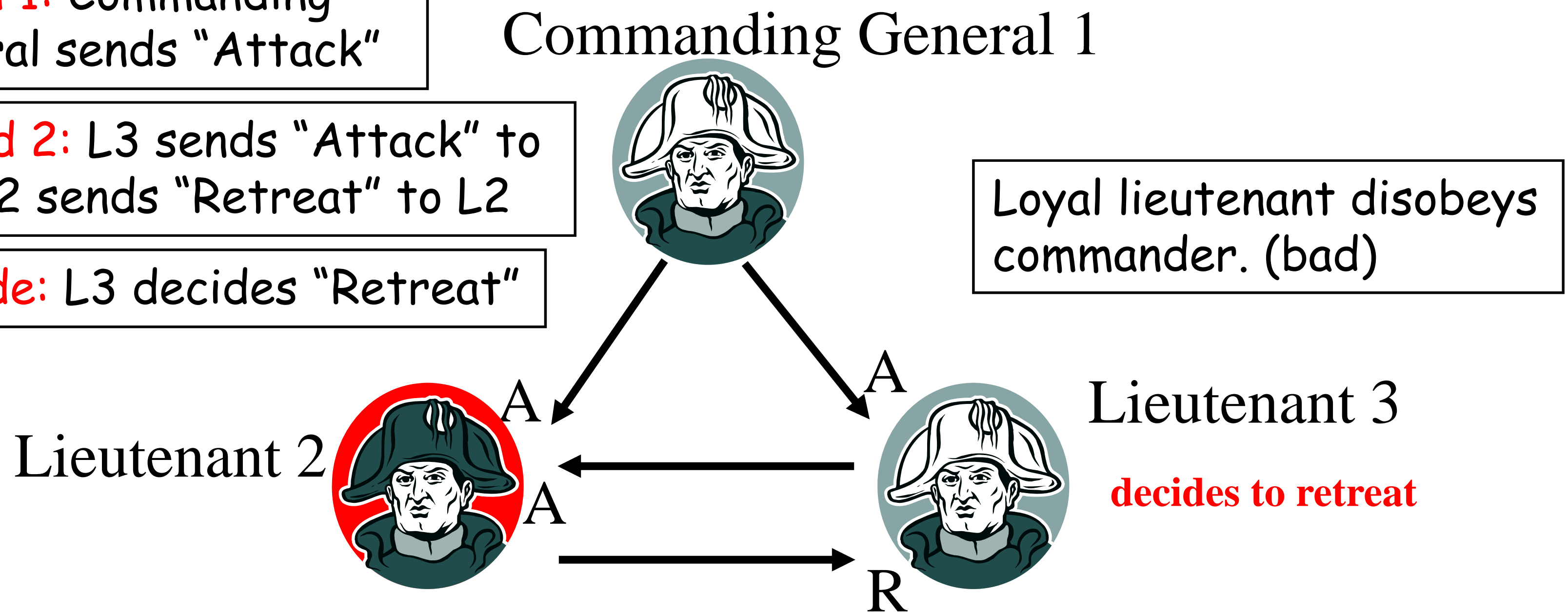
Case 2a

- Traitor lieutenant tries to foil consensus by lying about order sent by general



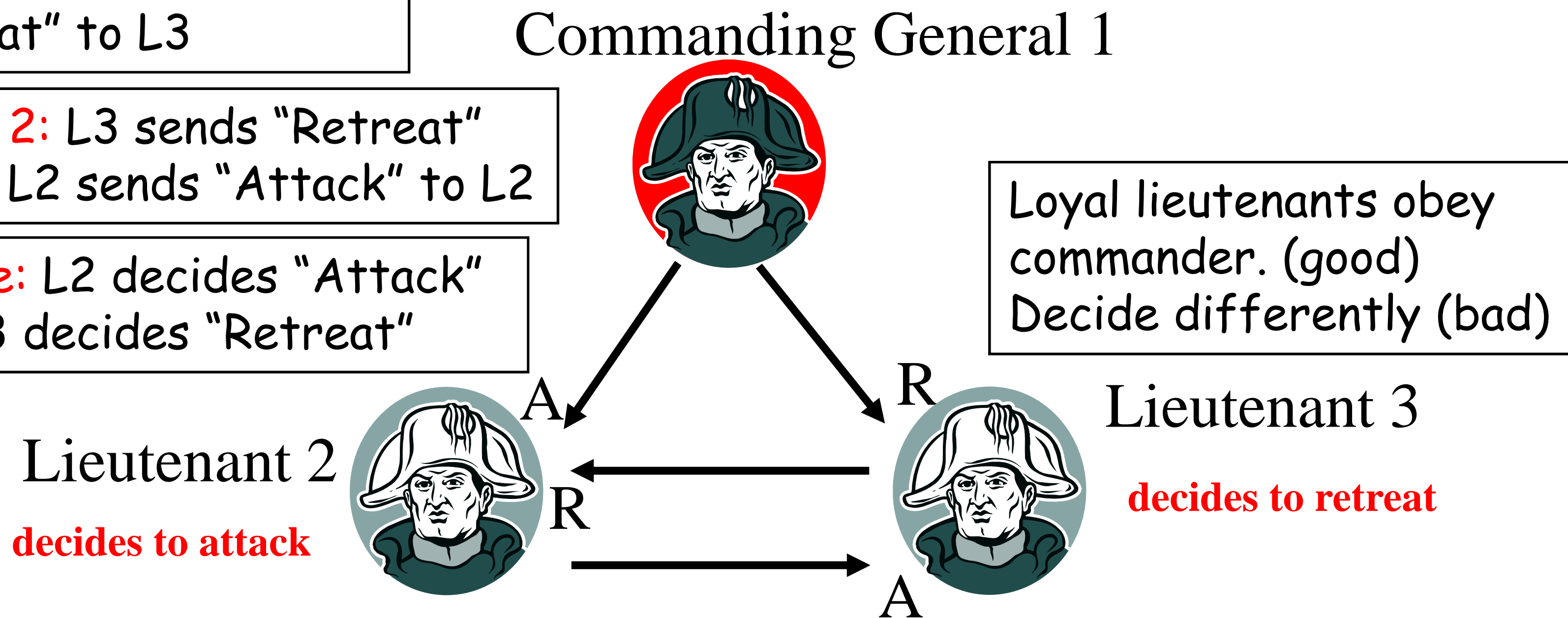
Case 2b

- Traitor lieutenant tries to foil consensus by lying about order sent by general



Case 3

- Traitor General tries to foil consensus by sending different orders to loyal lieutenants



Byzantine Consensus: n > 3f

- Oral Messages algorithm, OM(f)
- Consists of $f+1$ "phases"
- Algorithm OM(0) is the "base case" (no faults)
  1) Commander sends value to every lieutenant
  2) Each lieutenant uses value received from commander, or default "retreat" if no value was received
- Recursive algorithm handles up to f faults

OM(f): Recursive Algorithm

1) Commander sends value to every lieutenant
2) For each lieutenant *i*, let v_i be the value *i* received from commander , or "retreat" if no value was received. Lieutenant *i* acts as commander in Alg. OM(f-1) to send v_i to each of the *n-2* other lieutenants
3) For each *i*, and each *j* not equal to *i*, let v_j be the value Lieutenant *i* received from Lieutenant *j* in step (2) (using Alg. OM(f-1)), or else "retreat" if no such value was received. Lieutenant *i* uses the value *majority*$(v_1, \ldots , v_{n-1})$.

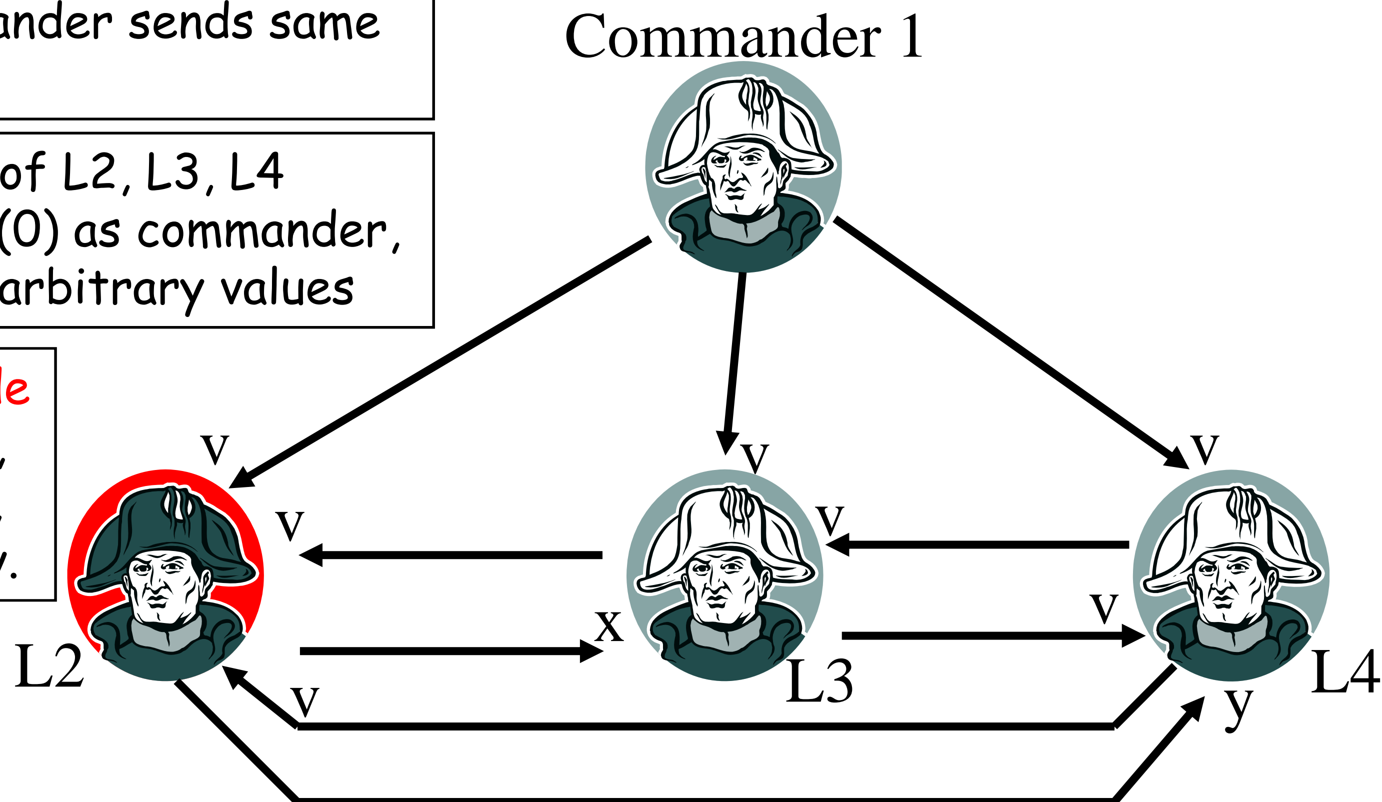
Example: f = 1, n = 4

- Loyal General, 1 traitor lieutenant

Step 1: Commander sends same value, v, to all

Step 2: Each of L2, L3, L4 executes OM(0) as commander, but L2 sends arbitrary values

Step 3: Decide
L3 has {v,v,x},
L4 has {v,v,y},
Both choose v.



Example: f = 1, n = 4

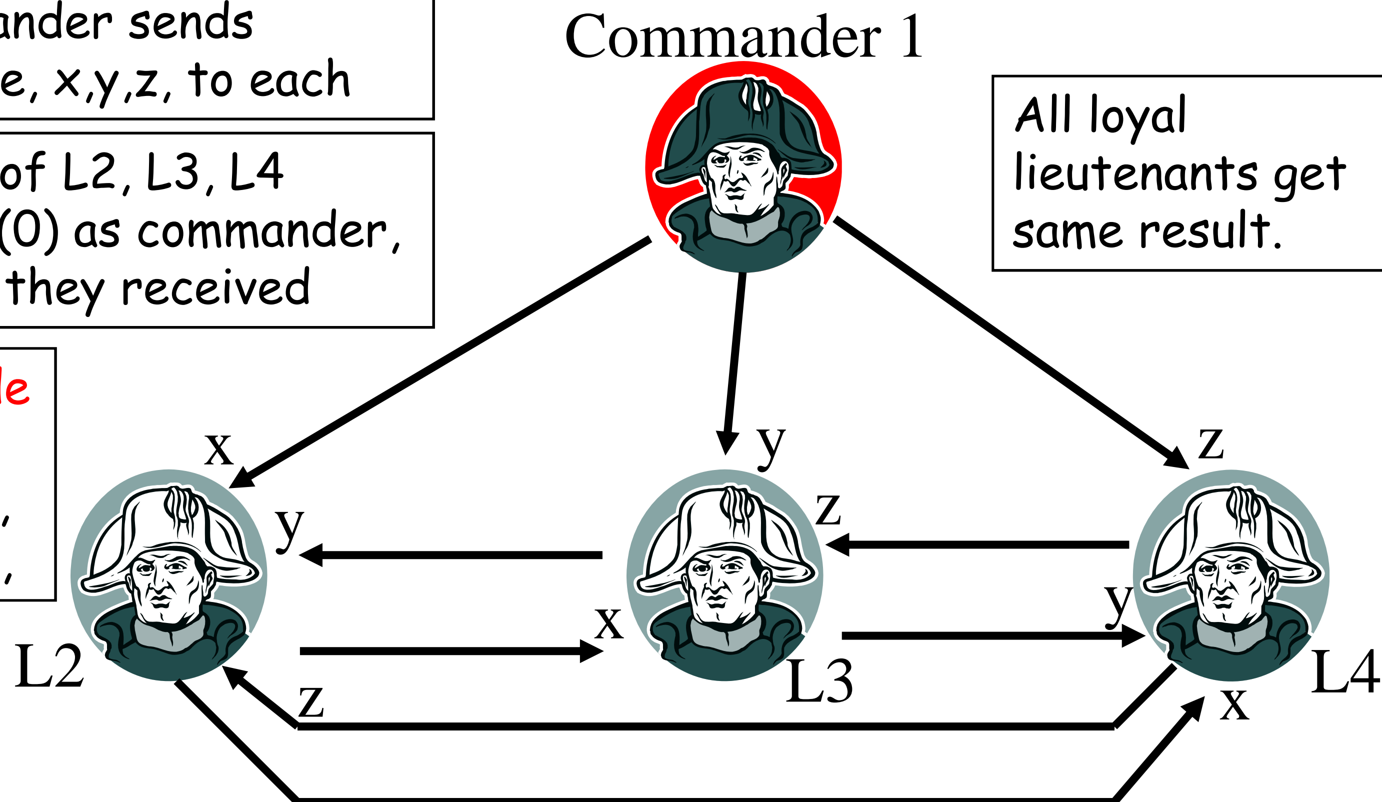
- Traitor General, all lieutenants loyal

Step 1: Commander sends different value, x,y,z, to each

Step 2: Each of L2, L3, L4 executes OM(0) as commander, sending value they received

Step 3: Decide
L2 has {x,y,z}
L3 has {x,y,z},
L4 has {x,y,x},

All loyal lieutenants get same result.



Problem

- Lots of messages required to handle even 1 faulty process
- Need minimum 4 processes to handle 1 fault, 7 to handle 2 faults, etc.
  - But as system gets larger, probability of a fault also increases
- If we use *signed messages*, instead of oral messages, can handle f faults with 2f+1 processes
  - Simple majority requirement
  - Still lots of messages sent though, plus cost of signing

Asynch. Distributed Consensus

- Fail-Stop/Byzantine → IMPOSSIBLE!
- FLP impossibility result
  - Fischer, Lynch and Patterson impossibility result
  - Asynchronous assumption makes it impossible to differentiate between failed and slow processes.
  - Therefore *termination* (**liveness**) cannot be guaranteed.
  - If an algorithm terminates it may violate *agreement* (**safety**).
    - A slow process may decide differently than other processes thus violating the agreement property

Castro: Practical Byz. Fault Tolerance

- Uses various optimizations to combine messages, reduce total communication
- Relies on partially synchronous assumption to guarantee **liveness**.
- Therefore attacks on system can only slow it down – **safety** is guaranteed.
- Assumes that an attack on **liveness** can be dealt with in a reasonable amount of time.
- Suitable for wide area deployment (e.g., internet)
- Being used in Microsoft Research's *Farsite* distributed file system

Partially Synchronous Consensus Algs

- Relies on a Fault-Detector
- Synchronous/Fail-stop distributed consensus algorithms (e.g. FloodSet) can be transformed to run in the partially synchronous environment
- Byzantine is still a problem though…
  - DoS attacks on correct processes result in the identification of correct processes as failed, reducing the number of processes that must be compromised to breach the *safety* property (i.e. attackers can manipulate f which is **not** cool)