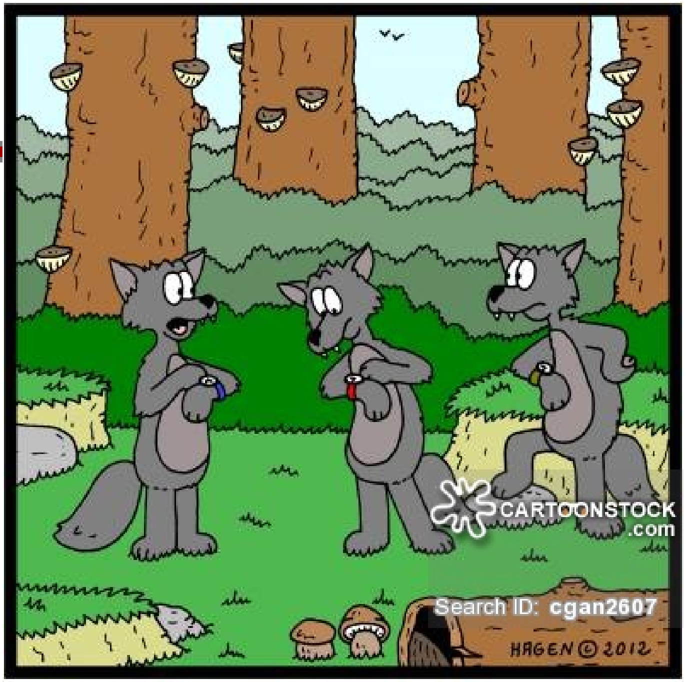
Lecture 9:

Time, Clocks and Event Ordering

Time in Distributed Systems



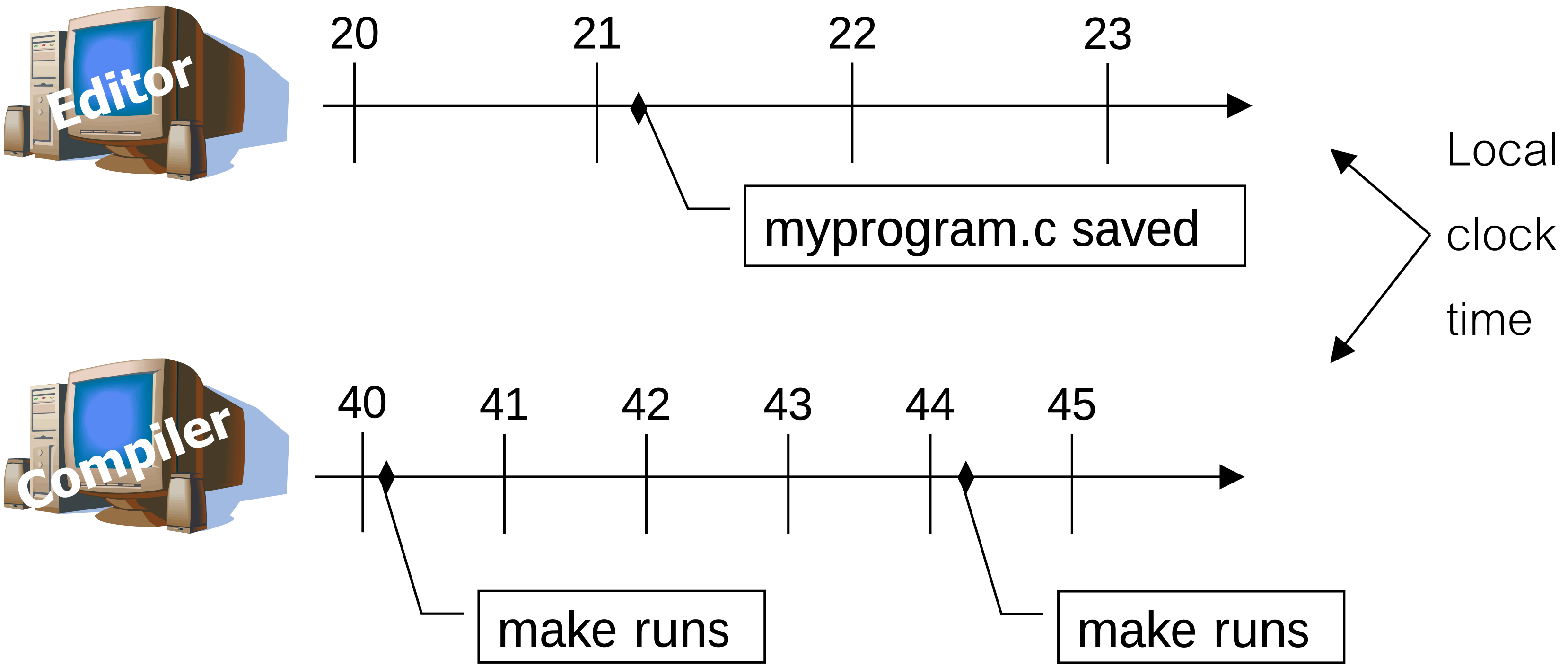
OK, it will be a complex hunt tonight:
Let's synchronize our watches...

- Each machine maintains its own time
  - No global shared clock
- Consider *make* program

```
myprogram: myprogram.c
    gcc –o myprogram myprogram.c
```

  - When does a target get re-built?
  - Unambiguous on single computer
  - What if timestamps are assigned on different machines?

Distributed Edit/Make



- Looks like myprogram should not get recompiled

Physical clocks

- Typical computer timer is a precisely-machined quartz crystal
  - Oscillates at a well-defined frequency when kept under tension
  - Freq depends on tension, kind of crystal, cut
- 2 associated registers, “counter” and “holding”
  - Counting register decremented by one on each oscillation
  - When zero, interrupt is generated (called a tick)
  - On each clock tick, adds 1 to the time stored in memory, and counter is reloaded from “holding”
- Can’t guarantee that two crystals oscillate at exactly the same frequency
  - => *clock skew*!

Clock synchronization

- Simple algorithm:
  - Time server maintains global notion of time
  - Each machine periodically contacts time server asking for current global time
  - Machine updates local time with global time
- Problems?

Cristian's algorithm (1989)

1. Client P requests the time from server S
2. S responds with the time T from its own clock.
3. P sets its time to be T + RTT/2

- Assumes propagation delay is the same for send and receive
- Accuracy can be improved by making multiple requests and using the minimum RTT.

Berkeley Algorithm (1989)

1. A *master* is chosen by election
2. The *master* polls the *slaves* who reply with their time
3. The *master* observes RTT of the messages and estimates the time of each *slave*
4. The *master* averages the slave and own clock times
   - Ignores values that are far outside of the others
5. The *master* sends out the amount (positive or negative) that each *slave* must adjust its clock

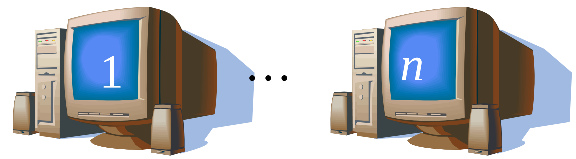
Better Clock Synchronization

- GPS receiver (+/- 10ns accuracy)
  - Not always available
- Precision Time Protocol, PTP (<1 us accuracy)
  - Takes advantage of time sources in network hardware

- But exact time is often less important than knowing how to order distributed events.
  - Which happened first?

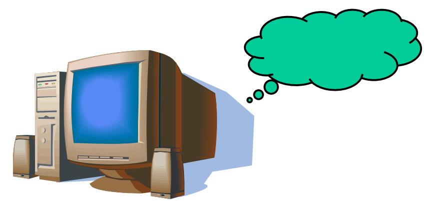
Basic “Message Passing” Model

- A collection of *n processes*

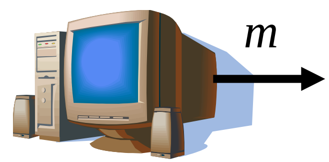


- A process executes a *sequence* of *events*

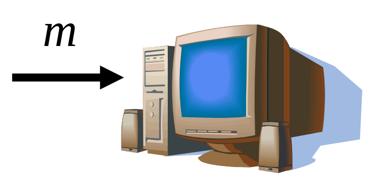
- Local computation



- Sending a message



- Receiving a message



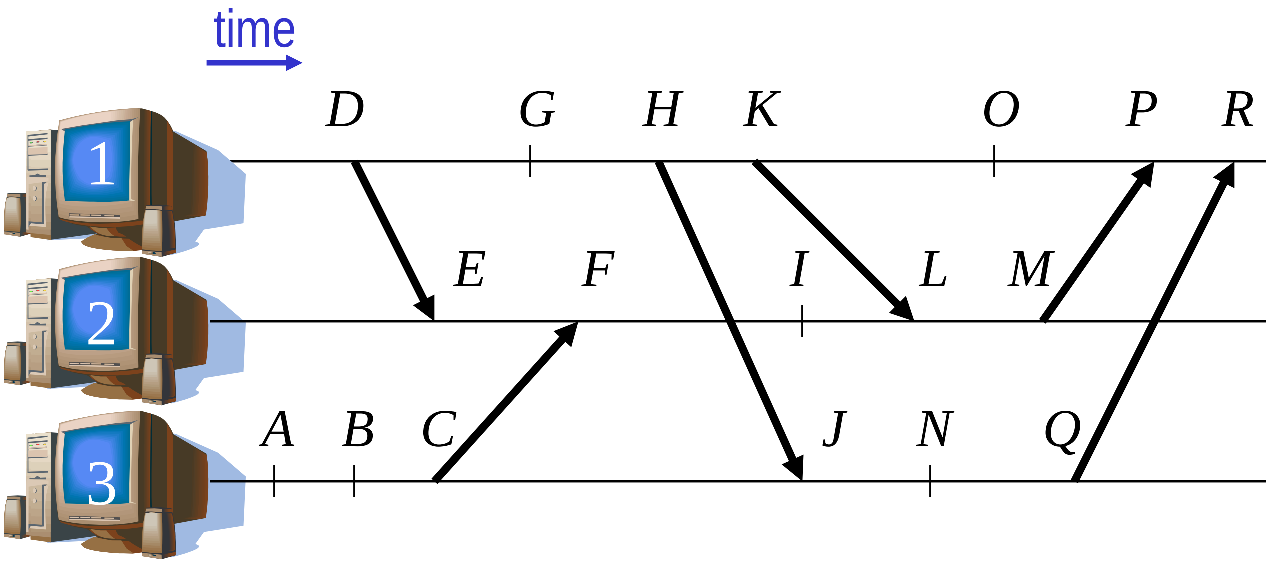
Logical Time in Distributed Systems

- Time gives us a reference with which to order events
  - Need not be consistent with external "real" time

- How do we define when one event occurs "before" another?
- Intuition: event *A* occurs before event *B* if *A* could have *influenced B*
  - It's a "causal" definition

The "Happens Before" Relation

- Given two events A and B, $A \Rightarrow B$ (A *happens before* B) if
  - 1. A and B are executed at the same process, and A occurs before B
  - 2. A = send(m) *and* B = receive(m) for some message m
  - 3. There is an event C such that $A \Rightarrow C$ and $C \Rightarrow B$
- No clear relationship => concurrent events



$D \Rightarrow I$? ✔

$G \Rightarrow I$? ✘

$A \Rightarrow O$? ✘

$A \Rightarrow R$? ✔

Observing "Happens Before" Relation

- Associate with each event a *logical timestamp* T such that:

$$\text{If } A \Rightarrow B \text{ then } T(A) < T(B).$$

- Logical clocks
  - Are local to each process/machine
  - Do not measure real time, only measure events
  - "Capture" the happened-before relation numerically
  - Provide a partial ordering (use logical clock values as timestamps)
- Algorithm to achieve it – **Lamport Clocks** [Leslie Lamport]

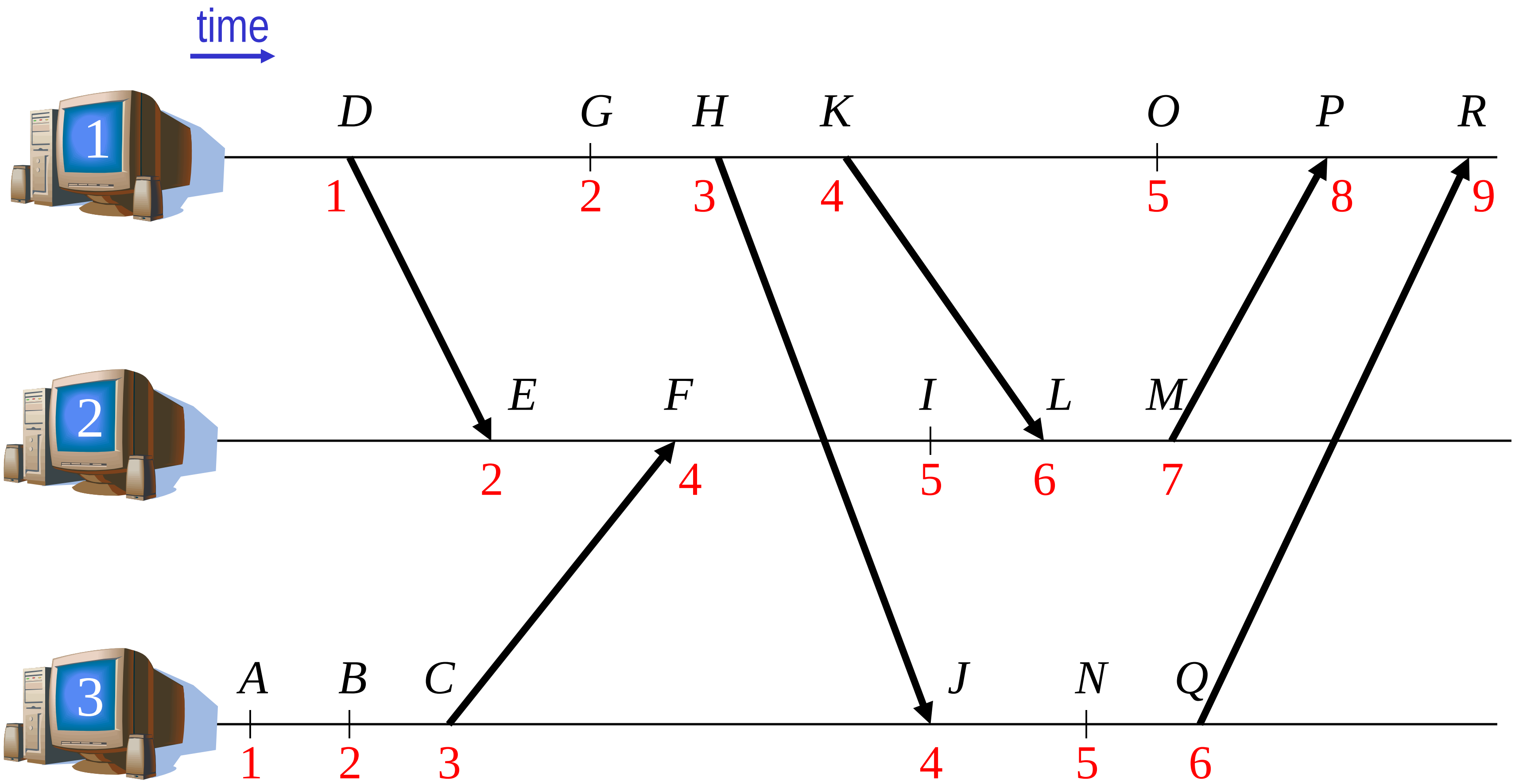
Observing "Happens Before" Relation

- Recall: each event has a *logical timestamp* T associated such that:

  If $A \Rightarrow B$ then $T(A) < T(B)$.

- Algorithm to achieve it – (**Lamport Clocks**):
  - 1. The i-th process keeps a non-negative integer counter T_i, initially 0
  - 2. When i-th process performs computation event, $T_i \leftarrow T_i + 1$
  - 3. When i-th process sends msg m, it computes $T_i \leftarrow T_i + 1$ and appends $T(m) \leftarrow T_i$ to m
  - 4. When i-th process receives msg m, $T_i \leftarrow \max\{T_i, T(m)\} + 1$
  - For event A at i-th process, define $T(A) = T_i$ computed during A
  - Can use $LC(A)$ notation to refer to Lamport Clock for event A

Example of Lamport's Algorithm



Lamport Clocks problem

- Lamport clock is used to create a partial causal ordering of events between processes
- Given a logical Lamport clock:
  - If $A \Rightarrow B$ then $LC(A) < LC(B)$
- The relation only goes one way
  - If an event A comes before another event B, then A's logical clock < B's
- What about?
  - If $LC(A) < LC(B)$ then $A \Rightarrow B$
- Problem: Lamport clocks do capture causal dependencies, but may imply more dependencies than truly exist.

More Accurate Logical Clocks

- Suppose we want a logical timestamp T such that:

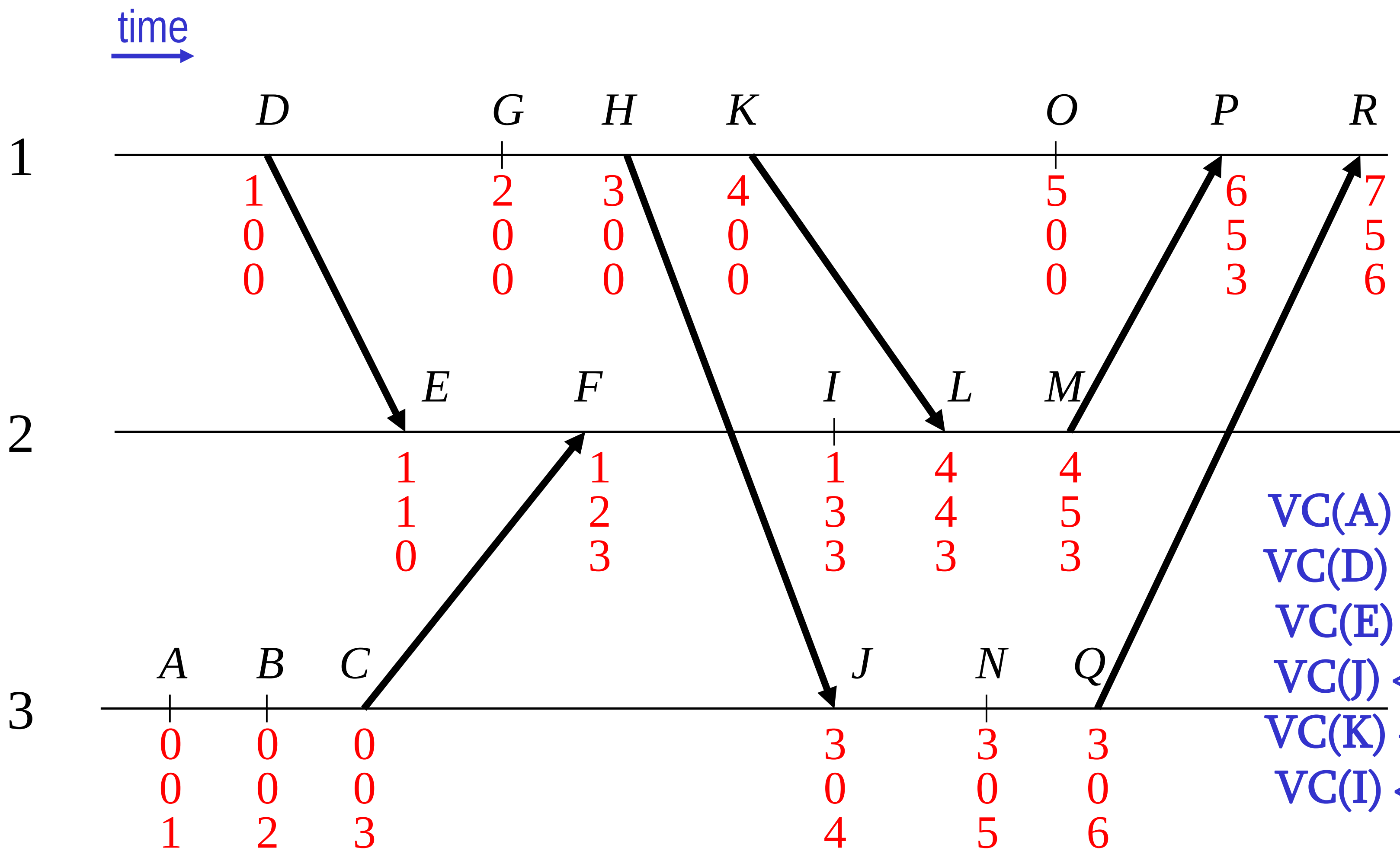
$$A \Rightarrow B \text{ \underline{if and only if} } T(A) < T(B).$$

- Algorithm to achieve it – **Vector Clocks** [Mattern; Fidge]:
  - i-th process keeps a vector T_i with n elements
    - Each element $T_i[j]$ is a non-negative integer counter, initially 0
  - When i-th process performs any event, $T_i[i] \leftarrow T_i[i] + 1$
  - When i-th process sends m, it also appends vector $T(m) \leftarrow T_i$ to m
  - When i-th process receives m, it also computes

$$T_i[j] \leftarrow \max\{T_i[j],\, T(m)[j]\} \text{ for each } j \neq i$$

  - For event A at i-th process, define $T(A) = T_i$ computed during A
  - $T(A) < T(B) \equiv [\forall j\!: T(A)[j] \leq T(B)[j] \;\wedge \exists i\!: T(A)[i] < T(B)[i]]$
  - Sometimes use $VC(A)$ to refer to vector clocks.

Example of Vector Clocks



VC(A) < VC(F) ? ✓

VC(D) < VC(N) ? ✓

VC(E) < VC(J) ? ✗

VC(J) < VC(R) ? ✓

VC(K) < VC(N) ? ✗

VC(I) < VC(P) ? ✓

Comparison

- Lamport clocks:
  - If $A \Rightarrow B$ then $LC(A) < LC(B)$
- Vector clocks:
  - $A \Rightarrow B$ if and only if $VC(A) < VC(B)$

- **Lamport clocks:** we have a guarantee that two causally-related events will have timestamps that reflect their order
- However, just by looking at LC timestamps, we cannot conclude that there is a causal happens-before relationship!
- **Vector clocks:** both implications are true (including that if A's vector clock is < B's vector clock, they are causally related).

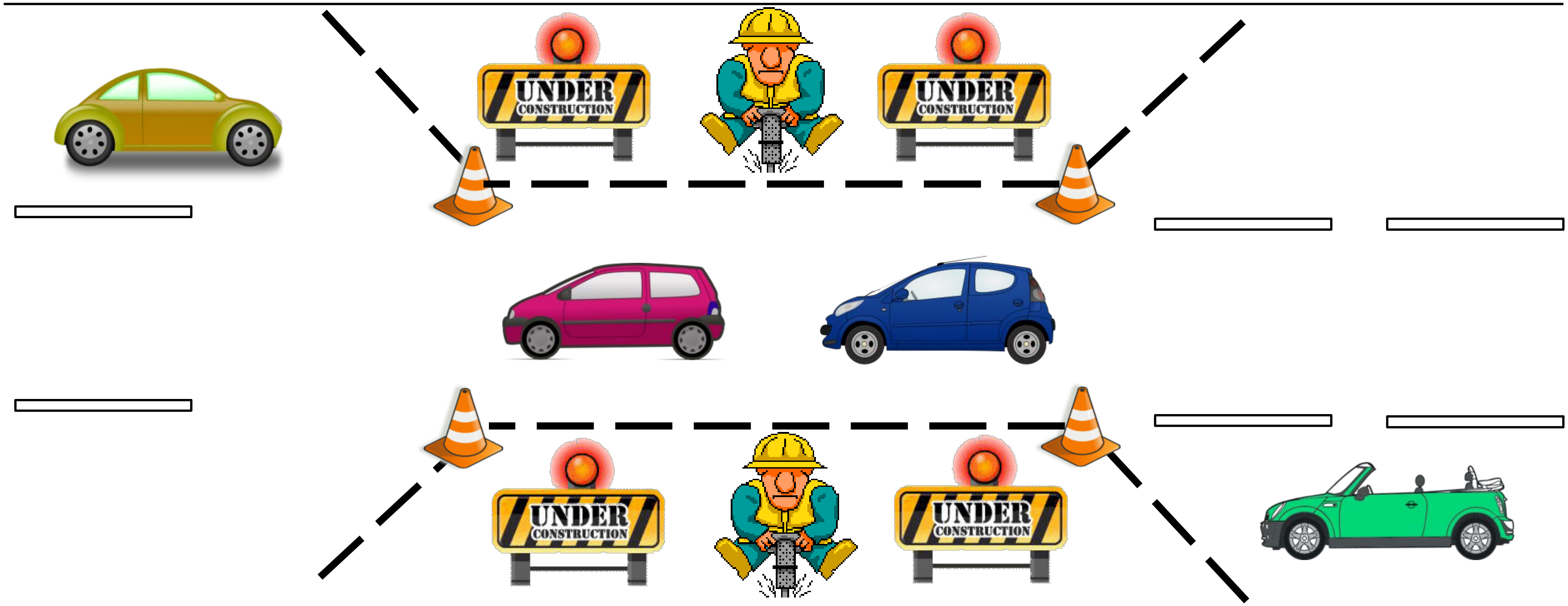
Distributed Algorithms

- 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

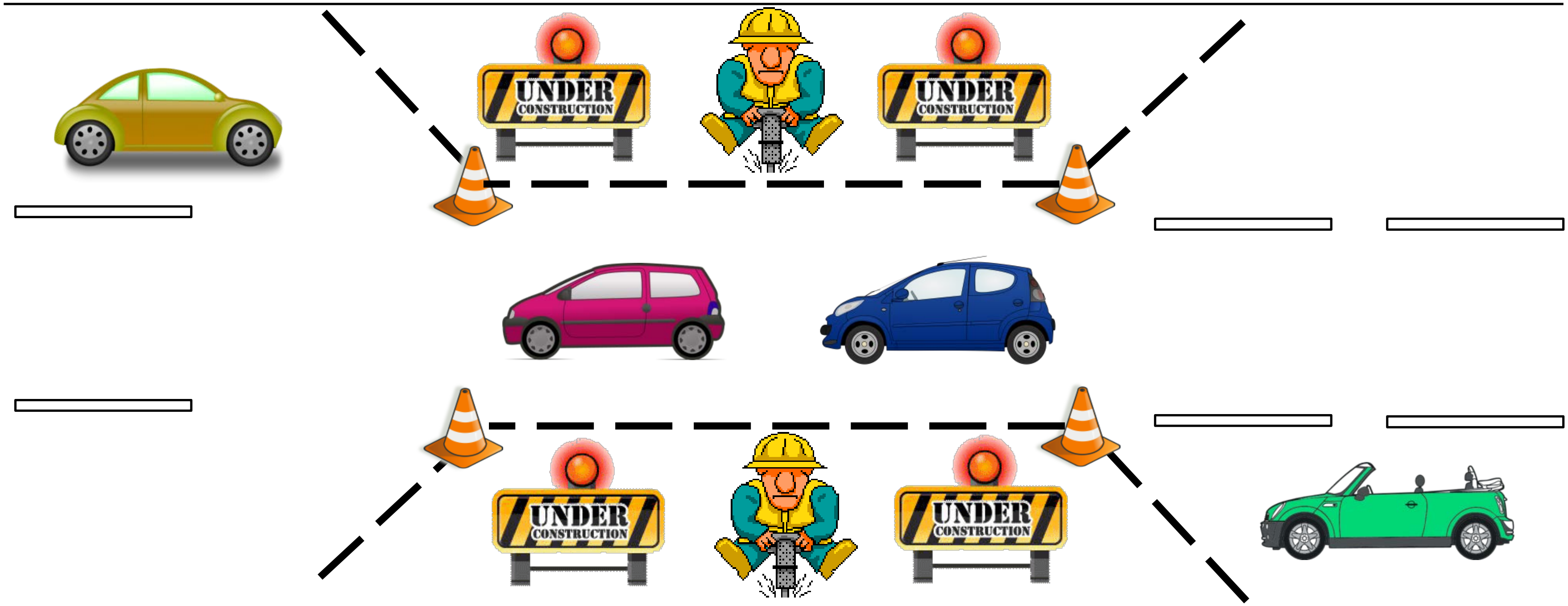
- Safety
  - Means that some particular “bad” thing never happens.
- Liveness
  - Indicates that some particular “good” thing will (eventually) happen.

Example



- Safety violation: if cars moving in opposite directions enter the lane at the same time.

Example



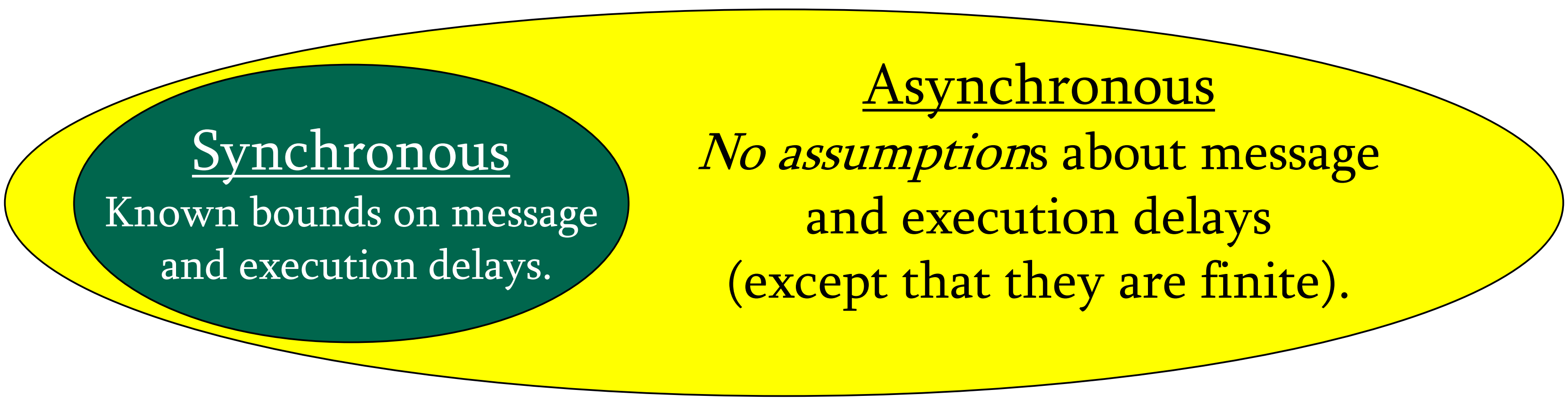
- Liveness: does every car eventually get a chance to go through (i.e., make progress)?
- Progress property (opposite of starvation)

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 <u>correct</u>
- A process that deviates from its specification is <u>faulty</u>
- Many gradations of faulty. Two of interest are:

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

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” (“heartbeat”) 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≠NP…)

Distributed Agreement

Agreement Problems

- High-level goal: Processes in a distributed system reach *agreement* on a value
- Numerous problems can be cast this way
  - Transactional commit, atomic broadcast, …
- The system model is critical to how to solve the agreement problem - or whether it can be solved at all
  - Failure assumptions
  - Timing assumptions

Review: Timing / Failure Models

- Timing assumptions:
  - Synchronous – shared clock, known bounds on message delivery
  - Asynchronous – no global clock, no time bounds on message delivery
  - Partial Synchrony – clocks synchronized within some bound, timeout to manage bounds on message delivery
- Failure assumptions:
  - Fail-stop – process is correct until it stops entirely
  - Byzantine – failed process behaves arbitrarily

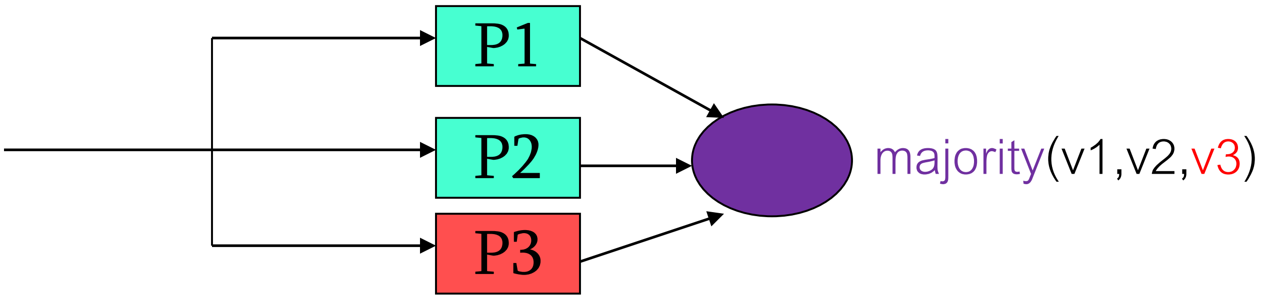
A rose by any other name…

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

"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.

High-level picture

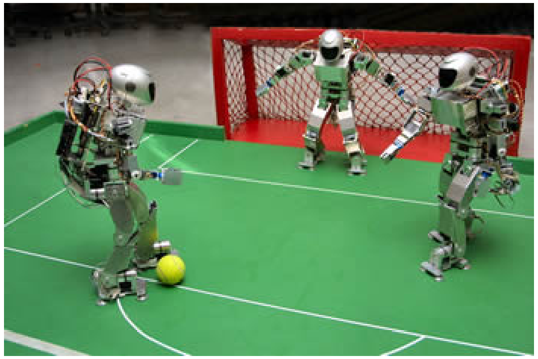
- Goal: Build reliable systems in presence of faulty components
- Common approach:
  - Send request (or input) to some “f-tolerant” server
  - Have multiple (potentially faulty) components compute same function
  - Perform majority vote on outputs to get the “correct” result



f faulty, f+1 good components => 2f+1 total

Setup of Distributed Consensus

- N processes have to agree on a single value.
  - e.g.,
    - 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*.
- Which of these are **Safety** properties and which are **Liveness** properties?

Fail-Stop Faults: Problem Description

- Assumptions:
  - N processes connected by a full graph
  - Each process starts with an initial value {0,1}
  - Synchronous setting: solution is required within a fixed r number of rounds of message exchanges
  - The number of Fail-Stop faults is bounded in advance to f. A process may fail in the middle of a message sending at some round. Once a process fails, it never recovers.
  - No omission failures.

Fail-Stop Faults: Problem Requirements

- *Agreement:* all correct processes decide on the same value
- *Validity:* If a correct process decides on a value, there was a process that started with that value

Synchronous Fail-stop Consensus Algorithm

- Each process maintains a vector containing a value for each process
- In each round:
  - Send your vector to all processes
  - Update local vector according to received vectors
- After f+1 rounds, decide according to local vector
  - e.g., If you have majority 1 in the vector => decide 1; otherwise => decide 0.
- Called “Flood Set algorithm”

Synchronous Fail-stop Consensus Algorithm

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

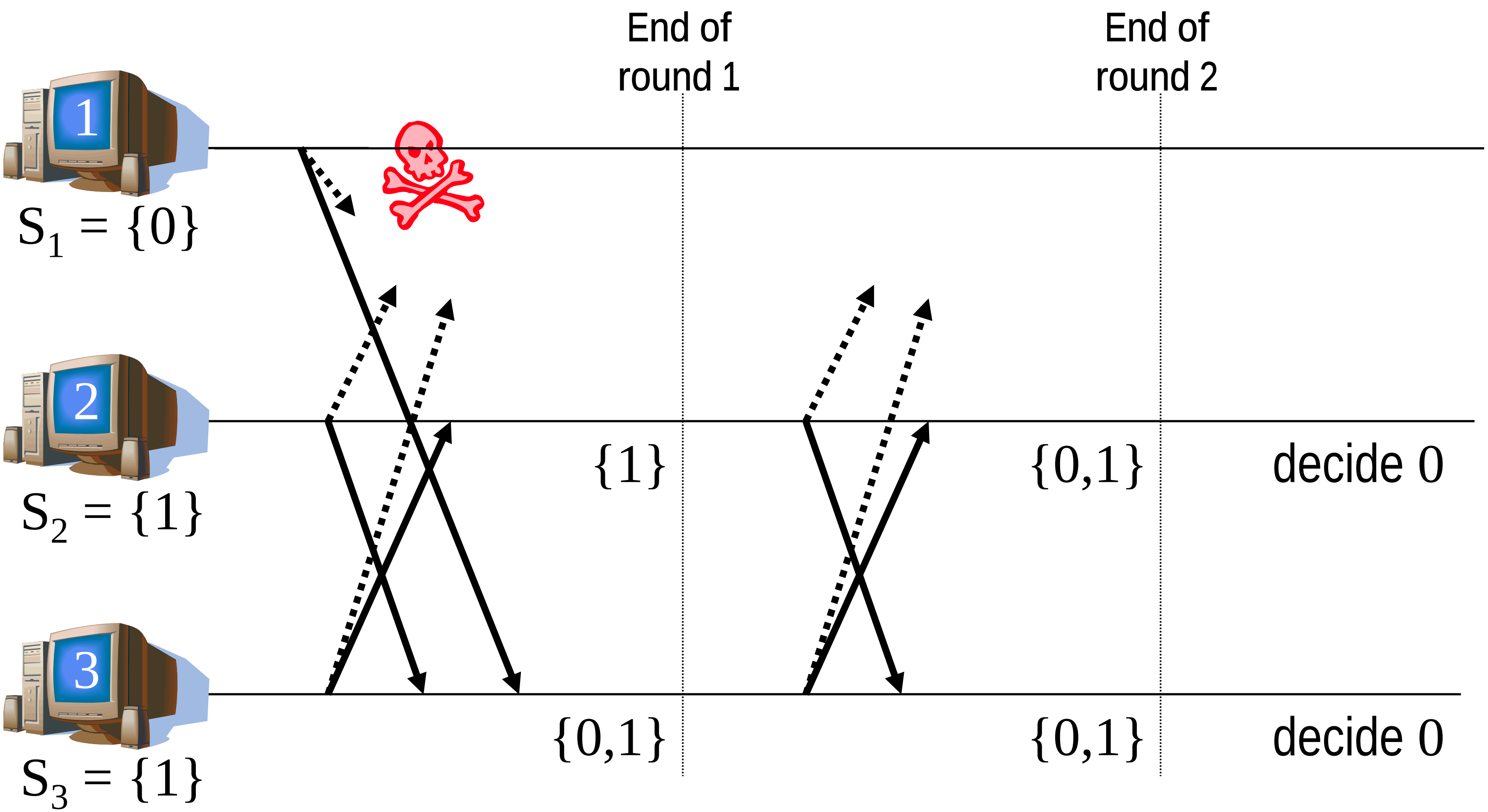
```
Si ← {initial value}
for k = 1 to f+1
    send Si to all processes
    receive Sj from all j != i
       Si ← Si U Sj  (for all j)
end for
Decide(Si)
```

- S is a set of values
- Decide(x) can technically 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
  - Must guarantee 1 round in which no failure occurs
- *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.
- The generals must have an algorithm to guarantee that:
  - A. All loyal generals decide on the same plan of action
    - Implies that all loyal generals must obtain the same information
  - B. 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
- A 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 i^{th} general is loyal, then the value that he 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 a Commanding 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

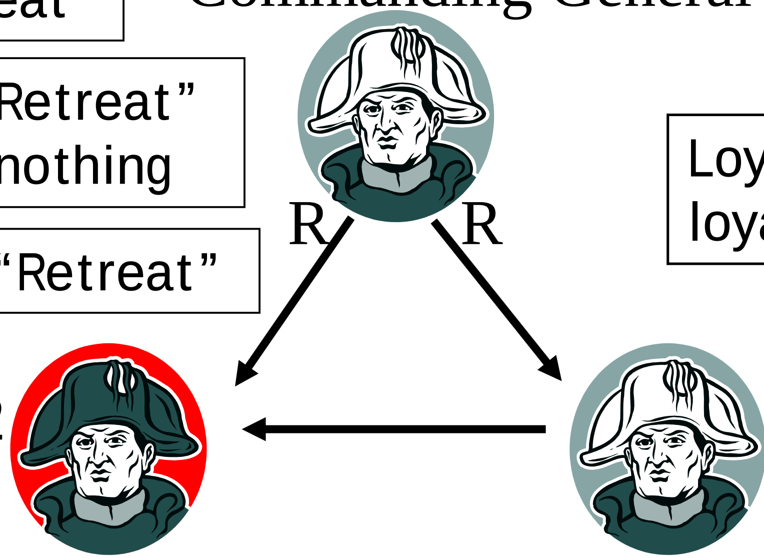
- Traitor lieutenant tries to foil consensus by refusing to participate

"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"



Loyal lieutenant obeys loyal commander. (good)

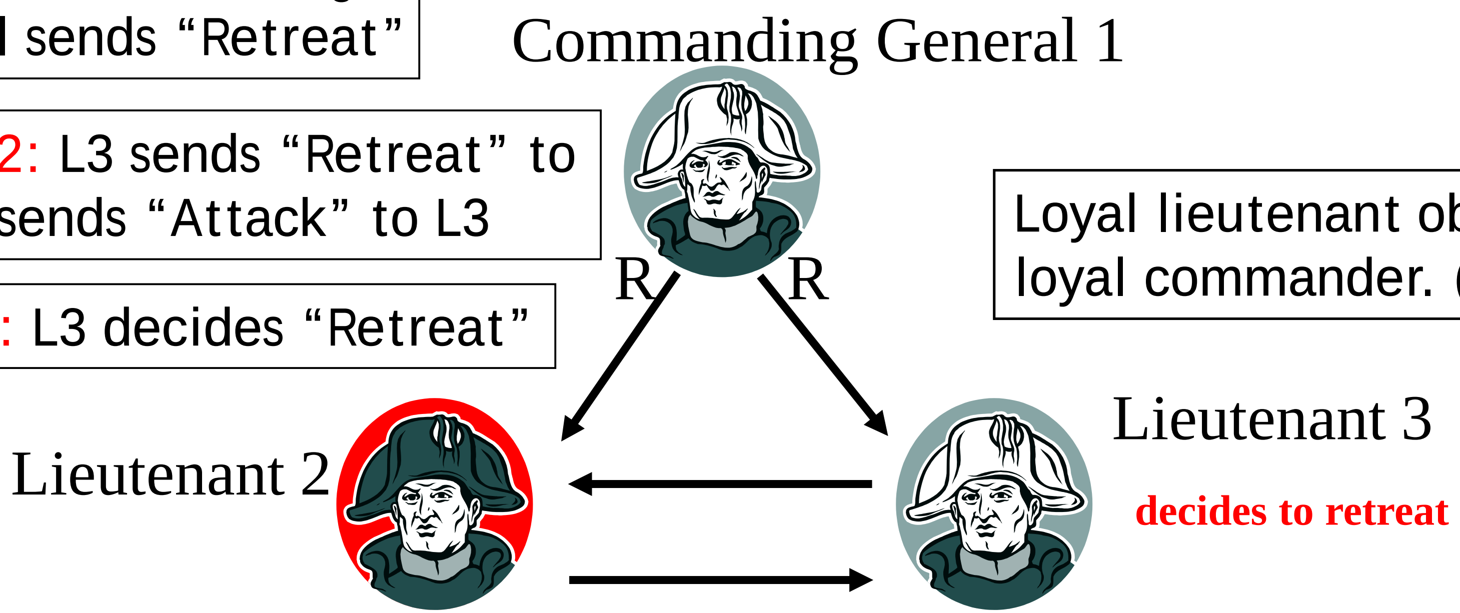
Case 2a

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

Round 1: Commanding General sends “Retreat”

Round 2: L3 sends “Retreat” to L2; L2 sends “Attack” to L3

Decide: L3 decides “Retreat”



Loyal lieutenant obeys loyal commander. (good)

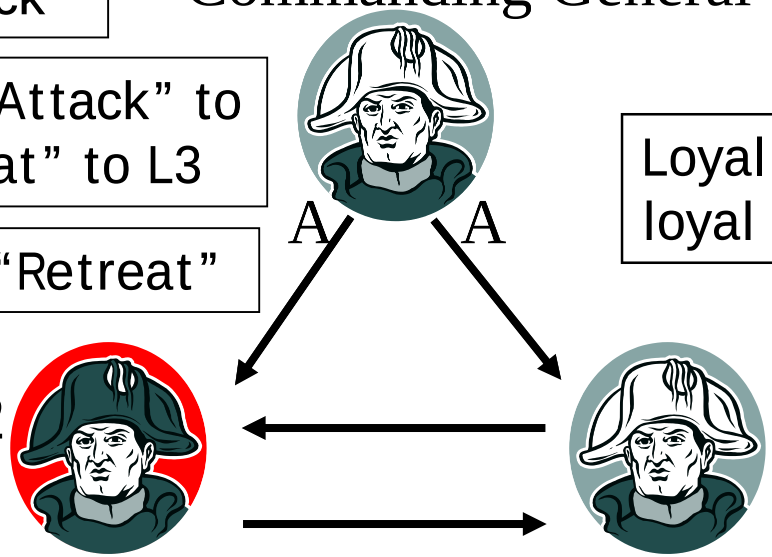
Case 2b

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

Round 1: Commanding General sends "Attack"

Round 2: L3 sends "Attack" to L2; L2 sends "Retreat" to L3

Decide: L3 decides "Retreat"



Loyal lieutenant disobeys loyal commander. (bad)

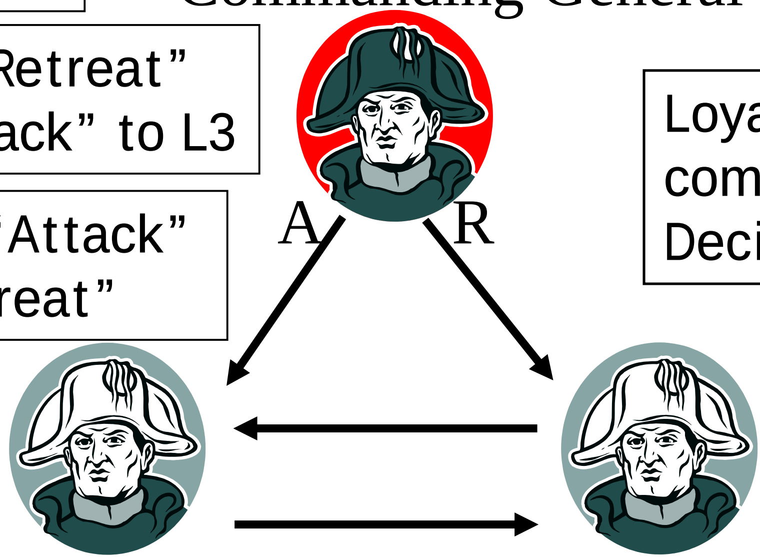
Case 3

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

Round 1: General sends "Attack" to L2 and "Retreat" to L3

Round 2: L3 sends "Retreat" to L2; L2 sends "Attack" to L3

Decide: L2 decides "Attack" and L3 decides "Retreat"



Loyal lieutenants obey commander. (good?)
Decide differently (bad)

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 his 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

f+1 rounds:

1) OM(f): Commander sends his 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* != *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 $majority(v_1, \ldots, v_{n-1})$.
4) Continue until OM(0).

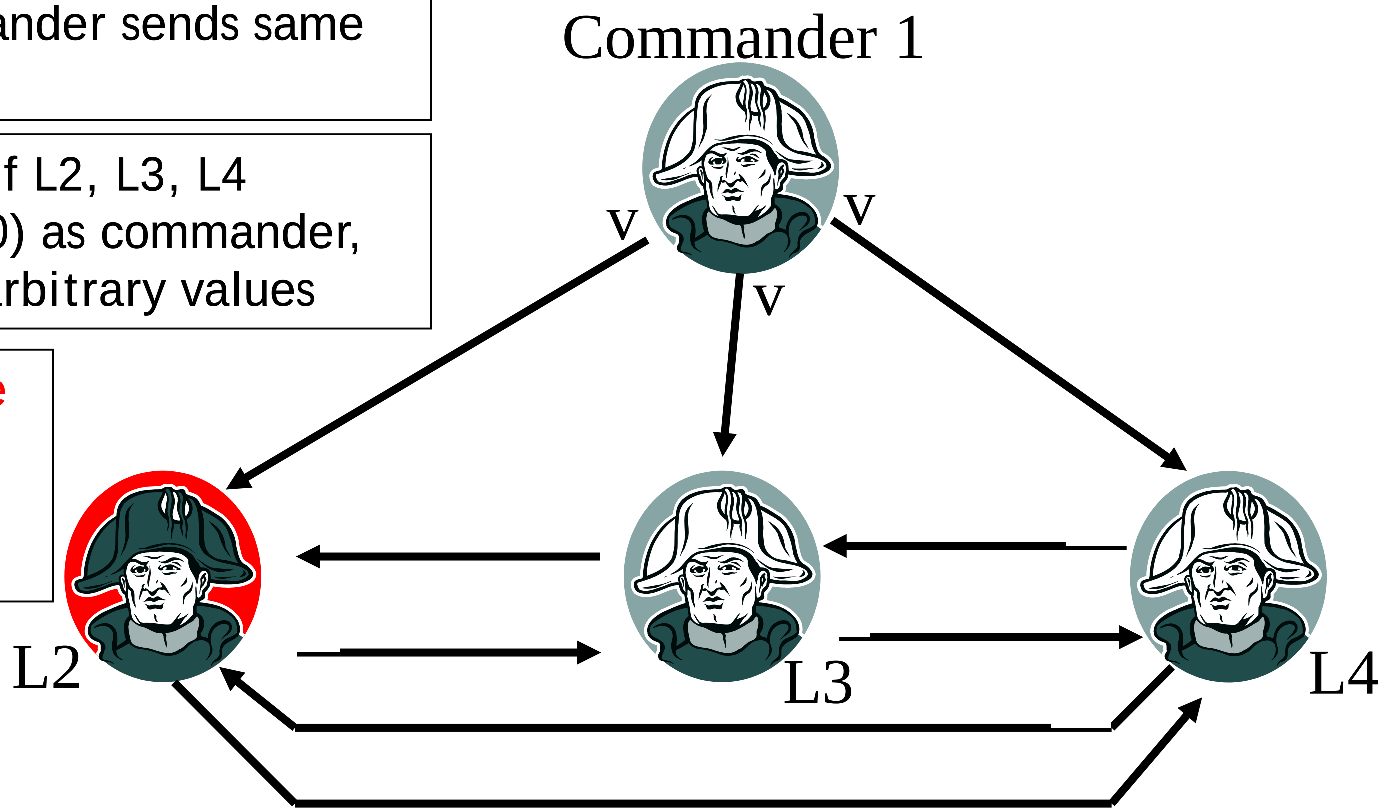
Example: f = 1, n = 4

- Loyal commander, 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.



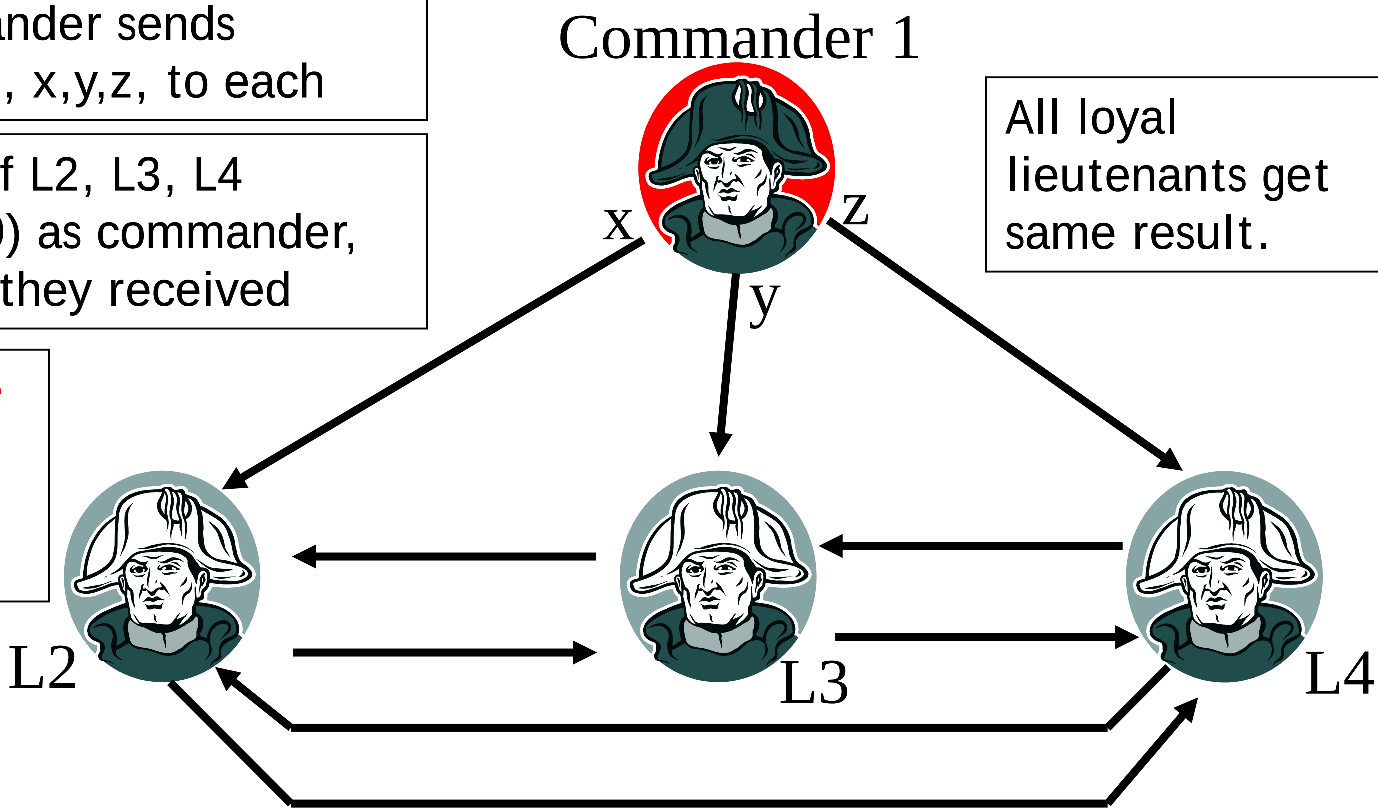
Example: f = 1, n = 4

- Traitor commander, 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,z},



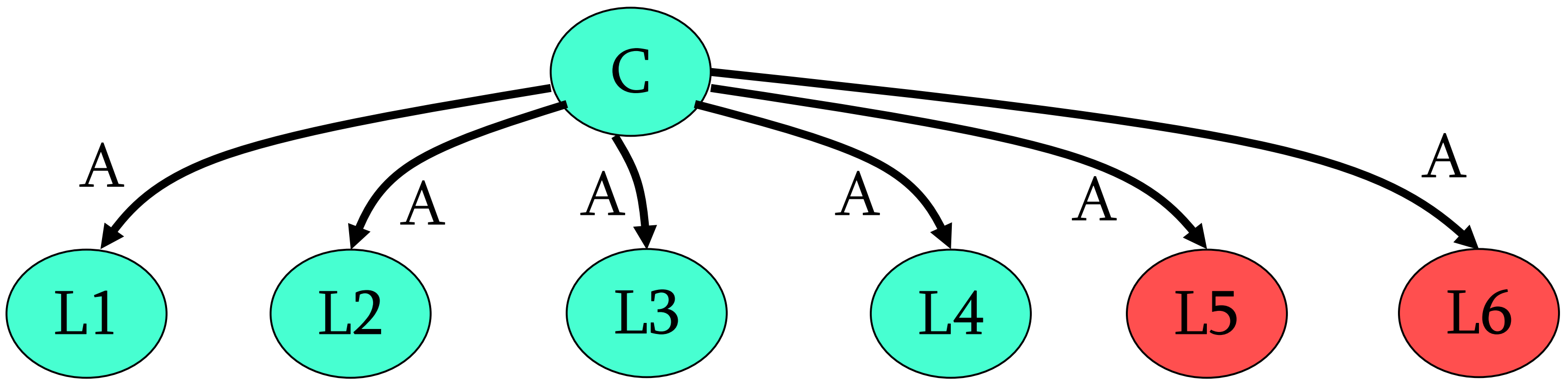
All loyal lieutenants get same result.

Example: OM(2), f=2, n=7

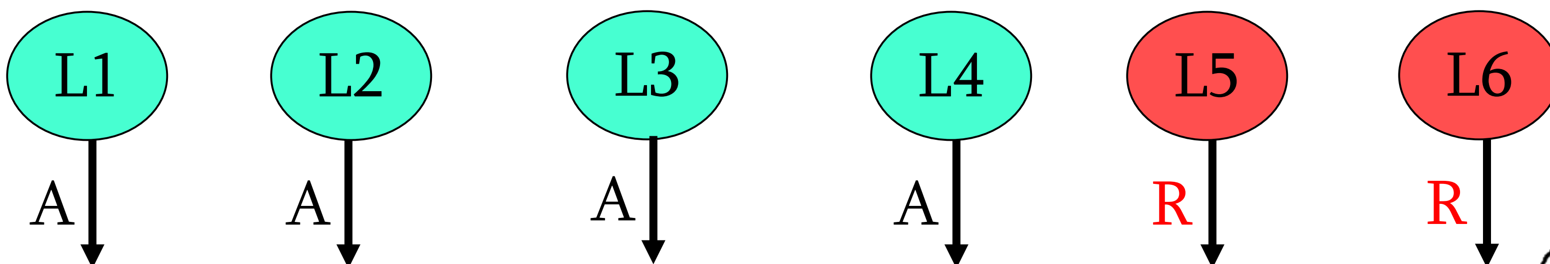
- OM(2): General sends value v to all six lieutenants
- Now run OM(1) six times
  - L_i takes turn as general to send value received from original general to others
  - At end of each OM(1), all lieutenants agree on the value to use for L_i
- Finally, OM(0): All receivers run OM(0) to exchange values
  - To verify that lieutenants tell each other the same thing
  - Msg from L_i of form: "L_0 said v0, L_1 said v1, etc.."
- All lieutenants are now using the same set of values to reach overall decision. Let's see how..

Example: OM(2), f=2, n=7

- **Traitors: L5, L6**



- Now run OM(1) six times

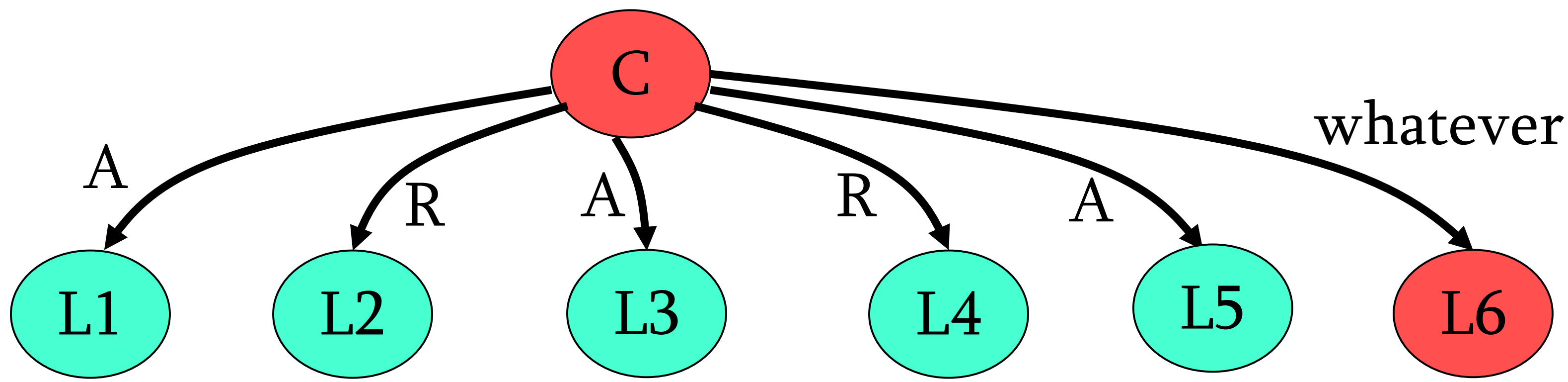


- All loyal lieutenants decide with maj(A, A, A, A, R, R)

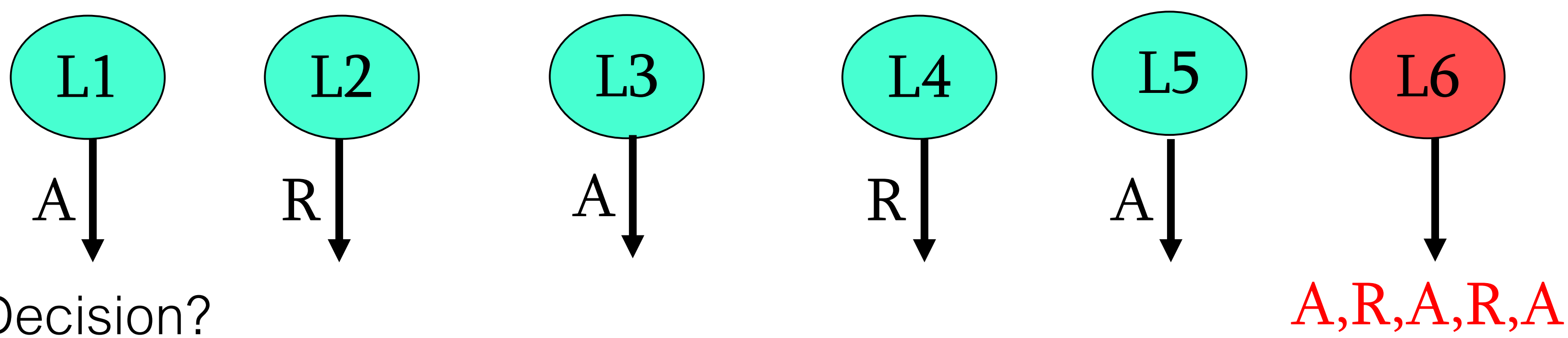
  => all loyal lieutenants attack!

Example: OM(2), f=2, n=7

- **Traitors: C, L6**



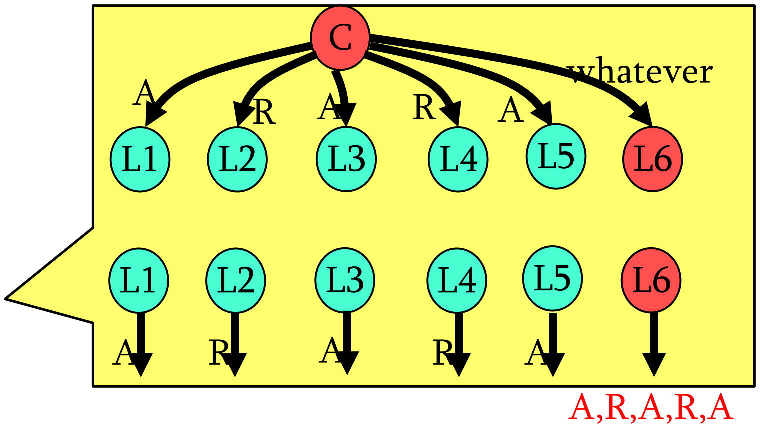
- Now run OM(1) six times



- Decision?

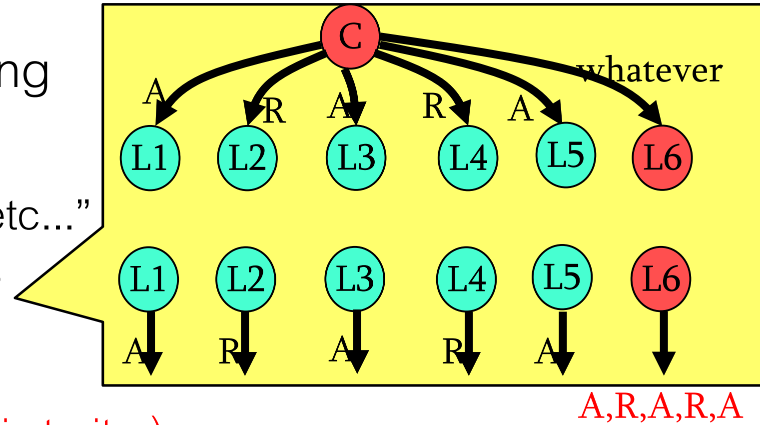
Decision with Bad Commander

- L1: maj(A,R,A,R,A,A) => Attack
- L2: maj(A,R,A,R,A,R) => Retreat
- L3: maj(A,R,A,R,A,A) => Attack
- L4: maj(A,R,A,R,A,R) => Retreat
- L5: maj(A,R,A,R,A,A) => Attack
- Problem: All loyal lieutenants do NOT choose same action



Next Step of Algorithm

- Verify that lieutenants tell each other the same thing
  - Requires rounds = f+1
  - OM(0): Msg from Li of form: “L0 said v0, L1 said v1, etc...”
- What messages does L1 receive in this example?
  - OM(2): A
  - OM(1): 2/R, 3/A, 4/R, 5/A, 6/A (doesn’t know 6 is traitor)
  - OM(0): L1 sees:
  - 2{ 3/A, 4/R, 5/A, 6/R}
  - 3{2/R, 4/R, 5/A, 6/A}
  - 4{2/R, 3/A, 5/A, 6/R}
  - 5{2/R, 3/A, 4/R, 6/A}
  - 6{ total confusion }
- All loyals see same messages in OM(0) from L1,2,3,4, and 5
- maj(1/A,2/R,3/A,4/R,5/A,-) => All attack



What if 6 “played nice” and sent everyone the same value?

Try this with f=2, n=6!

What happens in the end?

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
- Problem: Traitors can lie about what others said. => Restrict this ability!
- If we use *signed messages*, instead of oral messages, can handle f faults with 2f+1 processes
  - Loyal general's signature cannot be forged => limits traitors
  - Simple majority requirement

Signed messages: case 1

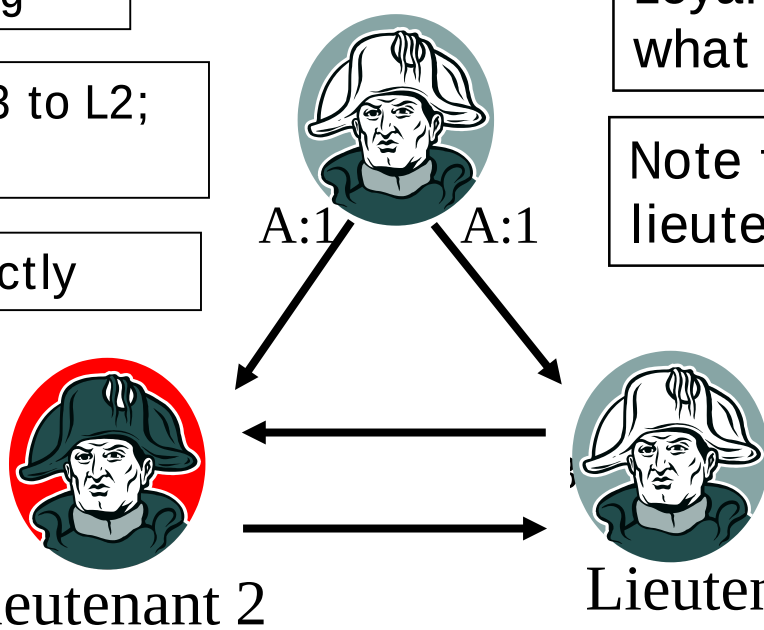
- Let x:i denote the value x signed by general i
  - v:j:i is value v signed by j, and then v:j signed by i

Traitor Lieutenant

Round 1: Commanding General sends signed “Attack” msg

Round 2: L3: signed A:1:3 to L2; L2 sends A:1:2 to L3

Decide: L3 decides correctly



Loyal lieutenant knows what order to obey. (good!)

Note that the traitor lieutenant cannot do much!

Signed messages: case 2

- Let x:i denote the value x signed by general i
  - v:j:i is value v signed by j, and then v:j signed by i

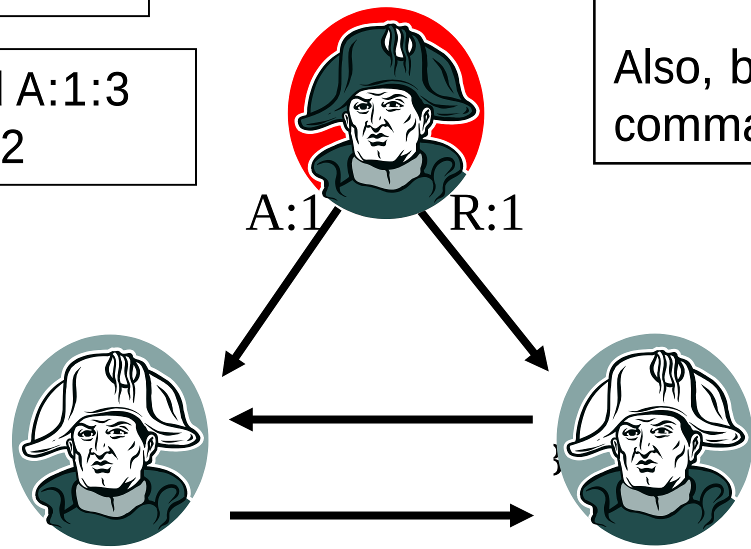
Traitor Commander

Round 1: Traitor General sends signed “A” to L2, and “R” to L3

Round 2: L2: sends signed A:1:3 to L3; L3 sends R:1:2 to L2

Decide: Both L2 and L3 have same set of orders: {A, R}

Commanding General 1



decides Choice(A, R)

Lieutenant 2

Lieutenant 3

decides Choice(A, R)

Both loyal lieutenants decide the same thing (good!)

Also, both lieutenants know the commander is a traitor. Why?

Conclusions

- Problem: To implement a fault-tolerant service with coordinated replicas, must agree on inputs
- Byzantine failures make agreement challenging
  - Produce arbitrary output, can't detect, collude
- Use different agreement protocol depending on assumptions
  - Oral messages: Need 3f+1 nodes to tolerate f failures
    - Difficult because traitors can lie about what others said
  - Signed messages: Need 2f+1 nodes
    - Easier because traitors can only lie about other traitors

Asynchronous Distributed Consensus

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

More Byzantine Fault Tolerance

- Castro and Liskov: Practical Byzantine 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
- Zyzzyva: Speculative Byzantine Fault Tolerance
- The Next 700 BFT Protocols, Guerraoui et al. (Eurosys 2010)
- A form of BFT is used in Bitcoin