Byzantine Generals Problem

"Byzantine Generals" metaphor used in the classical paper by Lamport et al. [Lamport et al., 1982]

The paper considered a *synchronous system*, i.e., a system in which there are known delay bounds for processing and communication.

Byzantine Generals

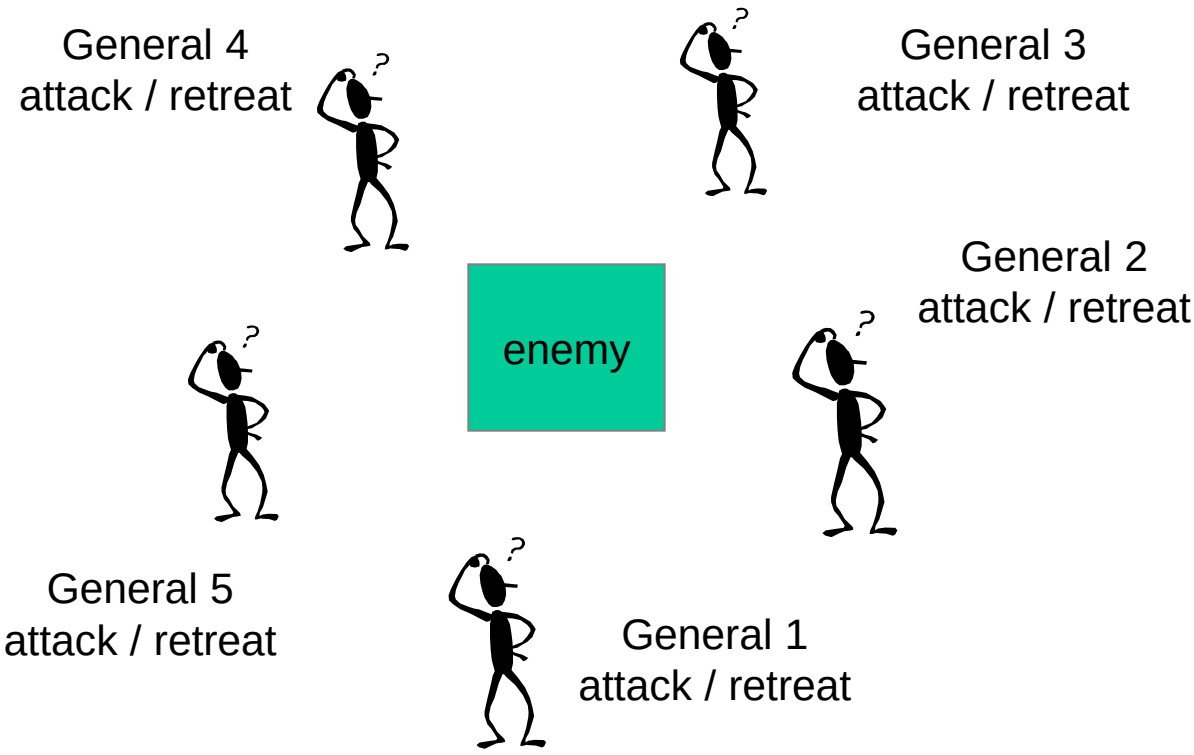
The problem is given in terms of generals who have surrounded the enemy.

Generals wish to organize a plan of action to attack or to retreat.
Each general observes the enemy and communicates his observations to the others.

Unfortunately there are traitors among generals and traitors want to influence this plan to the enemy's advantage. They may lie about whether they will support a particular plan and what other generals told them.

L. Lamport, R. Shostak, M. Pease
The Byzantine Generals Problem
ACM Trans. on Progr. Languages and Systems, 4(3),1982

Byzantine Generals Problem



<u>General: either a loyal general or a traitor</u>

Consensus:

A: All loyal generals decide upon the same plan of actions
B: A small number of traitors cannot cause loyal generals to adopt a bad plan

What algorithm for decision making should the generals use to reach a Consensus?

What percentage of liars can the algorithm tolerate and still correctly determine a Consensus?

Byzantine Generals Problem

Assume plan of actions: **attack or retreat**

Let

- n be the number of generals
- v(i) be the opinion of general i (attack/retreat)
- each general *i* communicate the value *v(i)* by messangers to each other general j
- each general final decision obtained by: majority vote among the values v(1), ..., v(n)

To satisfy condition A:

every general must apply the majority function to the same values *v(1),...,v(n).* **But** a traitor may send different values to different generals thus generals may receive different values

To satisfy condition B:

for each *i*, if the i-th general is loyal, then the value he sends must be used by every loyal general as the value *v(i)*

Byzantine Generals Problem

Let us consider the Consensus problem into a simpler situation in which we have:

1 commanding general (C)

n-1 lieutenant generals (L1, ..., Ln-1)

The Byzantine commanding general C wishes to organize a plan of action to attack or to retreat; he sends the command (attack/retrait) to every lieutenant general Li.

There are traitors among generals (commanding general and/or lieutenant general)

Consensus:

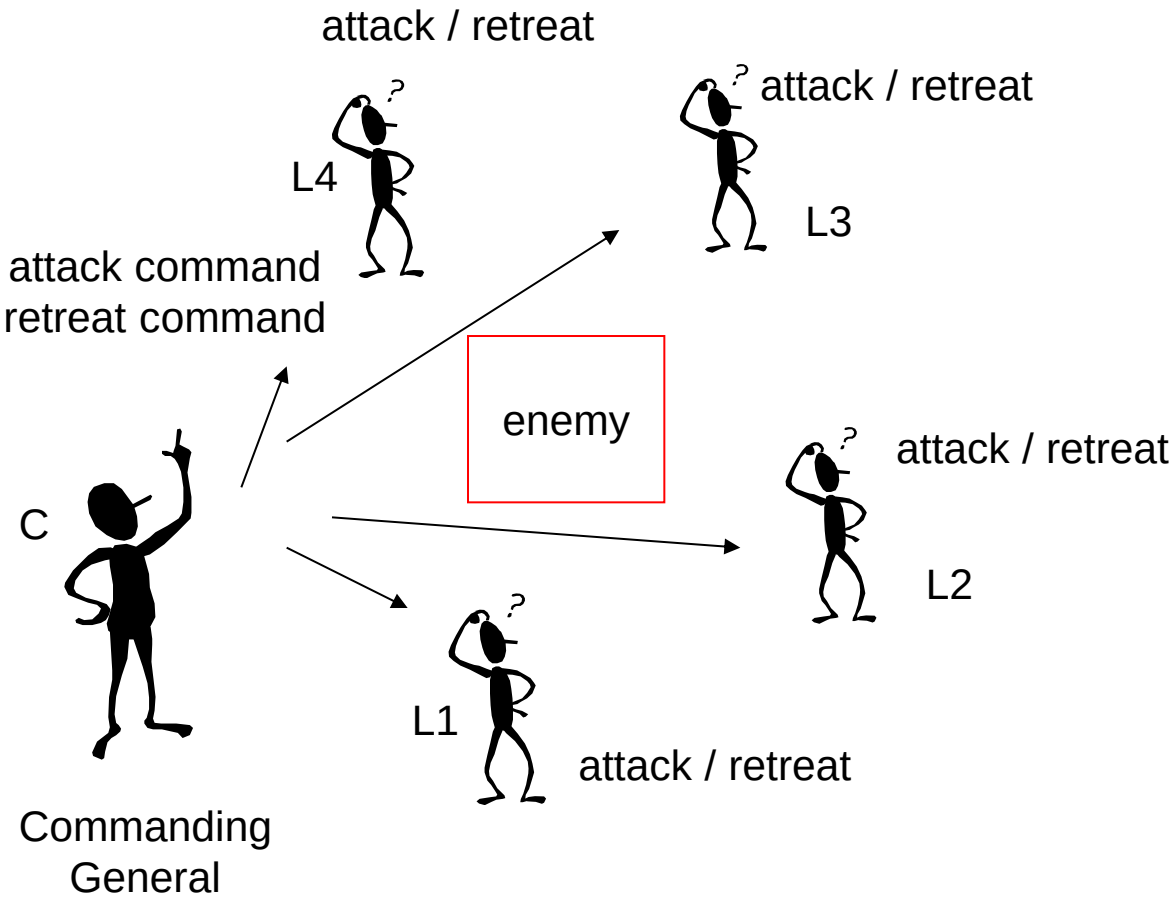
IC1: All loyal lieutenant generals obey the same command

IC2: The decision of loyal lieutenants must agree with the commanding general's order if he is loyal.

IC1 and IC2: Interactive Consistency conditions.

Note:
If the commander is loyal then IC1 follows from IC2

Byzantine Generals Problem



If the commanding general lies but sends the same command to lieutenants, IC1 and IC2 are satisfied.

Assume the commanding general lies and sends
- attack to some lieutenant generals
- retreat to some other lieutenant generals

How loyal lieutenant generals may all reach the same decision either to attack or to retreat ?

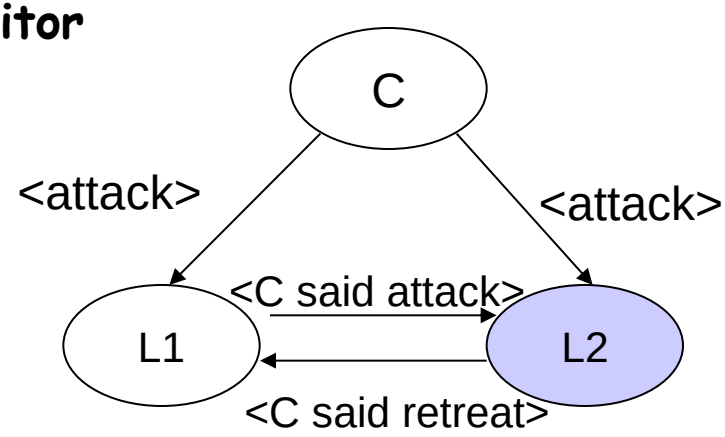
Lieutenant generals send messages back and forth among themselves reporting the command received by the Commanding General.

Byzantine Generals Problem

n = 3

no solution exists in presence of a traitor

L2 traitor



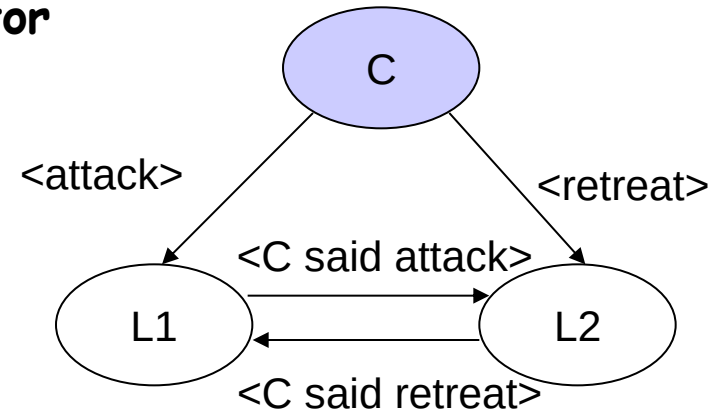
In this situation (two different commands, one from the commanding general and the other from a lieutenant general), assume L1 must obey the commanding general.

L1 decides attack.

IC1 and IC2 are satisfied.

(If L1 must obey the lieutenant general, IC2 is not satisfied)

C traitor



The situation is the same as before, and the same rule is applied

L1 must obey the commanding general and decides attack
L2 must obey the commanding general and decides retreat

IC1 is violated
IC2 is satisfied (the comanding general is a traitor)

To cope with 1 traitor, there must be at least 4 generals

Byzantine Generals Problem

In the following we show the Oral Message algorithm that gives a solution when

1. the system is synchronous
2. any two processes have direct communication across a network *not prone to failure itself* and *subject to negligible delay*
3. *the sender of a message can be identified by the receiver*

In particular, the following assumptions hold.

Assumption

A1. Every message that is sent by a non faulty process is correctly delivered

A2. The receiver of a message knows who sent it

A3. The absence of a message can be detected

Moreover, a traitor commander may decide not to send any order. In this case we assume a default order equal to "retreat".

Similarly the function majority(v1, ..., vn-1) returns "retrait" if there not exists a majoirity among values

Oral Message (OM) algorithm

The Oral Message algorithm OM(m) by which a commander sends an order to n-1 lieutenants, solves the Byzantine Generals Problem for n = (3m +1) or more generals, in presence of at most m traitors.

Function majority(v1, ..., vn-1)

---

majority(v1, ..., vn-1)

if a majority of values vi equals v,
then
majority(v1, ..., vn-1) equals v
else
majority(v1, ..., vn-1) equals retreat

---

deterministic majority vote on the values

The algorithm

Algorithm OM(0)
1. C sends its value to every Li, i∈{1, ..., n-1}
2. Each Li uses the received value, or the value retreat if no value is received

Algorithm OM(m), m>0
1. C sends its value to every Li, i∈{1, ..., n-1}

2. Let vi be the value received by Li from C (vi = retreat if Li receives no value) Li acts as C in OM(m-1) to send vi to each of the n-2 other lieutenants

3. For each i and j ≠ i, let vj be the value that Li received from Lj in step 2 using Algorithm OM(m-1) (vj = retreat if Li receives no value). Li uses the value of majority(v1, ..., vn-1)

OM(m) is a recursive algorithm that invokes n-1 separate executions of OM(m-1), each of which invokes n-2 executions of O(m-2), etc..

For m >1, a lieutenant sends many separated messages to the other lieutenants.

To distinguish these messages, each lieutenent *i* prefixes the number *i* to the value sent

→ messages are *prefixed* by a *sequence of numbers of lieutenants*

The algorithm

4 generals, 1 traitor

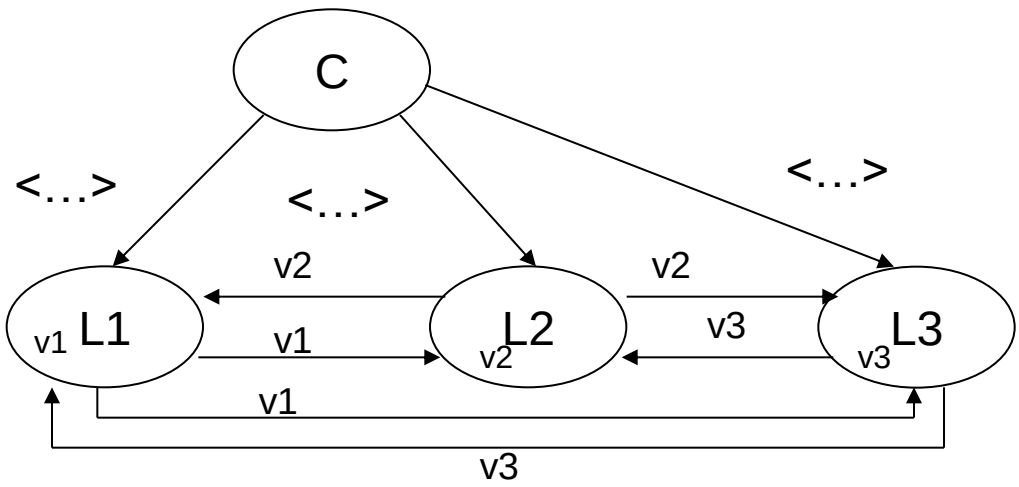
OM(1)

Point 1

- C sends the command to L1, L2, L3.
- L1 applies OM(0) and sends the command he received from C to L2 and L3
- L2 applies OM(0) and sends the command he received from C to L1and L3
- L3 applies OM(0) and sends the command he received from C to L1 and L2

Point 2

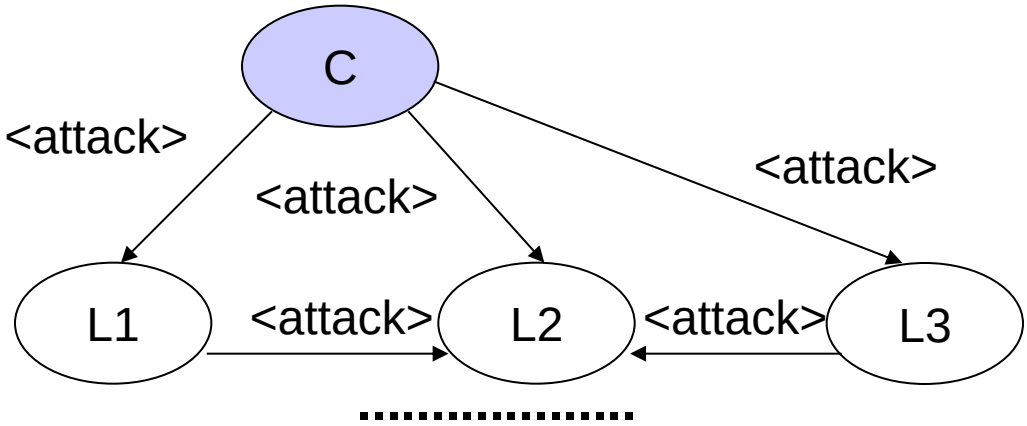
- L1: majority(v1, v2, v3)
- L2: majority(v1, v2, v3)
  //v1 comando che L1 dice di avere ricevuto
  //v3 comando che L3 dice di avere ricevuto
- L3: majority(v1, v2, v3)



The algorithm

4 generals, 1 traitor **n=4, m=1**

C is a traitor but sends the same command to L1, L2 ad L3

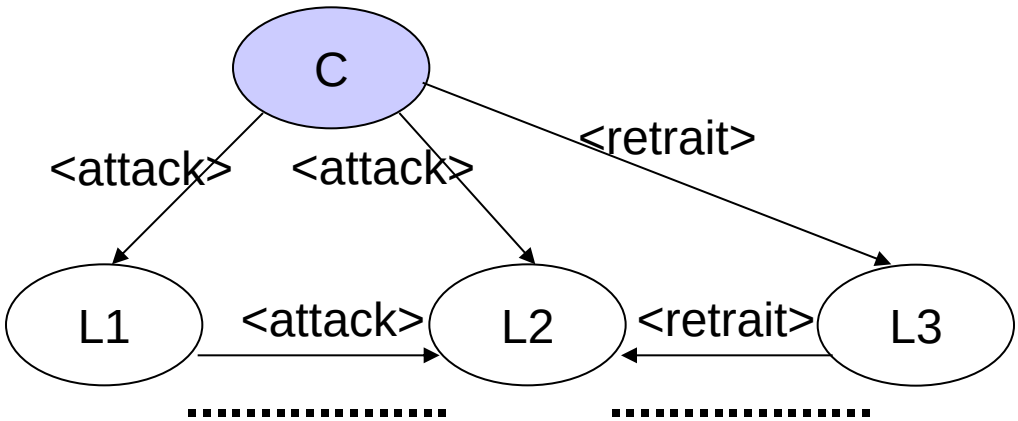


Li: v1 = attack, v2 =attack, v3 = attack
majority(....)= attack

L1, L2 and L3 are loyal. They send the same command when applying OM(0)
IC1 and IC2 are satisfied

The algorithm

C is a traitor and sends:
attack to L1 and L2
retrait to L3



L1, L2 and L3 are loyal.

L1: v1 = attack, v2 =attack, v3 = retrait
majority(...)= attack

L2: v1 = attack, v2 =attack, v3 = retrait
majority(...)= attack

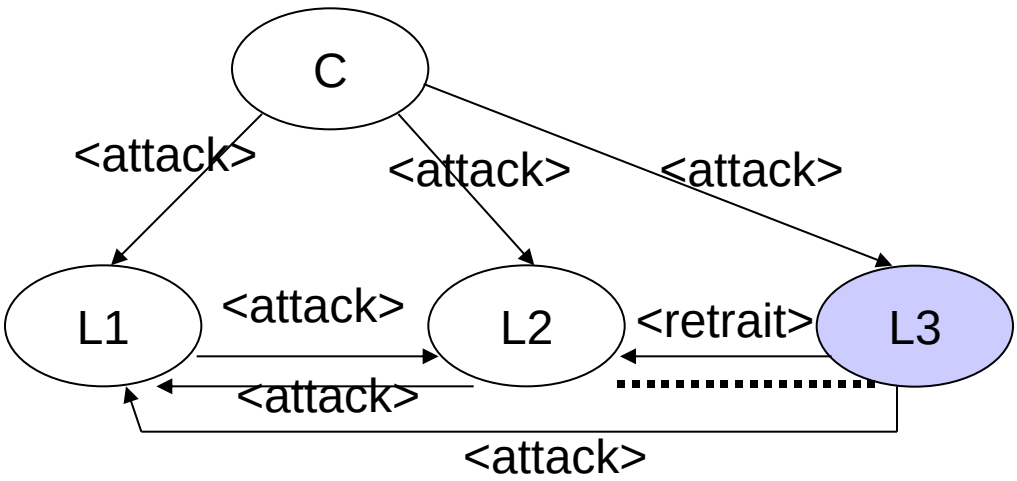
L3: v1 = attack, v2 =attack, v3 = retrait
majority(...)= attack

IC1 and IC2 satisfied

The algorithm

A leutenant is a traitor

L3 is a traitor:
sends retrait to L2 and attack to L1



L1: v1 = attack v2 = attack, v3 = attack
majority(...) = attack

L2: v1 = attack v2 = attack, v3 = retrait
majority(...) = attack

IC1 and IC2 satisfied

The algorithm

The following theorem has been formally proved:

Theorem:

For any **m**, algorithm **OM(m)** satisfies conditions **IC1** and **IC2** if there are more than **3m** generals and at most **m** traitors. Let **n** the number of generals: **n >= 3m +1**.

4 generals are needed to cope with 1 traitor;
7 generals are needed to cope with 2 traitors;
10 generals are neede to cope with 3 traitors
.....

Byzantine Generals Problem

Original Byzantine Generals Problem

Solved assigning the role of commanding general to every lieutenant general, and running the algorithms concurrently

Each general observes the enemy and communicates his observations to the others

→ Every general *i* sends the order "*use v(i) as my value*"

Consensus on the value sent by general *i*

→ algorithm OM

Each general combines *v(1),…,v(n)* into a plan of actions

→ Majority vote to decide attack/retreat

General agreement among n processors, m of which could be faulty and behave in arbirary manners. Each processor holds a secret value that wishes to share with other processors.

No assumptions on the characteristics of faulty processors

Conflicting values are solved taking a deterministic majority vote on the values received at each processor (completely distributed).

Remarks

Solutions of the Consensus problem are expensive:

Assume m be the maximum number of faulty nodes

OM(m):
each L_i waits for messages originated at C and relayed via m others L_j

OM(m) requires
n = 3m +1 nodes
m+1 rounds
message of the size $O(n^{m+1})$ - message size grows at each round

Algorithm evaluation using different metrics: number of fault processors / number of rounds / message size

In the literature, there are algorithms that are optimal for some of these aspects.

Signed messages

The ability of the traitor to lie makes the Byzantine Generals problem difficult

→ **restrict the ability of the traitor to lie**

A solution with signed messages:
allow generals to send
unforgeable signed messages

Signed messages (authenticated messages):
- Byzantine agreement becomes much simpler

A message is authenticated if:

1. a message signed by a fault-free processor cannot to be forged

2. any corruption of the message is detectable

3. the signature can be authenticated by any processors

Signed messages limit the capability of faulty-processors

Byzantine Generals Problem

…

Assumptions

A1. Every message that is sent by a non faulty process is correctly delivered

A2. The receiver of a message knows who sent it

A3. The absence of a message can be detected

Signed messages

Assumption A4

(a) The signature of a loyal general cannot be forged, and any alteration of the content of a signed message can be detected

(b) Anyone can verify the authenticity of the signature of a general

No assumptions about the signatures of traitor generals

Let V be a set of orders. The function choice(V) obtains a single order from a set of orders:

For choice(V) we require:

choice(∅) = retreat
choice(V) = v if V consists of the single element v

One possible definition of choice(V) is:
choice(V) = retrait if V consists of more than 1 element

---

x:i denotes the message x signed by general i
v:j:i denotes the value v signed by j and then the value v:j signed by i

General 0 is the commander

For each i, Vi contains the *set of properly signed orders* that lieutenant Li has received so far

Signed messages

Algorithm SM(m)

Vi = $\varnothing$

1. C signs and sends its value to every Li, i$\in\{1, ..., n-1\}$

2. For each i:
   (A) if Li receives v:0 and Vi is empty
   then Vi = $\{v\}$;
   sends v:0:i to every other Lj

   (B) if Li receives v:0:j1:...:jk and v $\notin$ Vi
   then Vi = Vi $\cup$ $\{v\}$;
   if k < m then
   sends v:0:j1:...:jk:i to every
   other Lj , j $\notin\{j1, ..., jk\}$

3. For each i: when Li will receive no more msgs,
   he obeys the order choice(Vi)

Observations:

- Li ignores msgs containing an order v$\in$Vi

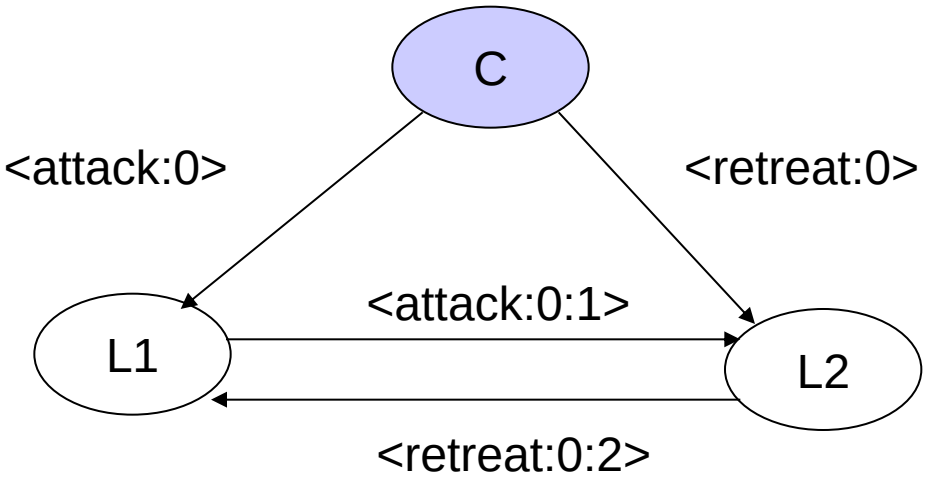
- Time-outs are used to determine when no more messages will arrive

- **If Li is the m-th lieutenant that adds the signature to the order, then the message is not relayed to anyone.**

Signed messages

3 generals, 1 traitor

C is a traitor and sends:
attack to L1 and L2
retrait to L3



V1 = {attack, retreat} V2 = {attack, retreat}

- L1 and L2 obey the order **choice({attack, retreat})**

- L1 and L2 know that **C** is a traitor because the signature of **C** appears in two different orders

The following theorem asserting the correctness of the algorithm has been formally proved.

Theorem :
For any m, algorithm SM(m) solves the Byzantine Generals Problem if there are at most m traitors.

Remarks

Consider Assumption **A1.**

Every message that is sent by a non faulty process is delivered correctly

→ **For the oral message algorithm:**
the failure of a communication line joining two processes is indistinguishable from the failure of one of the processes

→ **For the signed message algorithm:**
if a failed communication line cannot forge signed messages, the algorithm is insensitive to communication line failures.
Communication line failures lowers the connectivity

Consider Assumption **A2.**

The receiver of a message knows who sent it

→ **For the oral message algorithm**:
a process can determine the source of any message that it received.
Interprocess communications over fixed lines

→ **For the signed message algorithm:**
Interprocess communications over fixed lines or *switching network*

Remarks

Consider Assumption **A3**:

The absence of a message can be detected

For the oral/signed message algorithm: *timeouts*

- *requires a fixed maximum time for the generation and transmission of a message*
- *requires sender and receiver have clocks that are synchronised to within some fixed maximum error*

Consider Assumption **A4**:

(a) a loyal general signature cannot be forged, and any alteration of the content of a signed message can be detected
(b) anyone can verify the authenticity of a general signature

- *probability of this violation as small as possible*
- *cryptography*

Consensus in Asynchronous systems

Consensus in Asynchronous systems

Asynchronous distributed system:
no timing assumptions (no bounds on message delay, no bounds on the time necessary to execute a step)

Asynchronous model of computation: attractive.

- Applications programmed on this basis are easier to port than those incorporating specific timing assumptions.

- Synchronous assumptions are at best probabilistic:
in practice, variable or unexpected workloads are sources of asynchrony

Impossibility result

Consensus: cannot be solved deterministically in an asynchronous distributed system that is subject even to a single crash failure [Fisher, Lynch and Paterson 85]

→ due to the difficulty of determining whether a process has **actually crashed** or is **only very slow**.

If no assumptions are made about the upper bound on how long a message can be in transit, nor *the upper bound on the relative rates of processors*, then a single processor running the consensus protocol could simply halt and delay the procedure indefinitely.

Stopping a single process at an inopportune time can cause any distributed protocol to fail to reach consensus

M.Fisher, N. Lynch, M. Paterson
Impossibility of Distributed Consensus with one faulty process.
Journal of the Ass. for Computing Machinery, 32(2), 1985.

Circumventing FLP

Techniques to circumvent the impossibility result:

Augmenting the System Model with an Oracle

A (distributed) Oracle can be seen as some component that processes can query. An oracle provides information that algorithms can use to guiide their choices. The most used are **failure detectors**.

Since the information provided by these oracles makes the problem of consensus solvable, they augment the power of the asynchronous system model.

- *Failure detectors*

a failure detector is an oracle that provides information about the current status of processes, for instance, whether a given process has crashed or not.

A failure detector is modeled as a set of distributed modules, one module *Di* attached to each process *pi*. Any process *pi* can query its failure detector module *Di* about the status of other processes.

T. D. Chandra, S. Toueg
Unreliable Failure Detectors for Reliable Distributed Systems.
Journal of the Ass. For Computing Machinery, 43 (2), 1996.

Failure detectors

Failure detectors are considered *unreliable*, in the sense that they provide information that may not always correspond to the real state of the system.

For instance, a failure detector module *Di* may provide the erroneous information that some process *pj* has crashed whereas, in reality, *pj* is correct and running.
Conversely, *Di* may provide the information that a process *pk* is correct, while *pk* has actually crashed.

To reflect the unreliability of the information provided by failure detectors, we say that

a process *pi suspects* some process *pj* whenever *Di* , the failure detector module attached to *pi*, returns the (unreliable) information that *pj* has crashed.

In other words, a suspicion is a belief (e.g., "*pi* believes that *pj* has crashed") as opposed to a known fact (e.g., "*pj* has crashed and *pi* knows that").

Several failure detectors use sending/receiving of messages and **time-outs** as fault detection mechanism.

Randomized Byzantine consensus

- **Random Oracle**

  introduce the ability to generate random values.
  Processes could have access to a module that generates a random bit when queried
  Used by a class of algirithms called **randomized algorithms.**
  These algorithms solve consensus in a probabilistic manner.

The probability that such algorithms terminate before some time t, goes to 1, as t goes to infinity.

Almost all randomized algorithms choose to modify the Termination property, which becomes:

P-Termination: Every correct process eventually decides with probability 1.

Solving a problem deterministically and solving a problem with probability 1 are not the same.

“Termination: Every correct process eventually decides.”

Randomized Byzantine consensus

All randomized consensus algorithms are based on a random operation, tossing a coin, which returns values 0 or 1 with equal probability.

These algorithms can be divided in two classes depending on how the tossing operation is performed:

1) local coin mechanism in each process simpler but terminate in an expected exponential number of communication steps

2) shared coin that gives the same values to all processes require an additional coin sharing scheme but can terminate in an expected constant number of steps

Adding time to the model

Adding Time to the Model

- *using the notion of* partial synchrony

Partial synchrony model: captures the intuition that systems can behave asynchronously (i.e., with variable/unkown processing/ communication delays) for some time, but that they eventually stabilize and start to behave (more) synchronously.

The system is mostly asynchronous but we make assumptions about time properties that are eventually satisfied. Algorithms based on this model are typically guaranteed to terminate only when these time properties are satisfied.

Two basic partial synchrony models, each one extending the asynchronous model with a time property are:

• M1: For each execution, there is an unknown bound on the message delivery time, which is always satisfied.

• M2: For each execution, there is an unknown global stabilization time GST, such that a **known bound** on the message delivery time is always satisfied from GST.

Wormholes

Wormholes: enhanced components that provide processes with a means to obtain a few simple privileged functions with “good” properties otherwise not guaranteed by the normal.

Example, a wormhole can provide timely or secure functions in, respectively, asynchronous or Byzantine systems.

Consensus algorithms based on a wormhole device called Trusted Timely Computing Base (TTCB) have been defined.

TTCB is a secure real-time and fail-silent distributed component. Applications implementing the consensus algorithm run in the normal system, i.e., in the asynchronous Byzantine system.

TTCB is locally accessible to any process, and at certain points of the algorithm the processes can use it to execute correctly (small) crucial steps.