Basic buiding blocks in Fault Tolerant distributed systems

Lecture 4

Prof. Cinzia Bernardeschi
Department of Information Engineering
Univerisity of Pisa, Italy
cinzia.bernardeschi@unipi.it

May 7-10, 2019 – Thessaloniki, Greece

Outline

- Fault models in distributed systems
- Atomic actions
- Consensus problem
- Conclusions

Fault models in distributed systems

Multiple isolated processing nodes that operate concurrently on shared informations

Information is exchanged between the processes from time to time

The goal is to design the system in such a way that the distributed application is fault tolerant

- A set of high level faults are identified

- Systems are designed that tolerate those faults

Fault models in distributed systems

Node failures

-Byzantine

-Crash

-Fail-stop

-...

Communication failures

-Byzantine

-Link (message loss, ordering loss)

-Loss (message loss)

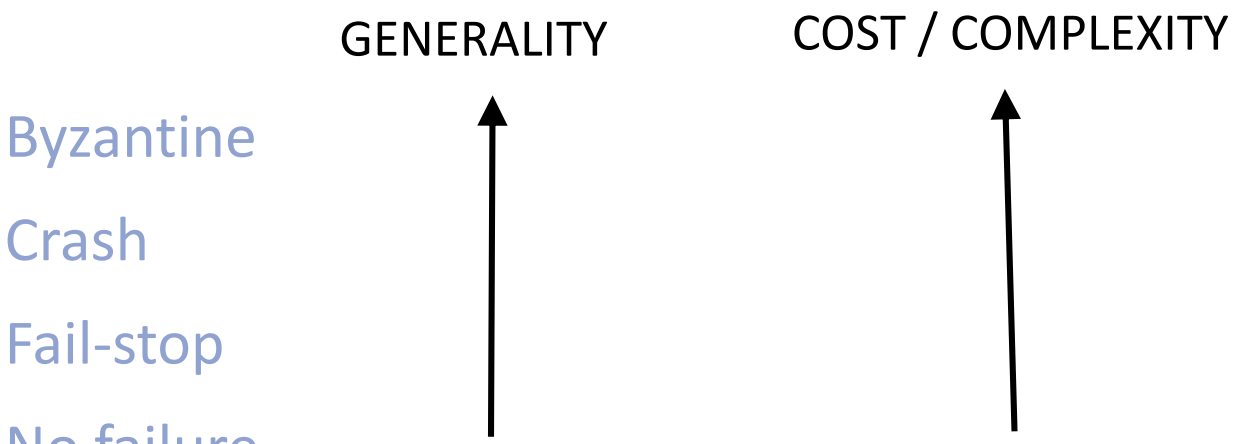
-...

Byzantine

- Processes :
  - can crash, disobey the protocol, send contradictory messages, collude with other malicious processes,...
- Network:
  - Can corrupt packets (due to accidental faults)
  - Modify, delete, and introduce messages in the network

Fault models in distributed systems

The more general the fault model, the more costly and complex the solution (for the same problem)



Arbitrary failure approach (Byzantine failure mode)

Architecting fault tolerant systems

We must consider the system model:

- Asynchronous
- Synchronous
- Partially synchronous
- …

Develop algorithms , protocolos that are useful building blocks for the architect of faut tolerant systems:

- Atomic actions
- Consensus
- Trusted components
- …….

Basic building blocks for fault tolerance

UNIVERSITÀ DI PISA

- Atomic actions

  action executed in full all or has no effect

- Consensus protocols

  correct replicas deliver the same result

- etc …

Atomic Actions

Atomic actions

Atomic action: an action that either is executed in full or has no effects at all

- Atomic actions in distributed systems:
  - an action is generally executed at more than one node
  - nodes must cooperate to guarantee that
  - either the execution of the action completes successfully at each node or the execution of the action has no effects

- The designer can associate fault tolerance mechanisms with the underlying atomic actions of the system:
  - limiting the extent of error propagation when faults occur and
  - localizing the subsequent error recovery

An example: Transactions in databases

UNIVERSITÀ DI PISA

- Transaction: a sequence of changes to data that move the data base from a consistent state to another consistent state.

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items

- Transactions must be atomic:

  all changes are executes successfully or data are not updated

Transactions in databases

Let T1 and T2 be transactions

Transaction T1

Transaction T2

1) A failure before the termination of the transaction, results into a rollback (abort) of the transaction

2) A failure after the termination with success (commit) of the transaction must have no consequences

Banking application

Account =(account_name, branch_name, balance)

t1: distributed transaction (access data at different sites)

t1: begin transaction

UPDATE account

SET balance=balance + 500

WHERE account_number=45;

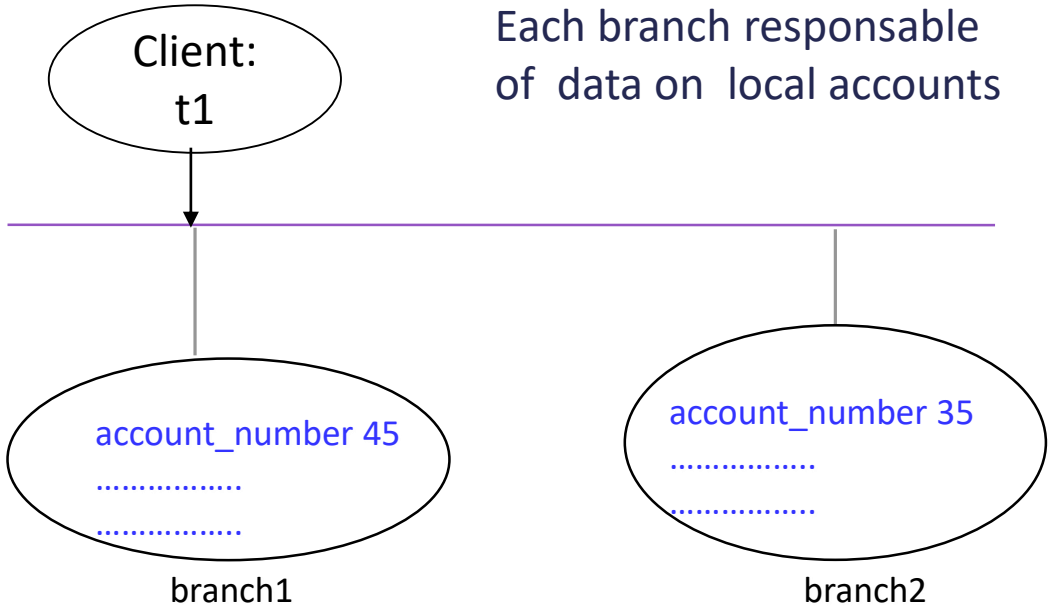
UPDATE account

SET balance=balance - 500

WHERE account_number=35;

commit

end transaction



Each branch responsable of data on local accounts

t1

t11: UPDATE account
SET balance=balance + 500
WHERE account_number=45;
site1

t12:UPDATE account
SET balance=balance - 500
WHERE account number=35;
site2

Atomicity requirement

- **Atomicity requirement**
  - if the transaction fails after the update of 45 and before the update of 35, money will be “lost” leading to an inconsistent database state
  - the system should ensure that updates of a partially executed transaction are not reflected in the database

A main issue: atomicity in case of **failures of various kinds, such as hardware failures and system crashes**

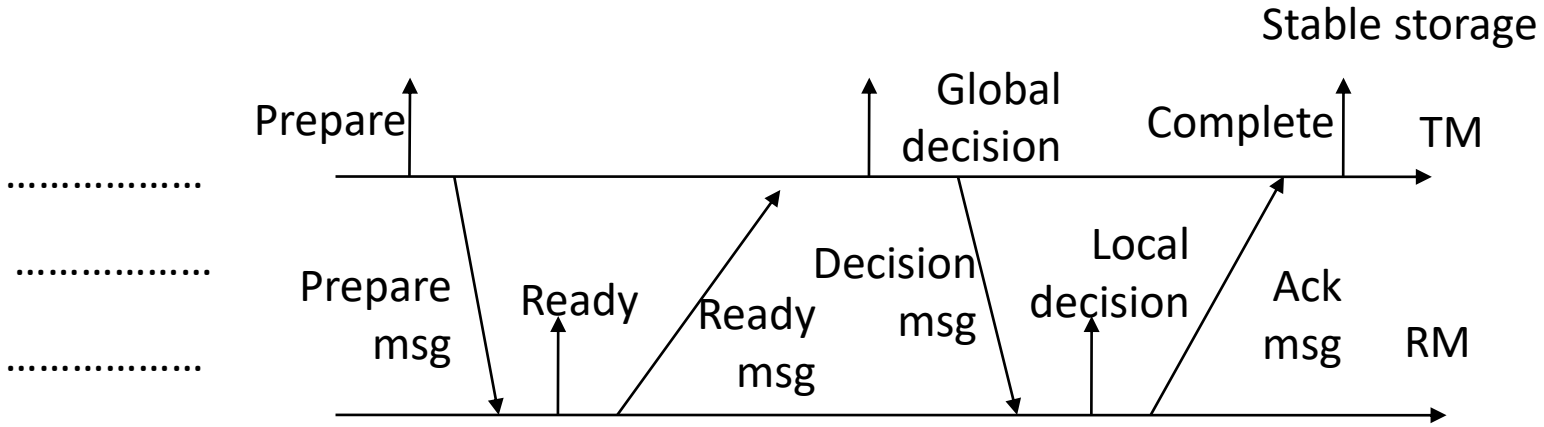
- Atomicity of a transaction:

  Commit protocol + Log in stable storage + Recovery algorithm

A programmer assumes atomicity of transactions

Two-phase commit protocol

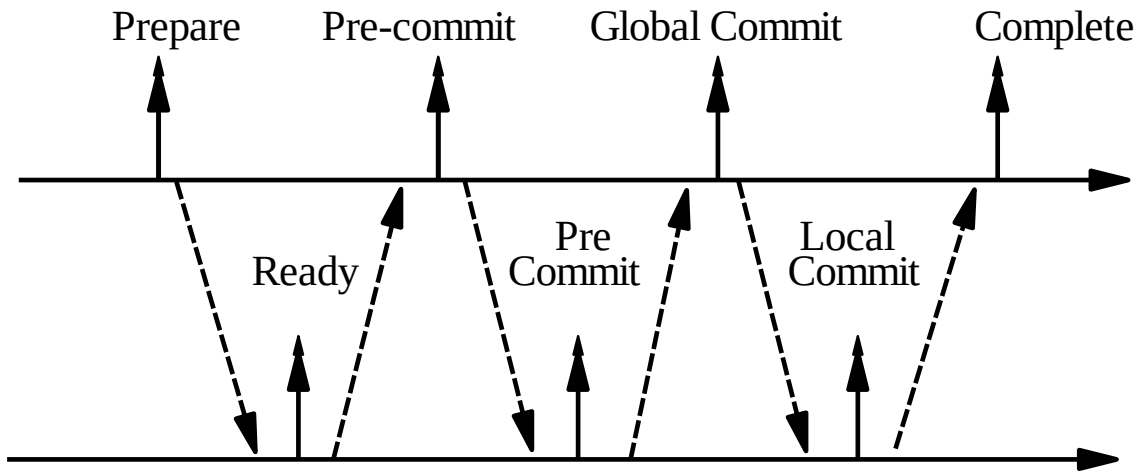
- One transaction manager TM
- Many resource managers RM
- Log file (persistent memory)
- Time-out



Tolerates: loss of messages
crash of nodes

Uncertain period:
if the transaction manager crash, a participant with Ready in its log cannot terminate the transaction

Three-phase commit



Precommit phase is added. Assume a permanent crash of the coordinator.
A participant can substitute the coordinator to terminate the transaction.

A participant assumes the role of coordinator and decides:

- Global Abort, if the last record in the log Ready
- Global Commit, if the last record in the log is Precommit

Recovery and Atomicity

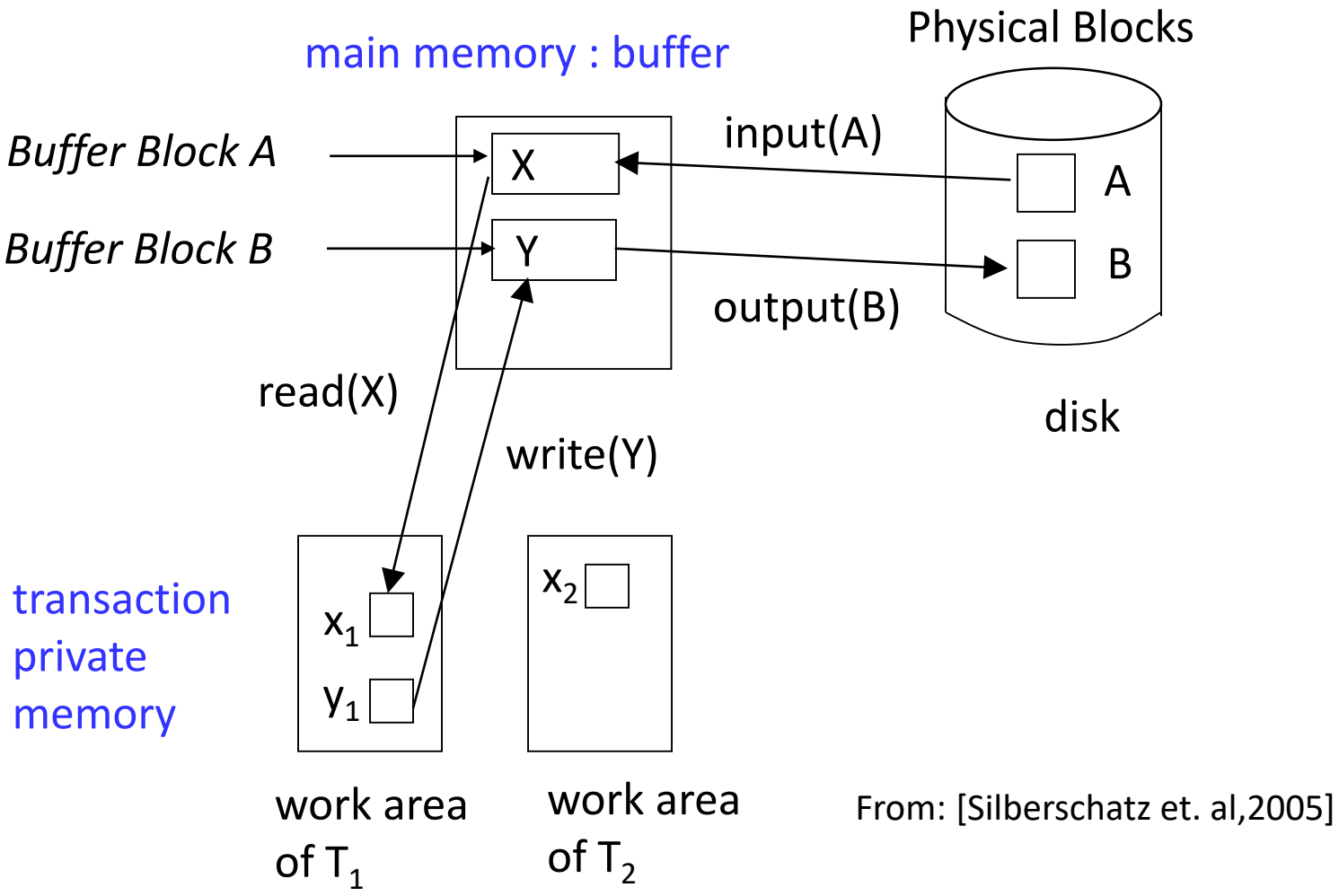
Physical blocks: blocks residing on the disk.
Buffer blocks: blocks residing temporarily in main memory

Block movements between disk and main memory through the following operations:
- **input**(*B*) transfers the physical block *B* to main memory.
- **output**(*B*) transfers the buffer block *B* to the disk

Transactions
- Each transaction *Ti* has its private work-area in which local copies of all data items accessed and updated by it are kept.
-perform **read**(*X*) while accessing *X* for the first time;
-executes **write**(*X*) after last access of X.

System can perform the **output** operation when it deems fit.
Let B_X denote block containing *X*.
output(B_X) need not immediately follow **write**(*X*)

Data Access



From: [Silberschatz et. al,2005]

Recovery and Atomicity

UNIVERSITÀ DI PISA

- Several output operations may be required for a transaction
- A transaction can be aborted after one of these modifications have been made permanent (transfer of block to disk)
- A transaction can be committed and a failure of the system can occur before all the modifications of the transaction are made permanent
- To ensure atomicity despite failures, we first output information describing the modifications to a Log file in stable storage without modifying the database itself

  Log-based recovery

DB Modification: an example

Log	Write	Output
<T_0 **start**>		
<T_0 , A, 1000, 950>		
	A = 950	
<T_0 , B, 2000, 2050>		
	B = 2050	
		Output(B_B)
<T_1 **start**>		
<T_0 **commit**>		
<T_1, C, 700, 600>		
	C = 600	
		Output(B_C)
CRASH		

Recovery actions

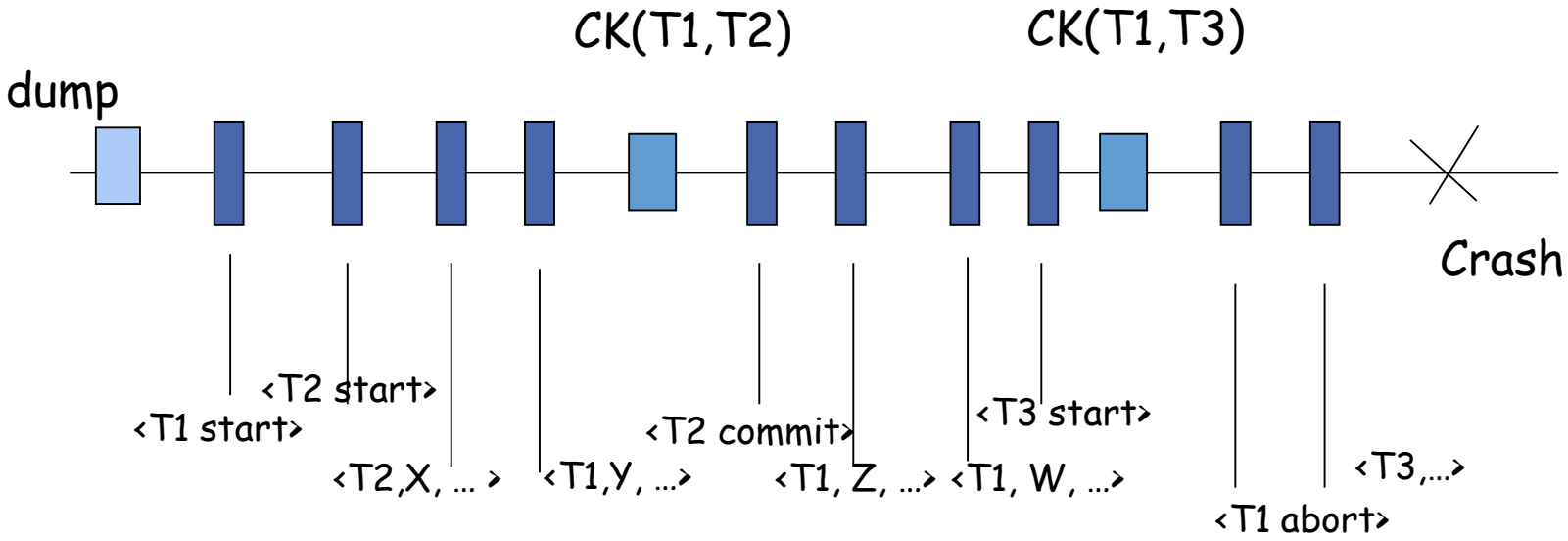
- undo (T_1) A reset to 950
  B reset to 2050
- redo (T_0) C is restored to 700

Checkpointing

CHECKPOINT operation: **output all modified buffer blocks to the disk**

To Recover from system failure:
- consult the Log
- redo all transactions in the checkpoint or started after the checkpoint that committed;
- undo all transaction in the checkpoint not committed or started after the checkpoint

To recover from disk failure:
- restore database from most recent dump
- apply the Log Recovery



Atomic actions

Advantages of atomic actions:

a designer can reason about system design as

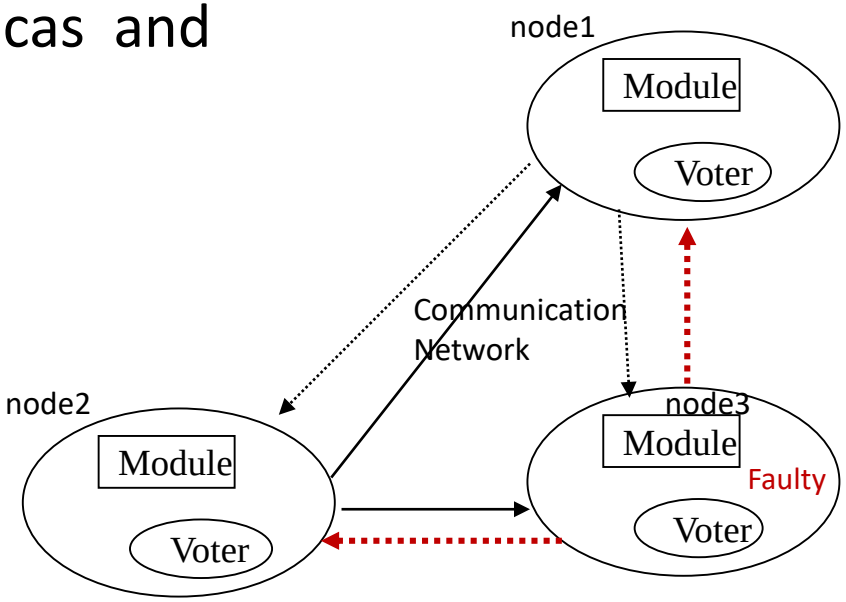
1) no failure happened in the middle of a atomic action

2) separate atomic actions access to consistent data (property called “serializability”, concurrency control).

Consensus protocols

Consensus problem

One way to achieve reliability is to have multiple replicas and take the majority voting among them



In order for the majority voting to yield a reliable system, the following two conditions should be satisfied:

- all non faulty components must use the same input value
- if the sender is non-faulty, then all non-faulty components use the value it provides as input

What happen with Byzantyne failures?

The faulty replica can send different values to the other replicas.

The inputs to the voter can be different

Consensus problem

The Consensus problem can be stated informally as:

> how to make a set of distributed processors achieve agreement on a value sent by one processor despite a number of failures

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

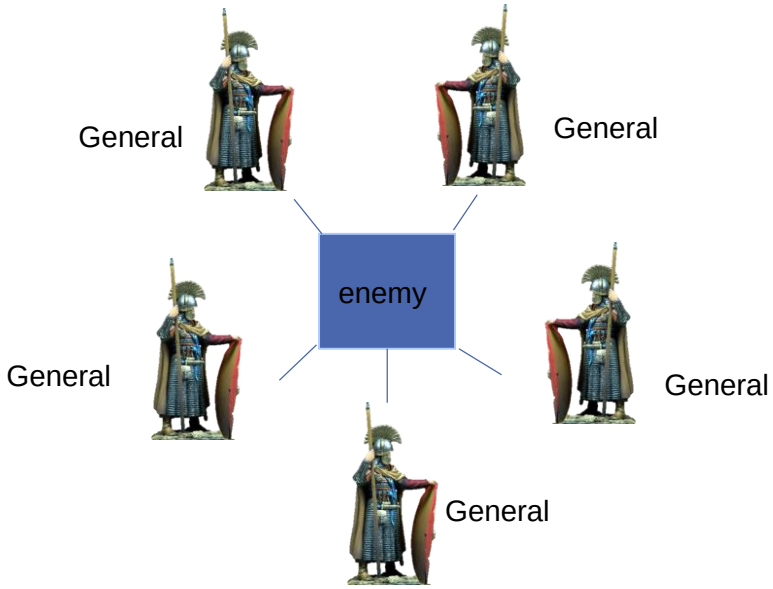
The problem is given in terms of generals who have surrounded the enemy.

Generals wish to organize a plan of action to attack or to retreat. They must take the same decision.

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.

Byzantine Generals Problem



General: either a loyal general or a traitor

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

Byzantine Generals Problem

Assume

- 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
- each general final decision obtained by:
  majority vote among the values *v(1), ..., v(n)*

Absence of traitors:
generals have the same values v(1), ..., v(n) and they take the same decision

Byzantine Generals Problem

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

In presence of traitors:

to satisfy condition A
every general must apply the majority function to the same values v(1),...,v(n)
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)

Interactive Consistency

Simpler situation:

1	Commanding general (C)
n-1	lieutenant generals (L1, ..., Ln-1)

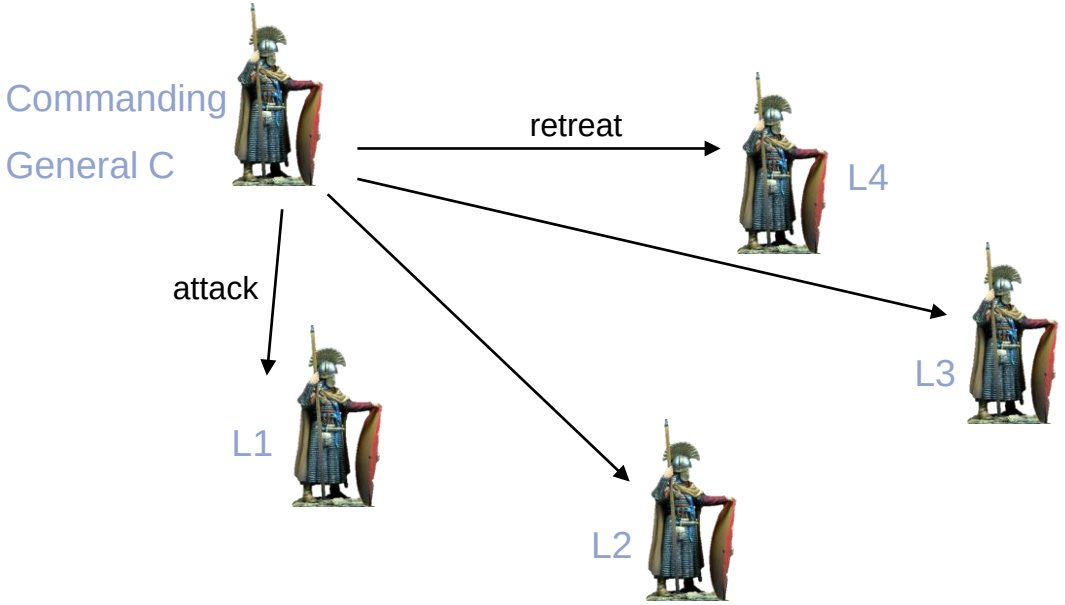
The Byzantine commanding general C wishes to organize a plan of action to attack or to retreat; he sends the command to every lieutenant general Li

Interactive Consistency

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

Byzantine Generals Problem



Commanding general is loyal: IC1 and IC2 are satisfied

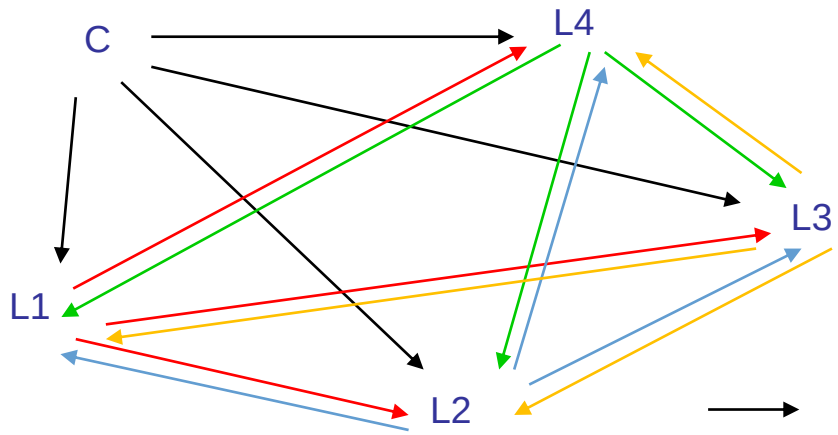
Commanding general lies but sends the same command to lieutenants:
IC1 and IC2 are satisfied

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 ?

Byzantine Generals Problem

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



L1= (v1, v2, v3, v4)
L2= (v1, v2, v3, v4)
L3= (v1, v2, v3, v4)
L4= (v1, v2, v3, v4)

decision sent by C

what L1 says he received by C

what L2 says he received by C

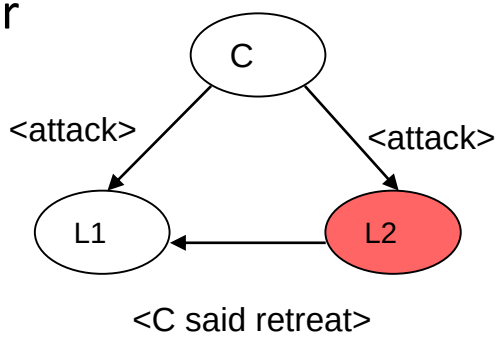
what L3 says he received by C

what L4 says he received by C

3 Generals: one lieutenant traitor

n = 3
no solution exists

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.

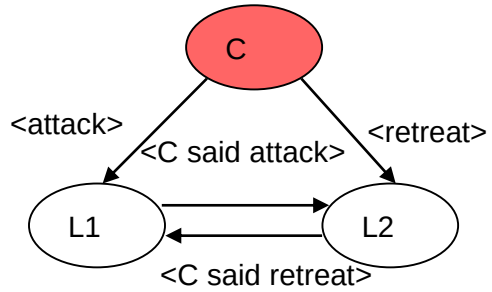
If L1 decides attack, IC1 and IC2 are satisfied.

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

RULE: if Li receives different messages, L1 takes the decision he received by the commander

3 Generals: Commander traitor

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

Oral Message (OM) algorithm

Assumptions

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

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”.

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.

---

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

The algorithm

Algorithm OM(0)

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

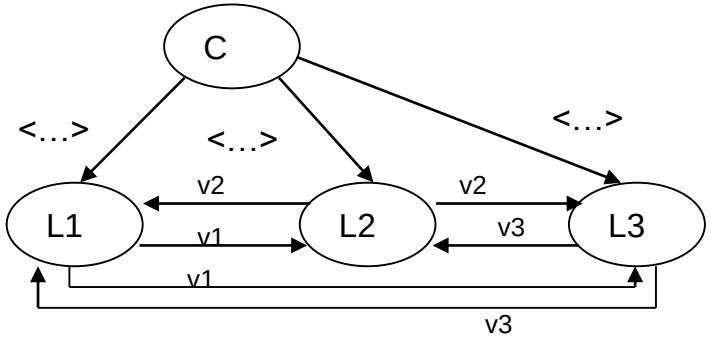
The algorithm OM(1)

4 generals, 1 traitor

Point 1

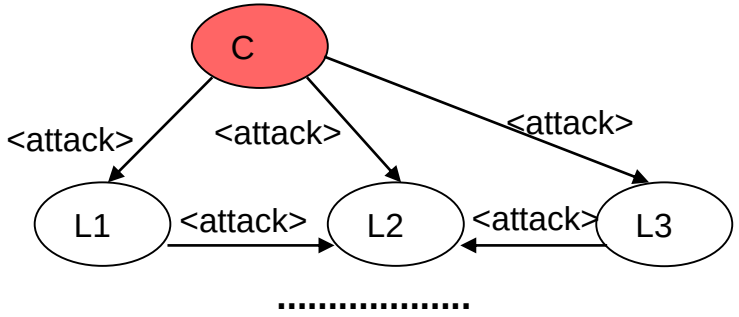
- C sends the command to L1, L2, L3.
- L1 applies OM(0) and sends the command he received from C to L2 and L3
- L2 applies OM(0) and sends the command he received from C to L1and L3
- L3 applies OM(0) and sends the command he received from C to L1 and L2

- Point 2
- L1: majority(v1, v2, v3)
- L2: majority(v1, v2, v3)
- //v1 command L1 says he received
- //v3 command L3 says he received
- L3: majority(v1, v2, v3)



4 Generals: Commander traitor

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



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

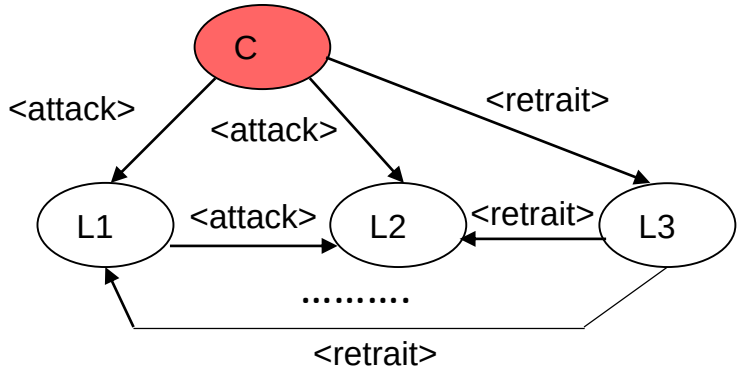
L1, L2 and L3 are loyal. They send the same command when applying OM(0)
IC1 and IC2 are satisfied

4 Generals: Commander traitor

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

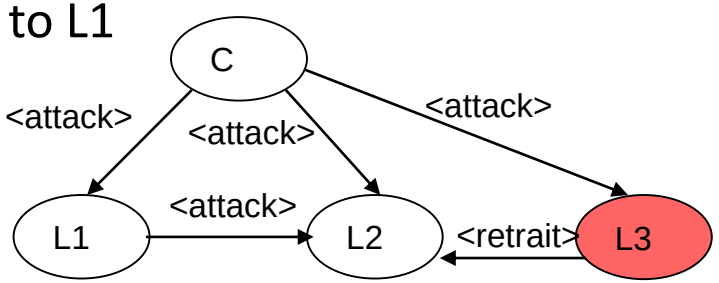
L2: v1 = attack, v2 =attack, v3 = retrait majority(...)= attack

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

IC1 and IC2 satisfied

4 Generals: one Lieutenant traitor

- 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

Oral message (OM) 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;

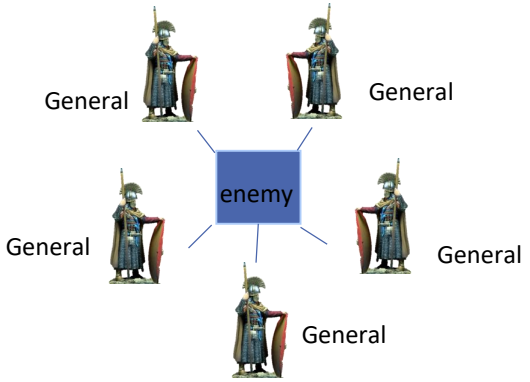
7 generals are needed to cope with 2 traitors;

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

General agreement among n processors, m of which could be faulty and behave in arbirary manners.

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).

Byzantine Generals Problem

Solutions of the Consensus problem are expensive

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.

Byzantine Generals Problem

- 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 (authenticated messages)

Byzantine agreement becomes much simpler

A message is authenticated if:

1. a message signed by a fault-free processor cannot be forged
2. any corruption of the message is detectable
3. the signature can be authenticated by any processors

Byzantine Generals Problem

Assumptions:

(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

Signed messages

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

choice(V) = retrait if V consists of more than 1 element

General 0 is the commander
For each i, Vi contains the *set of properly signed orders* that lieutenant Li has received so far

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

Signed messages SM(m) algorithm

Algorithm SM(m)

Vi = ∅

1. C signs and sends its value to every Li, i∈{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 ∉ Vi

then Vi = Vi ∪ {v};

if k < m then

sends v:0:j1:...:jk:i to every other Lj , j ∉{j1, ..., jk}

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

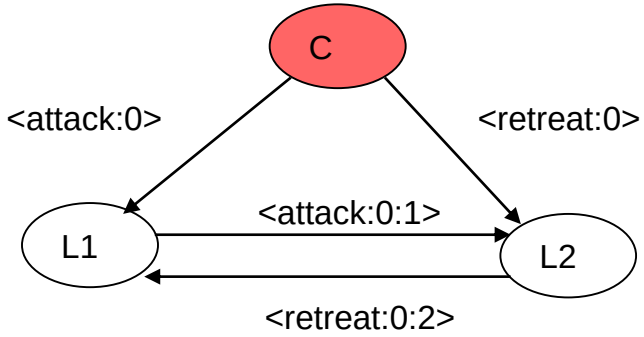
Observations:

- Li ignores msgs containing an order v∈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



- 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

Assumption A1.

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

Assumption A2.

The receiver of a message knows who sent it

Assumption A3:

The absence of a message can be detected

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

Impossibility result

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 et al. 1985]

difficulty of determining whether a process has actually crashed or is only very slow

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

Circumventing the problem: Adding Time to the Model (using the notion of partial synchrony), Randomized Byzantine consensus, Failure detectors, etc …

SIFT case study

SIFT (Software Implemented Fault Tolerance) is a Fault-Tolerant Computer for Aircraft Control

"a system capable of carrying out the calculations required for the control of an advanced commercial transport aircraft"

developed for NASA as an experimental case study for fault tolerant system research

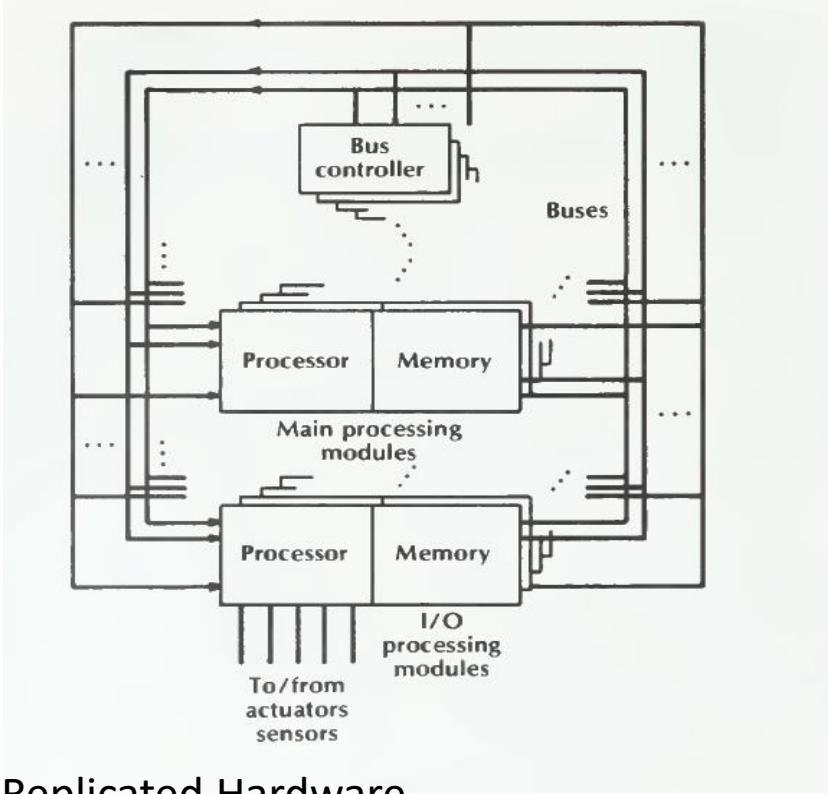
Reliability requirement:
probability of failure less than $\mathbf{10^{-9}}$ per hour in a flight of ten hours' duration.

The SIFT system executes a set of tasks, each of which consists of a **sequence of iterations**.

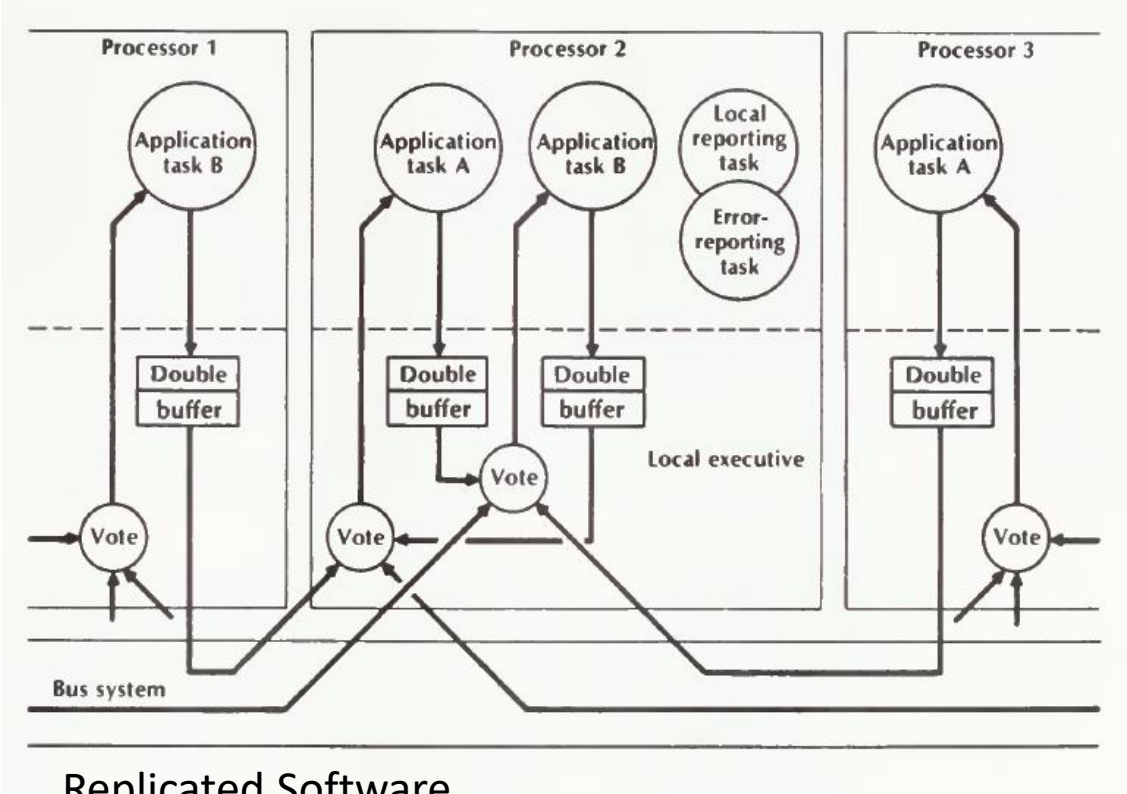
The **input data to each iteration of a task are the output data produced by the previous iteration of some collection of tasks** (which may include the task itself).

SIFT

Reliability is achieved by replication + voting



Replicated Hardware
A processor write only its private memory.



Replicated Software
each iteration of a task independently executed by a number of modules

Loose synchronization

- voting is executed only at the beginning of each iteration

  due to the iterative nature of the tasks

- processors need be only loosely synchronized

  guarantee that different processors allocated to a task are executing the same iteration, do not need tight synchronization to the instruction or clock level.

median clock algorithm

the traditional clock synchronization algorithm for reliable systems

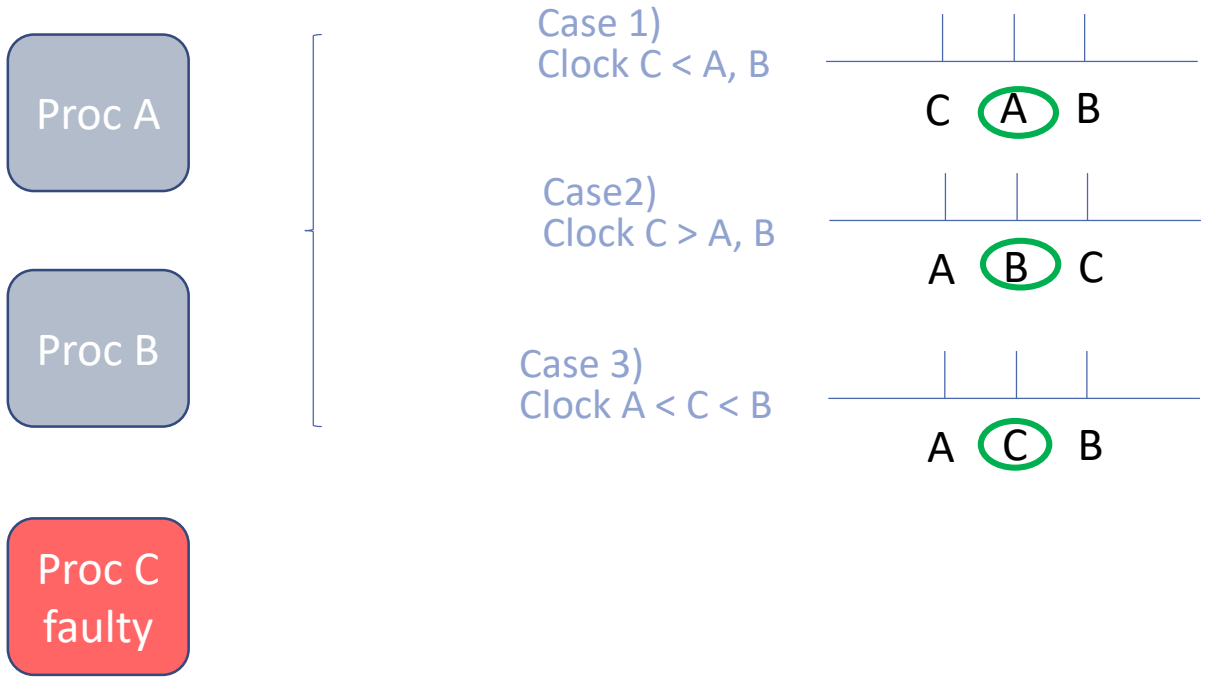
each clock observes every other clock and sets itself to the median of the values that it sees

Clock synchronization

Assumption:

in the presence of only a single fault, either the median value must be (i) the value of one of the valid clocks or else (ii) it must lie between a pair of valid clock values.

Let Clock A < Clock B.



The weakness of this algorithm is the Byzantine fault, that may cause other processors to observe different values for the failing clock

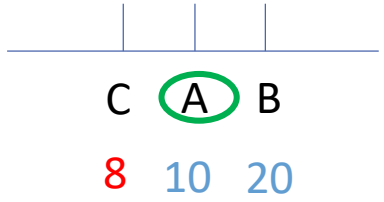
Clock synchronization

Let clock A < clock B. A: B: 20 C -> Clock A=
A:10 B: C:

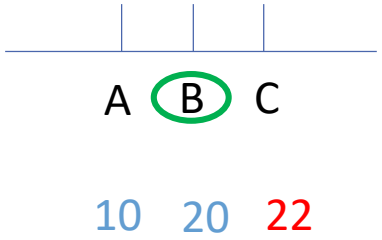
Assume failure mode of clock C is such that

- proc A sees a value for clock C that is slightly earlier than its own value, while
- proc B sees a value for clock C that is slightly later than its own value (Byzantine faults).

Proc A



Proc B



Proc C faulty

Assumption is violated

Processorss A and B will both see their own value as the median value, and therefore not change it.

To synchronise clocks SIFT applies a Consensus algorithm (5 processors)

Byzantyne fault tolerance

Many application fields:

- Airbone self-separation (Future generation of ATC)
  An operating environment where pilots are allowed to select their flight paths in real-time
  Byzantine Fault Tolerance algorithms for coordination between aircrafts to take local decisions
- Block-chains
  Byzantine Fault Tolerance algorithms for Block-chain
- etc …

Conclusions

In real world, reliability problems are really subtle

there is a cause that evolves. It propagates into the system, something happens in a subsystem, something else happens in another subsystem, …., and then we have a failure

- From Reliability to Resilience

  unforseen environmental changes and new type of threats

- Resilience

  the persistence of service delivery that can be justifiably be trusted when facing changes

- **Resilience engineering**

  how to design, implement operate, etc … comple systems so that they can be resilient

Other references

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

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