Basic building blocks for fault-tolerance in distributed systems

Fault tolerant distributed systems

Multiple isolated processing nodes that operate concurrently on shared informations

Information is exchanged between the processes from time to time

Algorithm construction:
the goal is to design the software in such a way that the distributed application is fault tolerant

- A set of high level faults are identified

- Algorithms 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

Architecting fault tolerant systems

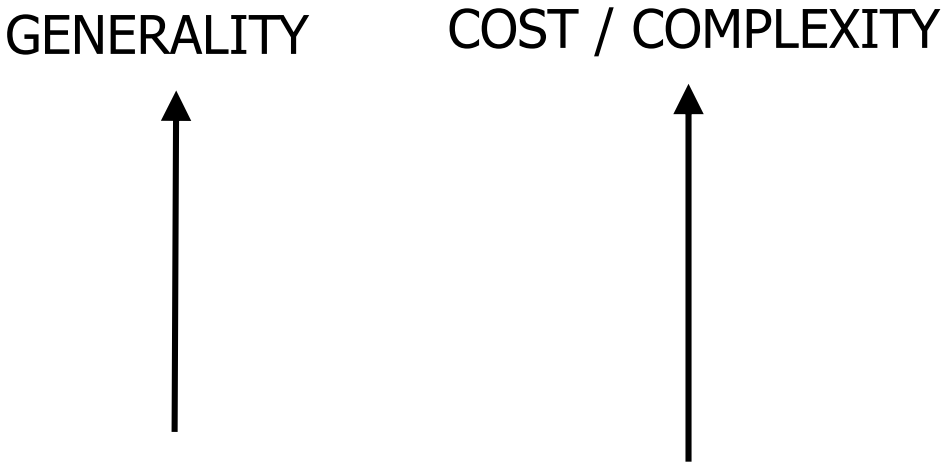
The more general the fault model, the more costly and complex the solution (for the same problem)

Byzantine

Crash

Fail-stop

No failure



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:

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

Basic building blocks for fault-tolerance

- Atomic actions

  action executed in full all or has no effect

- Consensus protocols

  correct replicas deliver the same result

- Reliable broadcast

  reliability of messages exchanged within a group of processes

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

J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A.F. Zorzo,E. Canver, F. von Henke. Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions. In FTCS-29, Madison, USA, pp. 68-75, 1999.

An example: Transactions in databases

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

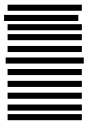
A **transaction** is a *unit* of program execution that accesses and possibly updates various data items

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

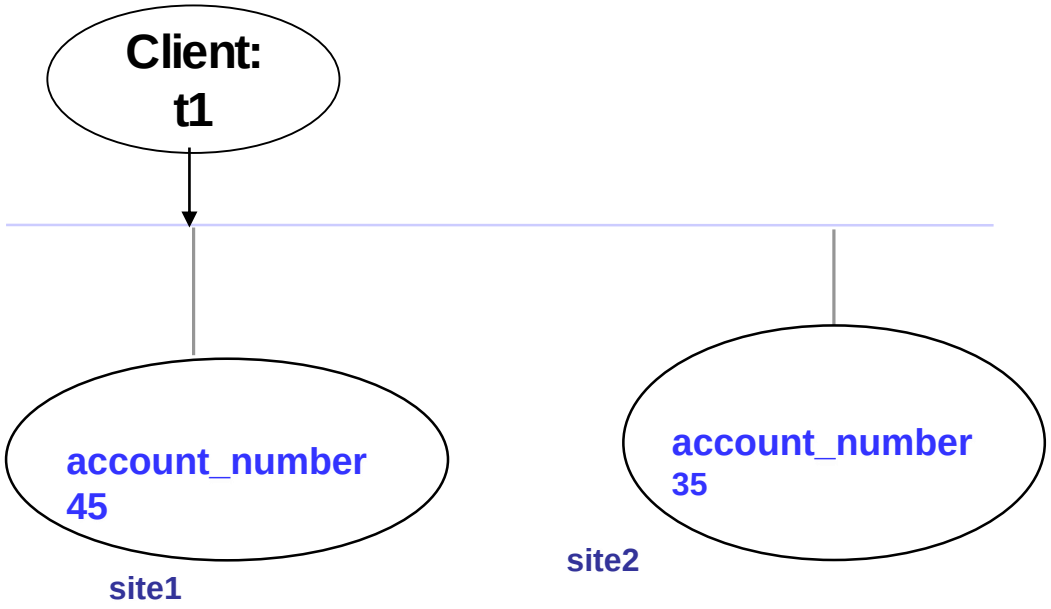
1) A failure before the termination of the transaction, results into a rollback (abort) of the transaction

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

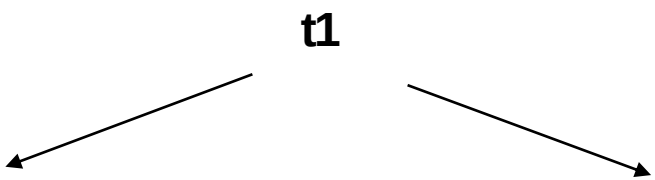
SET balance=balance - 500

WHERE account_number=35;

commit

end transaction

each branch responsable
of data on its accounts



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

- 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**

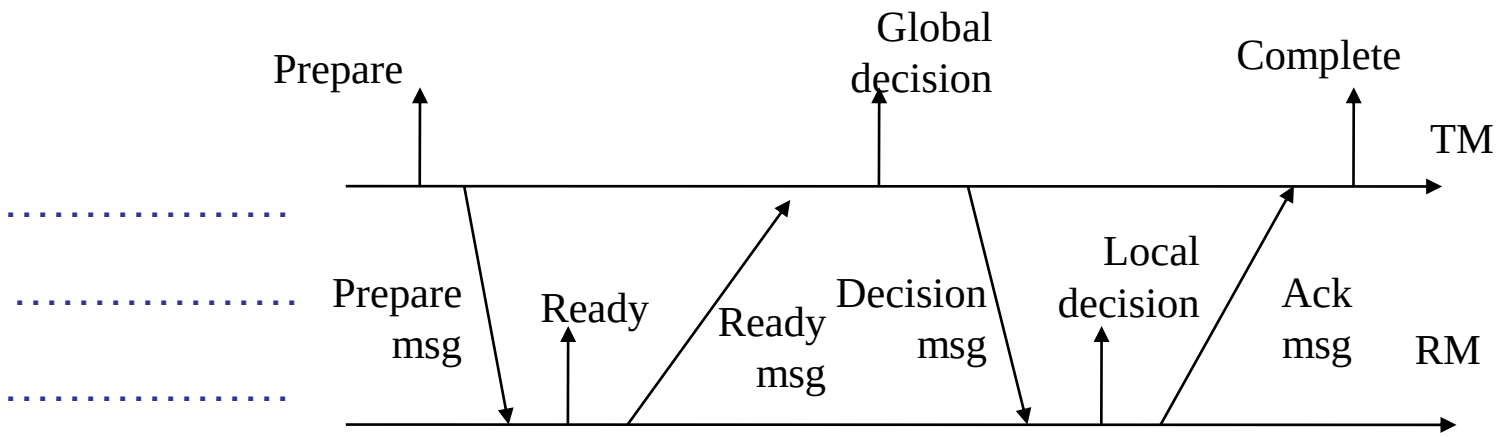
Atomicity of a transaction:

Commit protocol + Log in stable storage + Recovery algorithm

A programmer assumes atomicity of transactions

Two-phase commit protocol

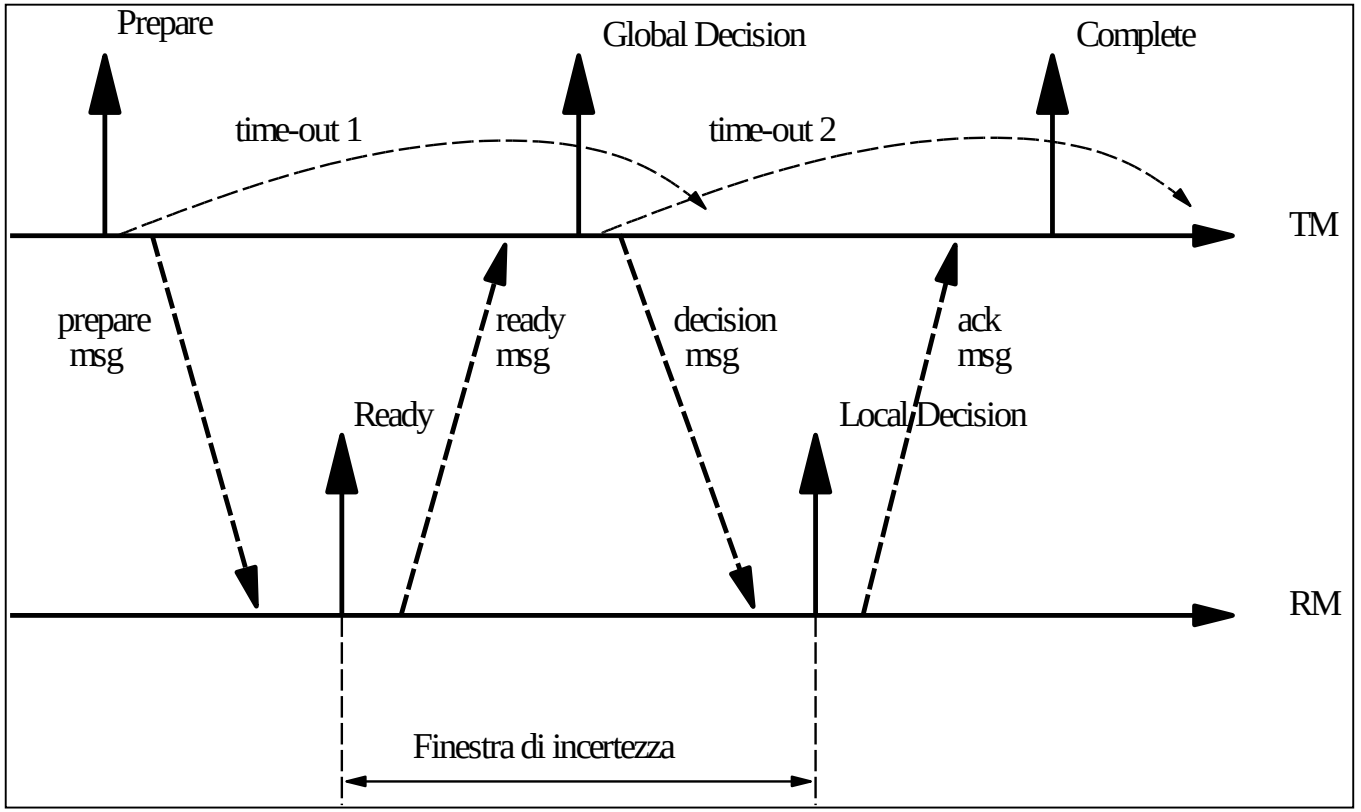
- One transaction manager TM
- Many resource managers RM
- Log file (persistent memory)
- time-out



Da: Atzeni, Ceri, Paraboschi, Torlone - Basi di Dati: Architetture e linee di evoluzione

Tolerates: loss of messages
crash of nodes

Timeout and uncertain period



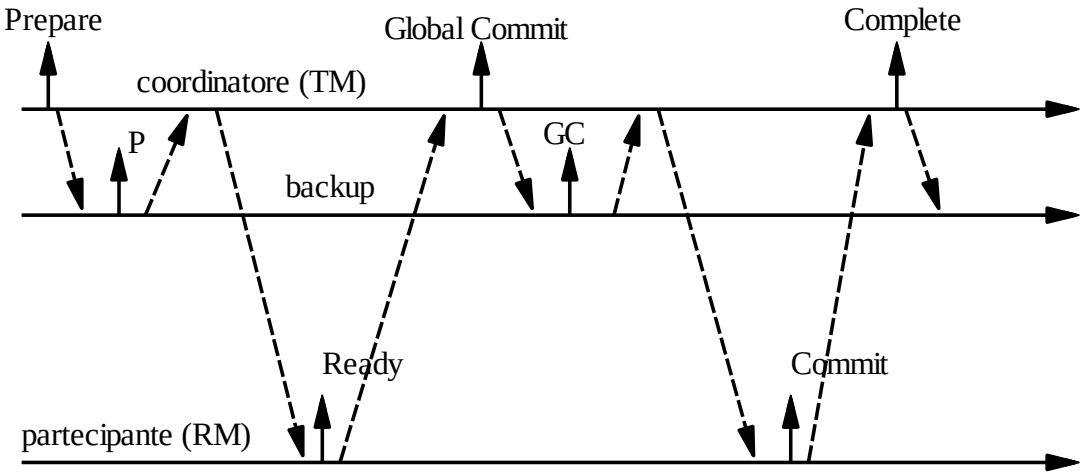
Da: Atzeni, Ceri, Paraboschi, Torlone - Basi di Dati: Architetture e linee di evoluzione

Uncertain period:

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

Four-phase commit

Da: Atzeni, Ceri, Paraboschi, Torlone - Basi di Dati: Architetture e linee di evoluzione



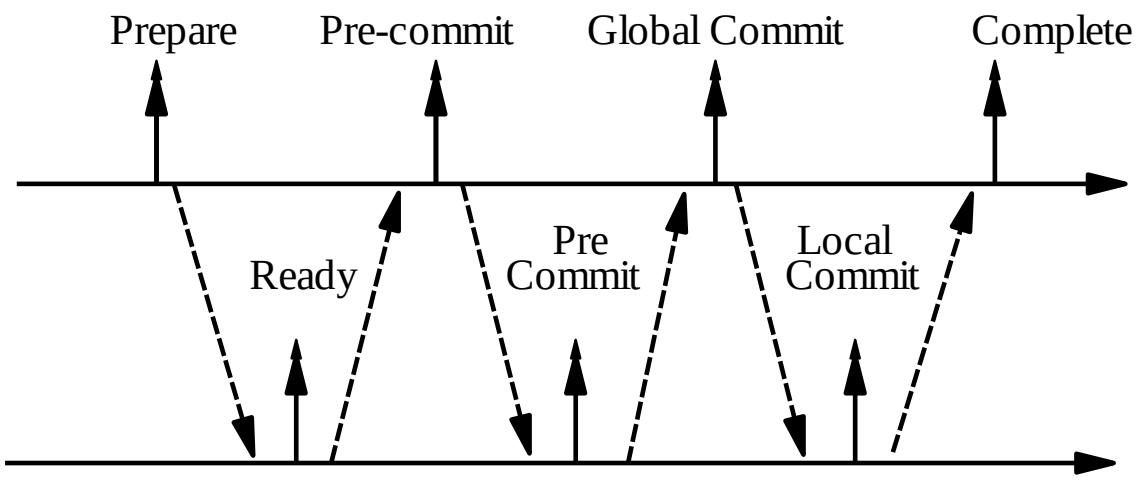
Coordinator backup is created at a different site
the backup maintains enough information to assume the role of coordinator if the actual coordinator crashes and does not recover.

The coordinator informs the backup of the actions taken.

If the coordinator crashes, the backup assume the role of coordinator:
1) Another backup is started.
2) The two-phase commit protocol is completed.

Three-phase commit

Da: Atzeni, Ceri, Paraboschi, Torlone - Basi di Dati: Architetture e linee di evoluzione



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*)

Recovery and Atomicity

Several output operations may be required for a transaction

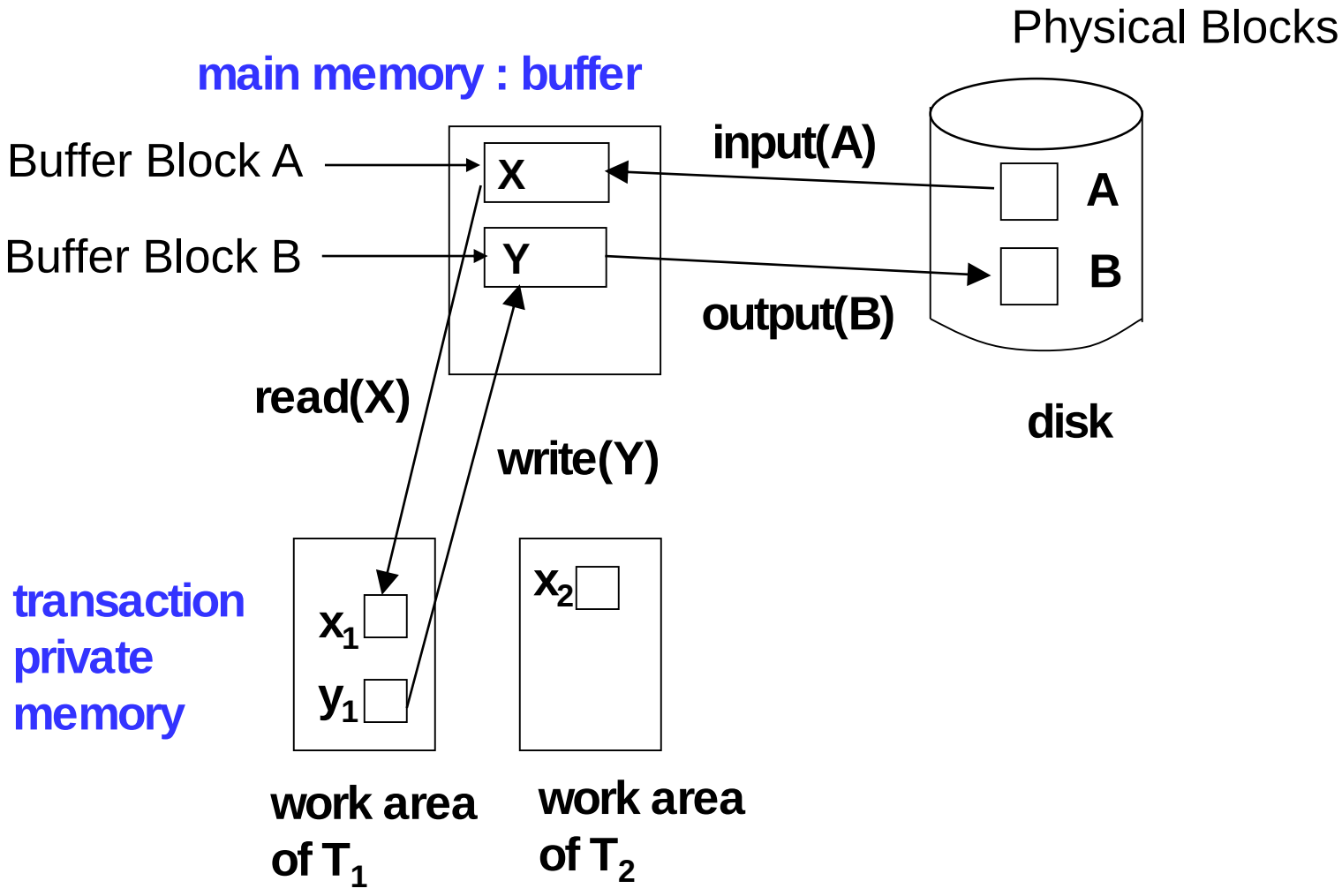
A transaction can be aborted after one of these modifications have been made permanent (transfer of block to disk)

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

Example of Data Access



From: Database System Concepts, 5th Ed., McGraw-Hill, by Silberschatz, Korth and Sudarshan

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_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		Output(B_C)
CRASH		

An Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then *A* and *B* are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

LOG file:

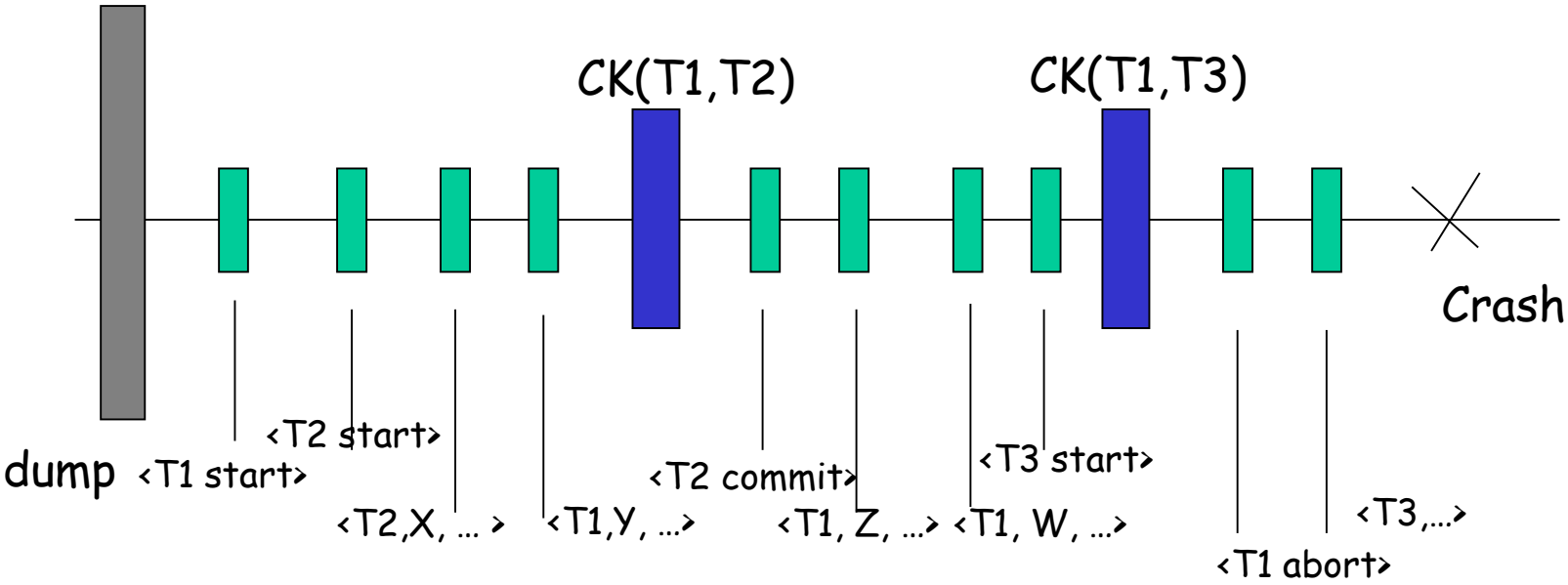
CHECKPOINT operation: **output all modified buffer blocks to the disk**

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



Example: concurrent transactions

Go over the steps of the recovery algorithm on the following log:

<T_0 **start**>
<T_0, A, 0, 10>
<T_0 **commit**>
<T_1 **start**> /* Scan at step 1 comes up to here */
<T_1, B, 0, 10>
<T_2 **start**>
<T_2, C, 0, 10>
<T_2, C, 10, 20>
<checkpoint {T_1, T_2}>
<T_3 **start**>
<T_3, A, 10, 20>
<T_3, D, 0, 10>
<T_3 **commit**>
crash

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

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.

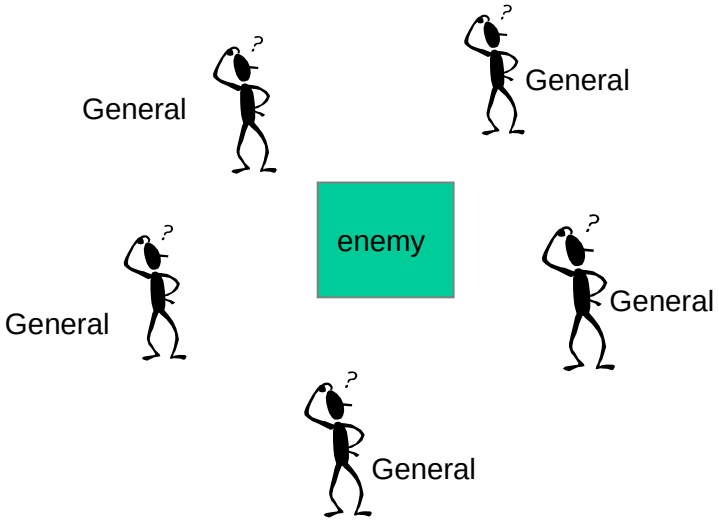
Generals wish to organize a plan of action to attack or to retreat.
Each general observes the enemy and communicates his observations to the others.

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.

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

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

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

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

- 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

In presence of traitors:

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 to every lieutenant general Li.

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

Consensus:

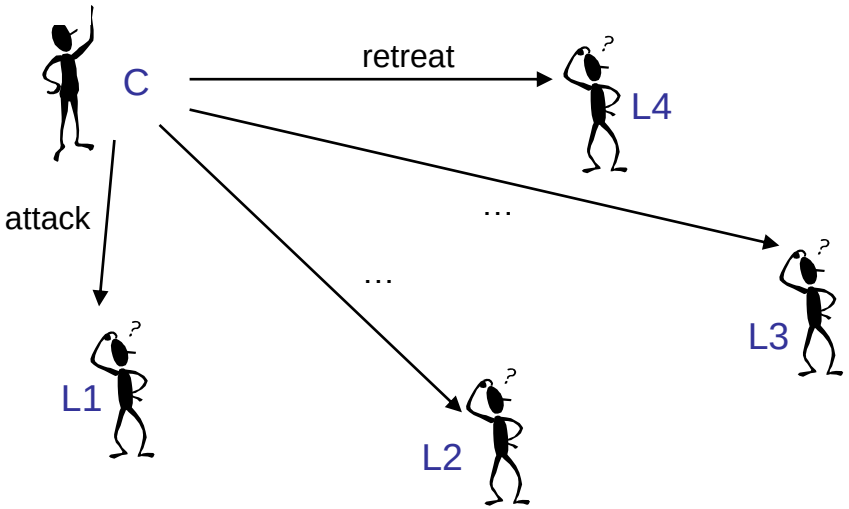
IC1: All loyal lieutenant generals obey the same command

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 C is loyal then IC1 follows from IC2

Byzantine Generals Problem



…

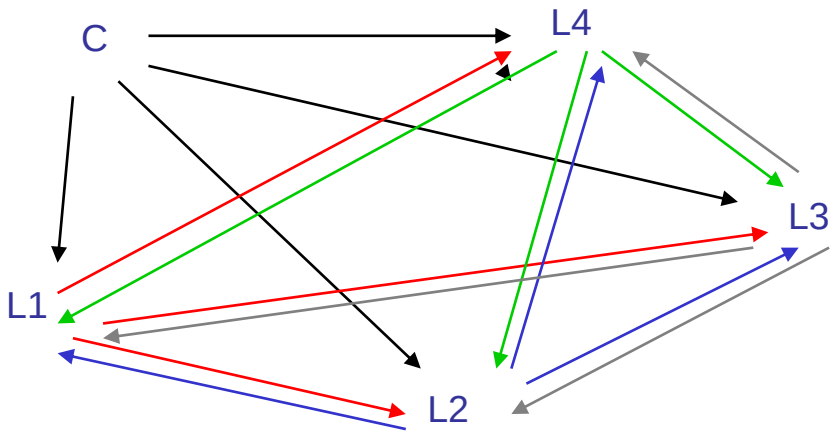
If the commanding general is loyal, IC1 and IC2 are satisfied.
If the commanding general lies but sends the same command to lieutenants, IC1 and IC2 are satisfied.

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 ?

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

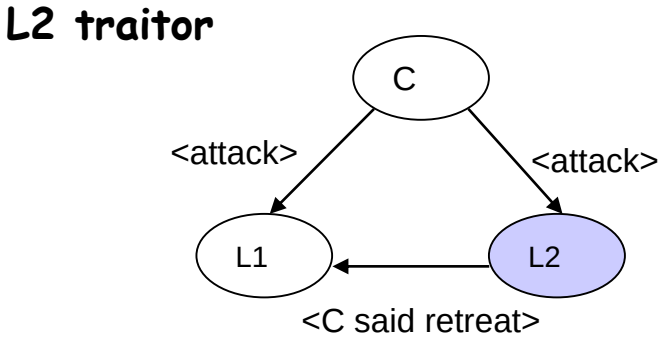
what L3 says he received by C

what L4 says he received by C

Byzantine Generals Problem

n = 3

no solution exists in presence of a 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.

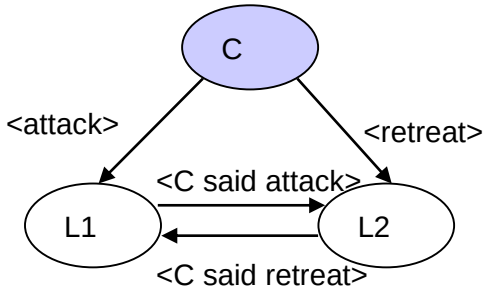
If L1 decides attack, IC1 and IC2 are satisfied.

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

Byzantine Generals Problem

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

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.

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 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

4 generals, 1 traitor OM(1)

Point 1

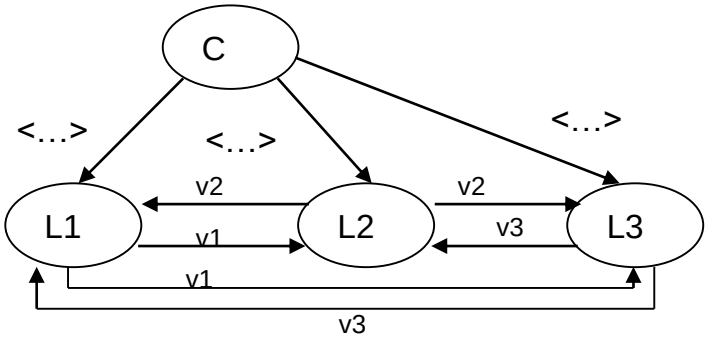
- C sends the command to L1, L2, L3.
- L1 applies OM(0) and sends the command he received from C to L2 and L3
- L2 applies OM(0) and sends the command he received from C to L1and L3
- L3 applies OM(0) and sends the command he received from C to L1 and L2

Point 2

- L1: majority(v1, v2, v3)
- L2: majority(v1, v2, v3)

  //v1 command L1 says he received

  //v3 command L3 says he received
- L3: majority(v1, v2, v3)

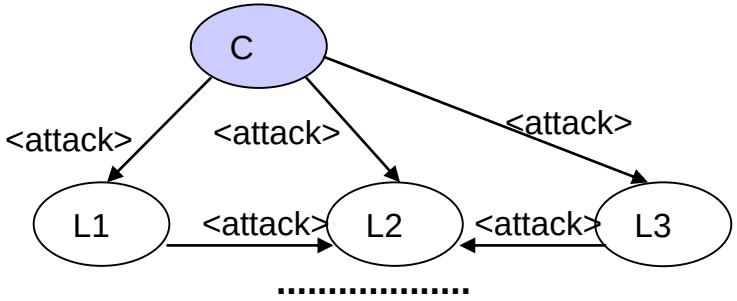


The algorithm

4 generals, 1 traitor

n=4, m=1

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



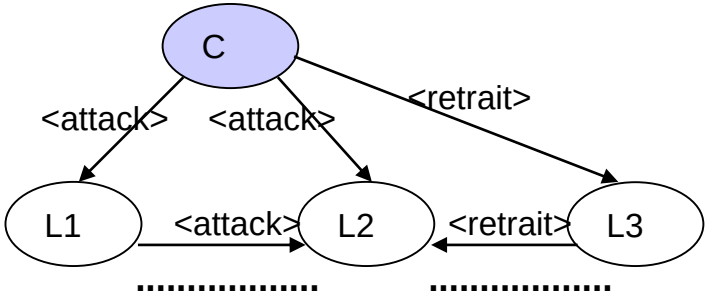
Li: v1 = attack, v2 =attack, v3 = attack
majority(....)= attack

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

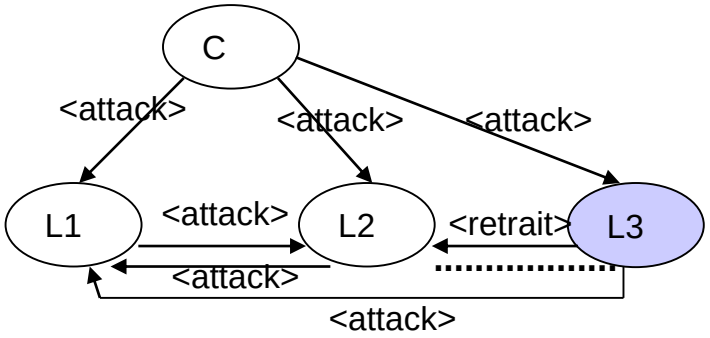
IC1 and IC2 satisfied

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.

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 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

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

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

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)

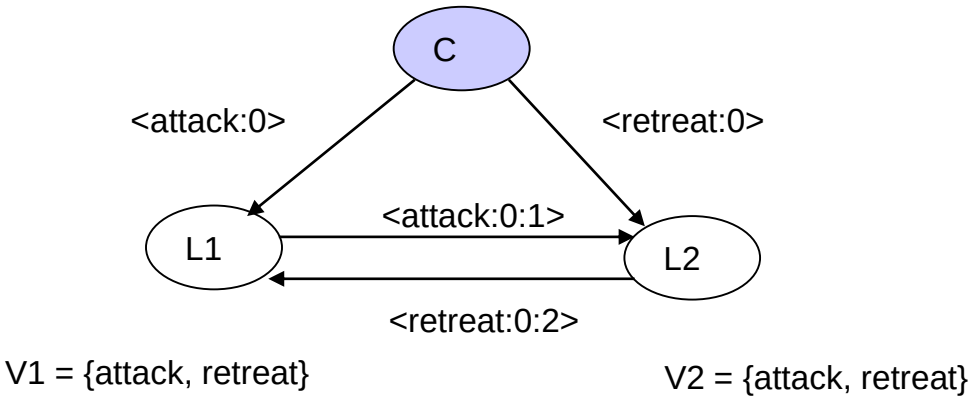
Observations:

- 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



- 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

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.

Circumventing FLP: 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.

Circumventing FLP: 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 algorithms 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.”

Circumventing FLP: 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

Circumventing FLP: 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.

Circumventing FLP: 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.