Fault tolerant distributed computing

Architecting 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

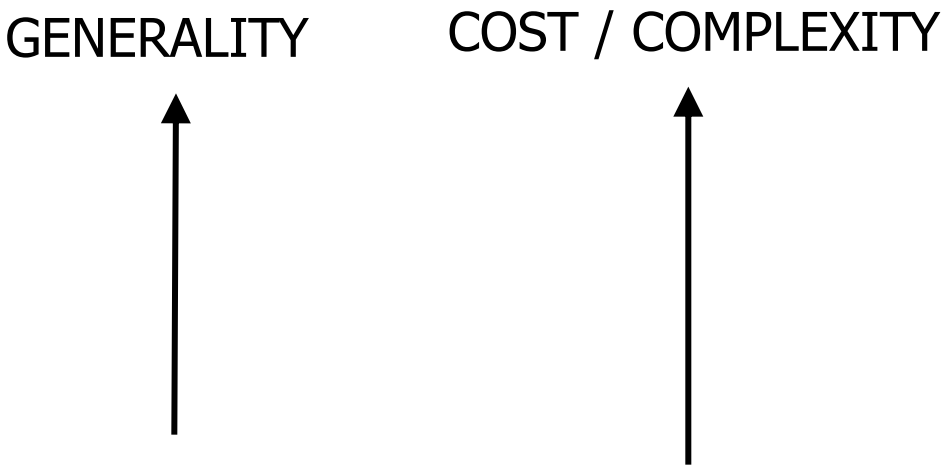
The more general the fault model, the more costly and complex the solution (for the same problem)

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
- …….

Atomic Actions

Atomic actions

An action that either is executed in full or has no effects at all is called "Atomic action"

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

Atomic actions are a basic building block in fault tolerant computing

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.

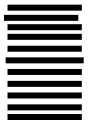
A **transaction** is a *unit* of program execution that accesses and possibly updates various data items

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

Transactions in databases

Concept of transaction and ACID properties of transactions:

A- Atomicity property

all or nothing property (with respect to failures)

C- Consistency

each transaction preserves required invariants over the data

I- Isolation (concurrency atomicity)

concurrent transaction have the same effect as though they were sequential

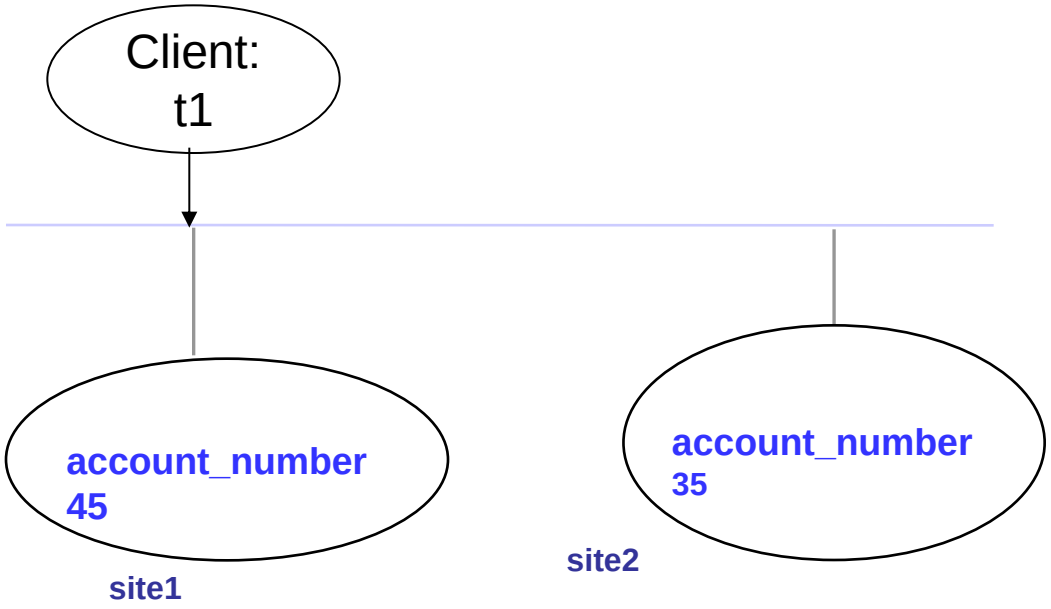
D- Durability(or permanence)

once a transaction is committed, failures cannot destroy its effects

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

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

- 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**

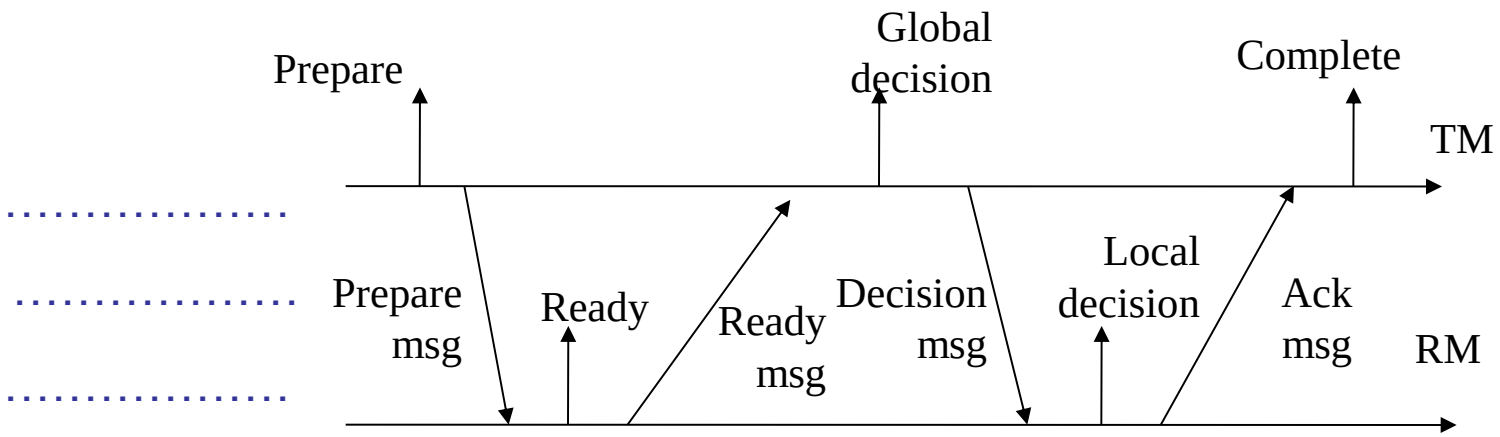
Atomicity of a transaction:

Commit protocol + Log in stable storage + Recovery algorithm

A programmer assumes atomicity of transactions

Two-phase commit protocol

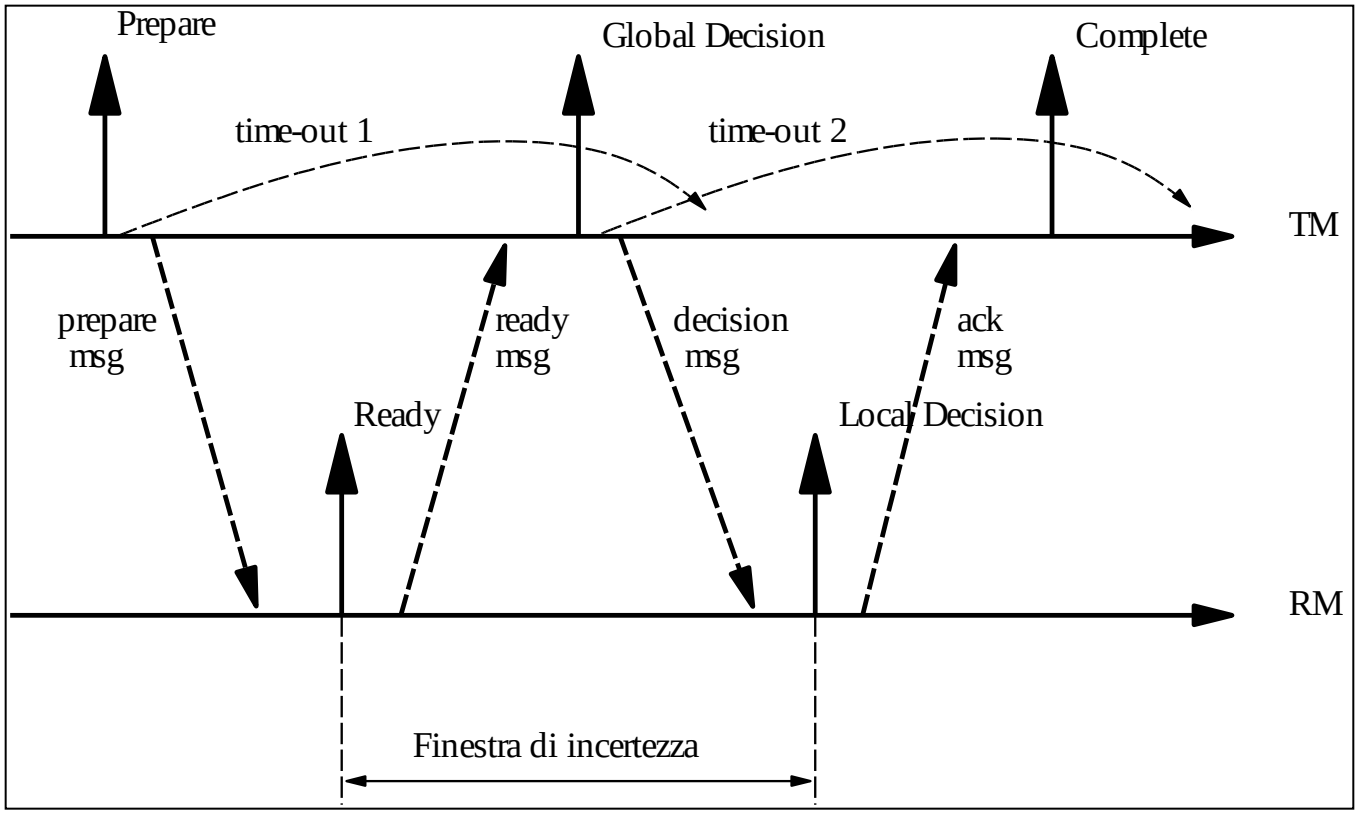
- One transaction manager TM
- Many resource managers RM
- Log file (persistent memory)
- time-out



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

Tolerates: loss of messages
crash of nodes

Timeout and uncertain period



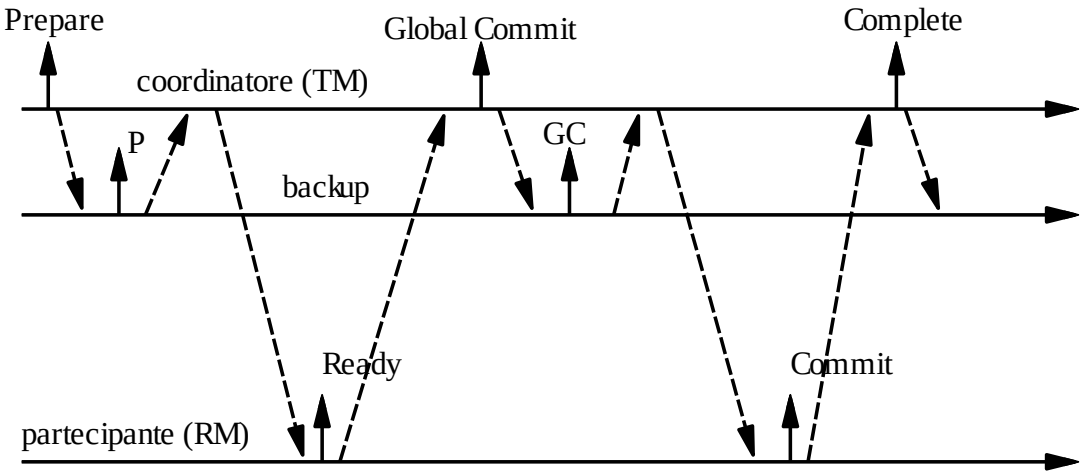
Da: Atzeni, Ceri, Paraboschi, Torlone - Basi di Dati: Architetture e linee di evoluzione

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



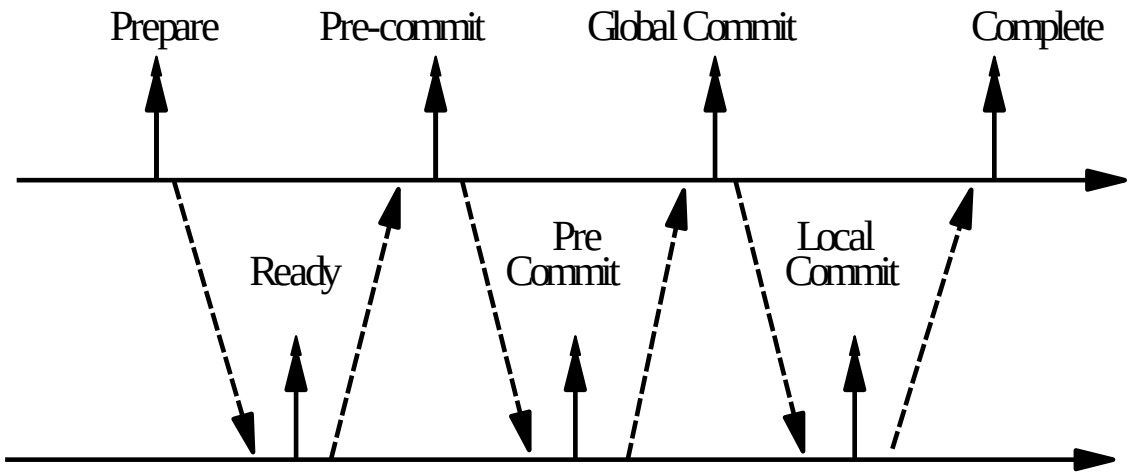
Coordinator backup is created at a different site
the backup maintains enough information to assume the role of coordinator if the actual coordinator crashes and does not recover.

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

TM crash

If the TM crashes and does not recover, any participant can assume the role of transaction manager and correctly terminate the transaction

The participant, scan the Log backward:

Last record in the Log for the transaction:

- Ready

  abort the transaction

- Pre-commit

  commit the transaction

Recovery and Atomicity

Block movements between disk and main memory are initiated

through the following two operations:

- **input**(*B*) transfers the physical block *B* to main memory.
- **output**(*B*) transfers the buffer block *B* to the disk

System can perform the **output** operation when it deems fit (Buffer manager, Replacement policies for the buffer manager)

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 stable storage without modifying the database itself

Log-based recovery

Recovery and Atomicity

Physical blocks are those blocks residing on the disk.

Buffer blocks are the blocks residing temporarily in main memory

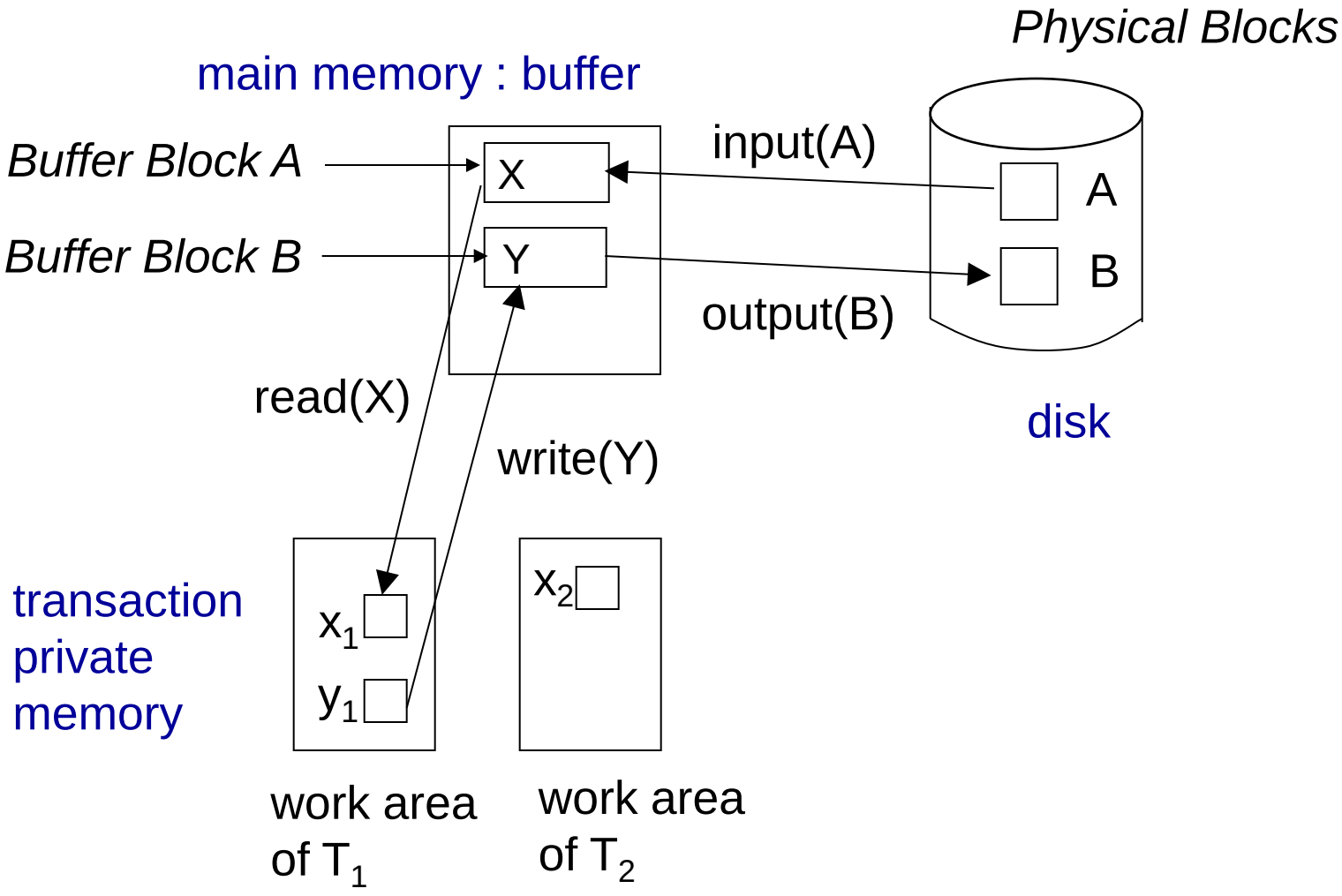
Transactions

- 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.

output(*BX*) need not immediately follow **write**(*X*)

System can perform the **output** operation when it deems fit

Example of Data Access



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

LOG file:

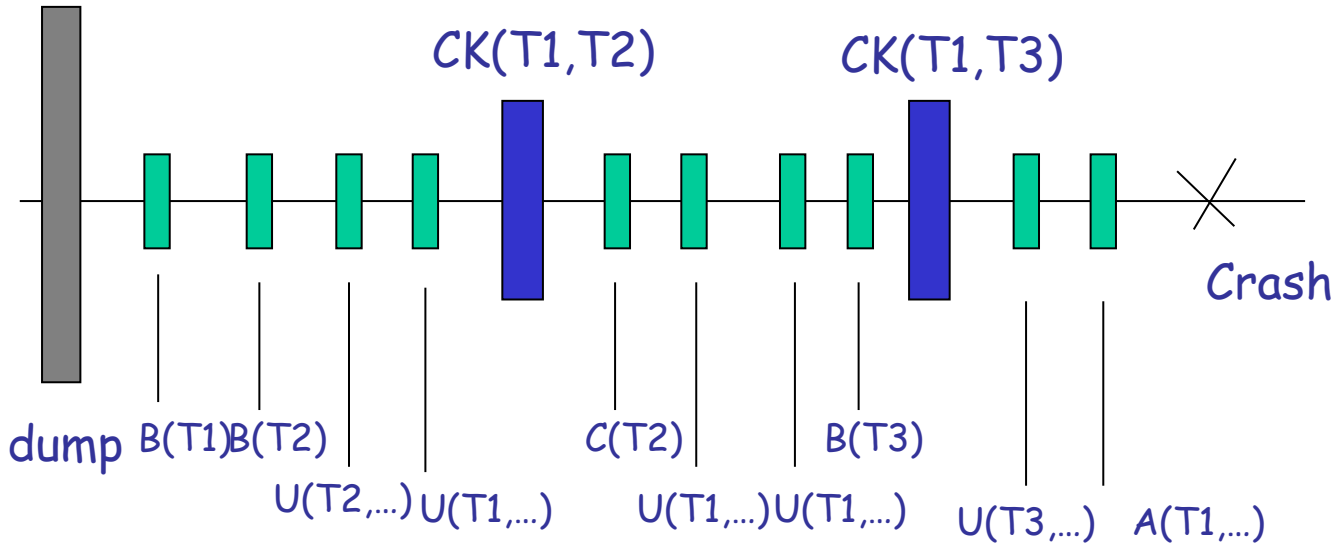
record types: B(T), U(T, O, B, A), A(T), C(T) ,CK(T1, …, Tn)

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 problem

Byzantine fault-tolerant (BFT) algorithms

they do not depend on trusted components for their correct operation

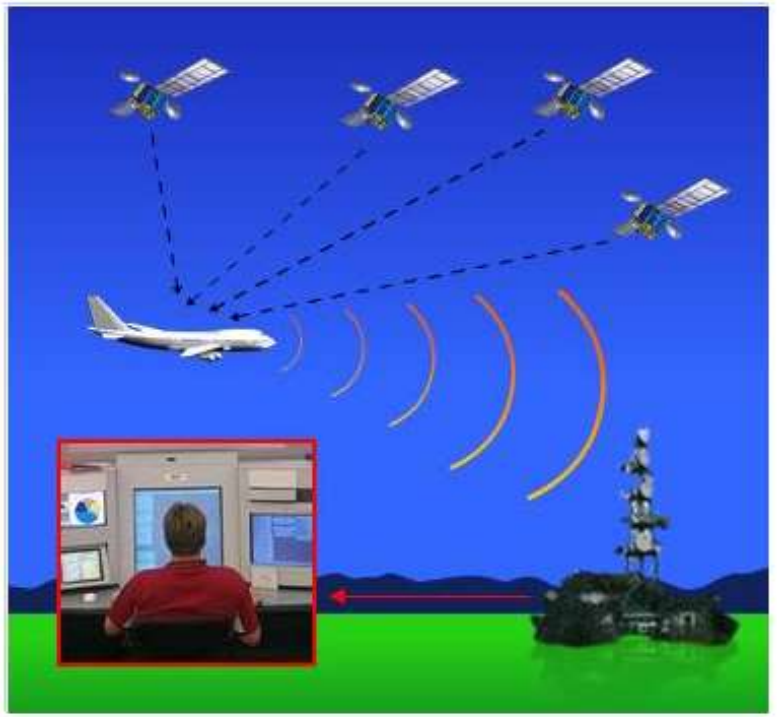
they must build trust during execution without trusting each other initially

they tolerate malicious components

Air Traffic Management

- Air Traffic Control (ATC) is a service provided by ground-based controllers who are responsible for maintaining a safe and efficient air traffic flow.

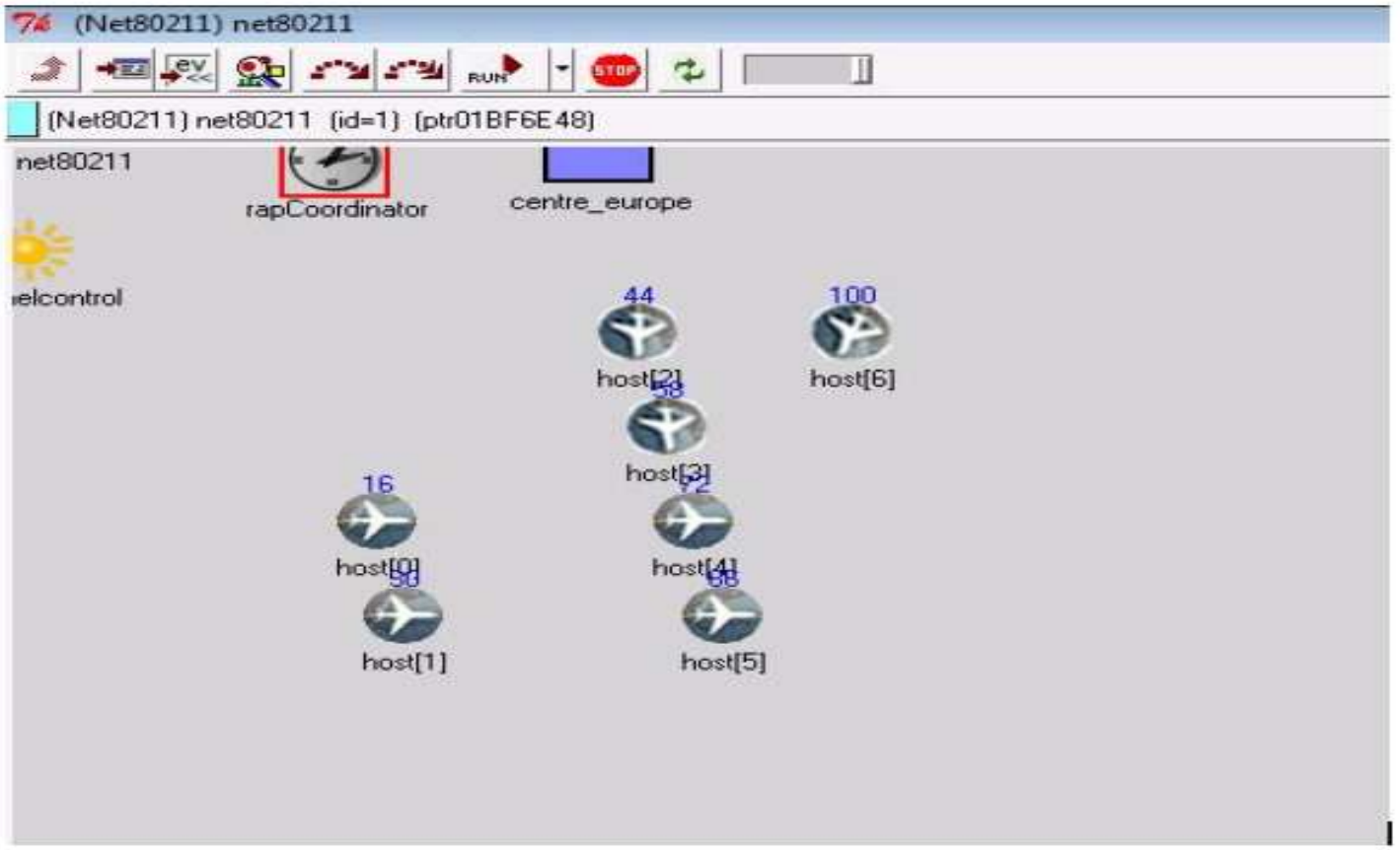
- Future generation of ATC: Airborne Self-Separation, an operating environment where pilots are allowed to select their flight paths in real-time.



ADS-B (AUTOMATIC DEPENDENT SURVELLEINCE BROADCAST): based on the Global Navigation Satellite System (*GNSS*) - broadcast communication links -

Airbone Self-Separation

- guarantee the correct behaviour of the system (i.e., the set of aircraft in a given area) even in the presence of component failures, or malicious attacks .



Airbone Self-Separation

Main challenge in Airborne Self-Separation:

coordination between aircrafts within a dynamic environment, where the set of surrounding aircraft is constantly changing, and where there is the possibility of arbitrary failures and malicious threats.

Conflict Resolution (and Traffic Optimisation) problem

Conflict Resolution algorithms are

decentralized and cooperative with the cooperation between aircraft being based on properties of the system

Conflict Resolution algorithms

based on theory of decision-making based on a multi-agent approach and Satisficing Game Theory (SGT)

- SGT as decision procedure requires the same information available at each node of the distributed system

- a fault-tolerant **Byzantine agreement protocol** that provides SGT the necessary services to execute correctly is necessary

- the agreement protocol is supported by suitable communication primitives realised for wireless networks

Conflict Resolution algorithms

System model:

multi-agent system

Aicraft: agent

Aircraft state (local information):
state_i = (aircraft id, destination, current_time, coordinates, speed, ….)

Region:
surrounding aircrafts involved in the calculation of the flight path (changing)

Decision procedure:

algorithm applied at every agent i based on

- agent state (local information)
- state of the agents in the region (information received from sorrounding aicrafts)

Decision procedure

Every aircraft must decide its flight path: to execute correctly it is necessary that every agent has the correct view of the system

Let n be number of agents in a region,

Agent_i: decision_procedure(i, state_1, state_2, …, state_n)

every agent applies the decision procedure starting from the same information on the state of the aircrafts in the region

Assume an attacker changes information exchanged through wireless communications.

What happen if information on the value of the position of aircraft 2 is modified and arrives wrong at some destination?

Agent_i: decision_procedure(i, state_1, state_2, …, state_n)

…

Agent_j: decision_procedure(j, state_1, state*_2, …, state_n)

Correctness of the decision procedure is compromised

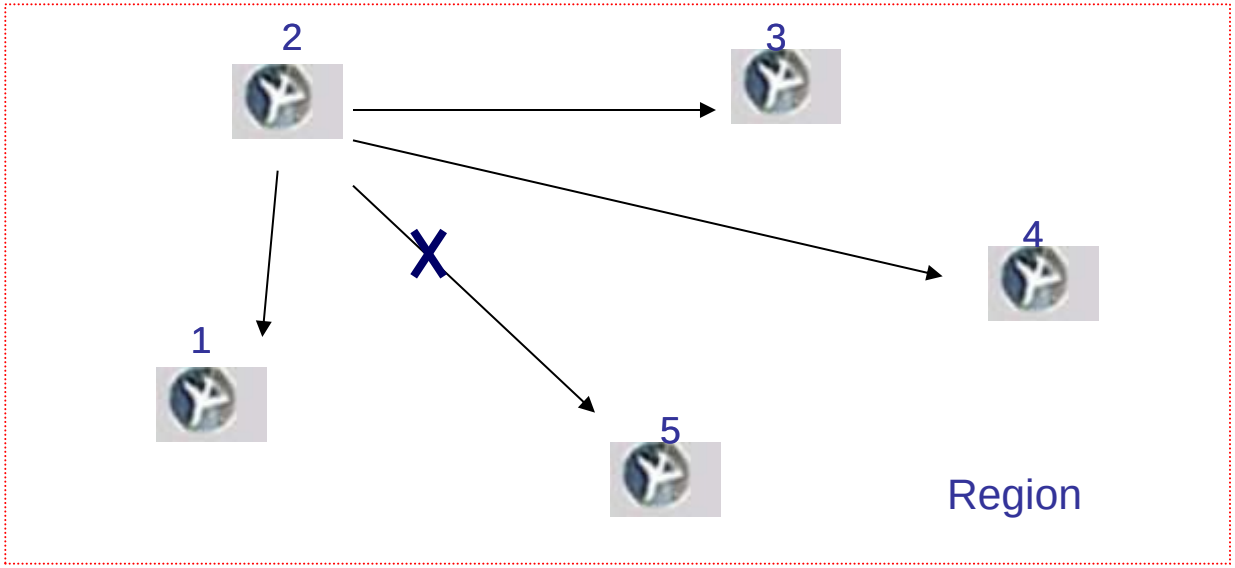
Agent_i and Agent_j must have the same information about the state of the aircraft in the region before applying the decision procedure.

Consensus problem

The Consensus problem can be stated informally as:

How to make a set of distributed agents achieve agreement on a value despite a number of threats?

Example: state of aircraft 2



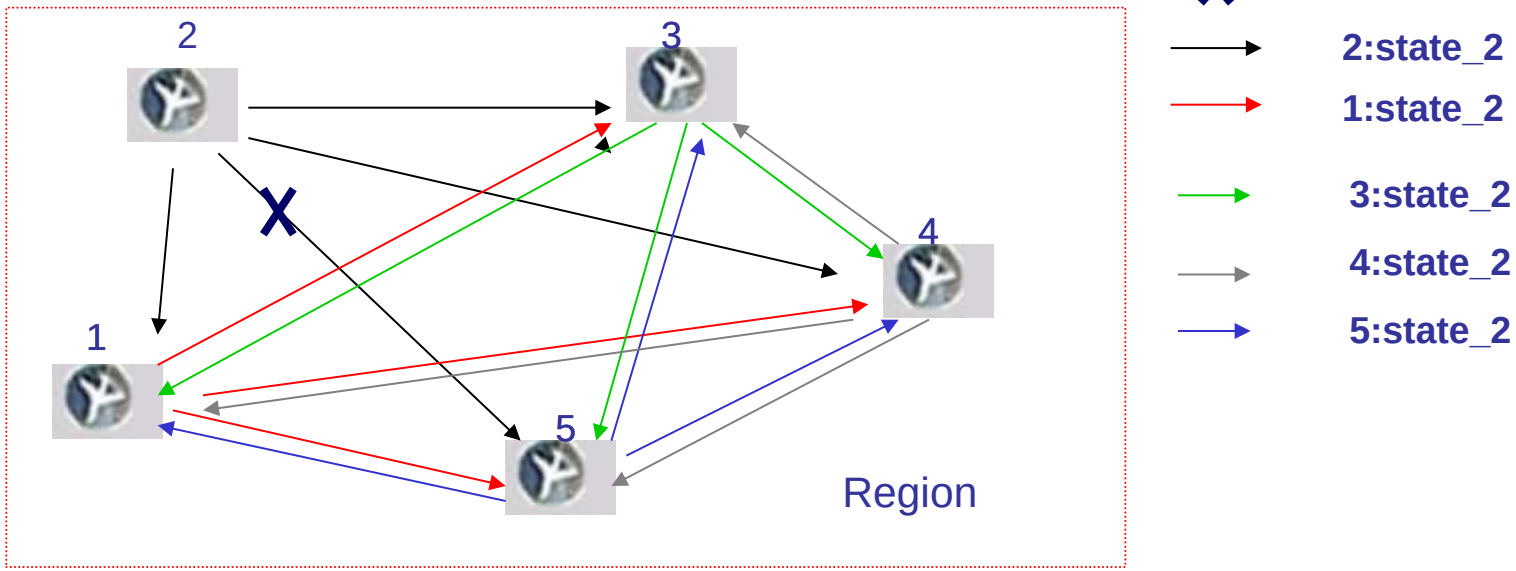
Byzantine fault-tolerant algorithm to solve the consensus problem

Consensus problem

Idea:

- use redundant messages exhanged among aircrafts reporting the position of the aircrafts

Example: value of the state of aircraft 2



Byzantine fault-tolerant (BFT) algorithms

From the abstract of Castro & Liskov OSDI'99 paper:

> *"We believe that Byzantine fault-tolerant algorithms will be increasingly important in the future because malicious attacks and software errors are increasingly common and can cause faulty nodes to exhibit arbitrary behavior."*