TELECOM ParisTech

Institut Mines-Télécom

Fault Tolerance

Thomas Robert
Office : 4D44
thomas.robert@telecom-paris.fr
Site web

Risk incident and general concepts

Motivation of this lecture

- **In the table below, you will find critical guideline about the grading policy**

- **This is an incident because it may have catastrophic consequences…**

Brainstorming : understanding the anatomy of an incident

- (easy) describe an incident for an automated train that involve the software controlling the train

- (bit more difficult) Describe me incidents for a market place putting customers and clients in touch to sell goods that involve the server code managing the search and transactions between clients

Incident anatomy : an abstract concept

- Incident = a state or event in a system + environment
- Derived concepts (often included in incident desrciptio):
  - responsibility, root cause, condition of occurrence, frequency of occurrence
  - Functional consequences, negative impact kind, cost, liability
- Problem managing incidents == A TRADE-OFF

Dependability (software systems)

- **Purpose :**

  Obtain a grounded trust in the **ability of a system to carry out and complete its expected services** given **identified use conditions**

- **Consequences**
  - Need to know what the system is expected to do, and to define «liabilities » between expectations and system components.
  - Determine how how confident you want to be and how you will share this confidence
  - Détermine acceptable use conditions

Abstract risk handling strategies

- **Mitigate (reduce) the risk**
  (changing use conditions or the system)
- **Delegate risk handling to a third party and consider the incident under control**
  (transfer the liability)
- **Accept the risk**
  (the incident will remain as is but it is ok)

- **Reject the situation (the incident cannot be handled, the system cannot be used nor produced - often appear later)**

Motivations: Zero defect theory not realistic

- In 1970’s : zero defect concept proposed as guideline for human task forces, then reinterpreted as a goal
- Principle :
  - Conformance to requirements (assume they are correct)
  - Fault handling = prevention
  - « Zero defect » is the target during production
  - Define a penalty to internal fault activation
- Criticism : defect = fault + responsibility + internal

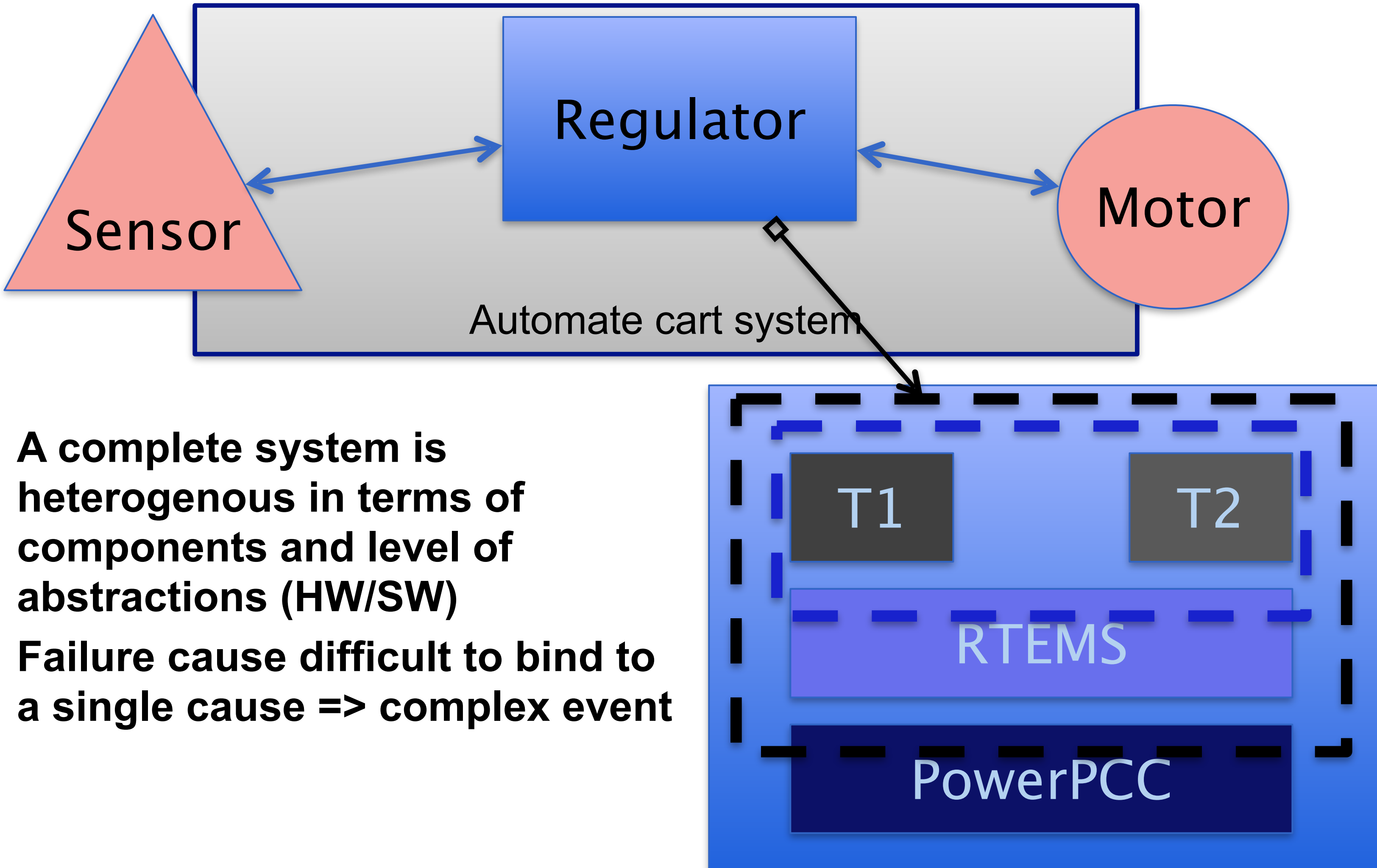
And so what ?

- Interaction faults and multiple conditions:
  - Dependable design => no single causes to catastrophic failures
  - What if not used as expected ? (zero defect ignore this point) … it depends (Therac 25 many causes but if no quick operation, no data race => no failure)
  - What if not correctly identified ? (overseen incident, Boeing 737 max)
- Hardware can fail, user can misuse the system, maintenance operation (software update) can go wrong
  => need to survive fault activation

Identify the scope : system, structure and dynamics

- **System : description unit that help distinguish the object of the analysis from its context (environnement)**
  - **System structure : the elements that are assumed to be fixed for once (usually the structure should not change)**
  - **System state et behavior : the information that can change during normal behavior of the system and that help define its expected behavior**
  - **System interface : part of the system state shared with the environnement (shared liabilities on the interface)**

Small Embedded system case study



- **A complete system is heterogenous in terms of components and level of abstractions (HW/SW)**
- **Failure cause difficult to bind to a single cause => complex event**

Means to Mitigate or decide to accept risk

- Prevention : prevent incident occurence eliminating their causes to occur (event), or to belong to the system or the environment (structure)
- Elimination : detect cause under the form of structural element and remove it
- Fault Tolerance : tolerate faute consequence but prevent the risk to be unbearable
- Assessment : determine entailed risk for given fault assumptions and a given system design.

Means to Mitigate or decide to accept risk

- Prevention : prevent incident occurence eliminating their causes to occur (event), or to belong to the system or the environment (structure)

- Elimination : detect cause under the form of structural element and remove it

- Fault Tolerance : tolerate faute consequence but prevent the risk to be unbearable

- Assessment : determine entailed risk for given fault assumptions and a given system design.

Means to Mitigate or decide to accept risk

- **Prevention : prevent incident occurence eliminating their causes to occur (event), or to belong to the system or the environment (structure)**
- **Elimination : detect cause under the form of structural element and remove it**
- **Fault Tolerance : tolerate faute consequence but prevent the risk to be unbearable**
- **Assessment : determine entailed risk for given fault assumptions and a given system design.**

Means to Mitigate or decide to accept risk

- **Prevention : prevent incident occurence eliminating their causes to occur (event), or to belong to the system or the environment (structure)**

- **Elimination : detect cause under the form of structural element and remove it**

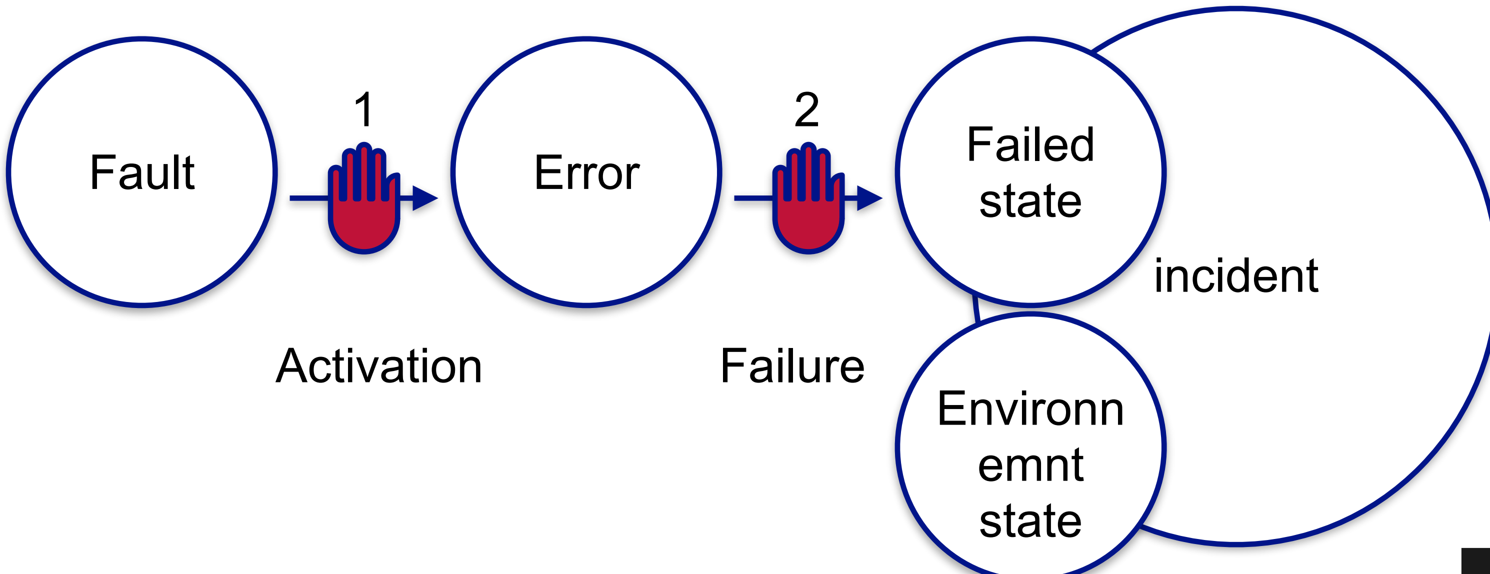
- **Fault Tolerance : tolerate faute consequence but prevent the risk to be unbearable**

- **Assessment : determine entailed risk for given fault assumption and a given system design.**

Fault Tolerance Definition, challenges and approaches

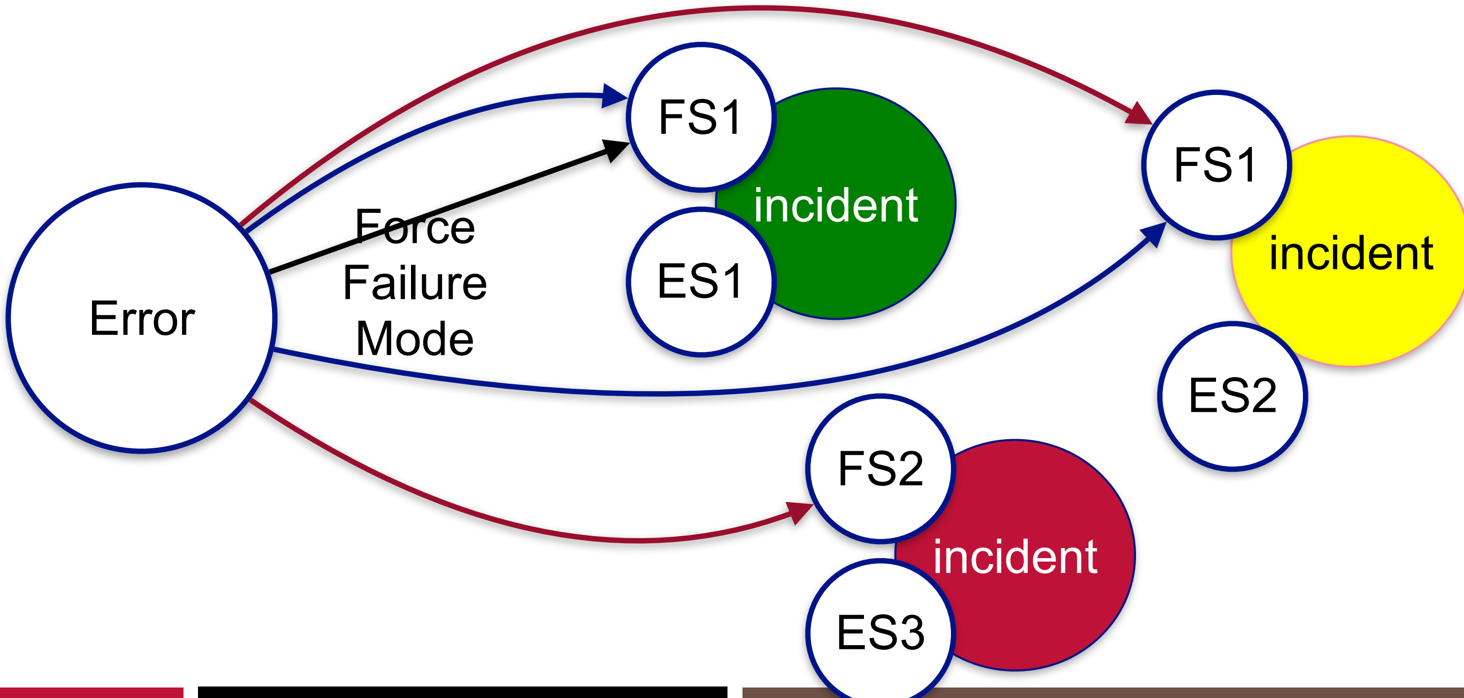
Definition I

- **Fault Tolerance : methods to deploys mechanisms that guarantees that failure impact can be mastered**
- **Limiting failure occurrence can be mitigated at run-time**
    1. **Detecting / controlling fault activation**
    2. **Detecting errors / preventing failed state**



Definition II

- **Fault Tolerance : methods to deploys mechanisms that guarantees that failure impact can be mastered**
- **Controlling failure impact**
    1. **Design failure signaling / recovery (manage)**
    2. **Steer system to master and select failure modes**



Generic vs dedicated / reuse vs efficiency

- **Error and failed state = specific to application**
- **Reliability / availability given failed state definition = non specific**
- **Generic solution for reliability / availability / integrity**
- **Dedicated solution required for safety (need to identify safe state first).**

Challenges with Faults

- Faults can be either structural feature or behavior
- Structural feature bound to the system
    - Poor code
    - Poor hardware
    - Inherent undesired behaviors (bit flips in memory)
- Behavior bound to the environnement
    - Wrong interaction on the system interface
    - Wrong context of use (the system entail an unwanted state in the env.)
    - Unknown interactions …. (hidden interface)
- Pb : how to inhibit a structural feature ? What is the best strategy w.r.t unwanted behaviors ?

Key idea : understand the full dynamics Root cause -(Activation/error/failure)$^{+}$- Failed state

- Controlling failure transition require to understand
  - When activation / error can be detected and where
  - How to prevent transition to failed state
    - Can we pause the dynamics ?
    - Can we determine the lower bound on time to failure ?
    - Can we revert state transitions ?
- System complexity make it difficult
  - More than one thread of state update
  - More than one abstraction level
  - Hardware/ software synchronous dynamics entails software error => hardware error and the way around.

Architectural description and error confinement

- Assumption : system interface, scope, expected behavior and use conditions defined
- A system provide Error Confinement
  - Identified failure modes that can be detected or are at least documented
  - Capabilities to detect errors before failure, and (optional) can mitigate them (no failure)
  - Fault assumption defining the use condition of this error confinement (≠ faults => confidence lost)

- Main objective : detects / signal / mitigate errors.

Course content

- Error confinement at the scale of the Hardware (the data storage case)
- Error confinement at the scale of the instruction sequence (programming language support and state of the art in API)
- Error confinement at the scale of the sequence 2 design pattern for sequential recovery, 1 pattern for diversification
- Next course content, replication strategies, and link to consensus algorithm + fault tolerance in real time systems.

Special case of storage failures - to get the intuition …

Error and failure in Hardware

- Failures = transition / error = state
- Control flow vs Data flow issues
  - Control flow : undesired instruction executed
  - Data Flow : accessed data with wrong value (not expected)
- Why Von Neumann architecture is so bad for fault tolerance ?
- Key idea : guarantee data integrity = top priority
- Fault model objective
  - Find realistic fault activation / impact,
  - Find realistic bounds to

Fault model, activation and confinement

- One storage unit + access function (store / read)
- Storage = fixed size array of bits
- Block model = partitioned in subintervals of fixed size (same for all).
- Objective : provide fault confinement on read access for fault that modify some of stored bits.
  - Pb 1: how to detect altered bits
  - Pb 2: how to recover from altered bits
- Coding theory provide a solution
  - See stored value as information quantity and not just the value
  - Work on the information encoding

Principle of the detection

- **Information : store K different values (K=2^P)**
- **Optimal encoding (number if bits) =**
  numbering values from 0 to 2^P-1 and bind it to the base 2 encoding of this number
- **PB: modify 1 bit encode a different value**
- **Idea: modify r bits does not represent a valid encoding of a value**
- **How : add extra information**

Encoding Function Enc : Val —> 2^n (n > log(|Val|)/log (2))
Decoding function Dec : Enc(Val)—> Val

Fault model, extension of Dec for detection/ correction

- Dec is not defined on 2^n a priori
- Consider y' not in Enc(Val)
- Fault activation = add ∆ to y in Env(Val), y is said faulted
- Error detection consist in extending Dec in Dec' so that
  - If y in Enc(Val) it returns x such that y= Enc(x)
  - Otherwise return « error »
  - The output domain is extended with the error case.
- Error correction under the additive assumption
  - For every element y' in 2^P there exist an element of Env(Val) that is considered as the most likely faulted code word leading to y'
  - Error correction returns x s.t. y= Enc(x) for any value y+∆ give ∆ is the fault activation logic

TELECOM ParisTech

Hamming distance : measure the space between code words

- **Hamming distance, Hdist, for two vectors from {0,1}^n**

  Hdist (v1,v2) =Card({i | v1[i]≠v2[i] })

- **Hamming weight of W(v) = Hdist (v,0) = number of non 0 element**

- **Hamming Ball of size r around v ={ v'| W (v xor v')≤r}**

- **Note that alternative notation of v xor v' is v-v'**

Principle of detection / correction based on Hamming distance, and surrounding ball

- Let assume we want to tolerate r errors in a block of n bits
- At decoding time : assume at most r bits have been modified from an element of Enc(Val) to obtain y'
- When does detection is possible ?
  For all y, correct encoding of a value in Val, ensure that ball(y,r) does contain a single element of Enc(Val)
- When does correction is possible ?
  For all element v in 2^n, ensure there is a single element of Enc(Val) in ball(v,r)
  Alternate criteria : For all y, correct encoding of a value in Val, ensure that ball(y,2r) only contain y from Enc(Val)

Hamming code (4,7)

- **Example on the whiteboard**

Software and Error confinement strategies

Software failure / fault assumptions

- System:
  - Structure=sequence of instruction
  - Interface=set of variables (typed or not)
  - Expected behavior=read interface state, compute, update
  - In the interface : application data + control data (ensure execution continuity - e.g. return conditions)
- Failure modes :
  - No W (system seems absent)
  - Bad W (wrong value or bad timing …) on data flow
  - Bad W on control flow
- Faults :
  - Code leading to data error or control flow error (e.g. may entail no W)
  - Hardware / execution platform issues
  - Interaction issues

What is the scope / interface of a sequential code

- State = 2 parts
  - Data flow (memory, variables …)
  - Control flow : register value, return address, call stack structure….
- Error properties :
  - Error in data bound to data flow = can alter the functional state and propagate as interaction faults
  - Error in data bound to control flow = can change le sequence of actions executed (and eventually the functional state but can propagate to the execution platform)

Confinement at Block level (Exceptions)

- {
- Statement1 —> data flow error e1 or e2 (don't know)
- Statement2 —> call to f => may detect data flow error e1
- Statement3 —> call to g => may detect data flow error e2
- }

- Handling code e1: { }
- Handling code e2: { }

- Exception principle : intercept error at the beginning of step 2 /3 as interaction fault + branch to recovery / signaling code
- Provide naming / typing and routing features

Confinement at Block level (Exceptions)

- Requirement: need branching capabilities in case of error detection, integrated to languages
- Design pattern : try / throw / catch model
  - Given a block of sequential code, N types of error can be detected
  - Detection entail branching (throw) to detection mitigation (catch)
  - Compatible with block nesting => capability to propagation error detection to upper level
- Criticism :
  - Do not encourage to manage interaction faults because seems already done …
  - Provide good localisation of fault activation
  - Ease interception of failure transition and resource management

Case 1 : functions

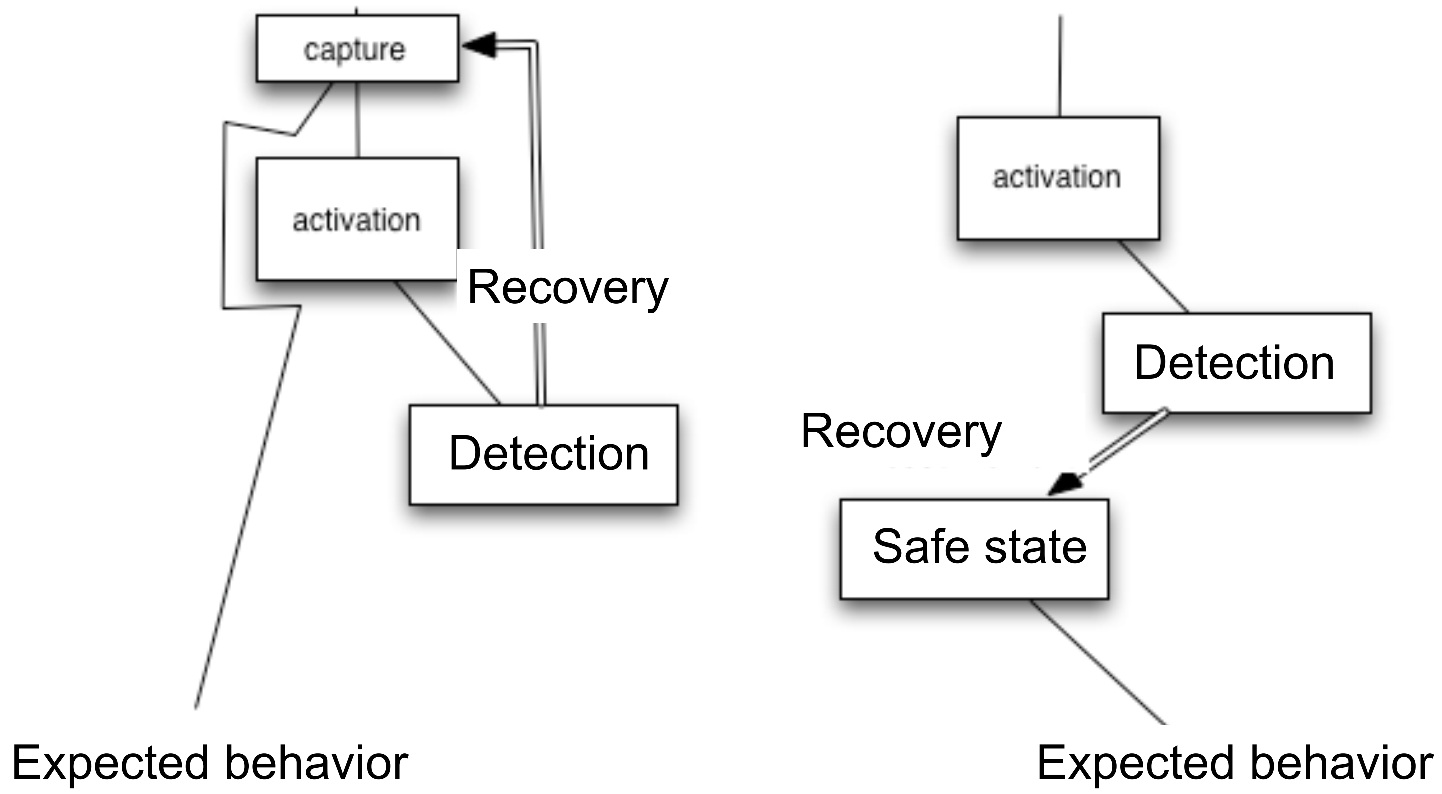
- Observation : Beginning and end of sequence well identified (can insert code to prevent propagation of errors from/to the interface
- Internal state : local variables + locally allocated variable on the heap
- Handling interaction faults consequences :
  - Stateless — Filter input parameter value (use predicates)
  - Stateful — use static local variables to keep track of issues
- Failure signaling : use globale variable (bad) or the return value (best practice if no other support)

Confinement at function scope (C example) How to make confinement afterwards.

- Specificities :
  - parameters can be adresses to share memory (so input / output parameters)
  - Return value of limited type
- Design pattern : function wrapping (in C)
  - Given retType f(Tp1, …, Tpn) a typed function
  - Build FMType g(Tp1, …, Tpn, Tout)
  - Call f from g but implement error confinement
    - Filter interaction faults on input parameters
    - Filter failure on output with
      - Assertions
      - Comparison to oracles
    - Manage ressource if error mitigation needed

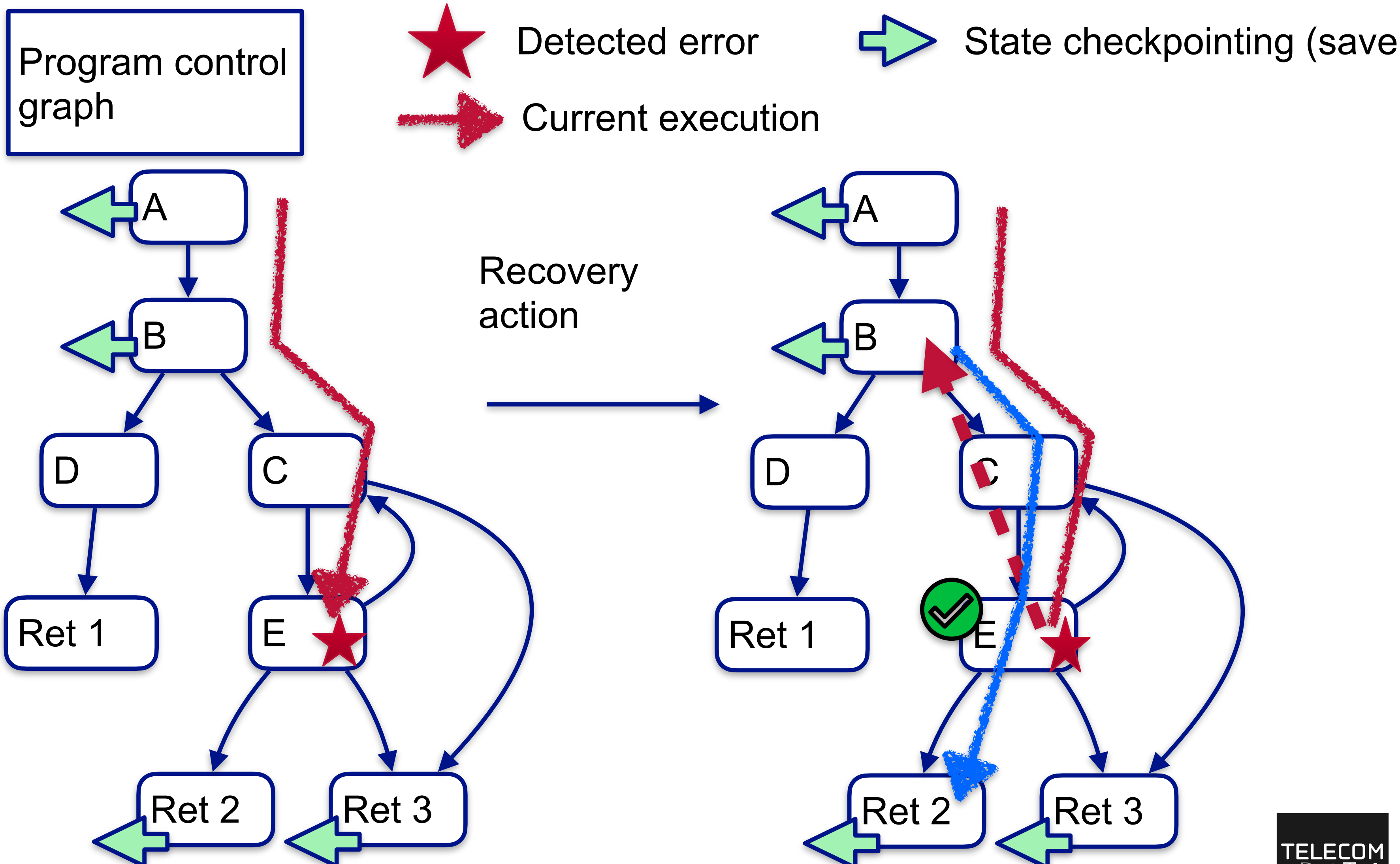
Error Detection / recovery generic templates

- **Forward vs Backward recovery**
- **Pb : how to mitigate errors**

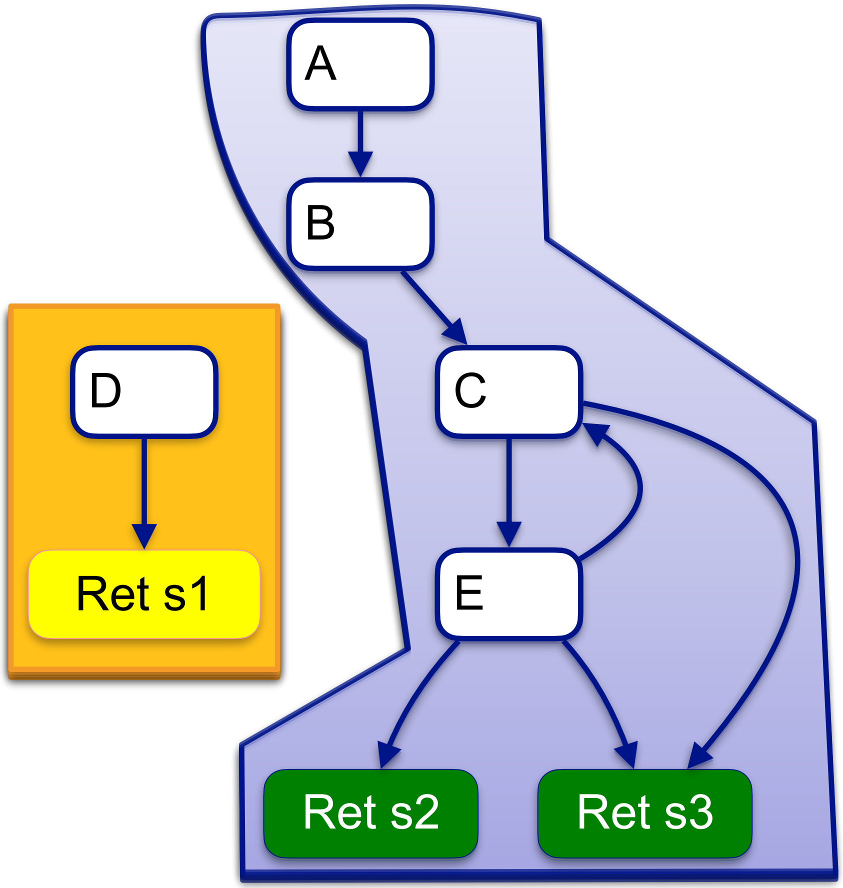


TELECOM ParisTech

Illustrating backward recovery

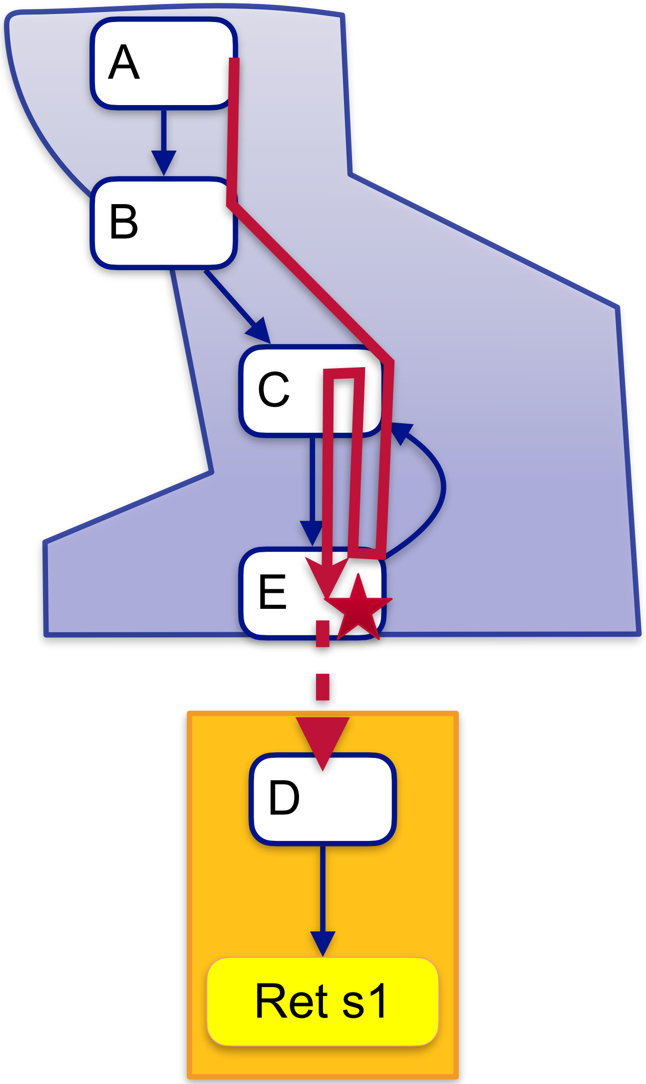


Forward recovery detailed



- Pb what is the solution for systematic activation ?
- Additional assumption :
  - 2 level of services « optimal » and « safe but degraded »
  - Blue graph = optimal
  - Yellow graph = safe but degraded

Execution logic of forward recovery



- Upon detection
  - Rerouting execution to D state
  - Continue from D independently from the past
- Example
  - Text editor
  - Network connectivity
  - … your turn

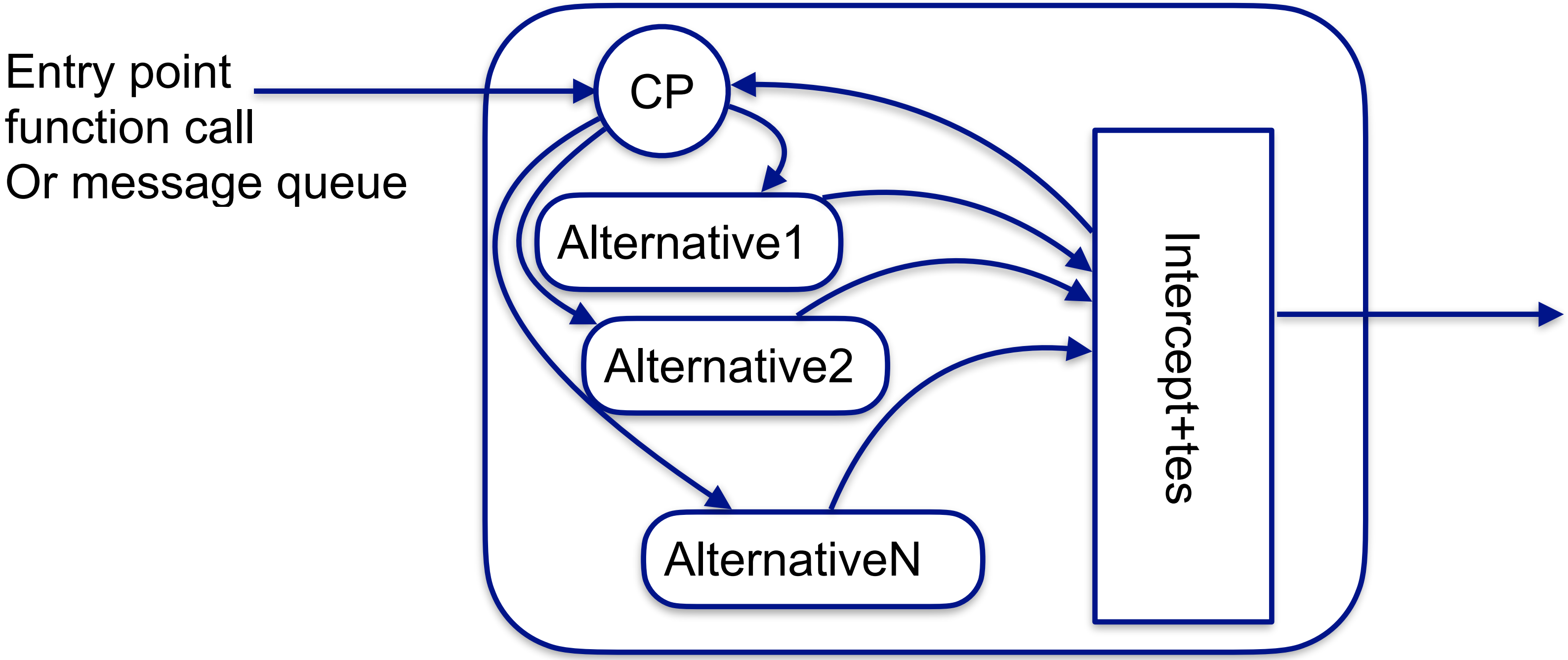
How to cope with systematic activation : Diversification (code)

- BWR recovery cannot cope with wrong pointer initialization for instance ….
- Idea : use different implementations of the same function

- IT IS THE DEFINITION OF DIVERSIFICATION

Recovery Blocks the main idea

- **CP = capture point or check point**



Possible execution scenarii

- **Without Failure of any alternative**

CP	Alternatiev1	TestOK

- **With an alternative implementation failing**

CP	Alternatiev1	TestFail	RestoreCP	Alternatiev2	TestOK

- **Cost model for the approach**
  - **Time : proportional to alternative Worst case execution time and number of failures**
  - **Memory : CP storage is not necessarily cheap**

Replica and failure modes

TELECOM ParisTech

The concept of replica

- Idea : use of N version programming + distinct hardware to support execution
- Consequences : confinement at the scope of a host, or a subnet.
- Given a functionality to deploy, a replica =
  - Software
  - Hardware
  - Integration (code or hardware)
- Fault tolerance dealt with multiple replica with different failure modes (e.g. replica failures are the system error)

Most popular failure modes

- 3 templates that help designing efficient strategy and cover many cases or tradeoffs
  - Crash (a host either produce correct output or stop emitting any data, permanently up to its repair)
  - Omission and commission : in a sequence of expected output, some are missing or some are duplicated
  - Byzantine failure : a host can exhibit an arbitrary behavior (covering any possible behavior — worst case)
- Can consider other cases but design patterns mostly for those three cases.

Passive replication

Passive Replication principle

- Problem to be solved :
  - Define a mechanism to resist crashes
  - Optimize the used CPU
  - Scale to an arbitrary upper bound to crashes count on a lifetime
- Assumption : network does not fail, do not alter message integrity nor availability

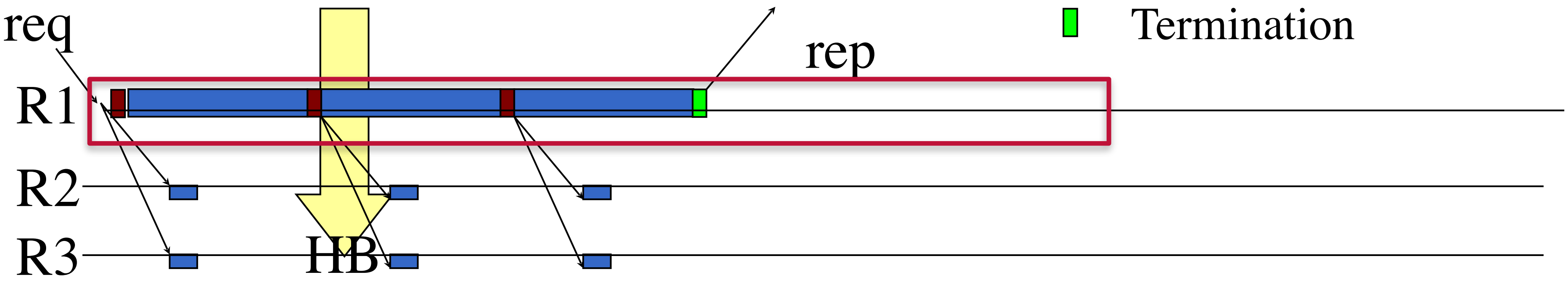
Passive replication behavior

- Idea : revisit backward recovery
- Replica equipped with integration code to capture internal state
- Additional (leader/follower) state
  - In leader mode : perform computation, produce output, perform state capture, and broadcast it
  - In follower mode : wait for state update + can decide whether leader failed & elect new leader
- Failure assumption covered : crash of #replica - 1.

3 replica execution scenarii

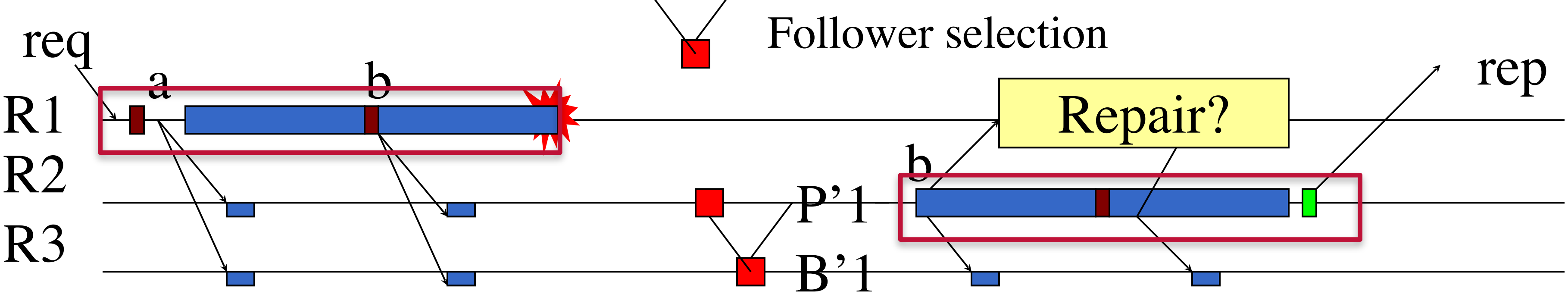
State Capture
Processing
Recovery state storage
Termination

- **Without Failures**



- **With a crash**

Crash
Follower selection



TELECOM ParisTech

Think it twice

- Small brainstorming :
  - Is it tractable for a real time task ? Why ?
  - In which condition does it save CPU, is it network friendly? Why ?
  - What does happen if we change the network behavior assumption (recall : perfect network)
    - What if it can loose sometime messages ? (But not too often) ?
    - What if it can alter the content of messages (not too often too) ?

Active Replication
The other extreme case

Active Replication principle

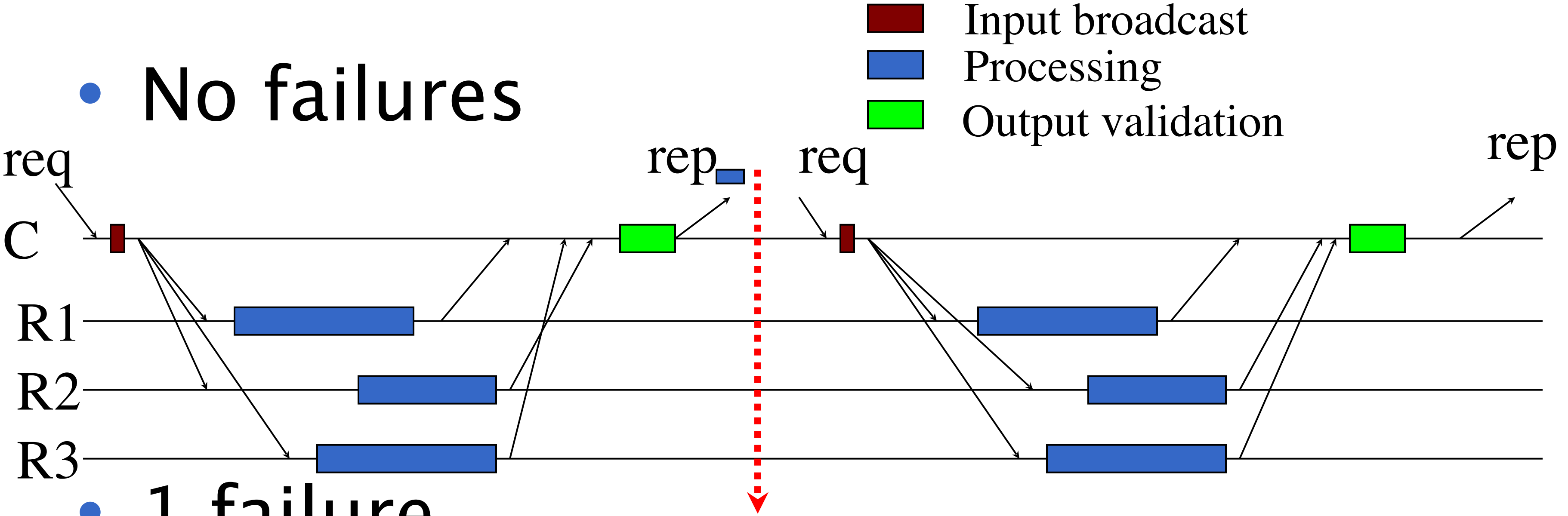
- Problem to be solved :
  - Define a mechanism to resist Byzantine fault to cover network as well as host failures
  - Optimize latency for recovery
  - Scale to an arbitrary upper bound to crashes count on a lifetime
- Assumption : possible to design integration code that does not fail if the host has not failed.

Active replication behavior

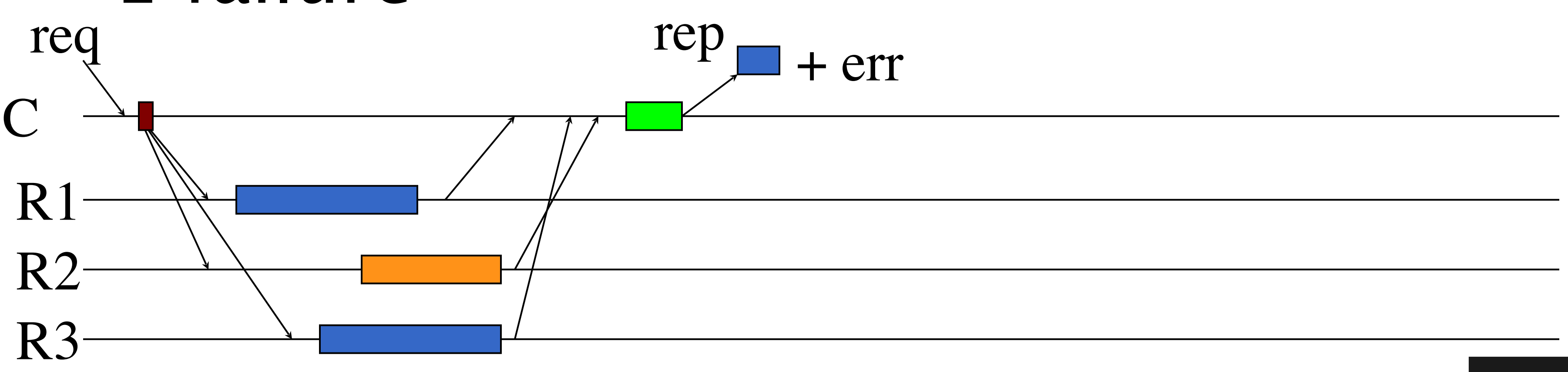
- Idea : revisit recovery block with parallel execution
- Architecture made of N replica plus 1 node in charge of input output (can be one of the replicas but not the usual assumption)
- Assumption : the input output node cannot fail (trustworthy)
- Replica communicate with the input output node.
- Input/output node behavior
  - When a processing start, send input to replicas, wait for reply
  - Upon reception of a sufficient number of reply, decide what should be produced (vote, average …)
- Failure assumption covered : 2 #Byzantines < #replicas - 1.

3 replica execution scenarii

- No failures



- 1 failure

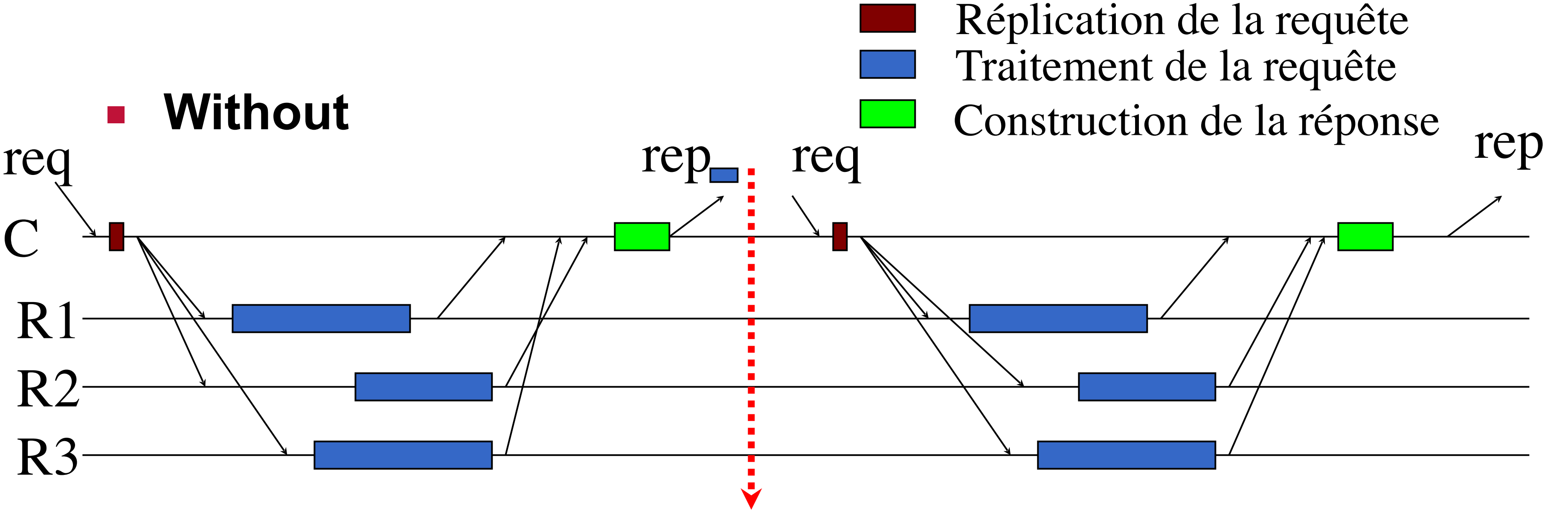


Think it twice

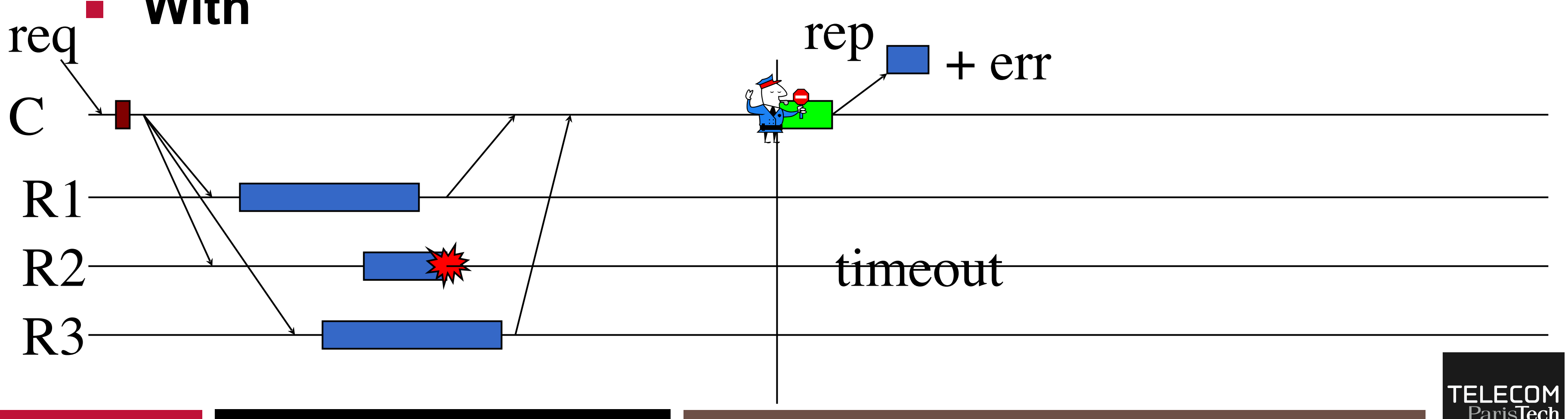
- **Propose for three replicas with bounded correct execution time a voting mechanism that guarantee bounded time reply**
- **Propose a state in which the active replication for 3 replicas with such mechanisms can signal an error but cannot correct it (nor produce wrong output).**
- **Comment about the « voter » in a case failure cannot recover without manual recovery (assume more than 3 replicas).**

3 replica execution scenarii

- **Without**



- **With**



Ressource Pool model

- Idea: could deploy this principle on clouds or on operating systems with processes
- Replica can be spawned on demand
  - Pro : offer tuning capabilities on dependability
  - Cons : consume ressources
- Solution : define pools of ressources with bounds
  - Create/destroy replicas
  - Pool elements : in and out need more synchronization to decide who participate
- Motivation the need for consensus algorithms