Tolérance aux Fautes des Systèmes Informatiques

Thomas ROBERT

2019

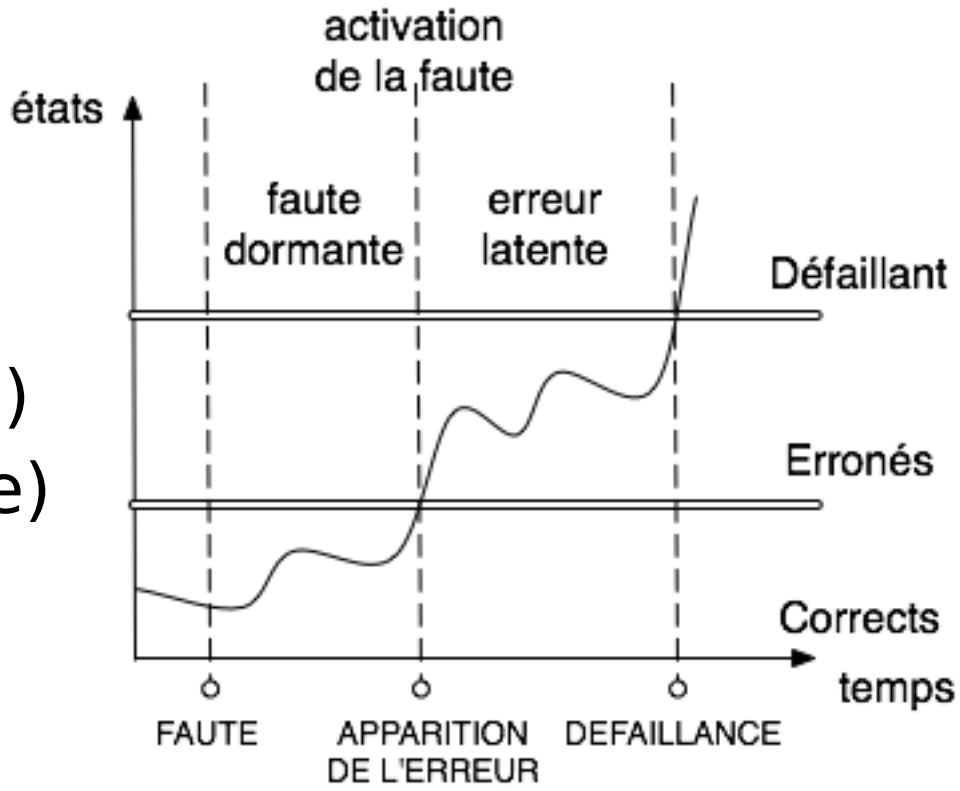
Plan du cours

- Tolérance aux fautes sur les données
- Tolérance aux fautes des traitements version «logicielle »
- Tolérance aux fautes des traitements version « matérielle »

Rappel : La chaîne faute/erreur/Défaillance

- Evénements :
  - Activation
  - Détection
  - Défaillance
- Etats :
  - Fautifs (non activés)
  - Corrects (sans faute)
  - Erronés (?)
  - Défaillants (KO)

Pas de détection == erreur dormante jusqu'à la défaillance …



Le modèle de faute I

- Une faute == cause adjugée ou effective d'une erreur et donc d'une défaillance

modèle de faute=caractériser la faute

- Quelques exemples :
  - **Corruption de la mémoire physique** -> les valeurs peuvent être modifiées aléatoirement
  - **Faute de développement**, pointeur mal initialisé -> donnée incorrecte / boucles infinies
  - **Défaillance du matériel ou sous-système** -> arrêt complet du système, comportement aléatoire

environnement

Production

Interaction

Le modèle de faute II

- Capturer leurs caractéristiques :
  - Phase de création ou activation
    - Modèle d'activation : déterministe / aléatoire
    - Phase de création : développement/opération
  - Positionnement % système (logicielle/matérielle) (interne externe)
  - Source (humaine / « naturelle »)
  - Persistance (permanente/transitoire)

=> Permet de déterminer la stratégie adaptée / permet de créer les modèles d'évaluation ....

Structure hiérarchique et systèmes de systèmes

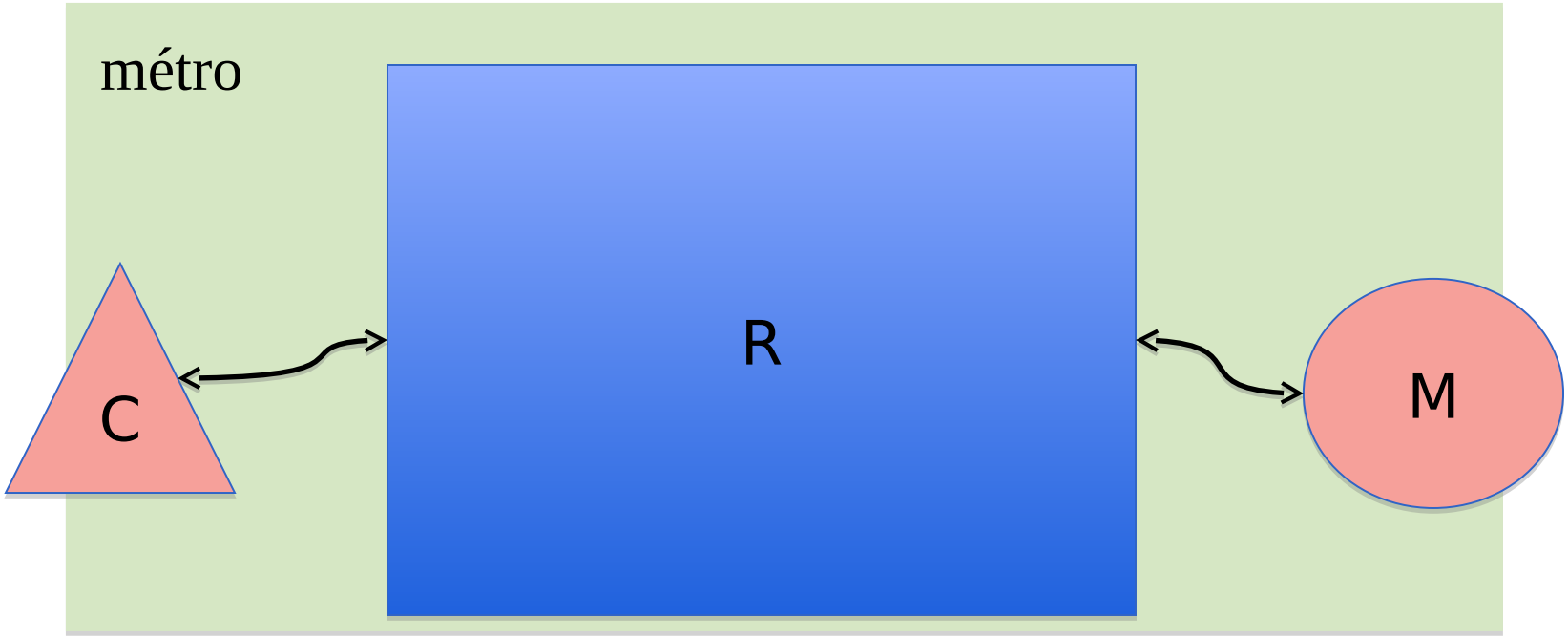
- La structure d'un système :
  - Hiérarchie
  - Dépendances matériel / logiciel
  - Description des interactions aux interfaces

- Principe de propagation :

  La défaillance d'une partie d'un système peut devenir une faute pour le reste du système

- La définition d'une architecture aide à raisonner (un des intérêts des ADL)

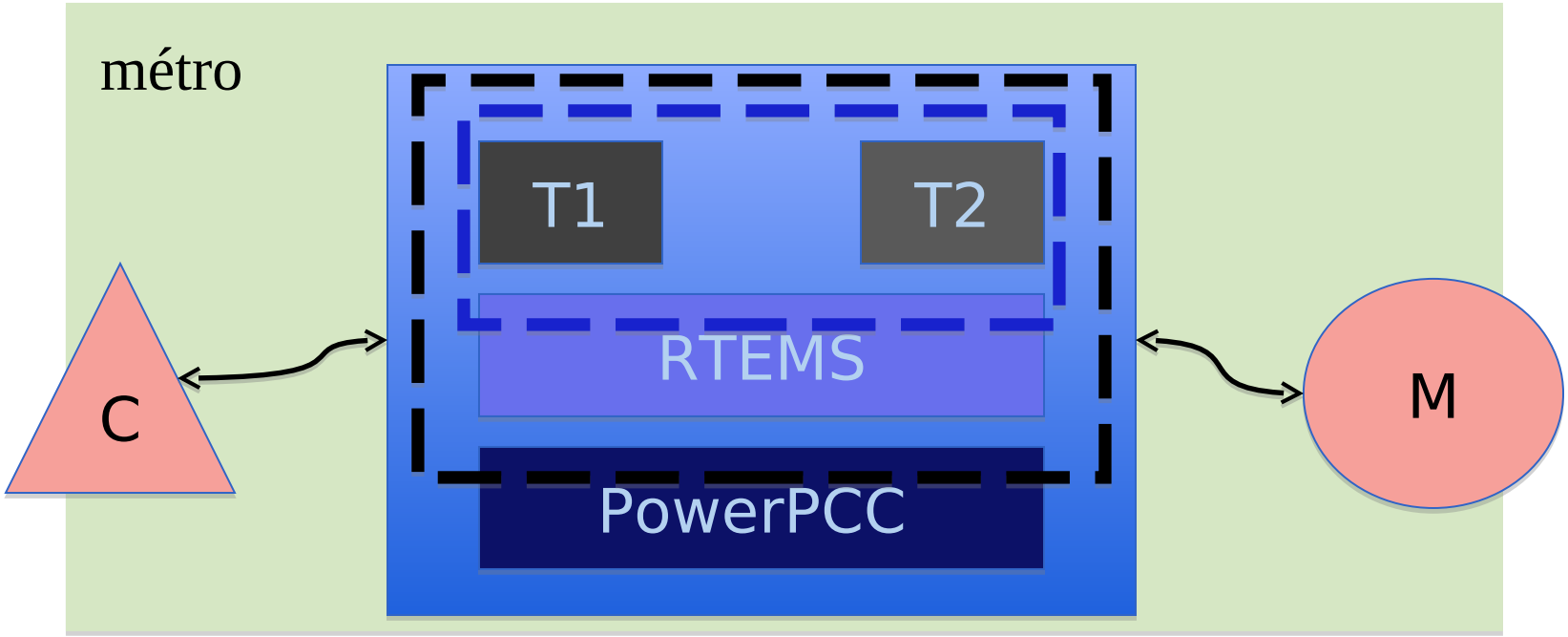
Défaillances, fautes et propagation des erreurs



- Fautes : interactions (propagées à travers l'interface)
- Propagation des erreurs : capteur (C) -> régulateur (R) -> moteur (M)
- Erreur dans C -> défaillance de C -> faute pour R -> erreur dans R

On peut « entrer » dans R…

Propagation : Logiciel/Matériel



- Décomposition du boitier en éléments relativement indépendants
- La défaillance d'une tâche peut altérer le fonctionnement de RTEMS déclenchant une altération du matériel (effacement du bios

⇒Propagation interne au boitier du logiciel vers le matériel

C'est très [redacted] aisonner par type de fautes, pas à pas

Le principe de la TaF

- Empêcher les défaillances :
  1. Éviter l'activation des fautes (différent de l'élimination au développement)
  2. Éviter qu'une erreur n'entraîne une défaillance (tolérance aux fautes)

- 1) Traitement des fautes,
  2) Traitement des erreurs
- Comment faire 2) ?
  - Détection & Rétablissement
  - Masquage (ou compensation)
- Cela va avoir coût

Principes fondamentaux

- Avoir des composants aux défaillances maitrisées
- Gérer la propagation des défaillances internes
- Mise en œuvre :
  - Connaître l'état du système
  - Savoir traiter / maitriser un état erroné
  - Comprendre l'origine des erreurs pour la maintenance

Zone de confinement des erreurs

- **Définition**
  périmètre/interface d'un système muni de mécanismes de protection empêchant une erreur de se propager sans être au moins détecté et signalée
  (ie de contaminer l'environnement du système)
- En pratique :
  - Description architecturale définissant la décomposition du système en sous-systèmes dépendant les uns des autres
  - Description des défaillances possibles associées au système et chaque sous-système.
  - modèle de fautes (apparition des erreurs) et de leur propagation
- Mise en œuvre=contrôle aux interfaces, modifications architecturales / comportementales internes

Traitement des Fautes

- **Détection (diagnostic)** : déterminer la cause première d'une défaillance. (ex. régulateur)

- **Isolation** : relier la plus petite partie du système à la faute cause de la défaillance
  Détermination d'une zone de confinement

- **Reconfiguration** :
  altération de la structure et de la logique du système pour inhiber la faute

Traitement des Erreurs

- **Détection**
  - Test de vraisemblance : erreur si Observé ≠ Attendu
  - Exécution multiple + Comparaison, erreur si V1≠V2
- **Recouvrement :** Démarche de correction de l'erreur par
  - Redéfinition de l'état ou reconstruction
  - Compensation de l'état erroné (cf redondance)
- **Utilisation de la détection**
  - Signalement/journalisation (exception Java)
  - Synchronisation d'action de Recouvrement

Tolérance aux fautes sur les données

Quel est le problème

- Donnée stockée hiérarchiquement % différents niveaux d'abstraction
  - Fichier
  - Bloc de k octets
  - Dispositif de stockage de masse

Solution orientée par le besoin de disponibilité + modèle de faute

Le cas traité

- Séquence de bits de taille fixe
  faute = modification aléatoire d'un bit de la séquence
- Séquence de bits de taille fixe

faute = modification de n bit contigues

- Solutions vues : codage par bloc Hamming + principe de dissémination

Abstraction du codage

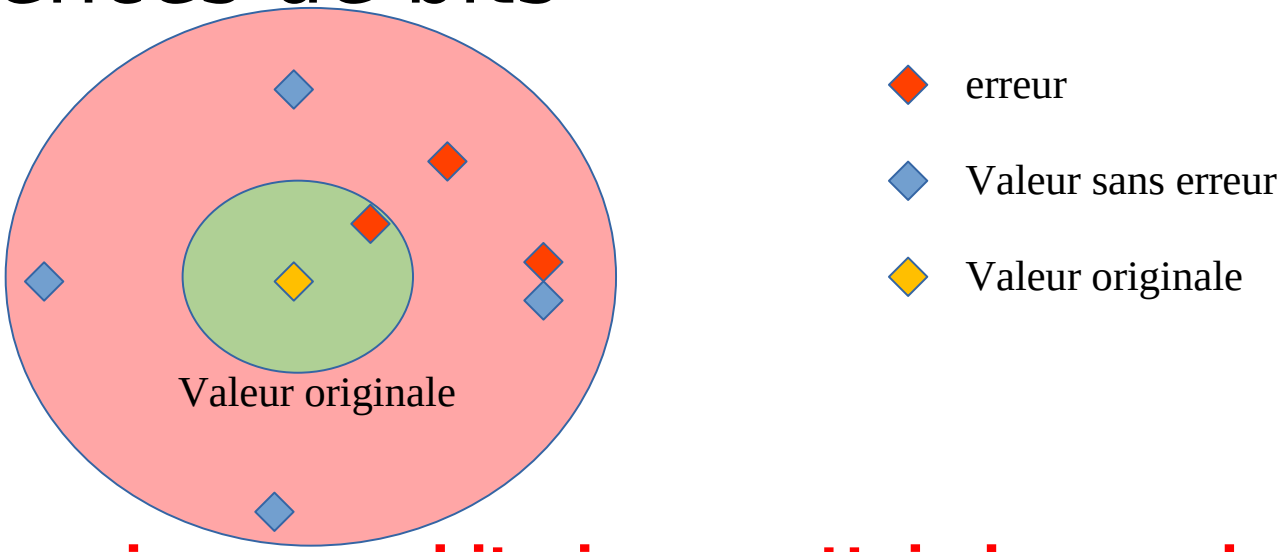
- Codage (binaire) : représentation sous forme de séquence de bits d'une information

D (information) => mot de code C tel que C =G(D) une séquence de n bits, n > m (G fonction génératrice)

- Soit V un ensemble de valeurs distinctes de taille 2^m => m bits au minimum pour son codage

Détection vs Correction I

- 1 une information codée = 1 point
- 1 distance pour comparer des séquences de bits



Principe : si erreur arbitraire => atteind une sphere autour de la valeur originale
Pb : si la sphère contient un mot de code pas de détection

Détection vs Correction II

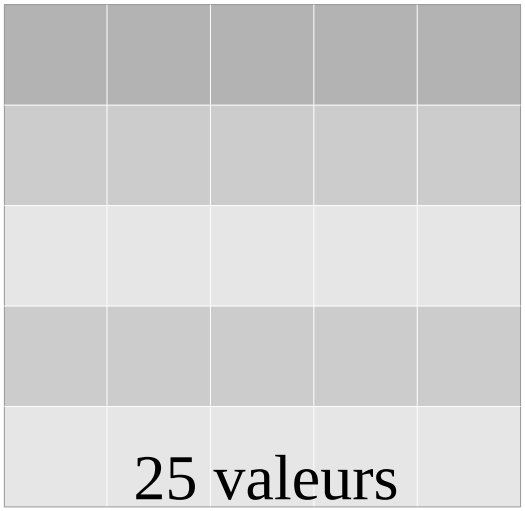
- Distance de Hamming ($d_H(v,v')$) = nombre de bits distincts entre deux séquences de tailles identiques.

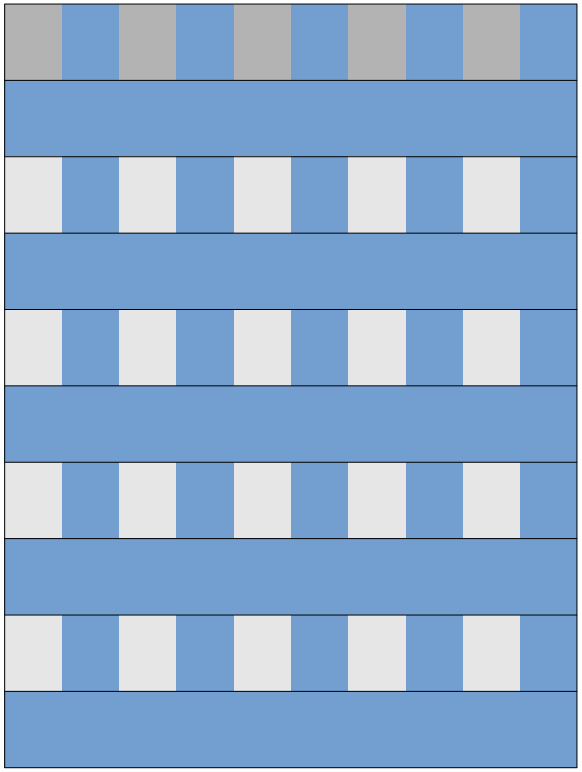
$d_H(0101,1100)=2$

- Distance d'un codage : minimum de la distance de hamming sur l'ensemble des mots de code obtenus par G (fonction génératrice). (noté dmin(G)).
- Détection de k erreurs si dmin(G)> k (néc.)
- Correction si dmin(G) > floor((k-1)/2) (suf.)

Codage pour la détection

- Visuellement





Principe le cas linéaire (mathématisé – opt.)

Vecteur de donnée (ex 4 bits)

X=(0100),

codage par G matrice

G.X= (0010110) =,Mc

Mc’ : mot potentiellement fauté

Décodage par matrice H

H.Mc’= 0 => pas de faute

H.Mc’ ≠ 0 => faute → reconstruction

Alternatives – la localité

- Idée : les altérations sont peu fréquentes mais groupées > taille d'une donnée élémentaire
- Codage de x dans V : x1, …. , xn
- **Pb si u > n bits modifiés ?**
- Créer K vecteurs de taille l ≤ n tels que V_i contient un sous ensemble des « coordonnées » de x (mélangé avec d'autres messages peut être)

- Propriété : il suffit de disposer de suffisamment de copies intègres pour retrouver x.

Mise en pratique

- Soit un Message avec 45 bits utiles, et 19 bit de redondance (supposons le code parfait: $d_H(C1, C2) \geq 19$
- Avant émission on accumule 8 Messages C1, C2 … C8 de façon à tranférer le premier octet de C1, puis C2 … jusqu'à C8 (puis on passe au second octet)
- Combien de bit altérés successif tolère t on ? (nb. 64Bits= 8 octets).

Tolérance aux fautes logicielle

Génie logiciel et TaF

- Les activités:
  - Identification / classification des défaillances
  - Conception / mise en œuvre des zones de confinement
  - Vérification des comportements/performances
- Les moyens
  - Des modèles de fautes (hypothèses de travail)
  - Des designs patterns (architectures)
  - Des modèles de performances (chaînes de markov)

TaF logicielle ≠ logiciel pour la TaF

- TaF logicielle => zone de confinement au niveau LOGICIEL (ce doit être justifié)
- **Pré-requis** :
  connaître les modes de défaillance du logiciel et leur propagation au matériel
- **Exemple au tableau sur une application Java hébergée sous Unix**
  - **OutOfBoundException**
  - **NullPointerException …**

Confinement logiciel quelle zone (de confinement) ?

- Défaillances concernées :
  tout ce qui ne compromet pas le matériel mais altère l'exécution
  - Valeurs incorrecte sur les interfaces des fonctions
  - Comportement aberrant de l'ordonnancement
  - Corruption de l'intégrités des donnée de programme

- Exemple : process Unix & sigsev
  - 1 process unix
  - 1 accès illicite à la mémoire du noyau (pointeur null)
  - Détection par la MMU (hw), et signalement au processus
  - Le processus se termine en indiquant au système d'exploitation (env du process) la cause de l'arrêt (SIGSEV)

Confinement par fonction

Idée : une fonction d'une bibliothèque = ZCE

Pb : comment signaler / confiner les erreurs.

- Pré-requis: taxonomie des erreurs
- Mise en œuvre :
  - encodage de l'état fonctionnel du composant dans la valeur de retour des fonctions
  - altération de la signature
- Pb : surcharge du type de retour ou altération de sa signature
- Exemple : code retour en C

Confinement par blocs

- Les exceptions :
  - Pré-requis :
    - encapsulation des traitements séquentiel dans des « blocs » (fonctions, accolades),
    - arrêt et déroutement de l'exécution d'un bloc sur réception d'un événement
    - taxonomie des erreurs + support pour le déroutement d'exécution
  - Composants : blocs → méthodes, fonctions, boucles
  - Mise en œuvre :
    - Utilisation du typage : 1 type d'exception=1 classe d'erreur
    - Définition d'une politique de propagation des signalements en l'absence de traitement explicite

Pb : souvent très mal utilisé malgré la puissance.

Décryptons une page de man

xterm

```
NAME
     pthread_mutex_lock -- lock a mutex

SYNOPSIS
     #include <pthread.h>

     int
     pthread_mutex_lock(pthread_mutex_t *mutex);

DESCRIPTION
     The pthread_mutex_lock() function locks mutex.  If the mutex is already
     locked, the calling thread will block until the mutex becomes available.

RETURN VALUES
     If successful, pthread_mutex_lock() will return zero, otherwise an error
     number will be returned to indicate the error.

ERRORS
     pthread_mutex_lock() will fail if:

     [EINVAL]           The value specified by mutex is invalid.

     [EDEADLK]          A deadlock would occur if the thread blocked waiting
                        for mutex.
:
```

Mode de défaillance non trivial

Le principe de l'enveloppe

- **Soit T1 f(Tp1,...Tpn) une fonction pouvant défaillir à cause de fautes d'interaction (mauvais paramètres).**
- **Pb : f ne peut signaler son état de fonctionnement….**
- **On crée ft_t**
  - **Type de retour status de fonctionnement**
  - **Utilisation d'une référence (en valeur) pour la valeur de sortie de f**
  - **Test des paramètre avant appel à f**
  - **Test de la valeur retournée par f**

L’enveloppe dans le détail

- **Soit T1 f(Tp1,...Tpn) une fonction pouvant défaillir à cause de fautes d'interaction (mauvais paramètres).**

- **Solution (en C, mais transposable)**

fonction ft :

  - **Tsig ft(Tp1,….,Tpn,T1*)**
  - **Tsig type utilisé pour l’état de fonctionnement**
  - **T1 * adresse vers emplacement du résultat**
  - **Lors d’un appel à ft, ft appelle f.**
  - **Valeur de retour de f copiée à l’adresse désignée par le dernier paramètre**

Confinement par moniteur externe

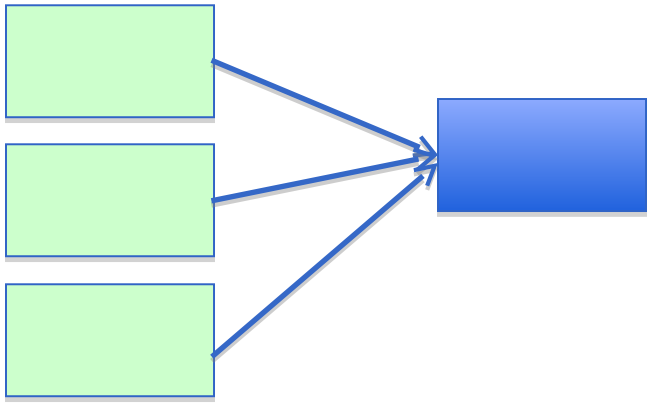
- Watchdog et interruption logicielle:
  - Défaillance constaté par l'absence de progrès dans l'exécution d'un programme
  - Causes possibles : boucle infinie, blocage sur un verrou pris et jamais restitué, récurrence trop longue …
  - Mise en œuvre : alarme, plus mécanisme de raz à des points critiques du code.

Recouvrement des erreurs 2 visions complémentaires

Corriger avant de poursuivre (on corrige l'erreur)

Recouvrement

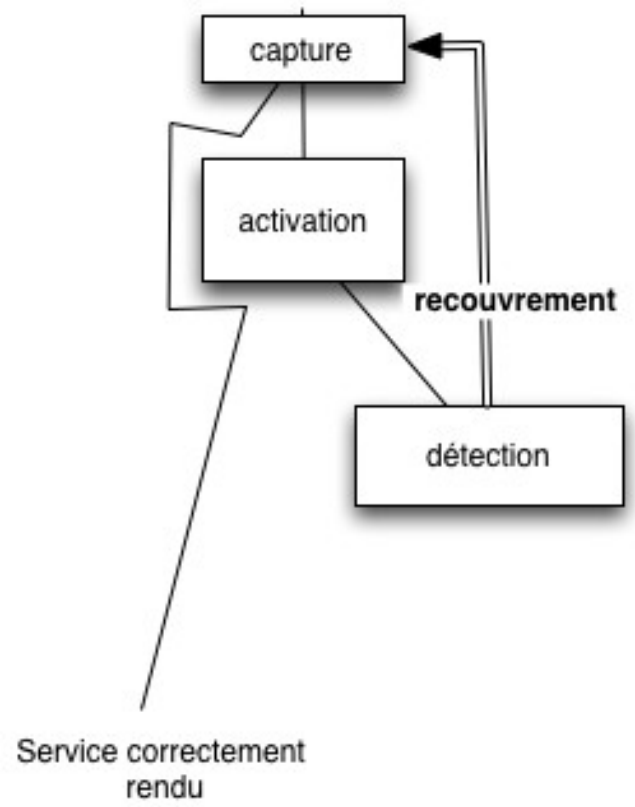
Choisir pour poursuivre (pas de correction nécessairement)

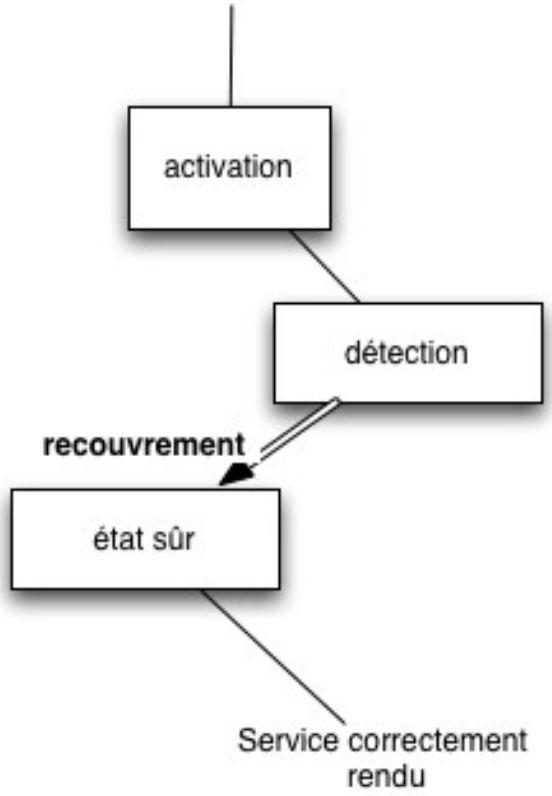


Masquage

Détection et recouvrement

- L’exécution du système est une séquence d’états
- **Détecter** la 1ere transition vers un état erroné
- **Reprendre** l’exécution depuis un état « correct »





Détection et recouvrement

Problème où trouver l'état correct ? :

- Dans le passé :
  - Hyp : il existe un mécanisme de capture d'états qui mémorise de manière répétée un état correct « récent »
  - Restauration sur détection d'erreur du dernier état sauvé

- Dans une liste d'états prédéfinis :
  - Hyp : Identification, a priori, d'états de poursuite d'exécution sûrs en fonction de l'erreur
  - Forçage d'une transition vers l'état de poursuite d'exécution correspondant à l'erreur détectée

Réflexions sur l'usage du recouvrement

- Dans l'approche « arrière », échec si :
  - la **faute est toujours active** et cause systématiquement l'erreur
  - la **latence de détection** est si grande que **l'état sauvegardé contient déjà une faute dormante ou que l'erreur s'est déjà propagé à l'extérieur du composant**
- Dans l'approche « avant » echec si
  - Les états de poursuite sûr sont erronés (mauvaise évaluation du lien erreur – état sûr).
  - La faute sous-jacente et toujours active et cause à nouveau l'erreur
- Comparaison sur une faute causée par un bug :
  - Le recouvrement arrière a de fortes de chances de rater si le bug est persistent
  - L'activation des états sûrs permet d'appeler un autre code.

Masquage et exécutions multiples

- Rappel : masquage possible car plusieurs instances de processus réalisant la fonction (e.g. le calcul)
- 2 visions : Vrai et faux parallélisme
  - Faux parallélisme == N instances exécutées en séquence
  - Vrai parallélisme == N instances exécutées en parallèle avec diffusion des données et récupération du résulat.

Intérêt de la redondance

- Pb: comment s'assurer après recouvrement que l'erreur ne revienne pas …
- Raisonnement alternatif « l'indépendance » :

  Il est peu très peu probable qu'une même faute s'active sur deux exécutions indépendantes d'une même fonction

- La redondance ~ création de données, de composants, de « résultats indépendants »

Intérêt de la redondance

- La redondance permet de masquer les erreurs
  - Vote-élection / reconstruction / moyenne
    - Consensus
    - Code correcteurs d'erreur
    - Capteur de pression consolidés
- Problème : comment caractérise-t-on l'indépendance ?
  - Physique : réplication et séparation (COM-MON)
  - Processus : développement diversifié (NVP

Design pattern

- Architecture flexible pour la SdF
- Approches :
  - Programmatiques (syntaxe et enchainement)
  - Architecturales (modèles de flux et indépendance)
- On peut souvent combiner les deux…

N-version programming (process)

- L'idée est la même :
  avoir N versions indépendantes d'un même système :
  - 1 seule spécification
  - N équipes indépendantes de développement (lieu, formation, hiérarchie …)
  - ⇒ Une faute a peu de chances de s'activer de la même manière dans 2 versions distinctes

La transformée de fourrier discrète :

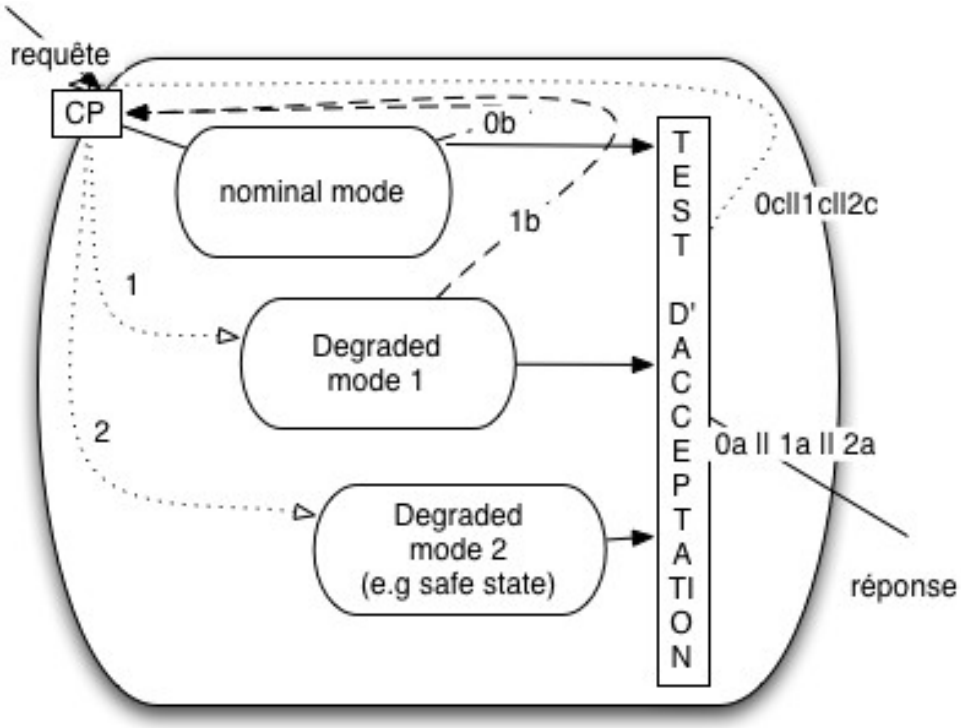
⇒ 2 algorithmes pour la calculer avec une sensibilité différente aux erreurs numériques

Recovery Blocks [Randall'95]

- Exemple d'intégration de NVP dans une application
- Un RB est un composant implémentant un service à la demande (client/server)
- La fonction est implémentée de N manières distinctes A1…AN
- Chaque version peut elle-même être un RB
- Il existe un test d'acceptation que doit pouvoir passer chaque alternative en l'absence d'erreur

Un exemple d'intégration de NVP : les Recovery Blocks

- A l'exécution :
  - Chaque requête génère la capture d'un point de reprise
  - La requête est transmise au premier alternat.
  - Si erreur ou échec du test, alors rétablir l'état du système et passer au bloc suivant
  - Dimension temporelle (watchdogs, timeouts)



CP : checkpoint; 1a,1b,1c chemins d'exécution possibles

Capture de contexte … et en vrai ?

- Principe :
  - Déterminer l'information représentative de l'état d'un système (variable, pile, ports de communication …)
  - Déterminer une méthode pour capturer un état cohérent de ces données
  - L'information enregistrée doit être suffisante pour recommencer l'exécution du système depuis cet état
- Obstacles
  - Usage de ressources systèmes et **effets de bords** (réservations de ressources, communications en cours)
  - **Implémentations concurrentes** du système
  - Quand doit-on capturer l'état ?

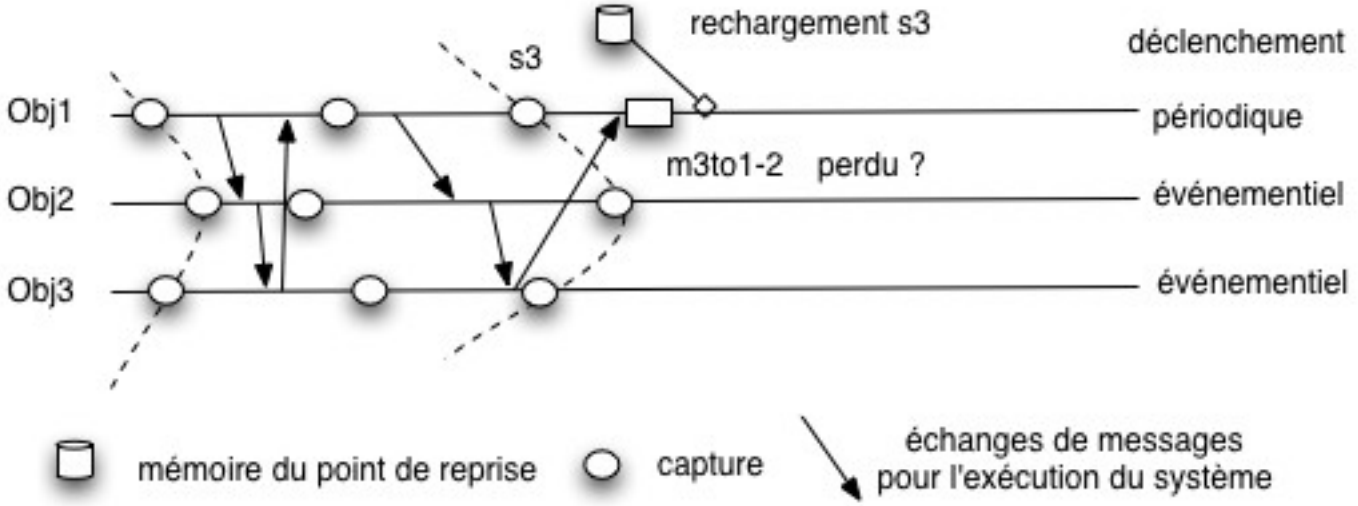
La vision centralisée

- Capture d'état ==
  capturer l'état d'une machine, d'un processus
- L'approche totale : recopier l'état de l'application et de sa plateforme d'exécution
  - Connaître l'OS (process / E-S / verrous)
  - Connaître le matériel (configuration des périphériques …)
- L'approche sémantique : identifier des états dans lesquels l'information utile est très faible
  - Connaître l'application à 100%

Le cas réparti et l'état inconsistant

- Principe : cas de figure désavantageux pour la capture d'états concurrents

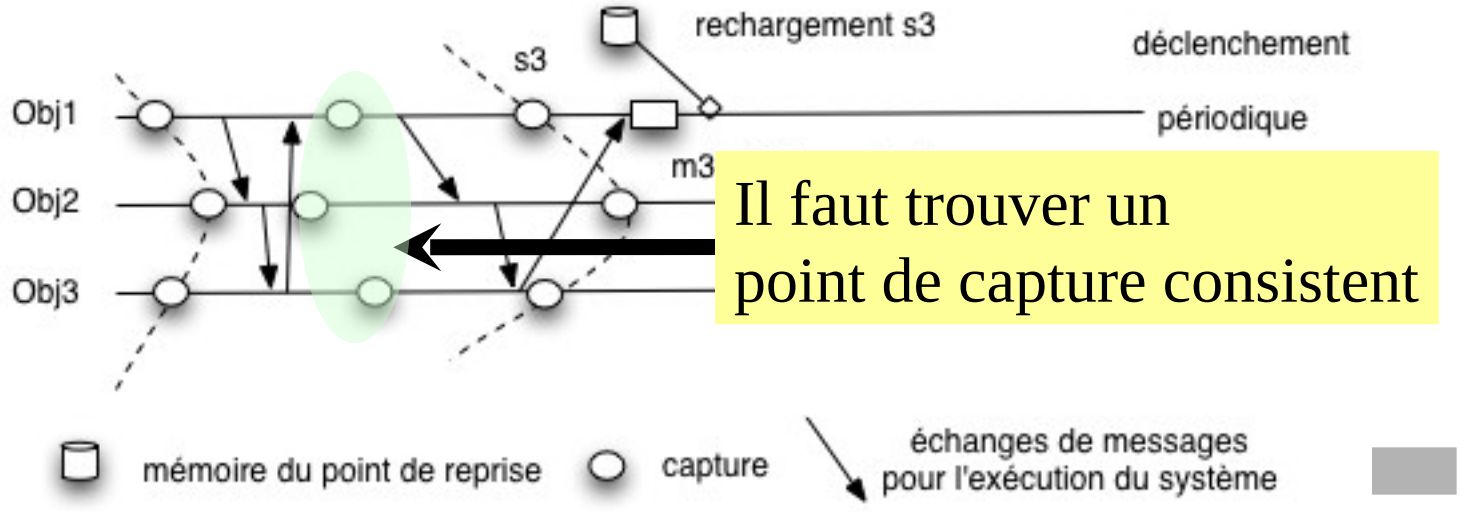
Le délai de transmission des messages fait qu'un site croit avoir transmis une donnée alors qu'un autre l'a ignorée à la suite d'une détection / reprise



Le cas réparti et la cause de l'effet domino

- Principe : cas de figure désavantageux pour la capture d'états concurrents

  Le délai de transmission des messages fait qu'un site croit avoir transmis une donnée alors qu'un autre l'a ignorée à la suite d'une détection / reprise



Il faut trouver un
point de capture consistent

48

Tolérance aux fautes matérielle

Intérêt de la redondance matérielle

- La redondance logicielle sans redondance matérielle suppose que le matériel est isolé du logiciel -> cela reste difficile à prouver
- La redondance matérielle permet de rendre un système informatique tolérant aux défaillances du logiciel sans isolation particulière
- Le système est modélisé comme système distribué : 1 ensemble de calculateurs communicant par message

Modèle de fautes dans un système distribué

- 4 gabarits classiques de fautes
  - Silence (perte définitive du service)
    Arrêt du matériel/blocage de l'OS
  - Omission (perte occasionnelle du service)
    lien réseau peu fiable ou timing très mauvais
  - Temporelle (mauvais timing)
    mauvaise estimation de WCET
  - Byzantine (service totalement incontrôlé)
    modélisation classique pour la sécurité
- Haut niveau d'abstraction
  système=réseau

1 faute == défaillance d'un site de calcul

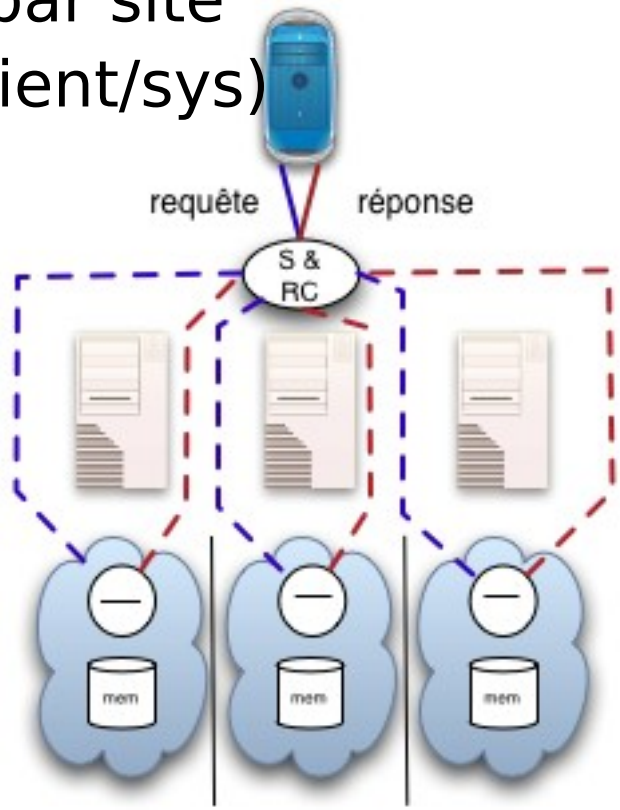
Stratégies de réplication

- Principe :
  - Déployer de manière concurrente plusieurs fois la même fonction
  - Utiliser un contrôleur pour
    - Piloter l'exécution de ces répliques
    - Assurer la transmission du résultat
- 3 gabarits (motifs de conception) :
  - Réplication active
  - Réplication passive
  - Réplication semi-active

Comment et avec quelles ressources ?

Réplication Active

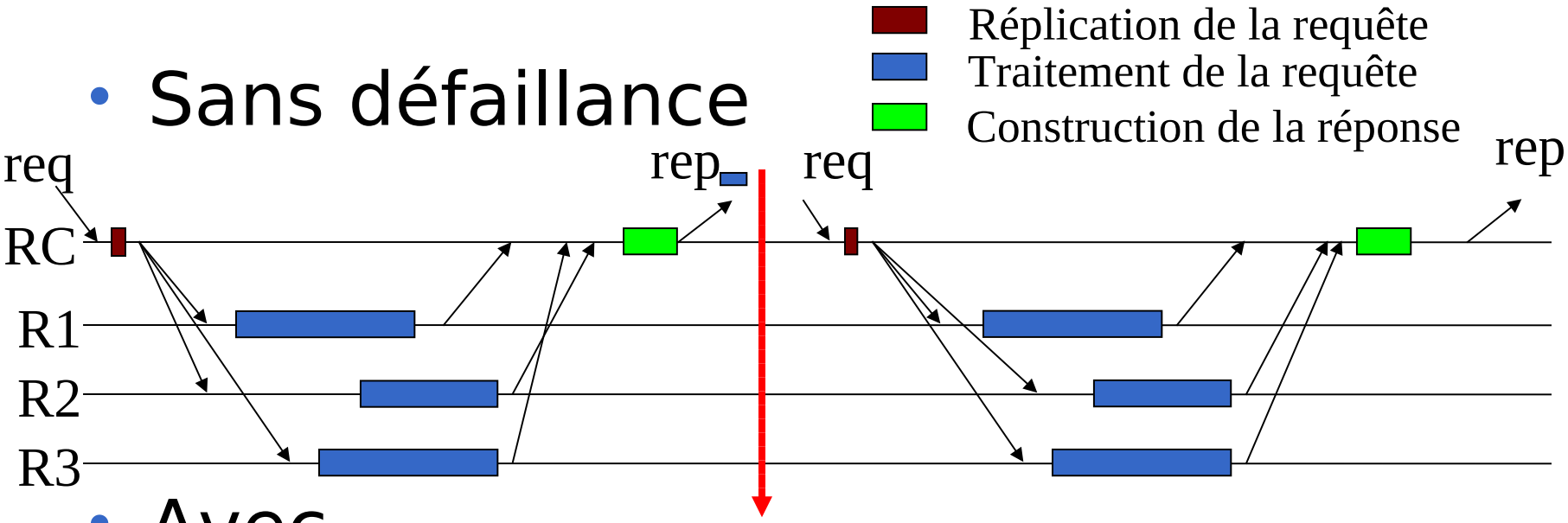
- N calculateur => 1 application par site
- 1 contrôleur pour interpréter (client/sys)
- Décision: vote
- Communications : diffusion des requête + agrément sur le résultat
- Déroulement
  1) Envoyer la requête à tous
  2) Chaque site exécute son service
  3) Construction de la réponse par vote majoritaire



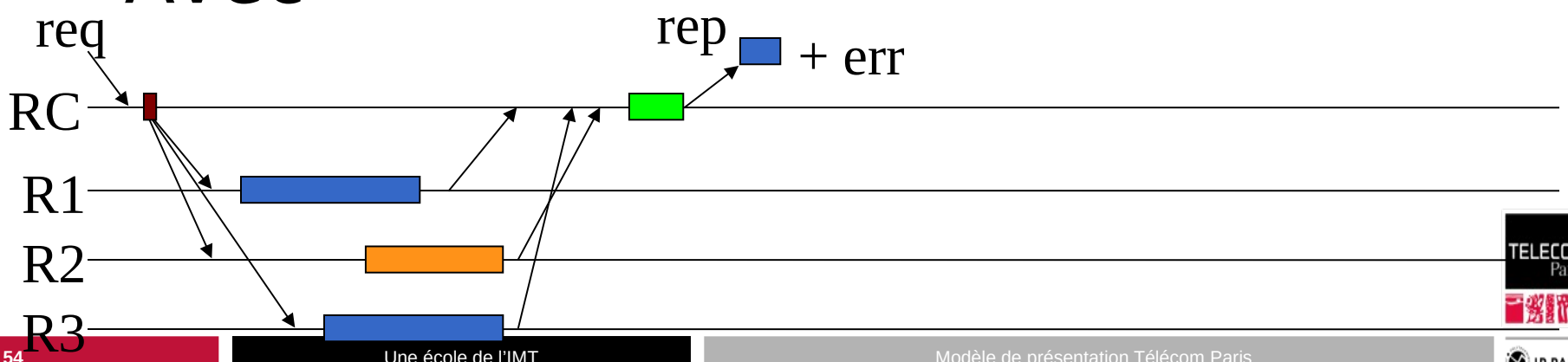
S & RC : vote et contrôle des répliques

Communication sans et avec défaillance

- Sans défaillance

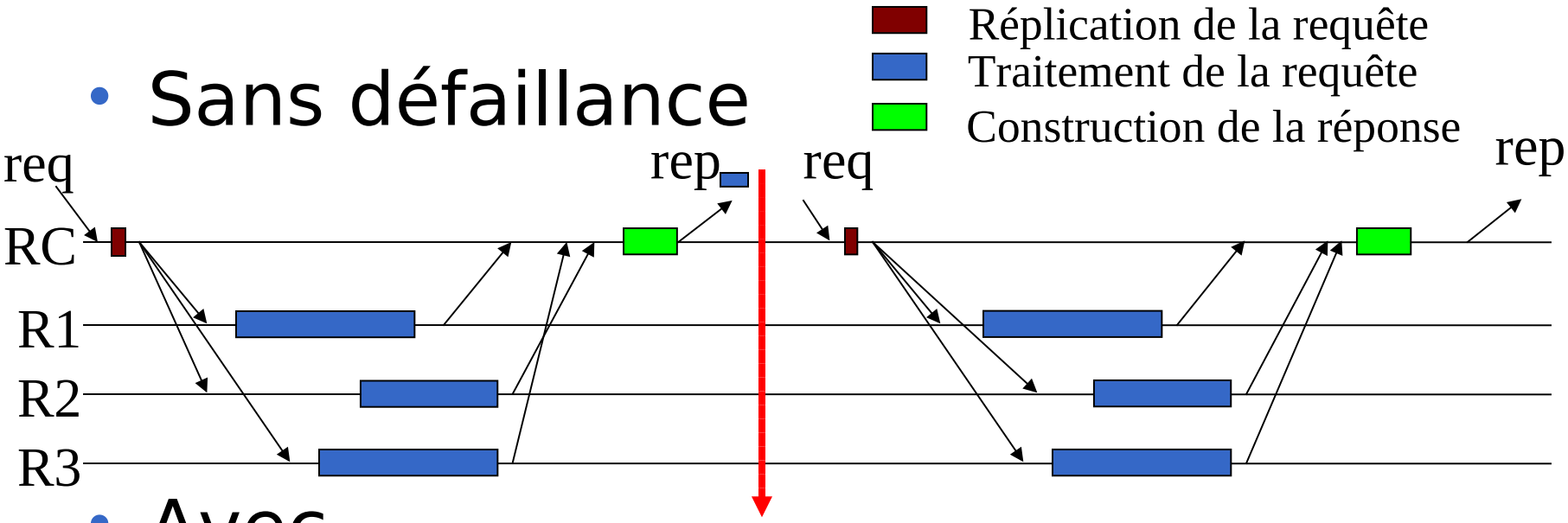


- Avec

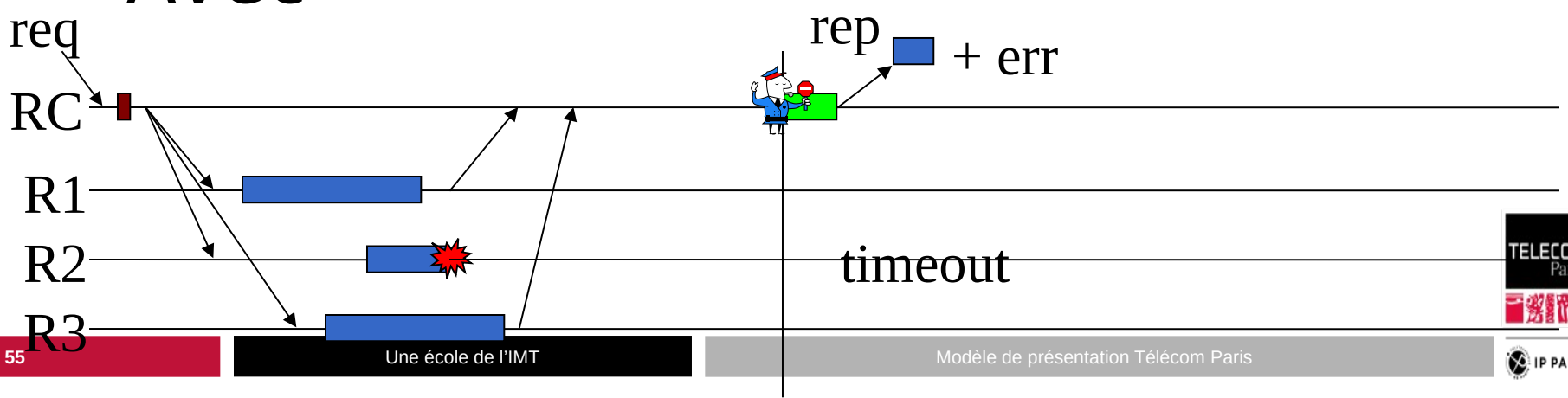


Communications sans et avec défaillance

- Sans défaillance



- Avec



Caractéristiques de la réplication active

- Plusieurs répliques actives en même temps
  Il faut définir la condition de disponibilité :

Quand produit t on une valeur ?

- Cas n°1 : 2 réponses identiques parmi 3 doivent être reçues
- Cas n°2 : dès que l'on reçoit une valeur on la transfère si c'est la seule reçue, sinon on applique n°1.
- Cas n°1 dispo = fiabilité, Cas n°2 c'est différent ! (i.e. on préfère produire un résultat même si il peut être faux.

Caractéristiques de la réplication active

Modèle de faute toléré pour N répliques :

N/2-1 fautes Byzantines, N-1 crash

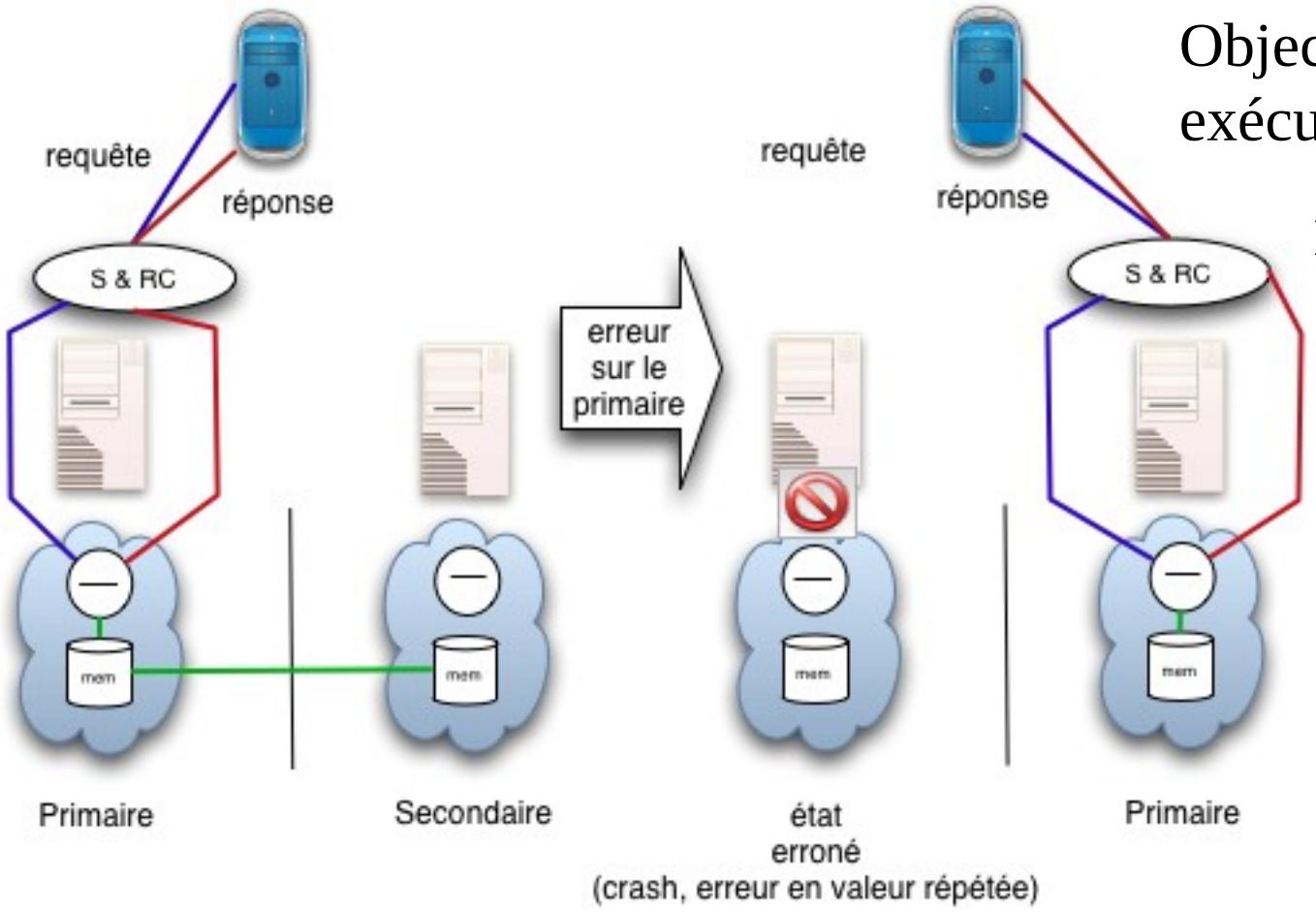
Détection réussie si au moins 1 correct

Pour qu'une faute entraîne une erreur non détectée, elle doit s'activer sur :

le voteur ou sur plus de N/2 répliques

Δt(régime normal / rétablissement) ~ nul

Réplication Passive



Objectif : éviter les exécutions concurrentes

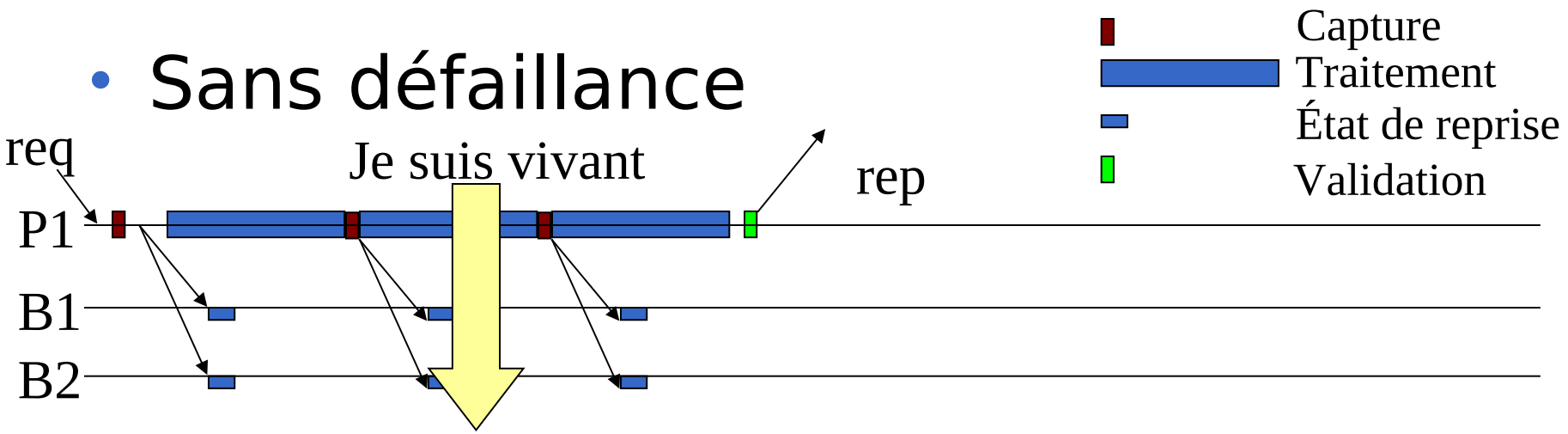
1 seule exécution à la fois

Communication : Transfert d'un état

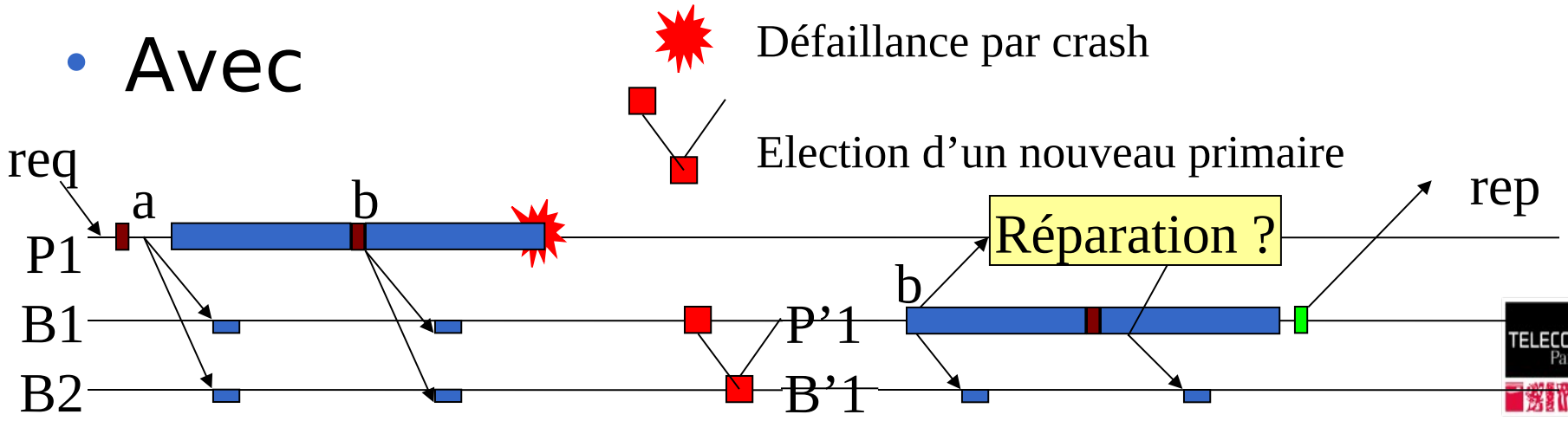
Le primaire doit capturer son état pour l'envoyer au secondaire

Communications et opérations significatives

- Sans défaillance



- Avec



Caractéristiques de la réplication passive

- 1 seul site exécute le service, jusqu'à la détection d'erreur (défaillance du primaire)
- Après détection d'erreur, basculement vers une réplique (nouveau primaire)
  - Identification du crash par « battements de cœur »
  - Élection d'un nouveau primaire
- Hypothèse forte : le primaire n'a qu'un seul mode de défaillance, le crash.
- Sans réparation la fiabilité service est identique à priori à la disponibilité service

Analyse : Caractériser l'état de fonctionnement

- Chaque réplique à un état de fonctionnement
  - Ok / byzantin
  - Ok / wrong result
  - Ok / crash
- On peut définir : disponibilité / fiabilité % répliques
- Il doit être possible de caractériser l'état de l'architecture (dispo / fiabilité) par une formule

Caractériser l'état un exemple

- Triplication (3 répliques) active
  - Cas 1 : 1 mode de défaillance - byzantin
  - Cas 2 : 1 mode de défaillance – crash
- Soit N le nombre de répliques défaillantes
- Caractériser dans chaque cas : l'état de disponibilité puis fiabilité

Hypothèse:

- le système est considéré disponible tant qu'il produit un résultat.
- Le voteur ne réalise pas de filtrage.

Modes de fonctionnement dégradés

- Modes dégradés :
  État du système tel qu'une partie du système est dysfonctionnelle sans pour autant compromettre l'intégralité du service
- Mode fail-safe: mode dégradé n'assurant aucun service mais garantissant la sûreté des biens et personnes

Conclusion

- Tolérance aux fautes = domaine établi
  - Des motifs de conception utilisés mais à adapter à chaque application
  - Pas de dogme mais une prise de conscience
    - La Sûreté de fonctionnement est difficile à obtenir
    - Il y a nécessairement des compromis à faire mais « les bons »
- Importance des zones de confinement
  - Savoir si les défaillances sont détectées/signalées
  - Lister les modes de défaillances connus, indiquer la cause si externe

Acronymes

- SdF : sûreté de fonctionnement
- TaF : Tolérance aux Fautes
- TMR, NMR : triple/ N - moduar replication
- RB : recovery blocks ou Rollback…
- ND : non - déterminisme (ou non déterministe)
- SR : stratégies de réplication

Références

Concepts de la SDF :

- PA Lee, T Anderson, JC Laprie, A Avizienis, ... - 1990 - Springer-Verlag New York, Inc. Secaucus, NJ, USA
- A. Avizienis; J. Laprie; B. Randell & C.E. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing“, IEEE Transaction Dependable Sec. Computing 2004, 1, 11-33
- J.C. Laprie, “Guide de la sûreté de fonctionnement (2° Ed.)“, ed. Lavoisier, 330p

Techniques de mise en place de la TaF

- B. Randel and J. Xu, "The Evolution of the Recovery Block Concept," Software Fault Tolerance, M.R. Lyu, ed., John Wiley & Sons, New York, 1995, chapter 1
- Elnozahy, E. N., Alvisi, L., Wang, Y., and Johnson, D. B. 2002. A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. 34, 3 (Sep. 2002), 375-408. DOI= http://doi.acm.org/10.1145/568522.568525
- [Cristian91] F. Cristian, "Understanding fault-tolerant distributed systems", Communications of the ACM, 34(2), February 1991
- [KD+89] Kopetz, Damm, Koza, Mulazzani, Schwabl, Senft, Zainlinger. "Distr[illegible]time systems: the Mars approach", IEEE Micro, pp. 25-40, February 1989

Références

- Xavier Défago and André Schiper, “Semi-passive replication and lazy consensus“, Journal of Parallel and Distributed Computing, 64(12):1380–1398, December 2004.
- Koo, R. and Toueg, S. Checkpointing and rollback-recovery for distributed systems. In Proceedings of 1986 ACM Fall Joint Computer Conference (Dallas, Texas, United States). IEEE Computer Society Press, Los Alamitos, CA, 1150-1158, 1986.
- Powell, D. 1994. “Distributed fault tolerance—lessons learnt from Delta-4“. In Papers of the Workshop on Hardware and Software Architectures For Fault Tolerance : Experiences and Perspectives: Experiences and Perspectives, M. Banâtre and P. A. Lee, Eds. Springer-Verlag, London, 199-217.

Algorithmique distribuée et prise de décision :

- Lamport, Leslie; Marshall Pease and Robert Shostak, "Reaching Agreement in the Presence of Faults". Journal of the ACM 27 (2): 228--234, April 1980
- Xavier Défago and André Schiper, “Semi-passive replication and lazy consensus“, Journal of Parallel and Distributed Computing, 64(12):1380–1398, December 2004.
- Chandra, T. D. and Toueg, S. 1996. Unreliable failure detectors for reliable distributed systems. J. ACM 43, 2 (Mar. 1996), 225-267.

Conception de stratégies de réplication :

- Schneider, F. B. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. ACM Comput. Surv. 22, 4 (Dec. 1990), 299-319.