

Tolérance aux Fautes des Systèmes Informatiques

Thomas ROBERT
INF 342

Motivations (1/3)

- 2 points de vue d'un système embarqué :
 - ♦ Un système informatique matériel + logiciel
 - ♦ Un système physique capteurs/actuateurs support de communication



Si le système fonctionne mal quelles sont les
Conséquences ?

Motivations (2 / 3)

- La conception et la réalisation dépendent de :
 - ◆ La nature de l'environnement considéré
 - ◆ La description du service à rendre
- A l'usage, l'environnement entrave toujours le bon fonctionnement du système
 - Environnement imprévisible
 - Le facteur humain à l'usage et à la conception
- Quelle **confiance**, quelles qualités peut on **associer** à un tel **système** ?

Motivations (3 / 3)

Sciences et techniques de l'ingénierie

-> Sûreté de fonctionnement

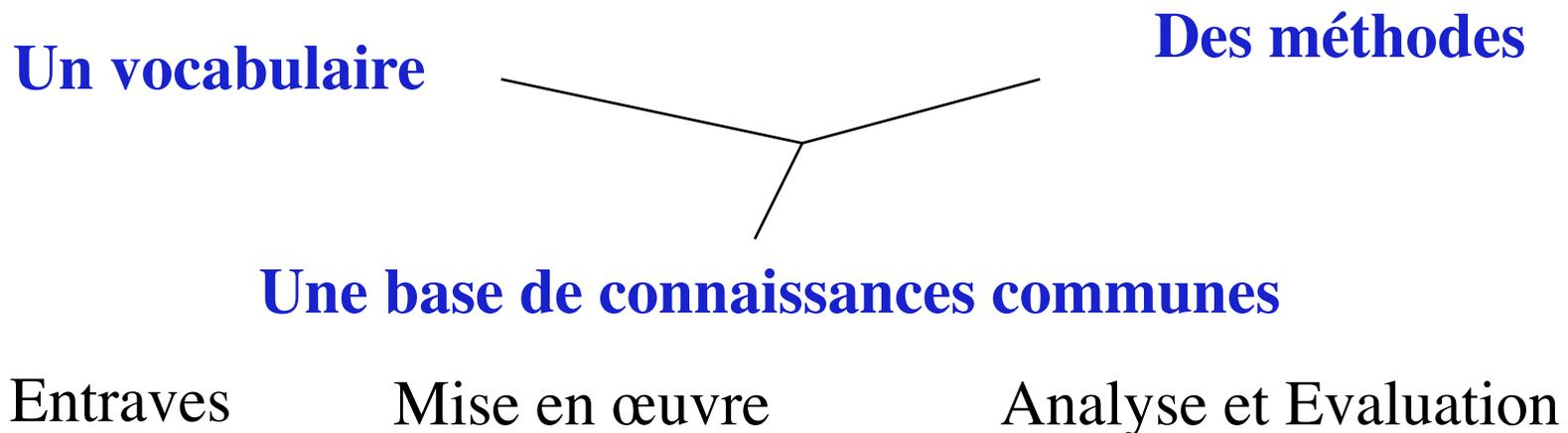
*Vous en faites sûrement un peu tous les jours,
Le but est de rationaliser cette pratique*

Plan du cours

- **Intro Sûreté de fonctionnement**
- Savoir faire élémentaire en TaF
- TaF dans le cadre temps réel
- Introduction au TP
- Bilan

La sûreté de fonctionnement en informatique...

- [Avizienis et al'04]
“*l'étude et mise en œuvre de « services » en lesquels on puisse placer une confiance justifiée*”



La vision système (def)

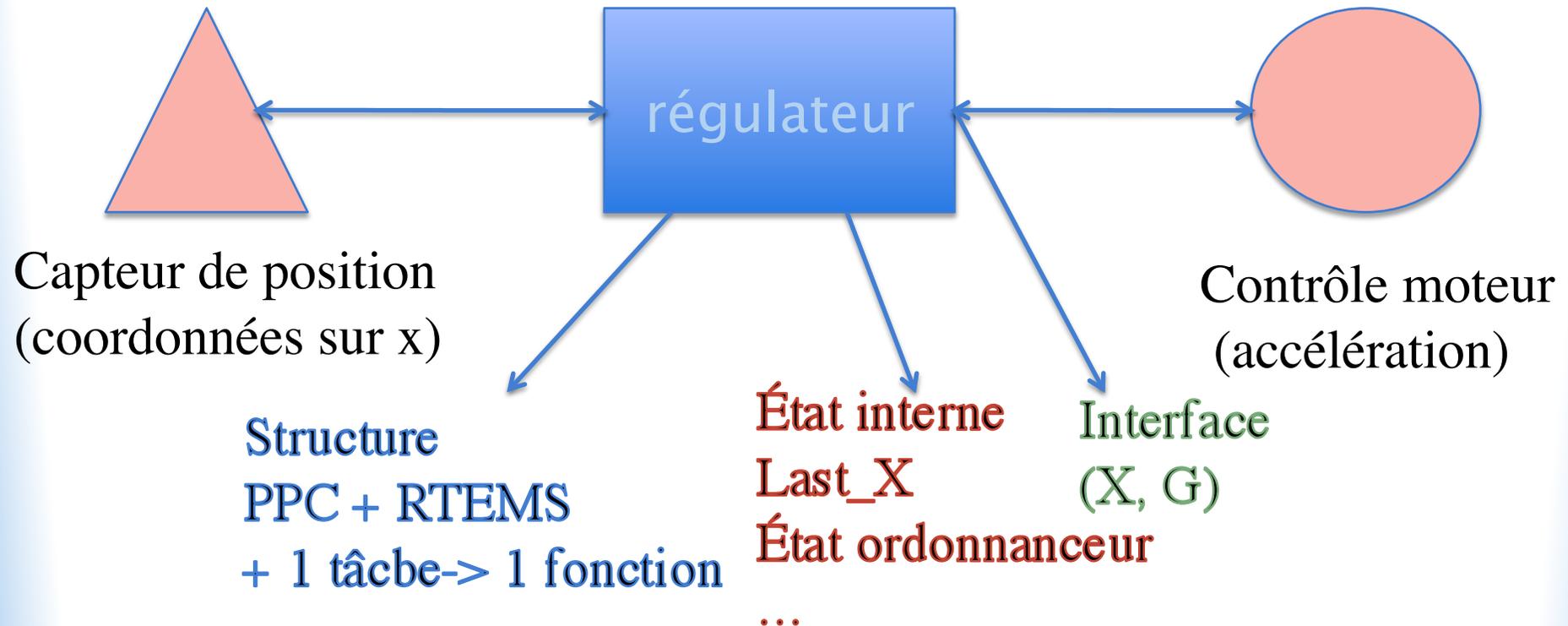
Un système: unité de description permettant de distinguer l'objet d'étude de son environnement

- ◆ Une structure ; description des éléments à priori immuables du système (architecture)
- ◆ État : information variable au cours de la vie opérationnelle du système
 - État interne: fraction non observable de l'état du système
 - Interface : fraction de l'état du système partagée avec son environnement (E/S)

La vision système par l'exemple

- La vision système ::
structure + comportement + ...

Exemple : Le régulateur de vitesse du métro



Spécification de l'attendu

- Les points de vue possibles :
 - ◆ Fonctionnel $F(x,y)=z$
 - ◆ Non fonctionnel Deadline =T0, Mem<M0
 - ◆ Structurel ...
- Les moyens :
 - ◆ Langage naturel et fichiers words
 - ◆ Semi formels (UML, SADT, SysML)

Syntaxe complète et sémantique mal déterminée

 - ◆ Langages Formels : Automates, langages synchrones, B logiciel, logiques, AADL ...

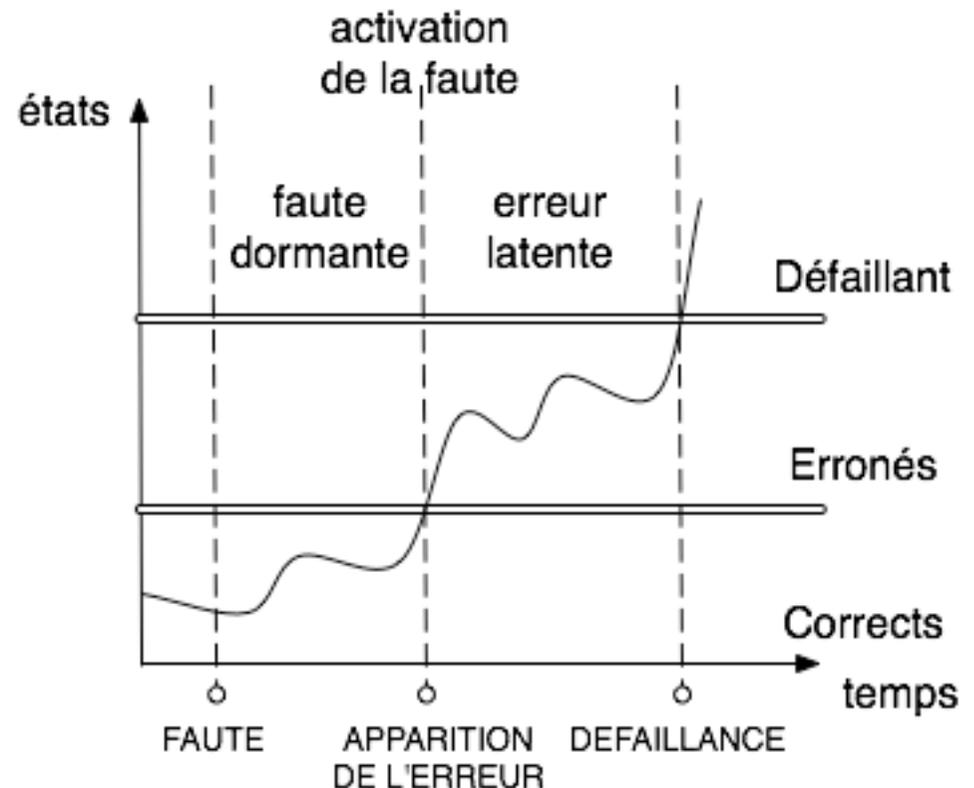
Les entraves (def)

- Un vocabulaire pour comprendre :
 - ◆ Défaillance : écart observable entre le service attendu et le service rendu
 - ◆ Erreur :
Tout ou partie de l'état interne du système pouvant causer sa défaillance
 - ◆ Faute :
Cause de l'apparition d'une erreur
- Quel est le lien entre ces concepts?

La chaîne faute/erreur/ Défaillance

- Evénements :
 - ◆ Faute
 - ◆ Activation
 - ◆ Détection
 - ◆ Défaillance
- Etats :
 - ◆ Corrects (OK)
 - ◆ Erronés (?)
 - ◆ Défaillants (KO)

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



Les attributs

- Par où commencer ? => Les attributs
 - ◆ Disponibilité (availability)
 - ◆ Fiabilité (reliability)
 - ◆ Sécurité–innocuité (safety)
 - ◆ Intégrité (integrity)
 - ◆ Confidentialité (confidentiality)
 - ◆ Maintenabilité (maintainability)

.... En pratique sur un exemple

Définitions attributs

- Disponibilité : capacité du système à offrir tout ou partie du service (n'est pas « visiblement » défaillant)
- Fiabilité : caractérise la capacité du système à effectivement délivrer un service correct (lorsqu'il est actif)
- Sécurité–innocuité (safety ou sûreté) caractérise la maîtrise/connaissance des conséquences d'une défaillance

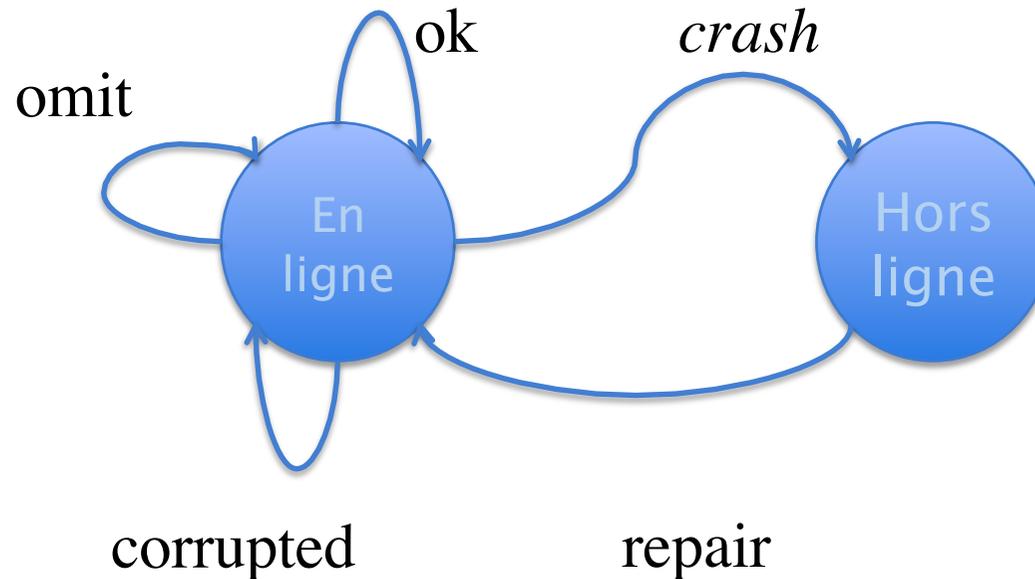
Définitions attributs

- Intégrité : caractérise la capacité du système à détecter ou empêcher toute altération non autorisée de sa structure ou de son état
- Maintenabilité : caractérise la capacité du système à faire évoluer sa structure ou son état pour faciliter le traitement des fautes ou le passage de l'état défaillant à l'état fonctionnel

Attributs en pratique (1)

- Système : un routeur réseau
 - ◆ 2 états: en ligne, hors ligne
 - ◆ 3 comportements possibles dans l'état en ligne
 - Ok
 - Corrupted packet
 - Omit
 - ◆ 2 composants :
 - Alimentation (réparable)
 - Mémoire (non réparable)

Attributs en pratique (2)



Disponibilité -> en-ligne / hors-ligne

Fiabilité -> probabilité de la transition Ok depuis « en ligne »

Intégrité -> détection de corrupted

Sécurité innocuité -> ??

Maintenabilité -> condition de succès de repair.

Les défaillances et leur analyse

- Une infinité de manières de défailir
- Une classification pour comparer (typage)
⇒ 4 attributs pour les définir
 - ◆ Domaine (nature de la défaillance)
 - ◆ Observabilité (conscience de la défaillance)
 - ◆ Homogénéité (perception de la défaillance)
 - ◆ Classification des Conséquences
- Un calculateur s'arrête
(content +timing, unsignaled, consistent ,major)

Les moyens pour obtenir un niveau voulu de SdF

- 4 Approches complémentaires
 - ◆ **Prévention des fautes :**
Méthodes destinées à empêcher l'occurrence même des fautes
 - ◆ **Élimination des fautes :**
Méthodes de recherche et de suppression des fautes de conception et développement
 - ◆ **Prévision des fautes :**
Analyse de la fréquence ou du nombre de fautes et de la gravité de leurs conséquences
 - ◆ **Tolérance aux fautes :**
Mécanismes au sein du système destinés à assurer les propriétés de SdF voulues en présence des fautes

Plan du cours

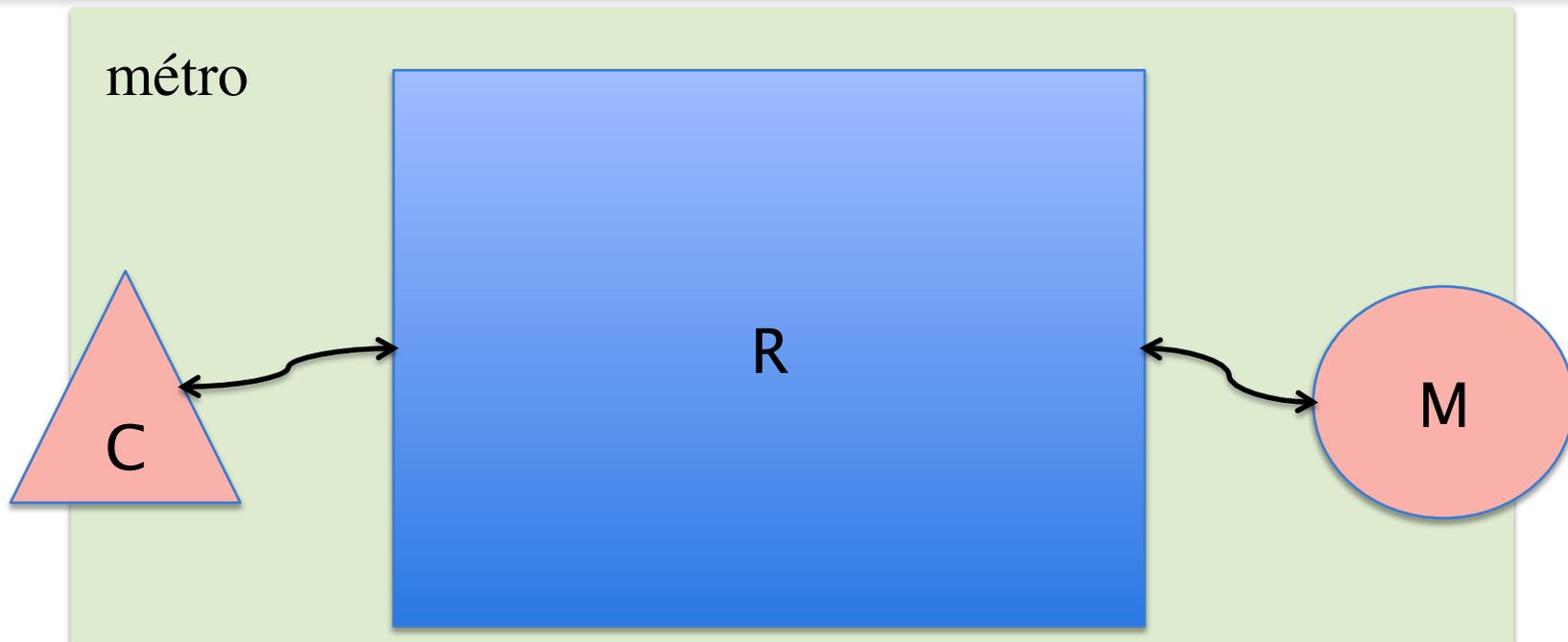
- Intro Sûreté de fonctionnement
- Savoir faire élémentaire en TaF
- TaF dans le cadre temps réel
- Introduction au TP
- Bilan

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

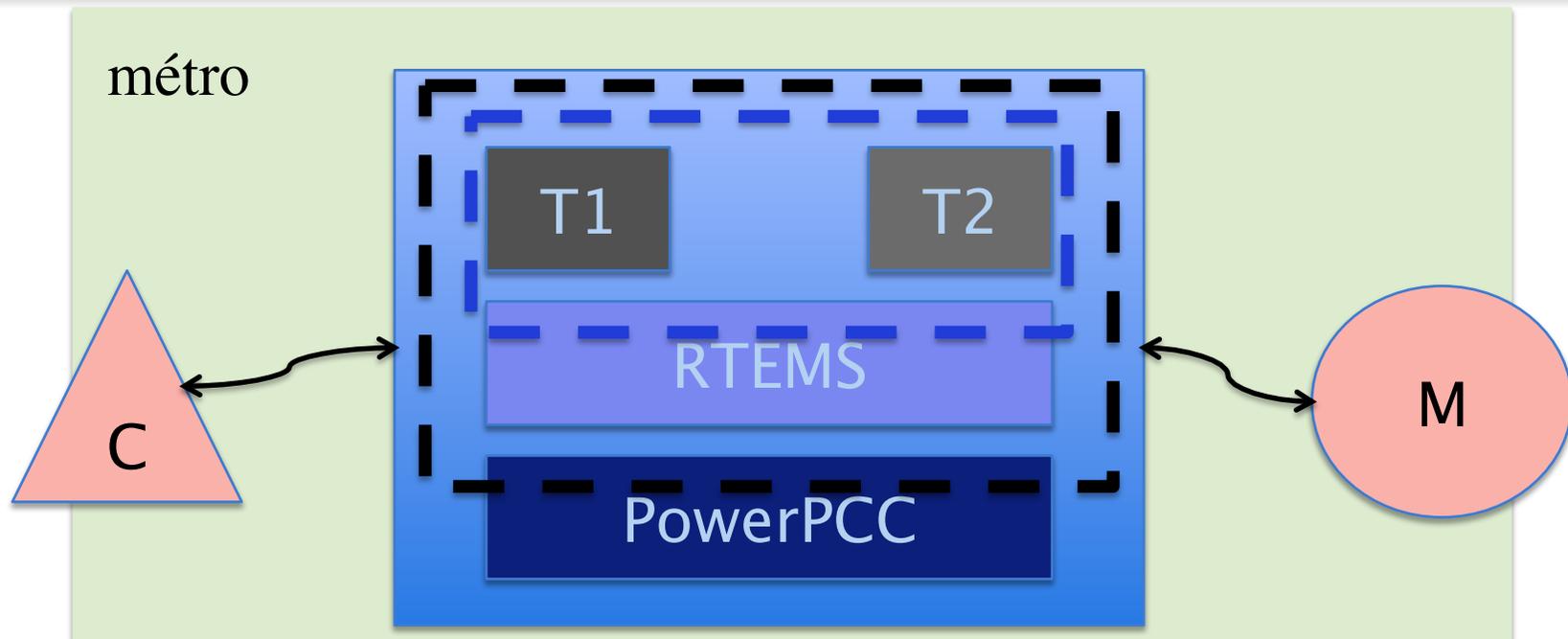
Défaillances, fautes et propagation des erreurs



- 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

ATTENTION IL NE FAUT PAS S'ARRÊTER A L'EVIDENCE

Propagations Logiciel/Matériel



- Fautes d'interaction sur des niveaux d'abstraction différents
- Chaque système est inclus dans l'autre : Propagation « verticale »
- La défaillance de T1 peut causer celle de RTEMS ou du processeur

Le principe de la TaF

- Empêcher les défaillances :
 1. Éviter l'activation des fautes
 2. Éviter qu'une erreur n'entraîne une défaillance
- Comment faire 2) ?
 - ◆ Détection & Rétablissement
 - ◆ Masquage (ou compensation)
- Concrètement ... cela va coûter du temps et changer la nature du système
- 1) nécessite souvent une altération en profondeur de la structure du système...

Zone de confinement des erreurs

- **Définition** interface d'un système munie de mécanismes de protection empêchant une erreur de se propager (ie de contaminer l'environnement du système)
- Pré requis :
 - ◆ Description architecturale définissant la décomposition du système en sous-systèmes dépendant les uns des autres
 - ◆ Un description des défaillances possibles associées à chaque système et sous-systèmes.
 - ◆ Un modèle de fautes précisant l'apparition de l'erreur dans le système.
- Mise en œuvre :
 - ◆ Détection et recouvrement
 - ◆ Compensation

Principe de fonctionnement

- Détection des erreurs :
 - ♦ Test de vraisemblance
si ($\text{sqrt}(x) < 0$) alors erreur
 - ♦ Comparaison et calcul de signature
Exécuter 2x la même tâche et comparer, CRC
- Traitement des fautes
(diagnostic, isolation, reconfiguration)
- Traitement des erreurs :
 - ♦ Recouvrement de la défaillance (ressource : temps)
 - ♦ Compensation de l'erreur (ressources : temps mémoire matériel réseau)

TaF logicielle

Détection et composant auto-testable

- But : éviter la propagation silencieuse des erreurs dans les différents sous systèmes
- Munir chaque composant du système d'un mécanisme de détection signalant les erreurs non masquables
- Attributs d'un détecteur :
 - ♦ Correction : erreur|alarme / alarme
estimation des faux positifs
 - ♦ Précision : alarme|erreur / occurrence d'erreur
estimation des faux négatifs
 - ♦ Latence :
temps de détection – temps d'activation

Confinement logiciel & logique

- Défaillances concernées :
tout ce qui ne compromet pas le matériel
 - ◆ Valeurs incorrecte sur les interfaces des fonctions
 - ◆ Comportement aberrant dans le « run-time »
 - ◆ Corruption de l'intégrités des donnée

Il faut identifier où placer les zones de confinement dans l'architecture : ça dépend des ressources disponibles

- 1 Composant logiciel « selfchecking » utilise souvent des ressources du système qui le contient

Exemple :

process Unix & détection d'accès illicites à la mémoire

ressource : matériel MMU, signaux et handlers

Surcharge des interfaces de programmation

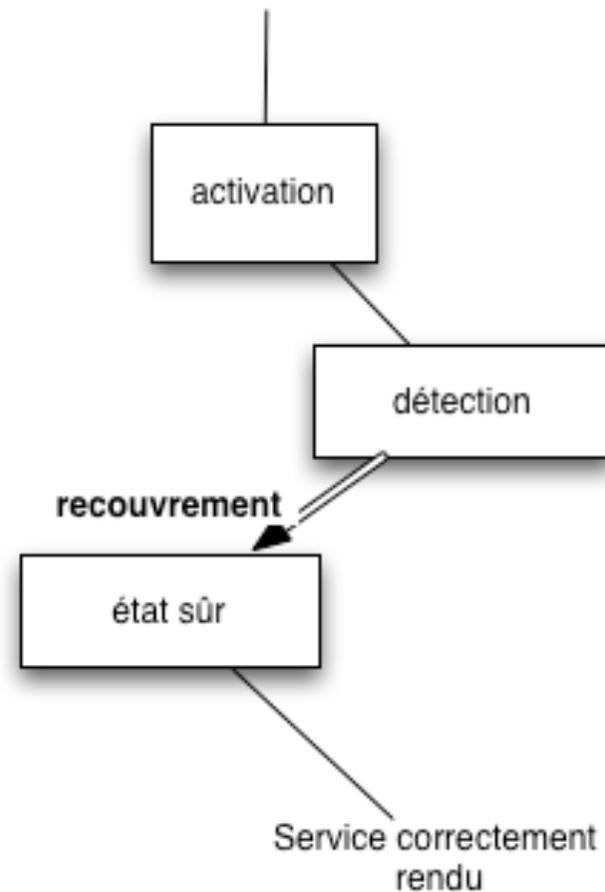
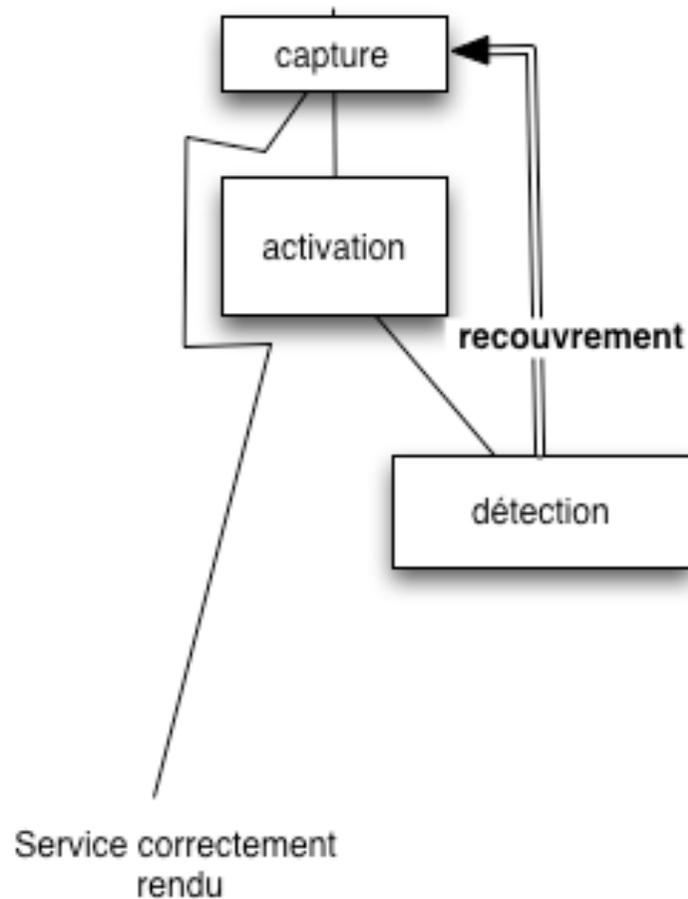
- Détection et confinement dans les APIs :
 - ◆ Moyens : Exceptions & code de retour
 - ◆ Implémentation sommaire de rétablissements
 - ◆ Evite les cas de propagation les plus sévères
- Code retour en C, le cas du mutex :
 - ◆ Son rôle : implémenter le verrou protégeant une section critique => 2 fonctions ...lock() et ...unlock()
 - ◆ pthread_mutex_lock() : attente sur un verrou local
 - Il existe une description de la nature des paramètres autorisés
 - Il existe une description des modes de fonctionnement de cette fonction

Détection et recouvrement

- Principe :
 - L'exécution du système est une séquence d'états
 - En remplaçant un état erroné par un état correct, il est possible d'effacer l'apparition de l'erreur
- 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'état 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

Détection et recouvrement

- Déroulement :



Traitements Ad'hoc

« en ligne »

- Si le traitement peut se faire localement « If then else » ou « while »
- Sinon Il faut monter d'un ou plusieurs niveaux dans l'architecture pour réagir
- Exemple Unix, les signaux :
 - ◆ SIGALARM
 - ◆ Segmentation fault
- La morale : **toujours inclure un minimum de gestion d'erreur quand vous utilisez une fonction des API classiques ... car elle possède une spécification de ses modes de défaillance**

Toujours signalés avec un comportement par défaut, l'arrêt du processus

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 erreur dormante ou que l'erreur s'est déjà propagé à l'extérieur du composant**
- Dans l'approche « avant » échec 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 forte de chance de raté si le bug est persistant
 - ♦ L'activation des états sûrs permet d'appeler un autre code...

La notion de modèle de faute

- Une faute == cause adjugée ou effective d'une erreur et donc d'une défaillance
- Quelques exemples :
 - ♦ **Corruption de la mémoire physique**
-> les valeurs peuvent être modifiées aléatoirement environnement
 - ♦ **Faute de développement, pointeur mal initialisé**
-> donnée incorrecte / boucle infinies Production
 - ♦ **Défaillance du matériel ou sous-système**
-> arrêt complet du système, comportement aléatoire Interaction

Modèles de fautes

- Une classification des fautes :
 - ◆ Phase de création ou occurrence
 - ◆ Positionnement % système
 - ◆ Source (humaine / « naturelle »)
(si source humaine)
 - Intention
 - « Compétence » (Délibéré/Accidentel/incompétence)
 - ◆ Persistance (permanente / transitoire)
- Il existe des gabarits si le système est vu comme un ensemble d'applications communicant par messages
- ATTENTION : cela ne résout pas tout !!

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 (bis)

- 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)

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 :

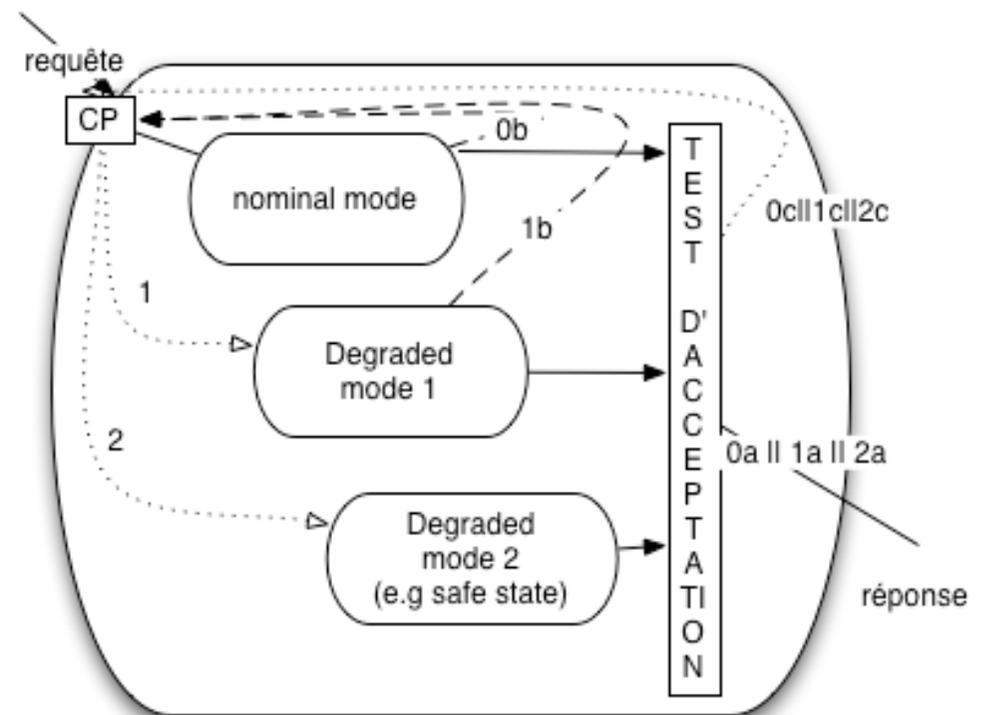
- ⇒ 2 algorithmes pour la calculer avec une sensibilité différente aux erreurs numériques

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 $A_1 \dots A_N$
- 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 ?

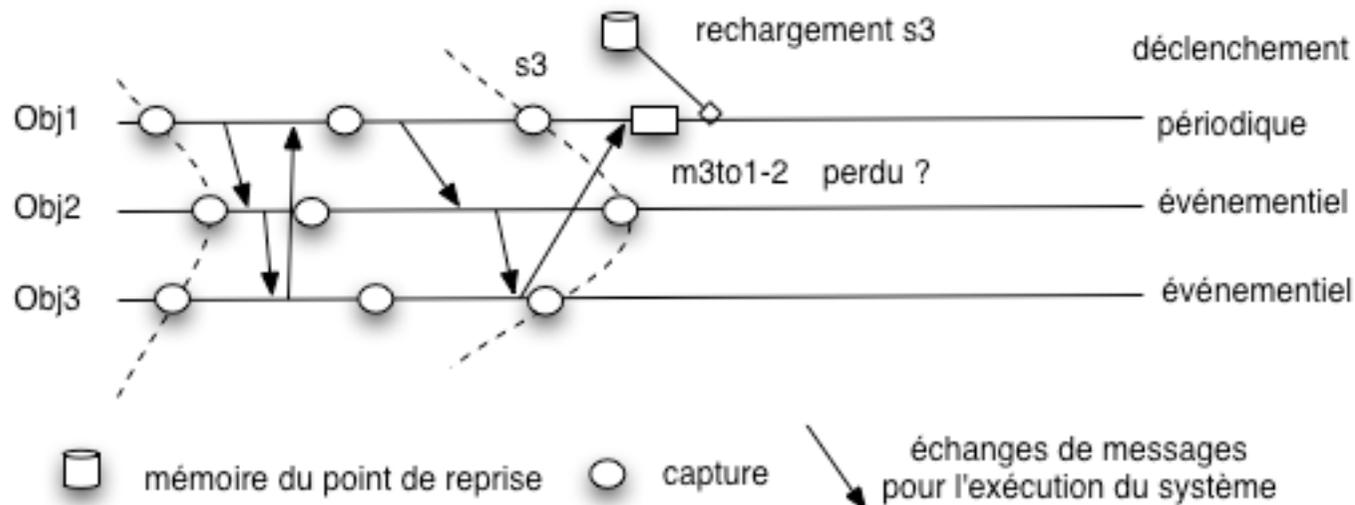
La vision centralisée

- 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

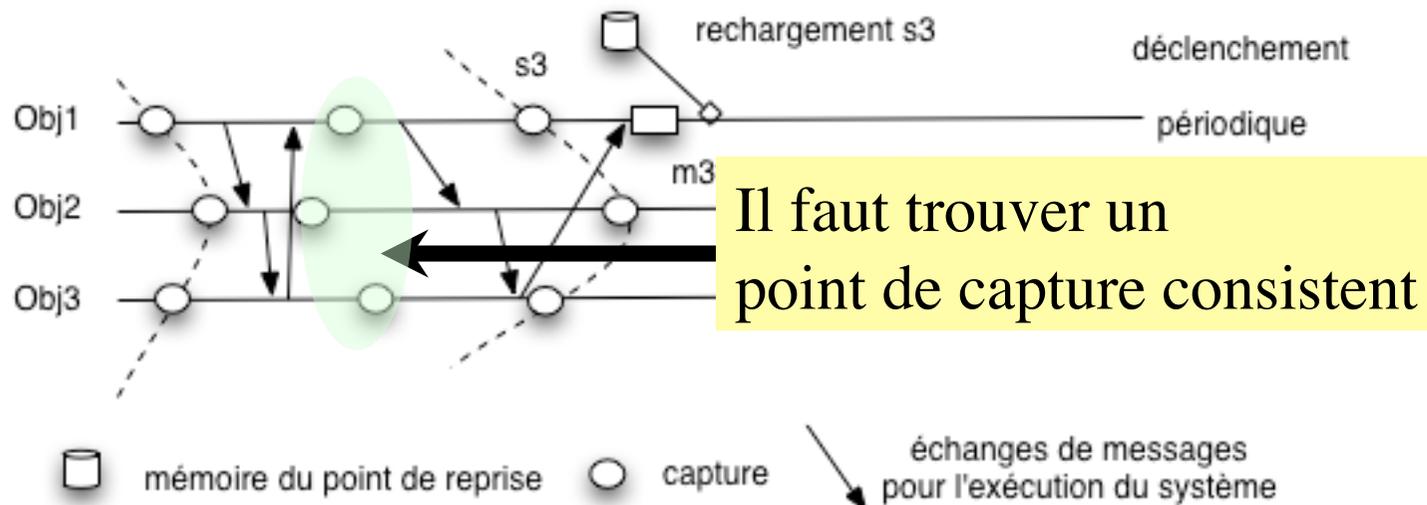
Le délai de transmission des messages fait qu'un site croit avoir transmis une donnée alors qu'un autre l'a ignorée à la suite d'une détection / reprise



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

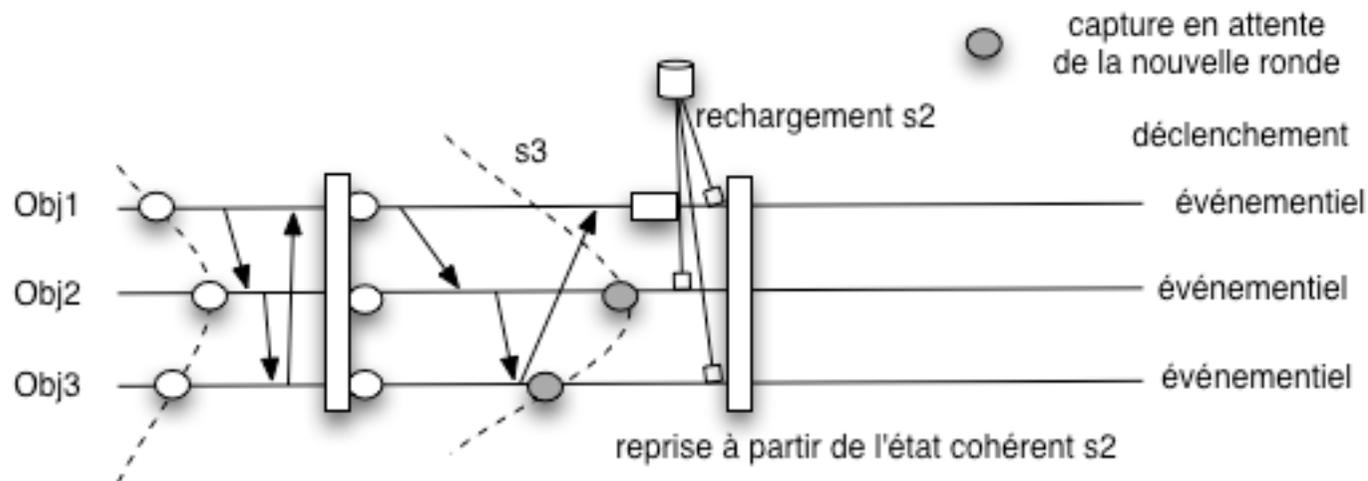


Les solutions pour la capture d'états concurrents

- La question : s'autorise-t-on à modifier en profondeur l'exécution du système ?
- Non – Utiliser une mémoire et un algorithme de reconstruction d'état global
 - Chaque site crée les points de capture qu'il désire.
 - Chaque point de capture est accompagné d'un historique des messages reçus (estampilles)
 - Sur défaillance, un état global consistant S est recherché dans l'historique des points de capture
 - => synchronisation + consensus/risque d'effet domino absolu ($S=S_0$ état initial)
- Des solutions hybrides (pour information) :
 - ♦ Vase communicant temps de rétablissement / coût en mode nominal
 - ♦ Tout va dépendre de la taille des messages à échanger (taille de l'état) et du nombre de sites à mettre d'accord
 - ♦ Le graal : perturber le moins possible l'exécution du système et éviter la recherche potentiellement infructueuse de points de reprise
 - Algorithme de Koo & Toueg sur la notion de Capture en deux temps [Koo&Toueg 1986]

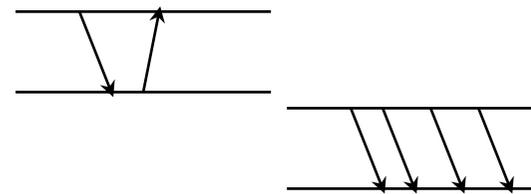
Les solutions pour la capture d'états concurrents

- La question : s'autorise-t-on à modifier en profondeur l'exécution du système ?
- Oui – Mise en place de rondes (ou conversations)
 - Aucun message ne peut être échangé entre deux rondes distinctes
 - Une nouvelle ronde ne peut commencer que si la précédente est terminée sur les autres sites



Un pied dans la théorie de la programmation distribuée

- La mise en œuvre de la **réplication** ou de la **capture d'état** nécessite différents **algorithmes distribués**
- Détection des répliques défaillantes vs communications lentes
 - ♦ Ping (aller – retour)
 - ♦ Heart-beat (message périodique)
- Les problèmes de détection et d'agrément distribués
 - ♦ Comment se mettre d'accord sur une valeur commune *consensus sur l'identité des sites défaillants*
 - ♦ Que se passe-t-il si en fonction du modèles des répliques (appelées souvent sites), du réseau ?
- Les protocoles de groupes : pratiques et nécessaires pour obtenir des preuves de correction des SR (cf tp)



TaF 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
- Le système est modélisé comme système distribué : 1 ensemble de calculateurs communicant par message

Modèle de fautes dans un système distribué

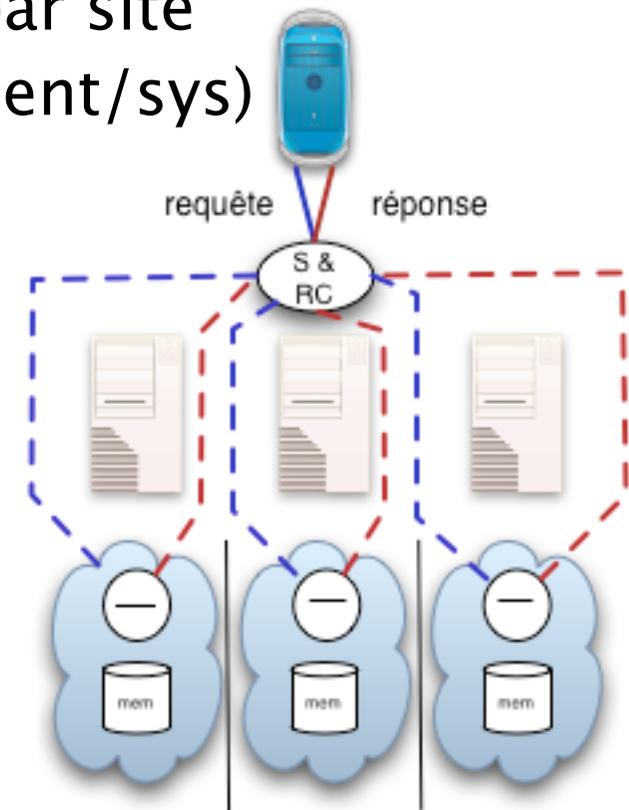
- 4 gabarits classiques de fautes
 - ◆ Silence (perte définitive du service)
Arrêt du matériel/blocage de l'OS
 - ◆ Omission (perte occasionnelle du service)
lien réseau peu fiable ou timing très mauvais
 - ◆ Temporelle (mauvais timing)
mauvaise estimation de WCET
 - ◆ Byzantine (service totalement incontrôlé)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

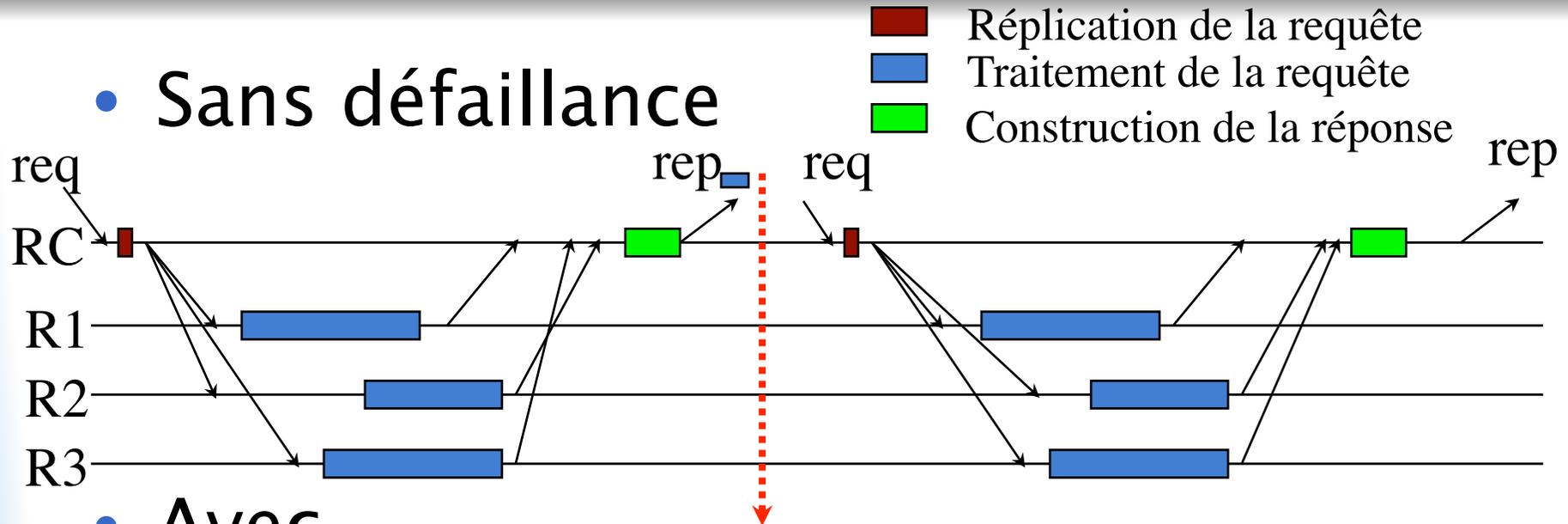
- N calculateur => 1 application par site
- 1 contrôleur pour interpréter (client/sys)
- Décision: vote
- Communications : diffusion des requête + agrément sur le résultat
- Déroulement
 - 1) Envoyer la requête à tous
 - 2) Chaque site exécute son service
 - 3) Construction de la réponse par vote majoritaire



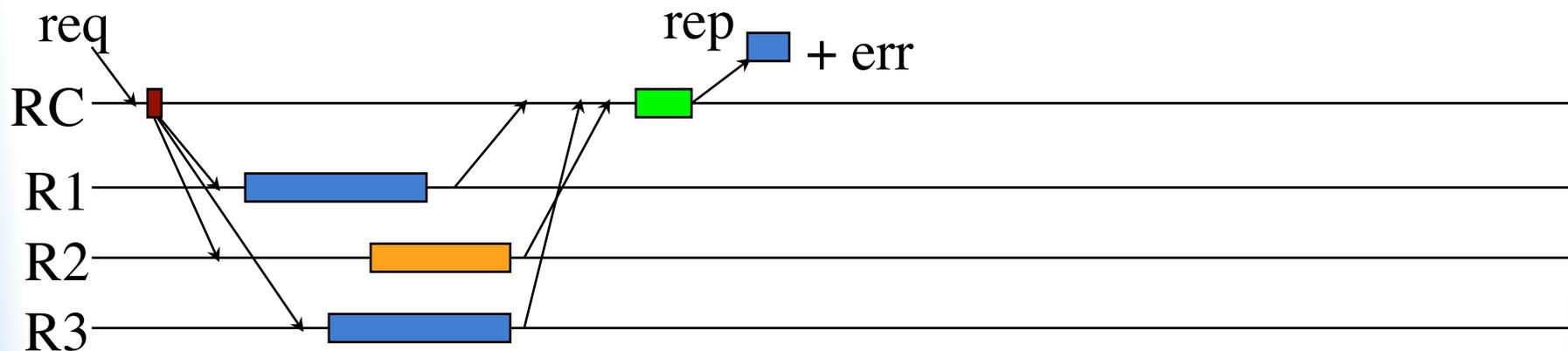
S & RC : vote et contrôle des répliques

Communication sans et avec défaillance

- Sans défaillance

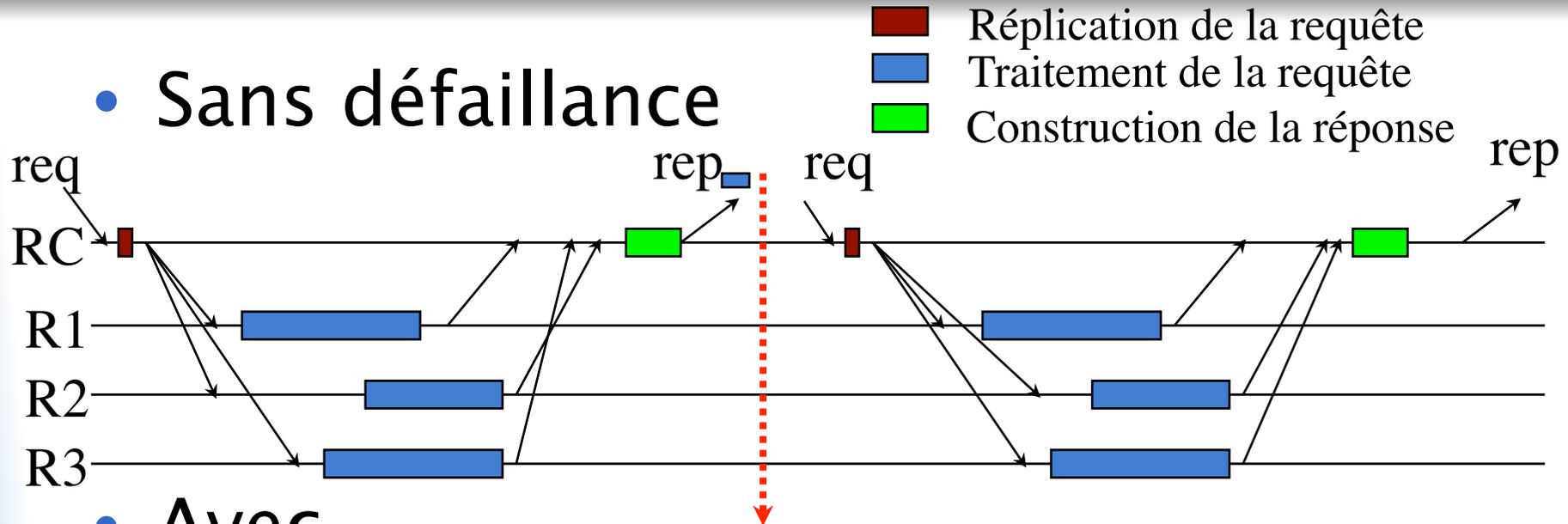


- Avec

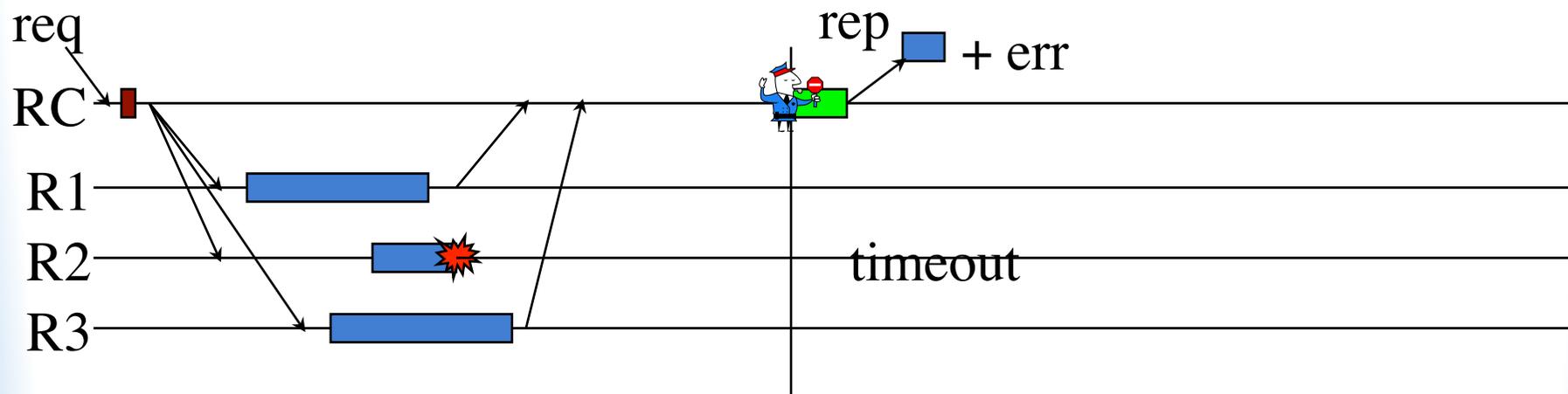


Communications sans et avec défaillance

- Sans défaillance



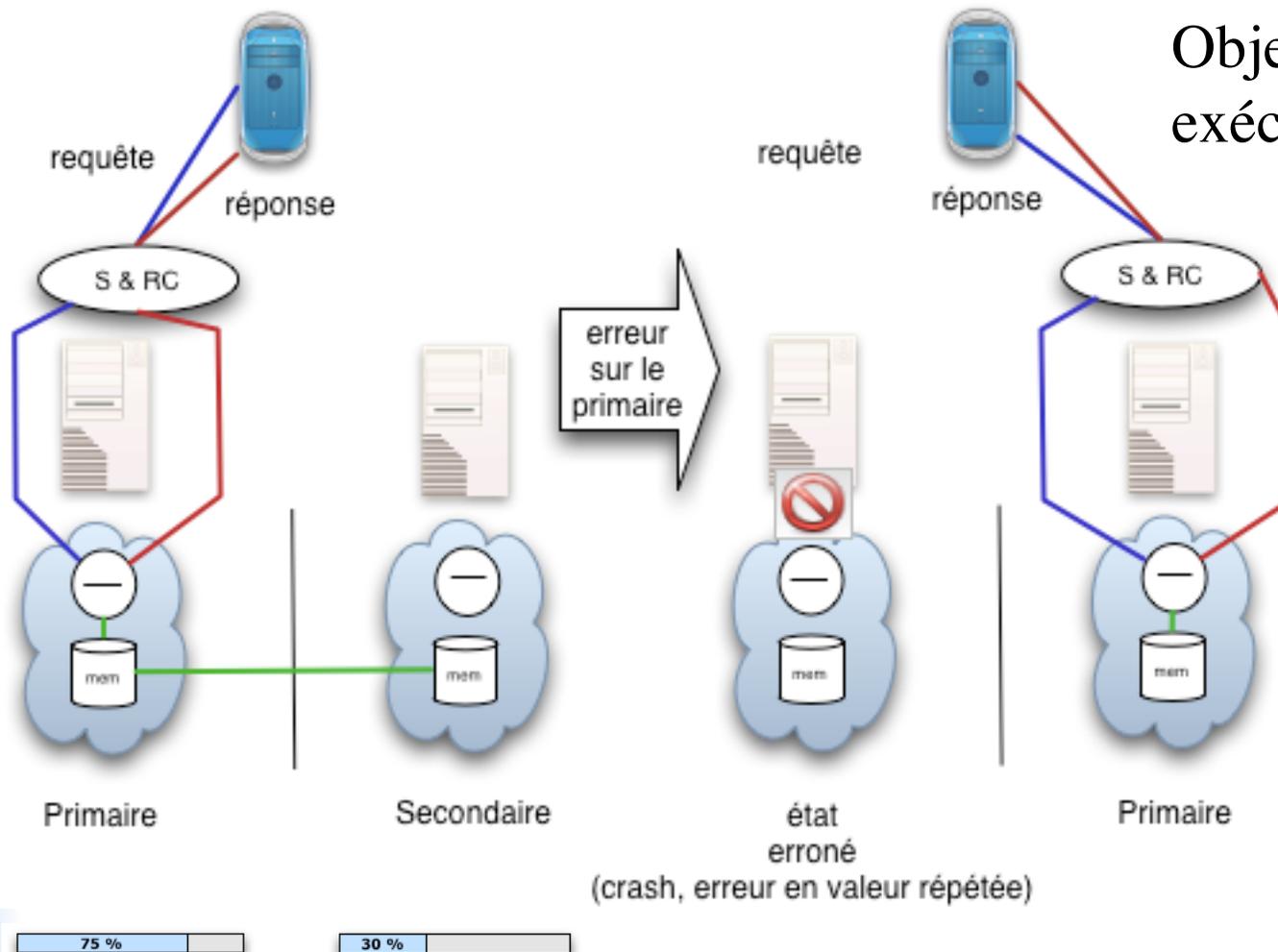
- Avec



Caractéristiques de la réplique active

- Modèle de faute toléré pour N répliques :
 - ♦ $N/2 - 1$ fautes Byzantines
- Détection réussie si au moins 1 correct
- Pour qu'une faute entraîne une erreur non détectée, elle doit s'activer sur :
 - ♦ le contrôleur des répliques,
 - ♦ Les conditions de fautes sur les répliques sont dépassées.
- $\Delta t(\text{régime normal} / \text{rétablissement}) \sim \text{nul}$

Réplication Passive



Objectif : éviter les exécutions concurrentes

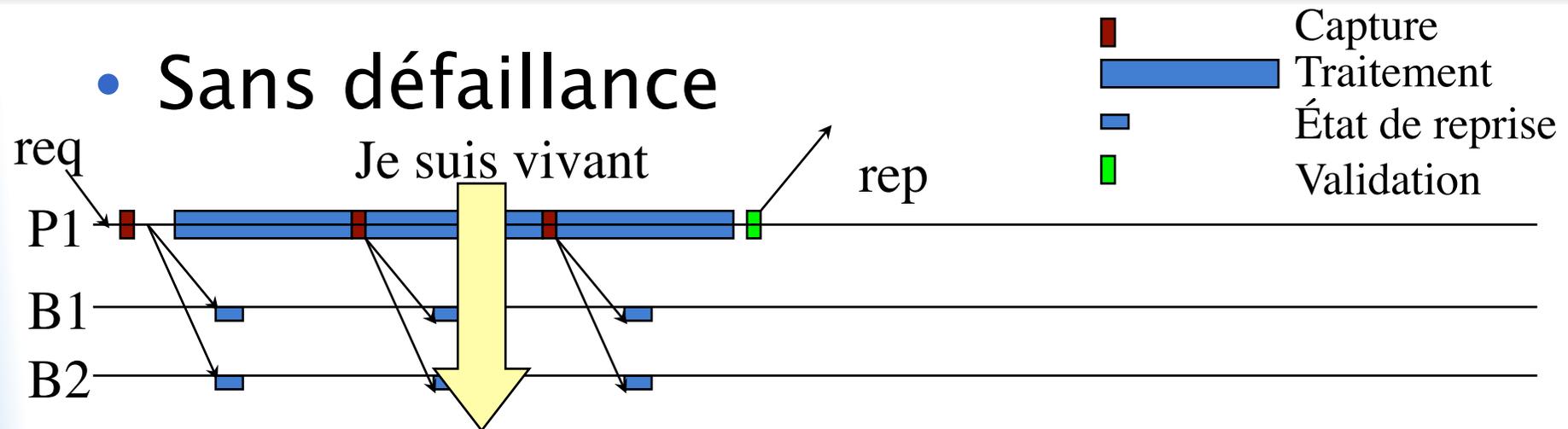
1 seule exécution à la fois

Communication : Transfert d'un état

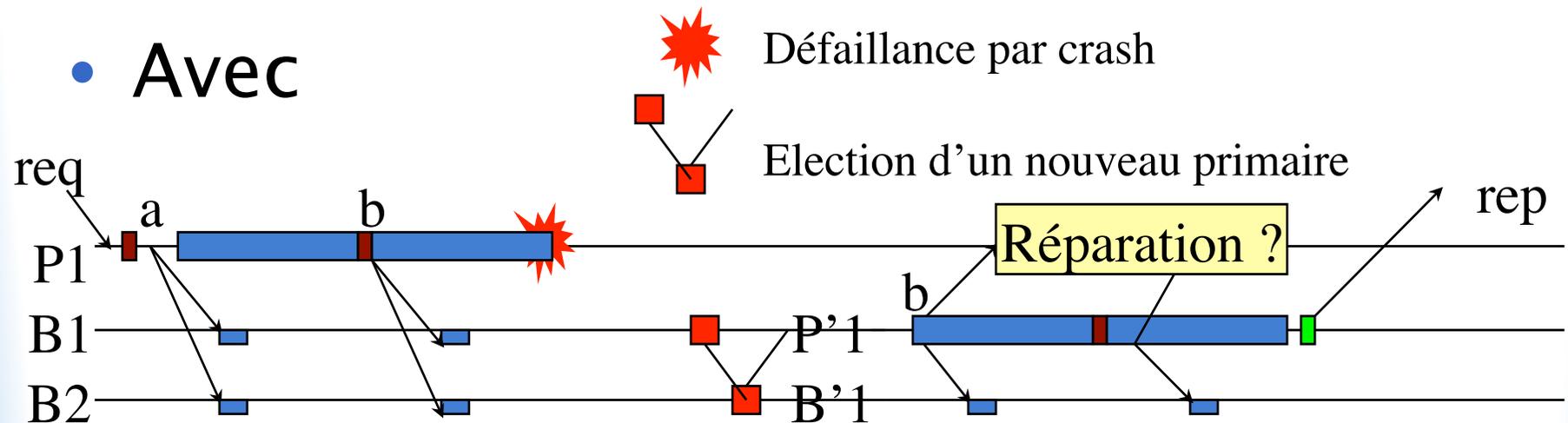
Le primaire doit capturer son état pour l'envoyer au secondaire

Communications et opérations significatives

- Sans défaillance



- Avec



Caractéristiques de la réplification 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.
- Grand Δt pour un service avec et sans faute
- Le coût en communication dépend de la taille de l'état de reprise.

Synthèse (1)

Stratégies	Hypothèse en présence de faute	Faute tolérée	Déterminisme des répliques	Coût en Ressources	Coût en Communication	Coût lors du Recouvrement
<i>Passive</i>	Silence sur défaillance	Crash	Non	1 serveur actif	haut (checkpoints)	Moyen (re-exécution)
<i>Semi-active</i>	Silence sur défaillance	Crash	Non	2 serveurs actifs	bas (notifications)	bas (switch)
<i>Active</i>	Défaillance arbitraire	Valeur	Oui	3 serveurs actifs	moyen (duplication messages)	Nul

Réplication passive	Entités	Action	Motivation	Moyens
<i>Communication</i>	Requêtes/réponses	Emission/réception	Synchronisation entre répliques.	Interception
<i>Exécution</i>	Points d'interruption asynchrone du flot de contrôle	activation / continuation / terminaison	Capture / restauration du contrôle lors du traitement des requêtes dans des composants concurrents.	Interception Instrumentation de la plate-forme
<i>Etat</i>	Données internes, état plate-forme	Mise-à-jour de données internes, interactions avec la plate-forme exécutive locale	Restauration cohérente de l'état pour un recouvrement transparent aux composants applicatifs clients.	Image mémoire Sérialisation Interactions Journaux

C-14

Tableau extrait du cours de Sdf de M. Fabre (INP Toulouse)

Synthèse (2)

Réplication semi-active	Entités	Action	Motivation	Moyens
<i>Communication</i>	<i>idem passive</i>	<i>idem passive</i>	Contrôle des notifications	<i>idem passive</i>
<i>Exécution</i>	<i>idem passive</i> + points de décision non-déterministes	<i>idem passive</i> + points de décision non-déterministes	Décisions non-déterministes	<i>idem passive</i>
<i>Etat</i>	<i>idem passive</i>	<i>idem passive</i>	Contrôle des interactions avec la plateforme produisant des résultats non-déterministes	<i>idem passive</i>

Réplication Active (TMR)	Entités	Action	Motivation	Moyens
<i>Communication</i>	<i>idem passive</i>		validation & propagation des réponses	<i>idem passive</i>
<i>Exécution</i>	<i>Pas nécessaire (déterminisme imposé de l'exécution des répliques)</i>			
<i>Etat</i>	<i>Pas nécessaire (si clonage à chaud non considéré)</i>			

Tableau extrait du cours de Sdf de M. Fabre (INP Toulouse)

Et pour le temps réel ?

- Les méthodes vues jusqu'à présent :
« best effort » ou « ultra synchrones »
- La synchronisation est
 - ♦ faible par rapport aux événements,
 - ♦ forte par rapport au temps
- Le dimensionnement du système est critique

Tolérance aux fautes & temps réel

- Le temps réel possède un modèle fortement contraint : le lot de tâches
- Prise en compte de l'exceptionnel
*le pire cas devient plus riche -> WCET erroné
+ tous les cas classiques vu jusqu'à présent*
- Détection de comportement erroné : timer, moniteurs couplé à
 - ♦ Architectures de masquage sans délai (TMR)
 - ♦ (Si la défaillance n'est pas liée au temps) des méthodes usuelles de recouvrement
 - ♦ (Si l'erreur détectée est liée au respect des contraintes temporelles) un déclenchement d'un mode de fonctionnement dégradé

Surdimensionnement et modes dégradés

- Pour assurer les bornes temporelles :
 - ♦ surdimensionnement
prévoir les reprises
 - ♦ Mode dégradés temporellement prédictibles et très fiables
 - ♦ Identification du mode pour la sélection d'un mode de fonctionnement

=> Le couplage du surdimensionnement et des modes opérationnels laisse une marge pour passer d'un lot de tâche TR défaillant à un autre lot de tâches TR ordonnançable
- Etude des lois d'occurrence des fautes

Notion de système intégré

TR et TaF

- Un support d'exécution logiciel &/ou matériel pour les deux aspects
- Idée : borner le temps du processus détection / rétablissement
- Exemples : MARS, Delta-4, ROAFTS ...
- Acceptation des fautes => prévoir le pire amène un compromis \neq temps réel souple
- Intégrer à l'ordonnancement les tâches de gestion des modes dégradés

Conclusion

- Tolérance aux fautes = domaine établi
 - ◆ Un vocabulaire & une communauté
 - ◆ 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
- Cohabitation TR / TaF
 - ◆ Tous les deux visent à contrôler le flux d'exécution
 - ◆ TaF a un impact sur l'ordonnancement
 - ◆ Il faut utiliser correctement les modes de fonctionnement et surtout les modes dégradés

Conclusion (suite)

- Cohabitation TR / TaF (suite)

Fixer la probabilité des modes dégradés est strictement différents de fixer la probabilité du respect des échéances

- La participation a un mode de fonctionnement est consciente (on connaît donc les échéances qui seront effectivement remplies)

Acronymes

- SdF : sûreté de fonctionnement
- TaF : Tolérance aux Fautes
- TMR, NMR : triple/ N – modular replication
- RB : recovery blocks ou Rollback...
- TR : temps-réel
- 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. "Distributed faulttolerant real-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.