



T e c h n o l o g y

There is a better way

Sponsors
Platinum



Sponsors
Gold



Les nouvelles architectures logicielles



Reactive

Version : 1.2

ParisJUG – 8 Décembre 2015

Changez d'approches – Soyez réactif

pprados@octo.com - Philippe PRADOS





- > Philippe PRADOS
- > Manager équipe « Architecture Réactive »

> +PhilippePrados



> @pprados



> in/pprados/fr



> BrownBagLunch.fr



> Conférences:

- + Solution Linux
- + MISC
- + Devoxx
- + PAUG
- + Scala.IO
- + JUG
- + ...



- > Conseil en Architecture et Management des SI
- > Un cabinet à taille humaine
- > 17 ans
- > 27 M€ de CA en 2014
- > Un positionnement Grands Comptes
- > 14% de croissance
- > 4 filiales : Maroc, Suisse, Brésil et Australie



- 1 POURQUOI ?
- 2 LES NOUVELLES APPROCHES
- 3 THREADS ?
- 4 MÉMOIRE ?
- 5 PERSISTANCE
- 6 LANGAGES DE DÉVELOPPEMENT
- 7 SYNTHÈSE



Reactive

Pourquoi ?



Evolutions des sollicitations des systèmes par les nouveaux usages et devices internet

Edition Collaborative
Google Docs
Office 365

Réseaux Sociaux
Like
+1

Analyses Financières
Flux Marché
Enchères

Informations Mutualisées
Trafic Pollution
Bon plan Parking

Multi-canal
Tablette / TV / Mobile / Montre
Session partagée
Synchro

Communication directe
Skype
Chats
Hang-Out

API
Ouverte
Fermée

Indicateurs
GPS
Capteurs
Objets Connectés

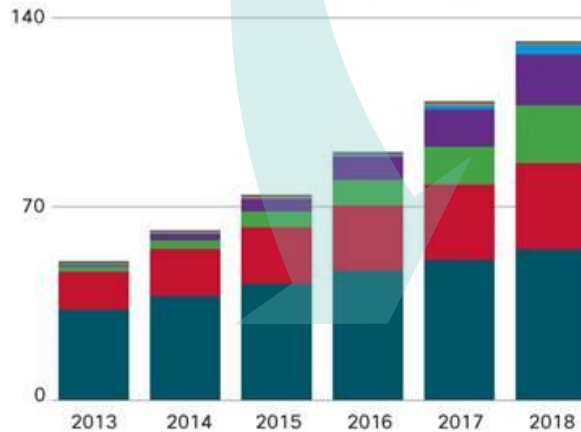
Jeux multi-joueurs
Cross-Devices
Serious Games
Centralisés Mondialement

Algorithmes
Big Data
Machine Learning
Séries temporelles

Pic de charge
Concert
Soldes
Pub TV

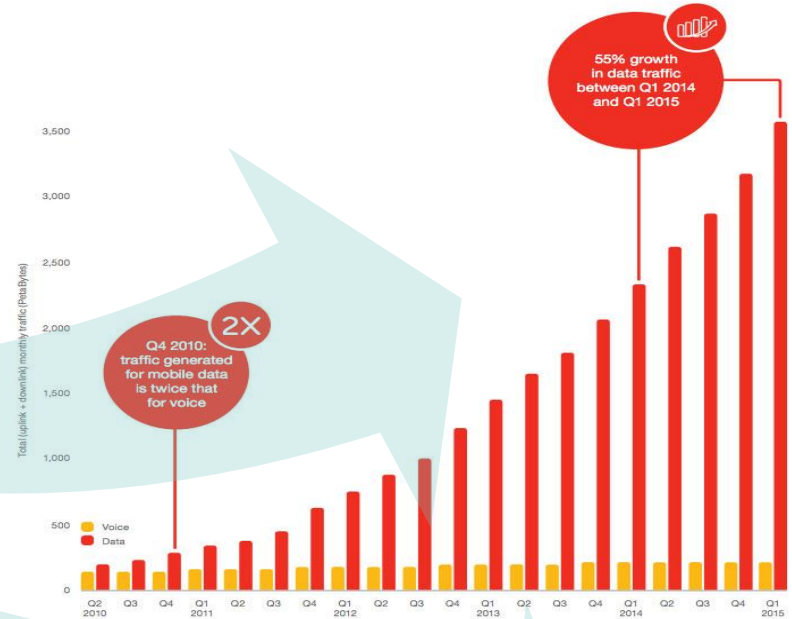
Exabytes per Month

21% CAGR 2013-2018

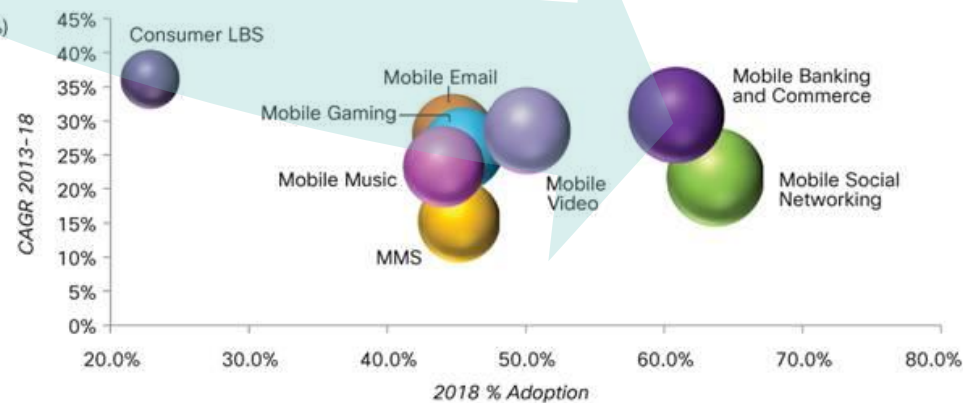


Source: Cisco VNI, 2014

The percentages in parentheses next to the legend denote the device traffic shares for the years 2013 and 2018, respectively.



* Traffic does not include DVB-H, Wi-Fi, or Mobile WIMAX. Voice does not include VoIP



Source: Cisco VNI Service Adoption Forecast, 2013-2018

Note: By 2018, the global consumer mobile population will be 4.8 billion.

- > Les nouveaux usages entraînent une **explosion des sollicitations des SI** : tout le monde est **connecté partout, tout le temps** avec n'importe quel **terminal** et s'attend à une **réactivité accrue** :
 - + Les systèmes doivent donc être capables de gérer des **milliards de petites requêtes**
 - + Dans un contexte de **démultiplication des devices** (Tablette, smartphone, montre, lunettes, voiture, capteurs, etc.)

- > Cela nécessite une **remise en cause des architectures logicielles « classiques »** notamment sur les volets de :
 - + Gestion de la latence
 - + Performances
 - + Et montée en charge

- > Les réponses actuelles se concentrent sur des **logiques d'amélioration de l'existant**, en termes de
 - + Machines (multiplication des serveurs et/ou augmentation de la puissance des serveurs)
 - + Logiciels (Ajout de composants, proxy, caches, etc.)
 - + Paramétrage (pour un gain relatif)
 - + Changement d'approche sur certaines couches (NoSQL, Oracle RAC)

- > Avec les **limitations intrinsèques** que portent les systèmes à améliorer
 - + Coût
 - + Limitation physiques
 - + Limitation des architectures et composants

- > Et qui nous amènent à proposer de **nouvelles approches**
 - + d'architecture
 - + et de conception logicielle

Ce qui change...	Il y a 10 ans	Maintenant
Nombre de serveurs	10	1000 (on demand)
Temps de réponse	Secondes	Millisecondes
Interruption de service	Heures	Jamais
Volume de donnée	GBs	TBs -> PBs
Traitements	Gros, rares, batchs	Unitaires, nombreux
HTTP	HTTP/1.1, Mode texte	HTTP/2.x, Mode binaire
Réseau	Cablé, stable	Mobile, instable
Clients	Workstation	Multiples
Sessions d'utilisation	Longue	Courte

Délais	Réaction de l'utilisateur
0-100ms	Instantané
100-300ms	Léthargique
300-1000ms	La machine travaille...
1s+	Switch mental
10s	Je reviens plus tard

* Latence moyenne en 3G : 150ms, 4G: 65ms

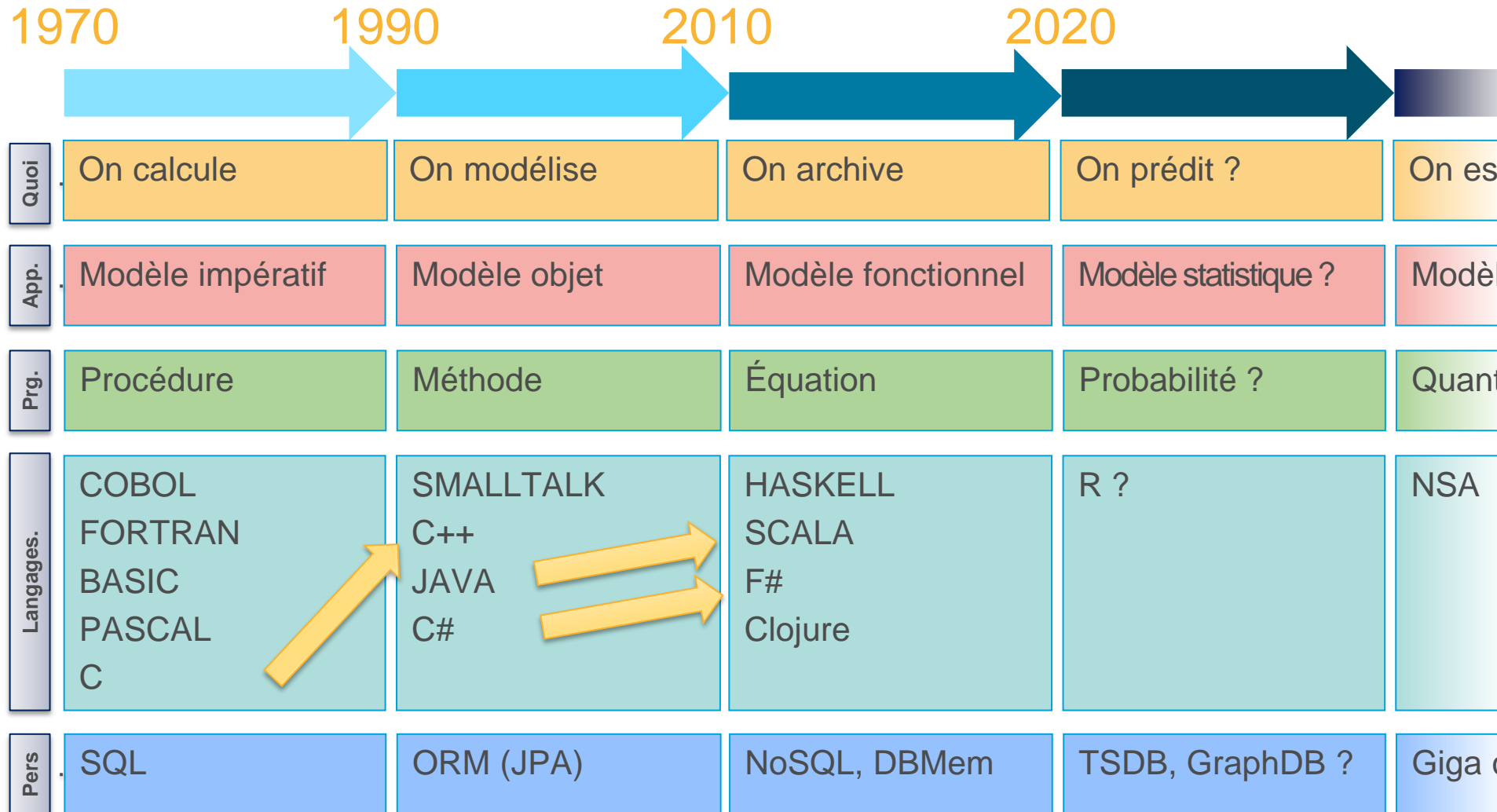
Vitesse processeurs

Exécution d'une instruction typique	1/1 000 000 000 sec = 1 nanoseconde
Mutex lock/unlock	25
Envoi de 2Ko sur réseau 1Gps	20 000
Lire 1MB séquentiellement en RAM	250 000
Lire 1MB séquentiellement du disque	20 000 000
Envoi d'un paquet vers US et retour	150 millisecondes = 150 000 000

Équivalent humain

Exécution d'une instruction typique	1 seconde
Mutex lock/unlock	½ minute
Envoi de 2Ko sur réseau 1Gps	5 heures ½
Lire 1MB séquentiellement en RAM	3 jours
Lire 1MB séquentiellement du disque	6 mois ½
Envoi d'un paquet vers US et retour	5 ans

Quatre grandes périodes de l'informatique



THE EVOLUTION OF
SOFTWARE ARCHITECTURE

1990's

SPAGHETTI-ORIENTED
ARCHITECTURE
(aka Copy & Paste)



2000's

LASAGNA-ORIENTED
ARCHITECTURE
(aka Layered Monolith)



2010's

RAVIOLI-ORIENTED
ARCHITECTURE
(aka Microservices)



WHAT'S NEXT?

PROBABLY PIZZA-ORIENTED ARCHITECTURE

By @benorama

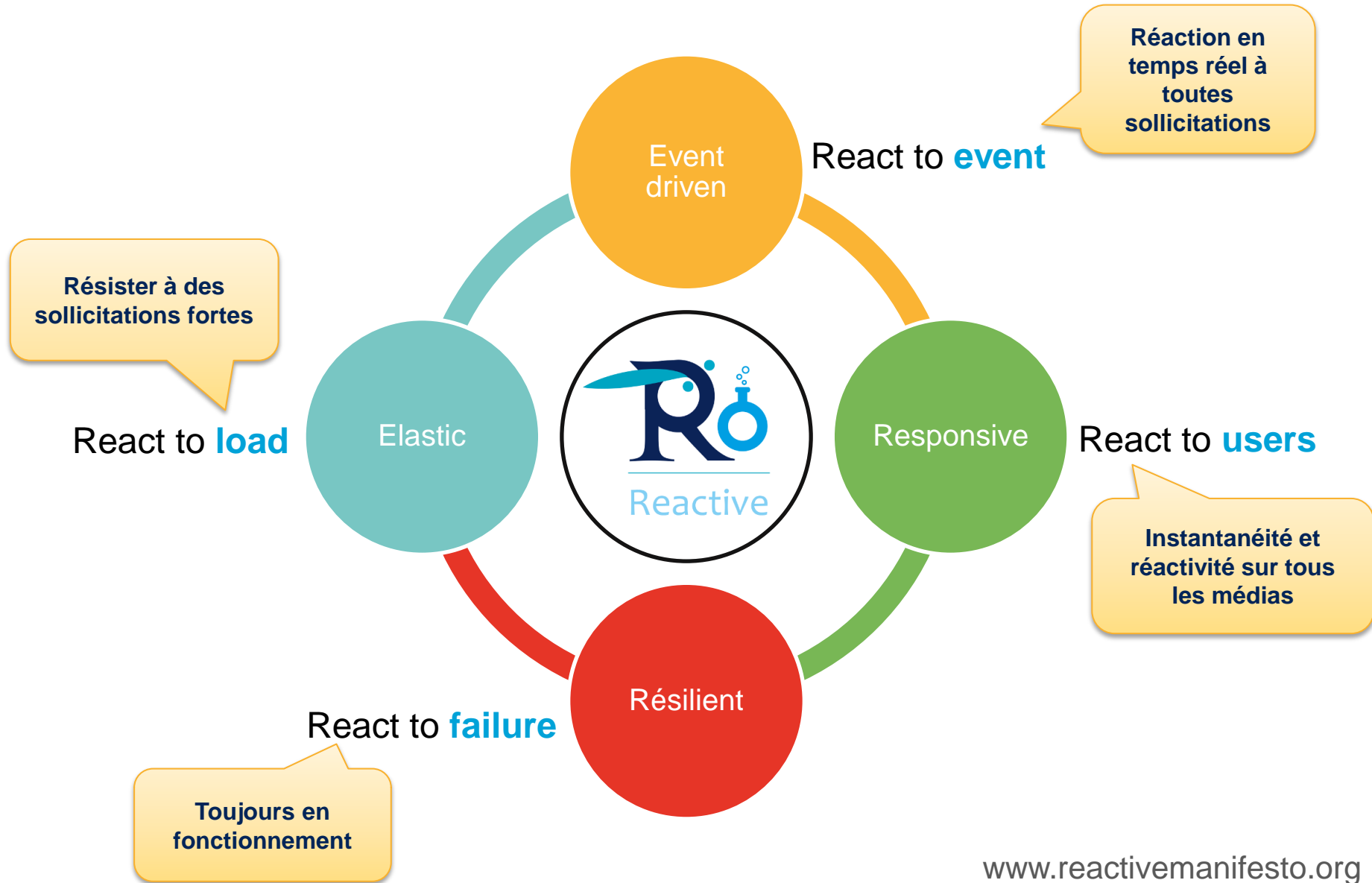


Reactive

Les nouvelles approches



Une **application** qui répond à des variations importantes **du nombre de connexions simultanées** sans remettre en cause sa structure et sans perturber son fonctionnement nominal.



Des langages enrichis

- > Async/await
- > Co-routine
- > Closure
- > Stream

et des évolutions technologiques

- > Drivers DB asynchrones (ODBC en Windows 8, AsyncMySQL, ...)
- > Frameworks (Node.js, Akka, Stream, Play, Rx*, ...)
- > Bases de données (NoSQL, Redis, ...)

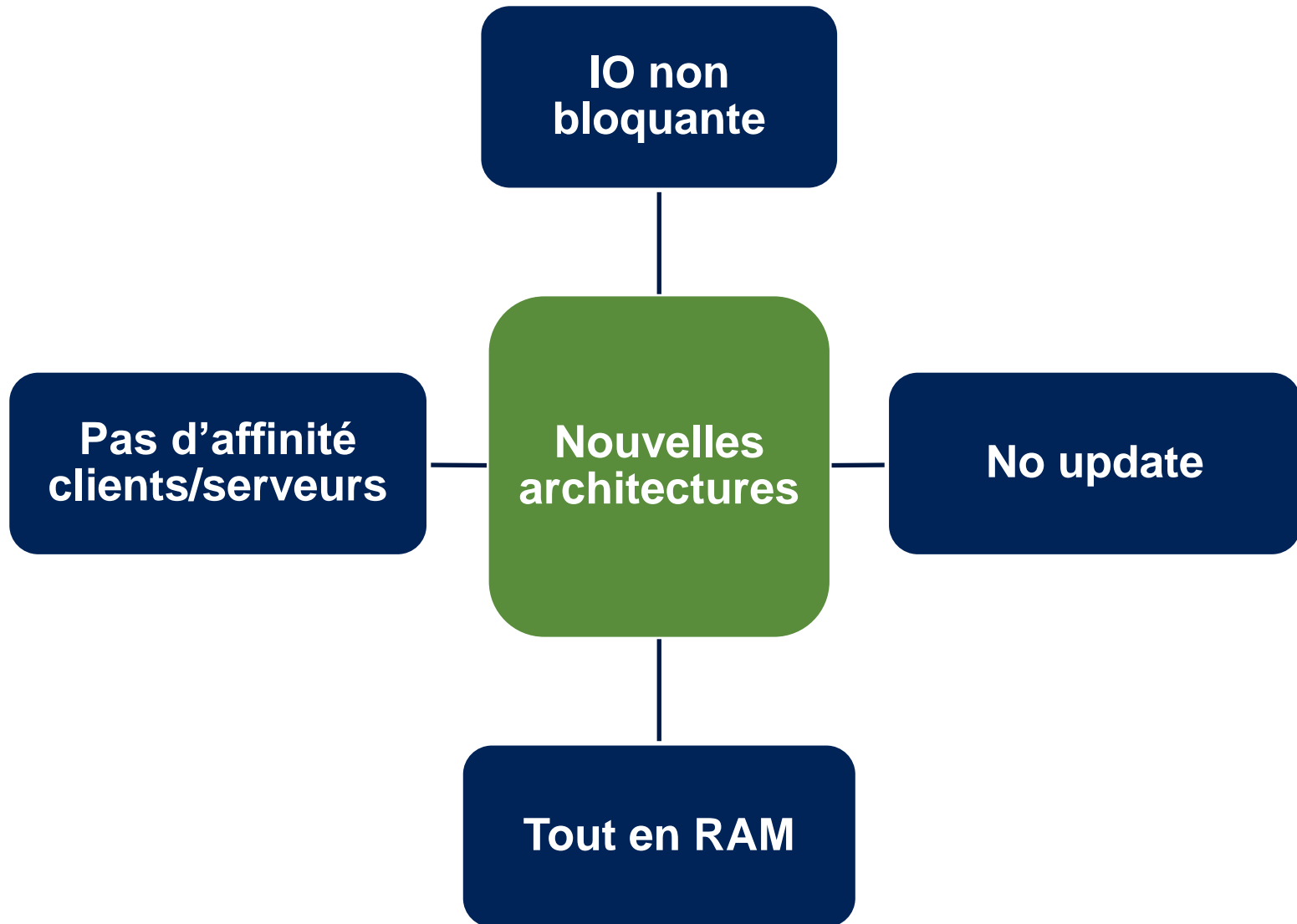
permettant de tirer le meilleur parti des concepts

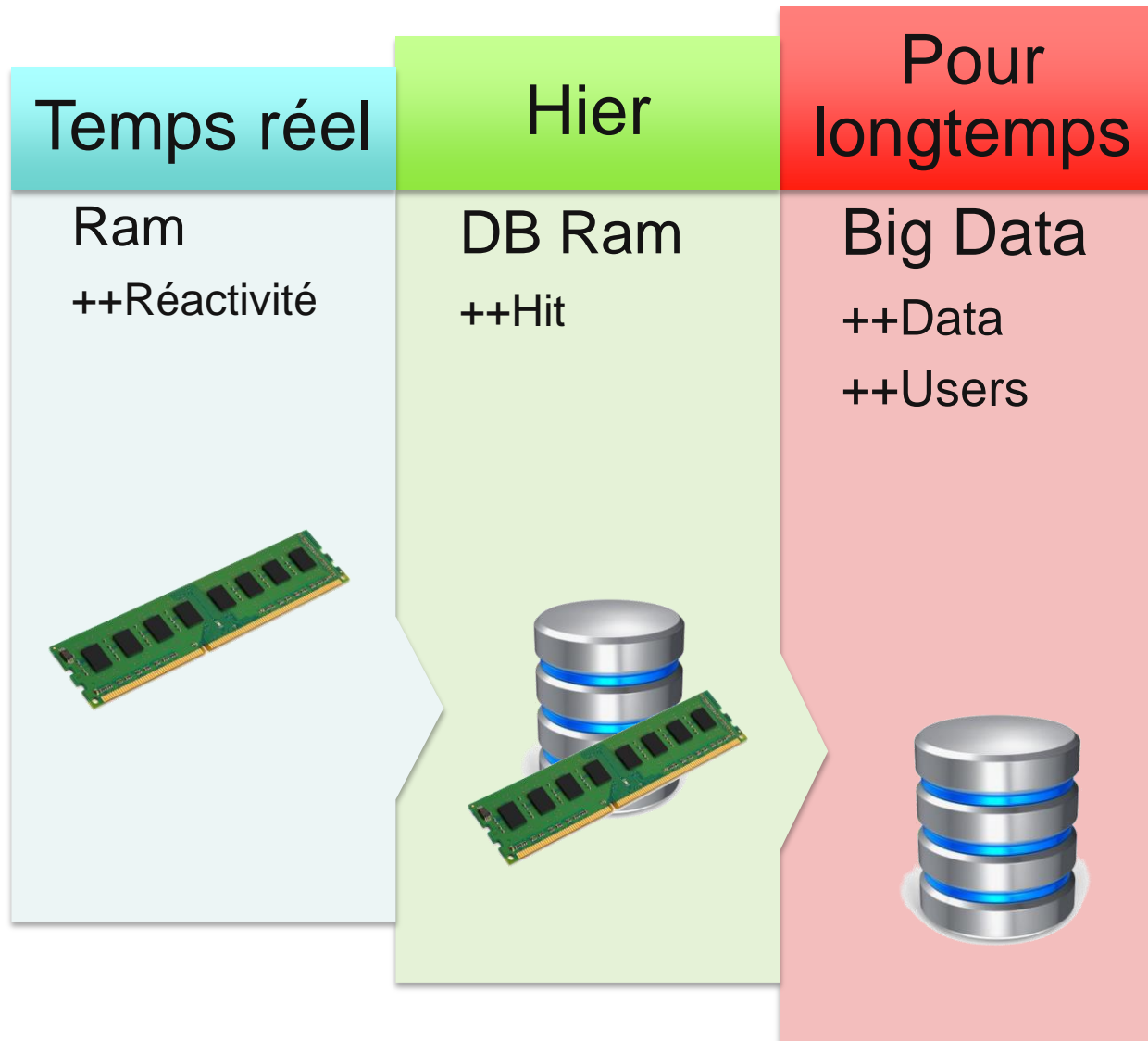
- > Événement
- > Flux
- > Acteur

dans le respect des principes clés

- > Pas plus de thread que de cœurs
- > Que des IO asynchrones

pour une meilleure exploitation des CPU







Reactive

Thread ?



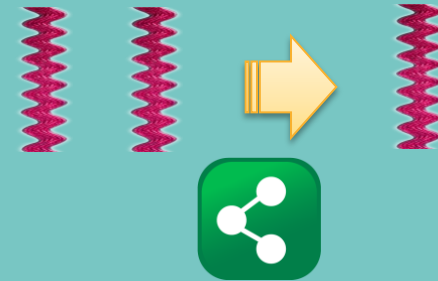
Un processus par client

- ++Ram, --CPU



Un thread maintenu par client

- ++Ram, --CPU



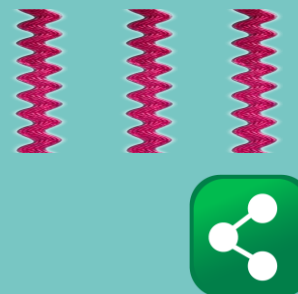
Un seul thread (NIO)

- --RAM, --CPU



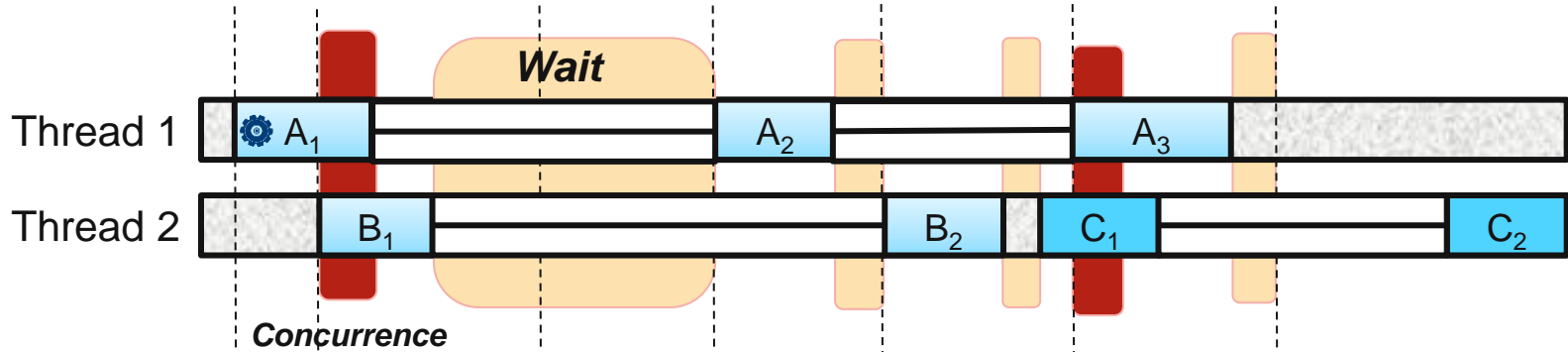
Pool de thread

- Limite par IO, ++Ram

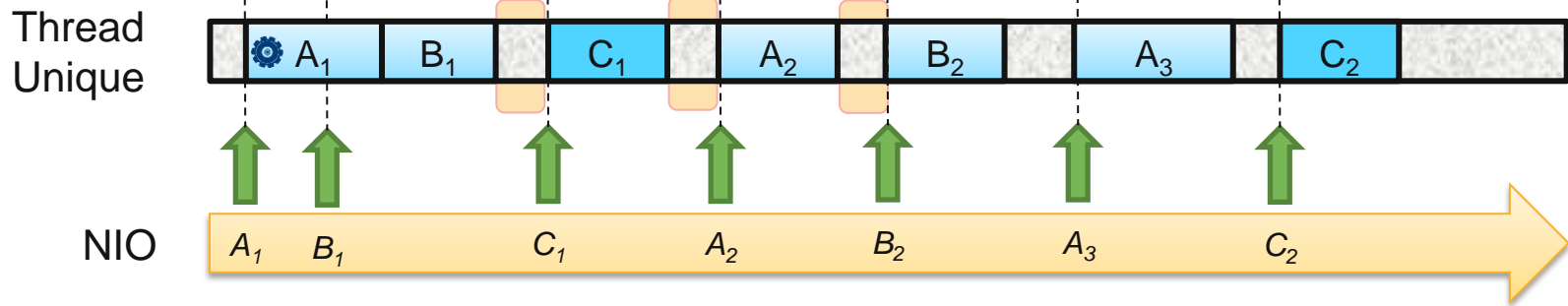


Gestion de la concurrence

Approche « probabiliste » (actuelle)



Approche Réactive (Optimise la CPU)



Legend:

- Blue box: Calcul
- White box: Attente I/O

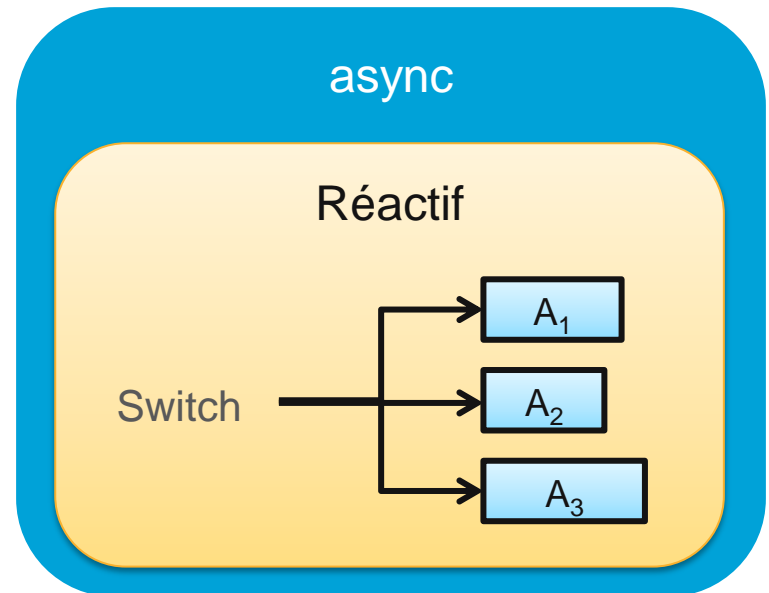
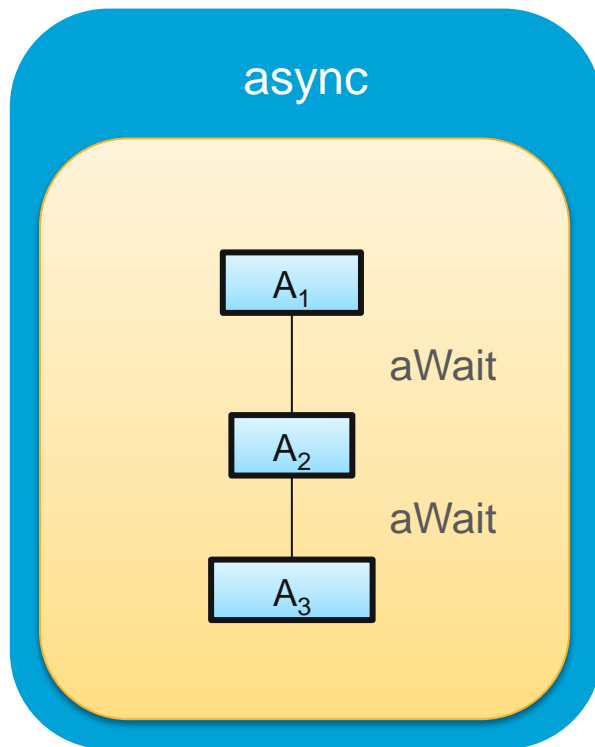


- > Depuis 20 ans, on **parie** que la concurrence sera optimale
- > Ce n'est **jamais** le cas sur les serveurs

- > Utilisons une approche **déterministe** et réactive
 - + Gain notable de performance



- > Des langages proposent une transformation d'un traitement classique en traitement réactif (Scala, .Net, JS, etc.)



Couche	Evolution
OS	Non blocking IO
Drivers	Asynchrone (NO-JDBC)
Protocoles	HTTP 2.0, Web Sockets
Frameworks, Composants	Play, Node.js, Netty Akka, JMS, Java8
Langages	Générateur, Continuation Co-routine, Async / Await, Functional

- > Un thread par cœurs CPU. Pas plus !
 - + Pas de perte de puissance dû au changement de contexte
 - + Pas de « cache-miss »
 - + Économie de mémoire





Reactive

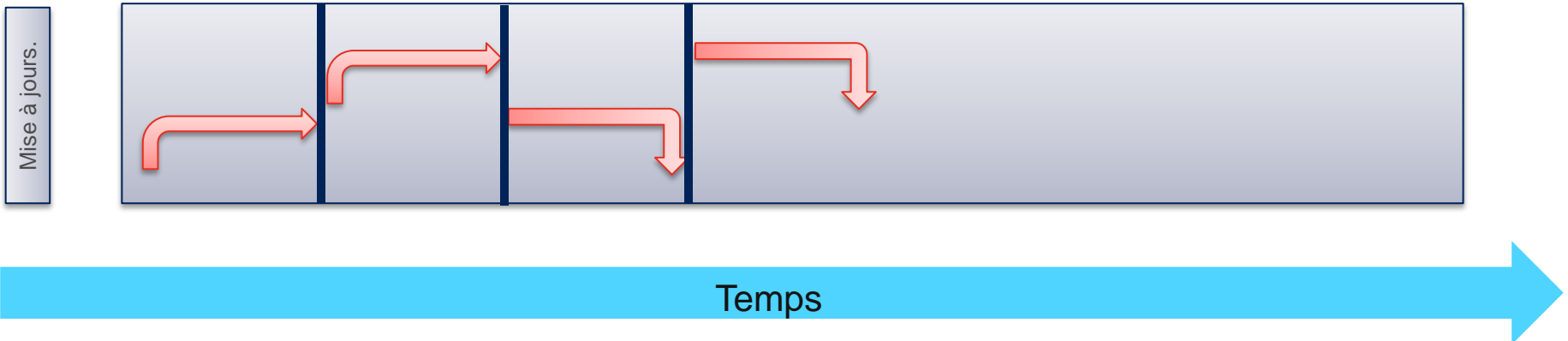
Tous en RAM ?



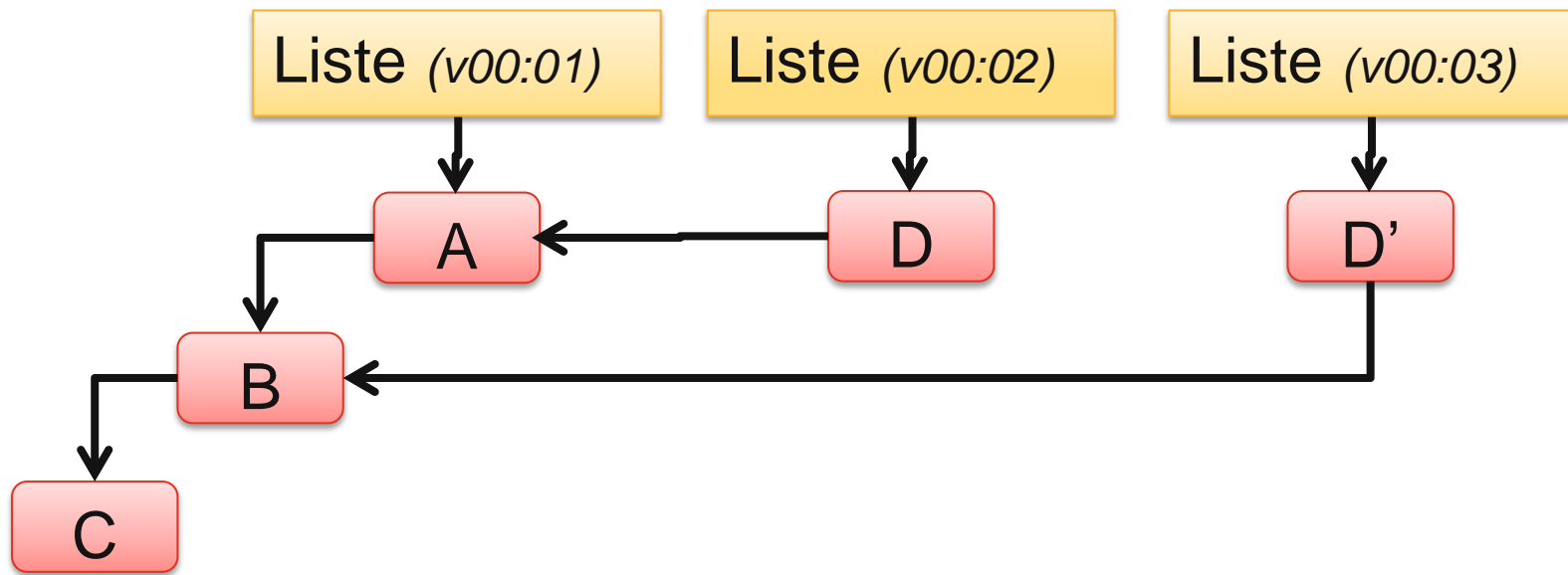
- > Approche par partage total
 - + L'**état** est manipulé simultanément par tous les threads
 - + Modèle classique avec verrous logiques (`synchronized`)
 - + Synchronisation entre les lecteurs et les écrivains
 - Les lectures peuvent être concurrents
 - Jusqu'à la demande de verrou d'un écrivain
- > Risque de dead-lock
- > Complexité du développement et du déverminage




- > Monothread (type **Node.js**, **Ruby**)
 - + Un seul thread manipule le modèle
 - + Pause des traitement lors des I/O
 - + Exploitation optimale d'un cœur du CPU
- > Possibilité d'utiliser un processus par cœurs, mais sans concurrence sur l'**état**.
- > Généralement épaulé d'une base de donnée en mémoire



- > Données intégralement **immuables** (type **Scala** ou **Haskell**)
 - + Les objets n'évoluent plus une fois créés
 - + La flèche du temps est figée lors de la consultation d'un état.
 - + Une multitude de lecteurs sur un état
 - + Il existe plusieurs versions des objets en mémoire
- + Séparation entre l'identité et l'état
 - Une identité **possède** un **état**
 - L'état de change pas
 - L'identité référence un nouvel état lors des modifications




 Persistent

Ajouter D

Supprime A

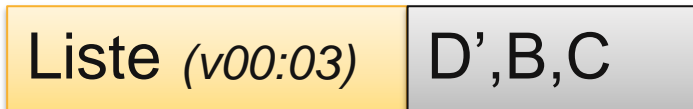
Liste (v00:01)	A,B,C
Liste (v00:02)	D,A,B,C
Liste (v00:03)	D',B,C

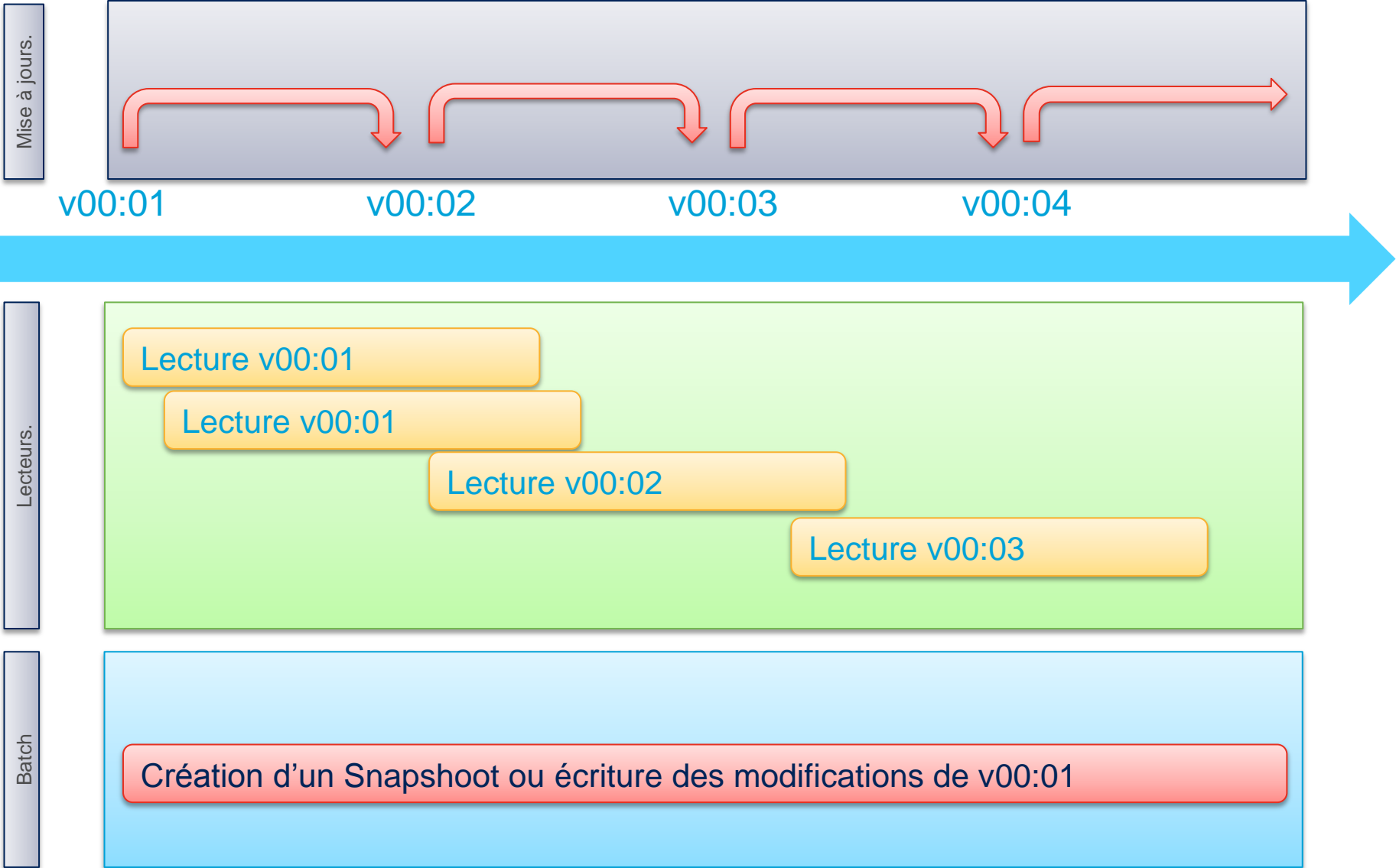


 Persistent

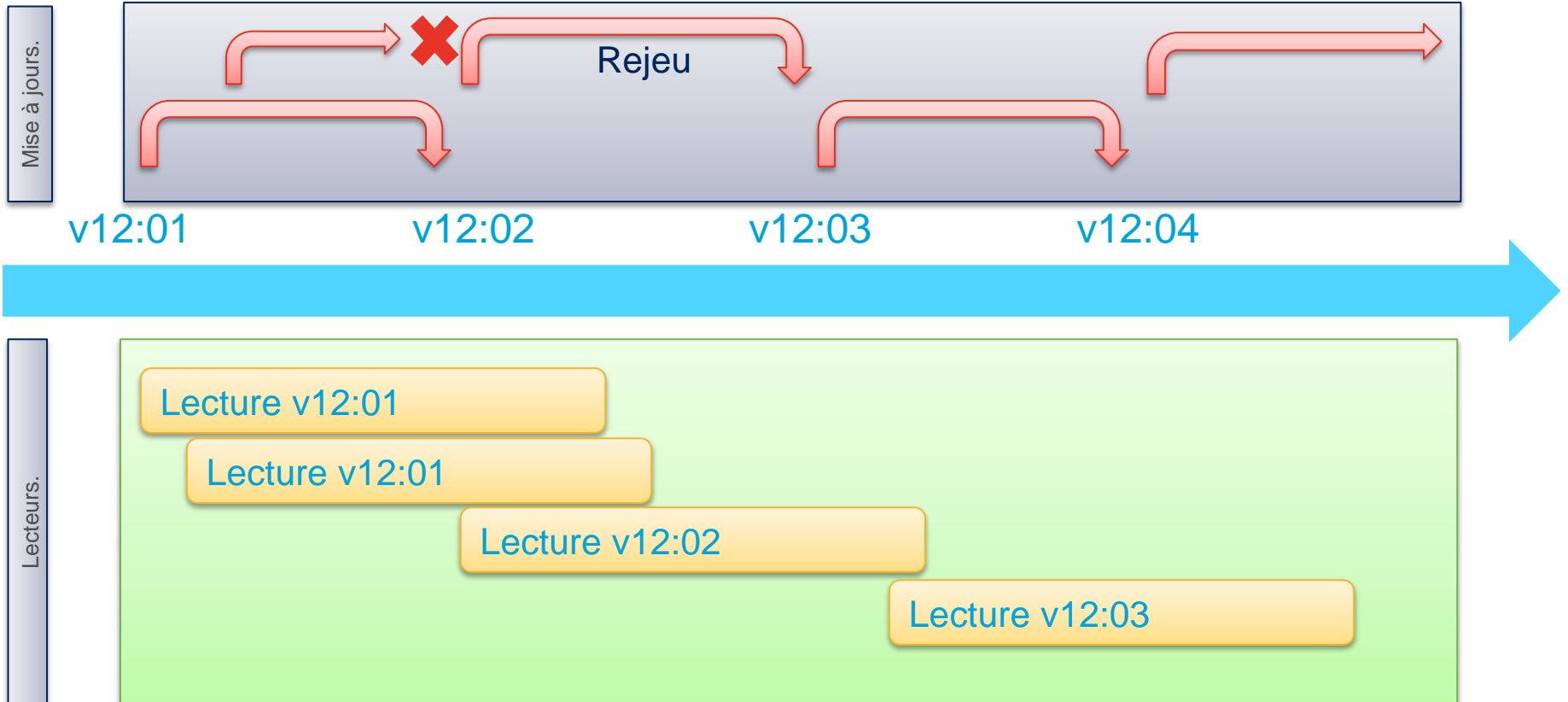
Ajouter D

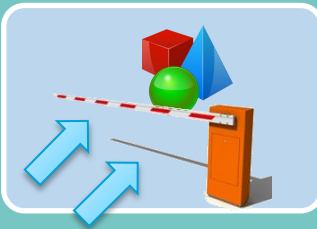
Supprime A





- STM: Transaction logicielle en mémoire (type Clojure ou Scala)
 - + Partage total de l'état mais modification sous contrôle
 - + Les modifications s'effectuent dans des transactions
 - + En cas de collisions, les modifications sont rejouées
 - + Les collisions doivent être rares





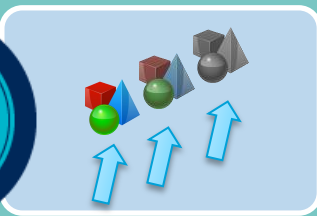
Partagé (classique)

- Verrous en lecture et écriture
- Complexe, Dead-lock
- Pessimiste



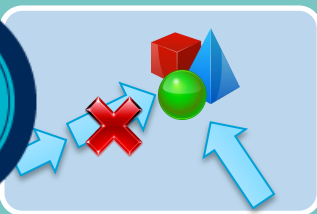
Pas d'état (stateless)

- État sur base de données



État immuable, lecture versionnée

- Pas de verrous en lecture
- Verrous en écriture



Transaction en mémoire (STM)

- Pas de verrous
- Rejeu en cas de collisions
- Optimiste





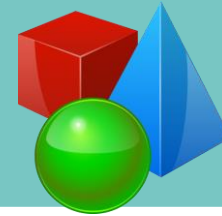
Reactive

La persistance



État

- La vision des données à l'instant
- Accès instantané
- Géré en mémoire



Flux

- Suite des modifications
- Intégralement rejouable
- Persistant
- Reconstitution de l'état



Avant

- Seul le **présent** est important
- Les données sont modifiées pour refléter leurs **états** présent
- Chaque modification fait disparaître l'état précédent

DELETE/INSERT/UPDATE

Maintenant

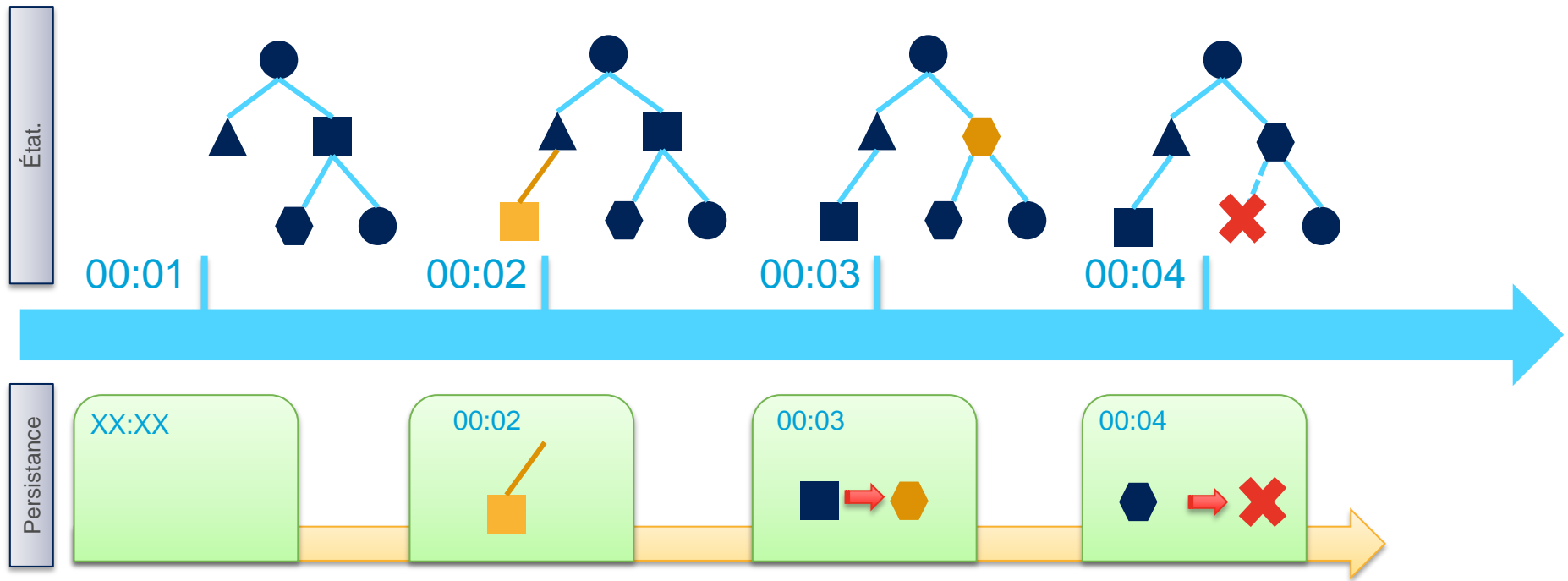
- L'**histoire** des données est important
- Rien ne disparaît
- Le **flux** des modifications porte les pépites
- Les archives deviennent numériques

INSERT ONLY

Demain

- Prédiction : Analyser le **passé** des données pour prédire leurs **futurs**

DATA LEARNING



- > Les dates de modifications sont importantes dans le **flux**.
- > Il peut y avoir des copies temporaires de l'**état** (Snapshot) dans des caches, pour
 - + accélérer de démarrage
 - + D'autres analyses asynchrones sur les données
- > Ou une optimisation du flux en supprimant les modifications devenu inutiles (attention à la perte d'information)



- > Écriture des modifications à la fin des fichiers
 - + I/O très rapide sur mémoire SSD

- > Permet de revoir complètement le modèle
 - + Un re-jeu de transactions permet de construire le nouveau modèle
- > Plus de migration de base de données
- > Backup rapide et à chaud, pendant l'exécution du service
 - + Sauvegarde des dernières logs
- > Capacité à découvrir du passé (déménager n'est pas changer l'adresse)
- > Le démarrage est rapide, car tous est calculé en mémoire
 - + Plusieurs millions d'évènements joués en quelques minutes
 - + Pendant que d'autres serveurs sont toujours actifs
- > Optimisations possibles :
 - + Possibilité de ré-écrire le fichier de log pour supprimer les modifications multiples
 - + Possibilité de persister des copies intégrales de l'état du système.

- > La persistance de l'**état** n'est pas importante
 - + S'il est vivant quelque part
 - + S'il est possible de le reconstituer



- > Persistance limitée aux **messages** de modification
 - + Sauvegarde du **flux**
 - + Ajout à la fin d'un fichier de transactions (AOF)



La base de donnée n'est qu'une  vue des logs de modifications

Chaude

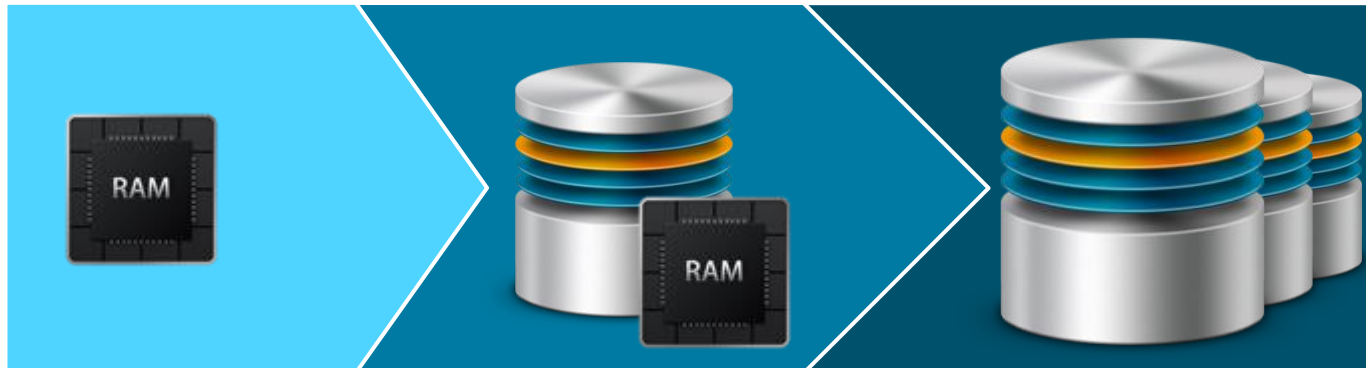
Cache en RAM
++ Réactivité

Tiède

DB snapshot
++ Réactivité
+ Volume

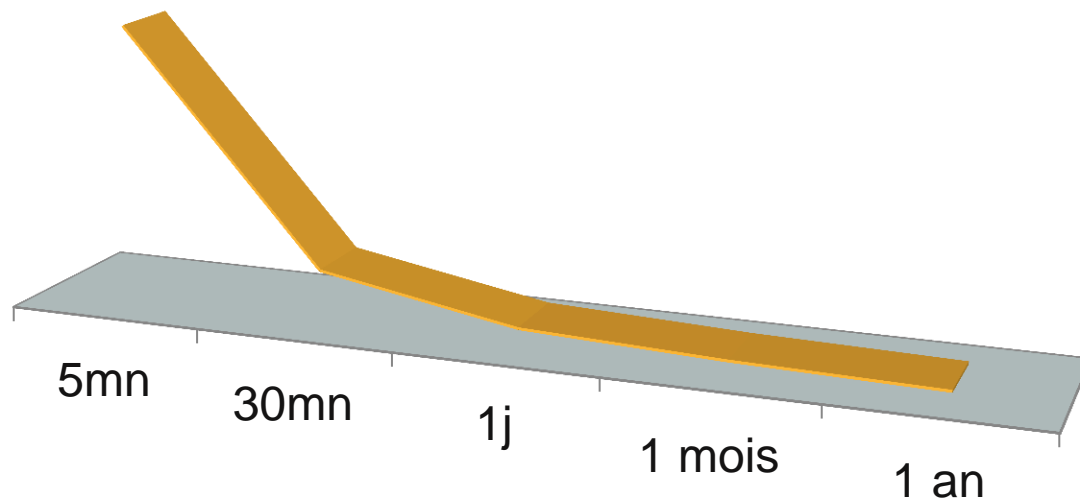
Froide

Big Data, Flux
++ Data
+ Requêtes longues



BATCH
(lambda architecture)

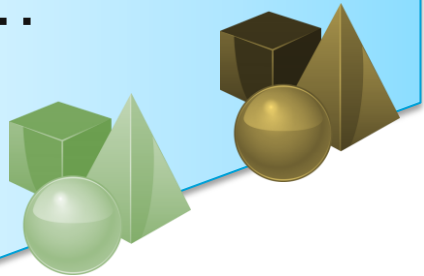
Un logiciel sans passé n'a pas d'avenir



Horizontal

(sharding)

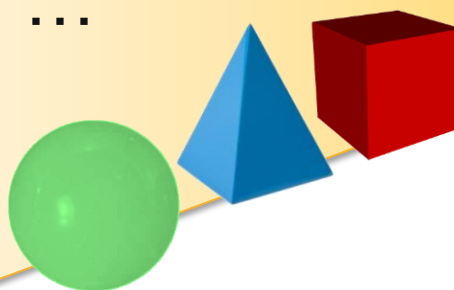
- Date
- Régions
- Métier
- ...



Vertical

(acteurs)

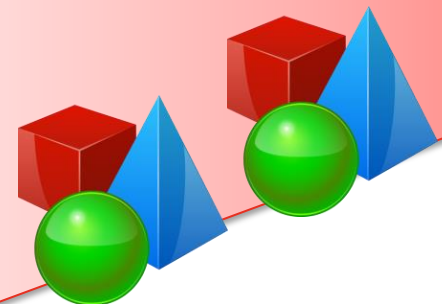
- Compta
- Produit
- Clients
- ...



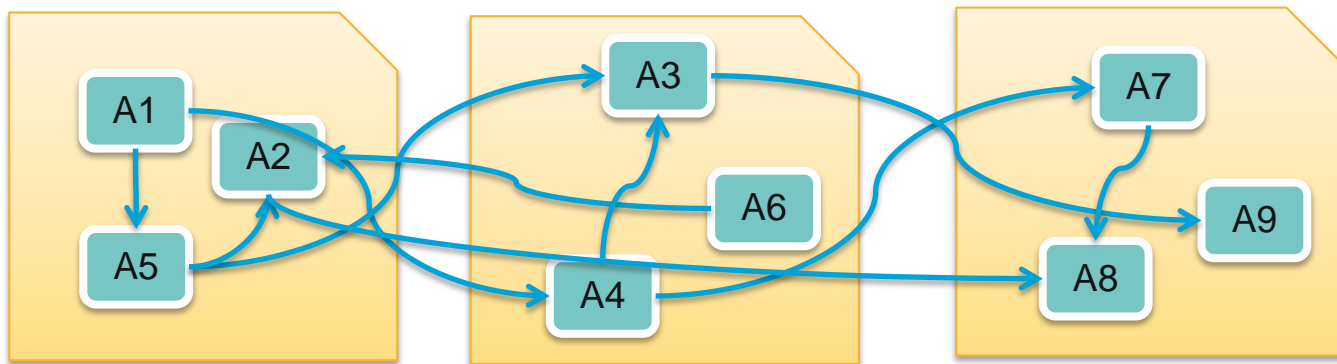
Attentions aux jointures !

Réplication

- Synchronne
- Asynchrone



- > Utilisation d'**Acteurs** (type Akka)
 - + Une file de message par acteur
 - + L'acteur traite les messages un par un
 - + Utilise des I/O asynchrones pour ne pas bloquer
- > Fractionnement de l'**état** en silo
 - + Un seul Acteur pour manipuler le silo de données
 - + Communication asynchrone entre les acteurs
 - + Distribution des acteurs dans le cluster
 - + Possibilité d'avoir des STM entres Acteurs



<http://www.reactivemanifesto.org/>



Reactive

Langages de développement



- > L'utilisation de langages orienté objets n'est plus la seule solution
- > Des concepts adaptés aux nouveaux besoins
- > Exploités par les géants du web
 - + **Twitter** ou **LinkedIn** utilisent **Scala**
 - + **Akamai** utilise **Clojure**
- > Ils offrent de nouveaux paradigmes
 - + Procédural
 - + Objet
 - + **Fonctionnel**
 - + **Transaction in Memory**
 - + **Acteurs**
 - + **DSL** (capacité à créer de nouveau langage spécifique)

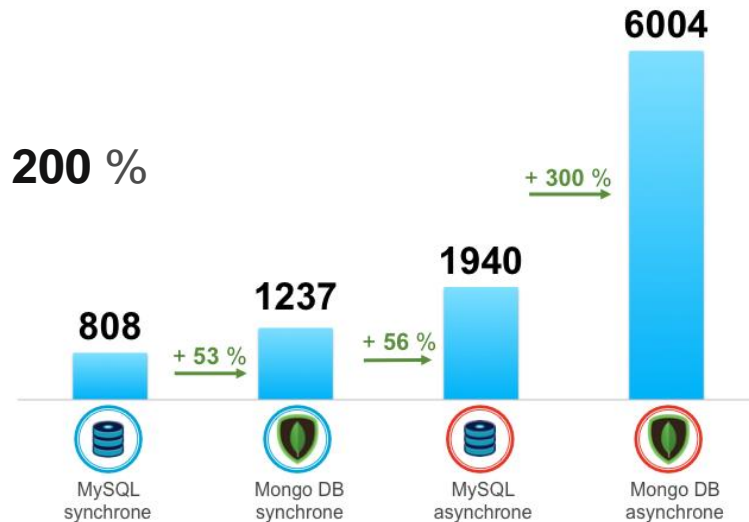


- > **Scala** est compatible avec **Java**.
- > Peut réutiliser le capital logiciel actuel
- > Pour les nouveaux projets, avec les nouvelles architectures
 - + La migration d'un projet existant est difficile
 - + Les nouvelles fonctionnalités peuvent bénéficier des nouveaux paradigmes
- > Nécessite une formation des développeurs
 - + Au même titre qu'ils ont été formé à la POO en son temps
- > **Plus efficace** pour accompagner les nouvelles architectures (Acteur, STM, Immutabilité, réactivité, DSL)



- > JDBC n'est pas réactif !
 - + JDBC synchrone vs Driver asynchrone > **200 %**

- > Scala propose des drivers réactifs
 - + Projets Open Sources
 - + Pas toujours stables
 - + Mongo, Redis, Mysql, etc.



- > Scala propose différents modèles pour les accès concurrents
 - + Modèle immuable
 - + Transaction en mémoire
 - + Répartition par type (Akka)
- > Scala facilite le code réactif
 - + Closure avec possibilité de modifier les variables externes

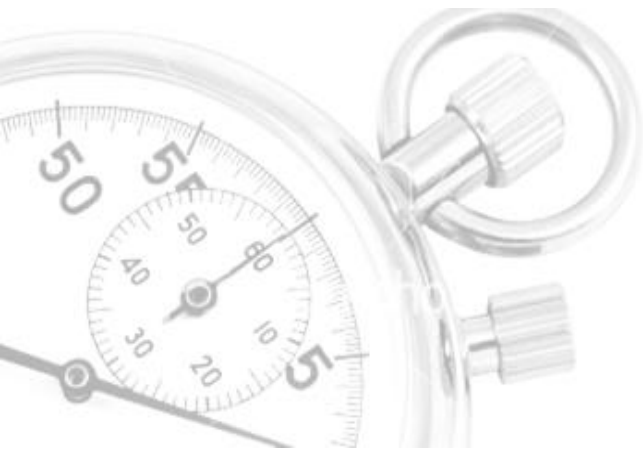


Reactive

Synthèse



Code réactif non bloquant Persistance orienté flux



Haute dispo Fail over

Capacité

Répartition par valeur
(Sharding)

Répartition par type
(agent) ✓

Réplication

Multi-thread

Threads

Un processus
par client

Un thread par
client

Un thread par
client actif (pool)

Un thread par
plusieurs clients ✓

État en mémoire

Partage complet

Immuable ✓

Stateless

Monothread

STM

Persistance

Historique

UPDATE

NO UPDATE ✓



**Haute dispo
Fail over**

Capacité

Répartition par
valeur

Répartition par
type (agent)

Réplication

Multi-thread

Threads

Un processus
par client

Un thread par
client

Un thread par
client actif (pool)

Un thread
plusieurs clients

Persistance

Historique

UPDATE

NO UPDATE



**Haute dispo
Fail over**

Multi-thread

Persistance

Capacité

Répartition par valeur

Répartition par type (agent)

Réplication

Threads

Un processus par client

Un thread par client

Un thread par client actif (pool)

Un thread plusieurs clients

État en mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Historique

UPDATE

NO UPDATE



Haute dispo
Fail over

Multi-thread

Persistance

Capacité

Répartition par
valeur

Répartition par
type (agent)

Réplication

Threads

Un processus
par client

Un thread par
client

Un thread par
client actif (pool)

Un thread
plusieurs clients

État en
mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Historique

UPDATE

NO UPDATE



**Haute dispo
Fail over**

Multi-thread

Persistance

Capacité

Répartition par
valeur

Répartition par
type (agent)

Réplication

Threads

Un processus
par client

Un thread par
client

Un thread par
client actif (pool)

Un thread
plusieurs clients

État en
mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Historique

UPDATE

NO UPDATE



Haute dispo Fail over

Capacité

Répartition par valeur

Répartition par type (agent)

Réplication

Multi-thread

Threads

Un processus par client

Un thread par client

Un thread par client actif (pool)

Un thread plusieurs clients

État en mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Persistence

Historique

UPDATE

NO UPDATE



Haute dispo Fail over

Capacité

Répartition par valeur

Répartition par type (agent)

Réplication

Multi-thread

Threads

Un processus par client

Un thread par client

Un thread par client actif (pool)

Un thread plusieurs clients

Persistence

État en mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Historique

UPDATE

NO UPDATE



Haute dispo Fail over

Capacité

Répartition par valeur

Répartition par type (agent)

Réplication

Multi-thread

Threads

Un processus par client

Un thread par client

Un thread par client actif (pool)

Un thread plusieurs clients

État en mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Persistance

Historique

UPDATE

NO UPDATE



Haute dispo
Fail over

Multi-thread

Persistence

Capacité

Répartition par valeur

Répartition par type (agent)

Réplication

Threads

Un processus par client

Un thread par client

Un thread par client actif (pool)

Un thread plusieurs clients

État en mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Historique

UPDATE

NO UPDATE



**Haute dispo
Fail over**

Multi-thread

Persistence

Capacité

Répartition par
valeur

Répartition par
type (agent)

Réplication

Threads

Un processus
par client

Un thread par
client

Un thread par
client actif (pool)

Un thread
plusieurs clients

État en
mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Historique

UPDATE

NO UPDATE



**Haute dispo
Fail over**

Capacité

Répartition par
valeur

Répartition par
type (agent)

Réplication

Multi-thread

Threads

Un processus
par client

Un thread par
client

Un thread par
client actif (pool)

Un thread
plusieurs clients

État en
mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Persistence

Historique

UPDATE

NO UPDATE

 **EHCACHE**

Haute dispo
Fail over

Multi-thread

Persistence

Capacité

Répartition par valeur

Répartition par type (agent)

Réplication

Threads

Un processus par client

Un thread par client

Un thread par client actif (pool)

Un thread plusieurs clients

État en mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Historique

UPDATE

NO UPDATE



Haute dispo
Fail over

Multi-thread

Persistence

Capacité

Répartition par valeur

Répartition par type (agent)

Réplication

Threads

Un processus par client

Un thread par client

Un thread par client actif (pool)

Un thread plusieurs clients

État en mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Historique

UPDATE

NO UPDATE



**Haute dispo
Fail over**

Multi-thread

Persistance

Capacité

Répartition par
valeur

Répartition par
type (agent)

Réplication

Threads

Un processus
par client

Un thread par
client

Un thread par
client actif (pool)

Un thread
plusieurs clients

État en
mémoire

Partage complet

Immuable

Stateless

Monothread

STM

Historique

UPDATE

NO UPDATE



- > Révisez vos architectures
 - + Réduction des coûts
 - + Amélioration notablement des performances
 - + Adaptation aux nouveaux besoins
- > Approche réactive (Dataflow program)
 - + Consultez le www.reactivemanifesto.org !
- > Supprimez les DELETE/UPDATES
 - + Vos revenus en dépendent
- > Allez y maintenant !!!





<https://www.linkedin.com/jobs2/view/91749910>

Vous croyez que *les technologies* changent le monde ?

Nous aussi ! Rejoignez-nous !

recrutement@octo.com

Questions ?



Merci de votre attention

contact :

PHILIPPE PRADOS

Architecte Senior, Leader Tribu Réactive

pprados@octo.com

Tél. : 06 20 66 71 00

> Notes

