# Algorithmique répartie

## Notion de cohérence dans les mémoires réparties

Gérard Padiou

Département Informatique et Mathématiques appliquées
ENSEEIHT

5 novembre 2012

# plan



# Plan



# Notion de mémoire répartie

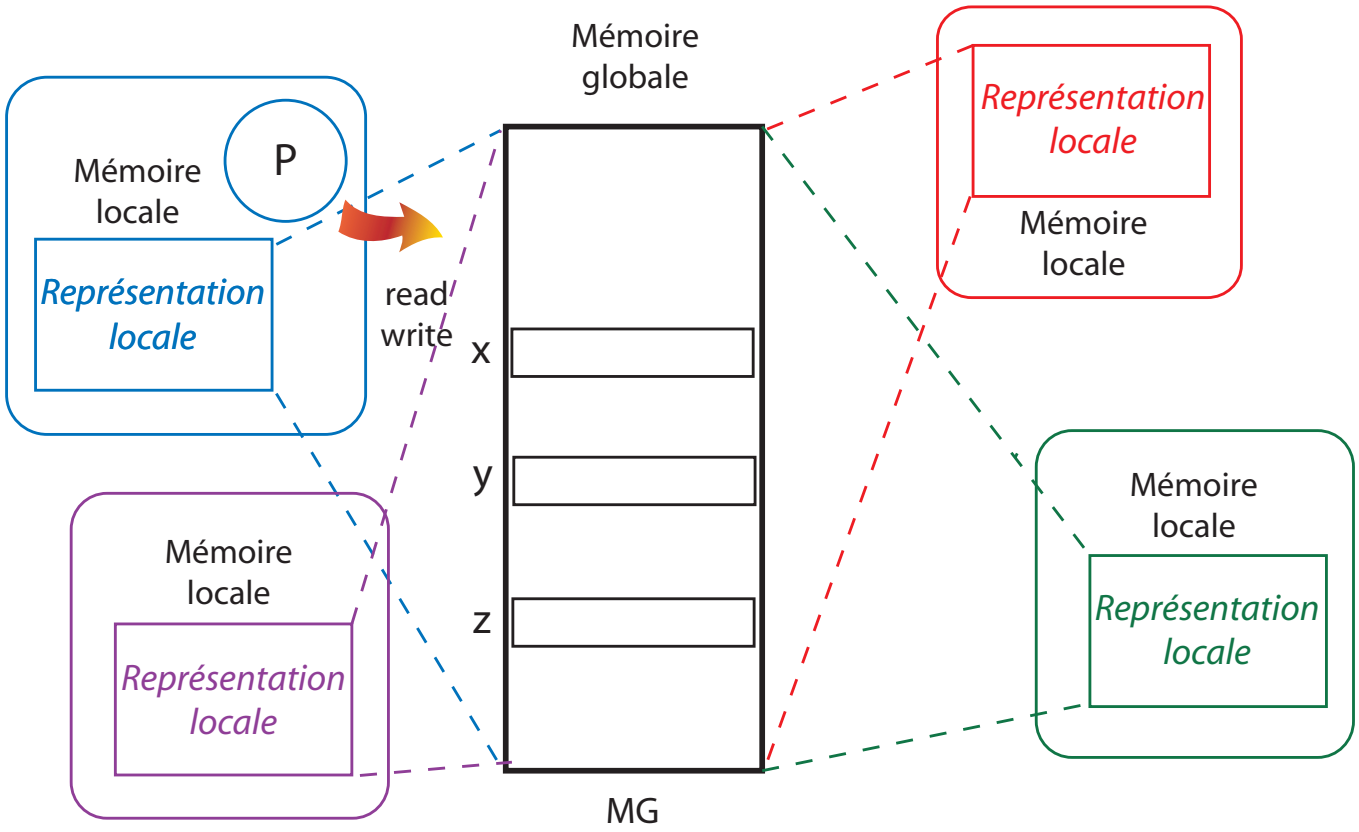
## Objectif

- Fournir une mémoire globale sur une architecture répartie ;
- Plus simple à utiliser ;
- Transparence de la répartition ;
- Utiliser les mémoires locales pour construire une abstraction « centralisée » ;

## Difficulté

- Problème de sémantique en réparti ;
- Plusieurs types de cohérence.

# Notion de mémoire répartie



# Modélisation d'une mémoire

## Approche événementielle

### Classes d'événements

- $W_i(x)v$ : le processus $i$ affecte la valeur $v$ à la variable $x$ ;
- $R_i(x)v$ : le processus $i$ lit la valeur $v$ dans la variable $x$ ;

### Cohérence mémoire habituelle (centralisée)

La valeur lue est la dernière écrite

- Problème : sous-entend un ordre total entre lectures et écritures ;
- En réparti : seulement un ordre partiel (causal).

# Cohérence forte

## Sémantique centralisée

Valeur lue = dernière valeur écrite dans le temps réel

## Difficultés en réparti

- Pas de temps global ;
- Délai des messages.

## Solution

- Nécessite de construire un ordre total dans un temps logique ;
- Trop coûteux, délais prohibitifs.

# Plan



# Exemple avec 2 processus

## Valeurs initiales des variables globales

```
int x := 0 ; int y := 0;
```

## process $P_1$

```
x := 1 ;
print(x,y);
```

## process $P_2$

```
y := 2 ;
print(x,y);
```

En terme d'événements d'écriture et lecture :

## process $P_1$

$W_1(x)1$ ;
$R_1(x, y)[1, ?]$

## process $P_2$

$W_2(y)2$ ;
$R_2(x, y)[?, 2]$

# Exemple

## Les exécutions possibles

### Histoires individuelles

- pour $P_1$ :
  - soit $W_1(x)1; R_1(x,y)[1,0]$
  - soit $W_1(x)1; R_1(x,y)[1,2]$
- pour $P_2$ :
  - soit $W_2(y)2; R_2(x,y)[0,2]$
  - soit $W_2(y)2; R_2(x,y)[1,2]$

### Histoire « globale » des écritures

- soit $W_1(x)1; W_2(y)2$
- soit $W_2(y)2; W_1(x)1$

# Exemple
## Notion de cohérence

### Histoires cohérentes

- soit $\{x = 0 \wedge y = 0\}\langle P_1 \| P_2 \rangle \{R_1(x, y)[1, 0] \wedge R_2(x, y)[1, 2]\}$
- soit $\{x = 0 \wedge y = 0\}\langle P_1 \| P_2 \rangle \{R_1(x, y)[1, 2] \wedge R_2(x, y)[0, 2]\}$
- soit $\{x = 0 \wedge y = 0\}\langle P_1 \| P_2 \rangle \{R_1(x, y)[1, 2] \wedge R_2(x, y)[1, 2]\}$

### Certains états n'ont pas d'histoires globales

$$\{x = 0 \wedge y = 0\}\langle P_1 \| P_2 \rangle \{R_1(x, y)[1, 0] \wedge R_2(x, y)[0, 2]\}$$

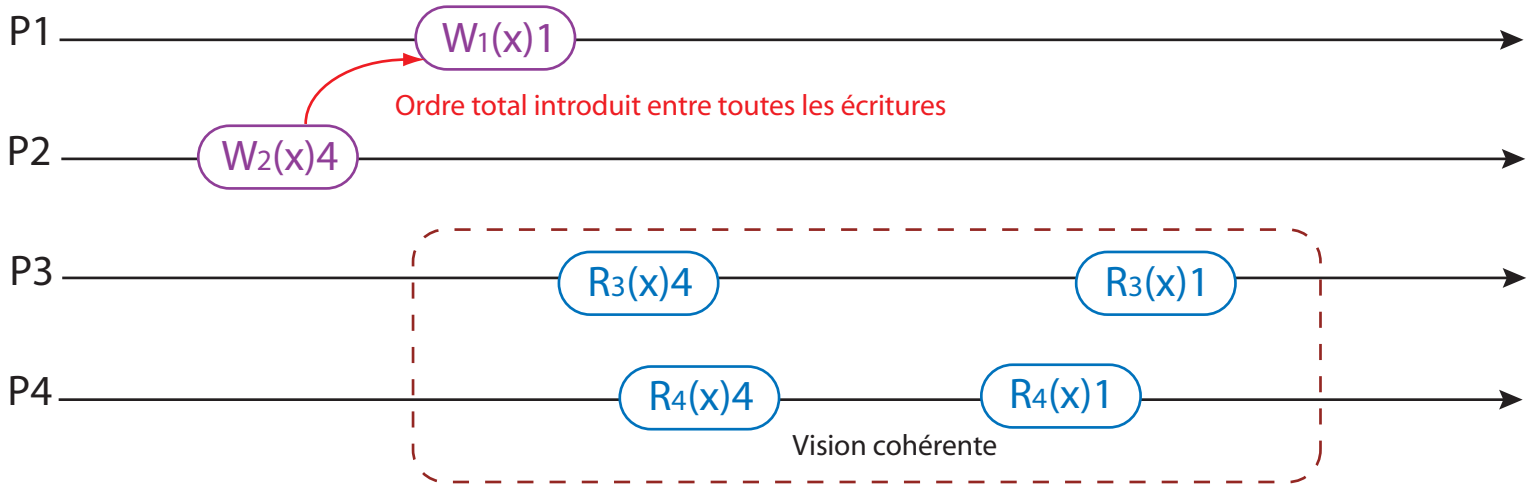
| | | | |
|---|---|---|---|
| ? | $W_1(x)1$ | $W_2(y)2$ | ? |
| $\uparrow$ | $\uparrow$ | $\uparrow$ | |
| $\uparrow$ | $R_1(x, y)[1, 0]$ | $\uparrow$ | |
| $R_2(x, y)[0, 2]$ | | $R_2(x, y)[0, 2]$ | |

# Plan



# Cohérence séquentielle

- On impose un ordre total sur les écritures ;
- Les lectures locales à chaque processus lisent la dernière valeur écrite ;



# Plus formellement

- Chaque processus $P_i$ engendre une trace (histoire) totalement ordonnée de lectures/écritures $h_i$ ;
- Une exécution engendre une histoire globale partiellement ordonnée par la relation causale $\prec_H = \cup_i \prec_{h_i}$ ;
- Une histoire est séquentielle si l'ordre associé est totalement ordonné ;
- Une histoire séquentielle est légale ssi à tout événement de lecture $r(x)v$ est associé un événement d'écriture $w(x)v$ tel que [1] :

$$\nexists\ w(x)v' : w(x)v \prec w(x)v' \prec r(x)v$$

1. Par convention (et simplification), on suppose que toutes les valeurs écrites sont différentes.

# Plus formellement (suite)

- Deux histoires $H_1$ et $H_2$ sont équivalentes $H_1 \equiv H_2$ ssi $H_1|_i = H_2|_i$ (histoires locales identiques) ;
- Un système de mémoire réparti est séquentiellement cohérent ssi pour toute exécution, l'histoire $H$ de l'exécution est équivalente à une histoire séquentielle légale $WR$ : $H \equiv WR$.

## Exemple

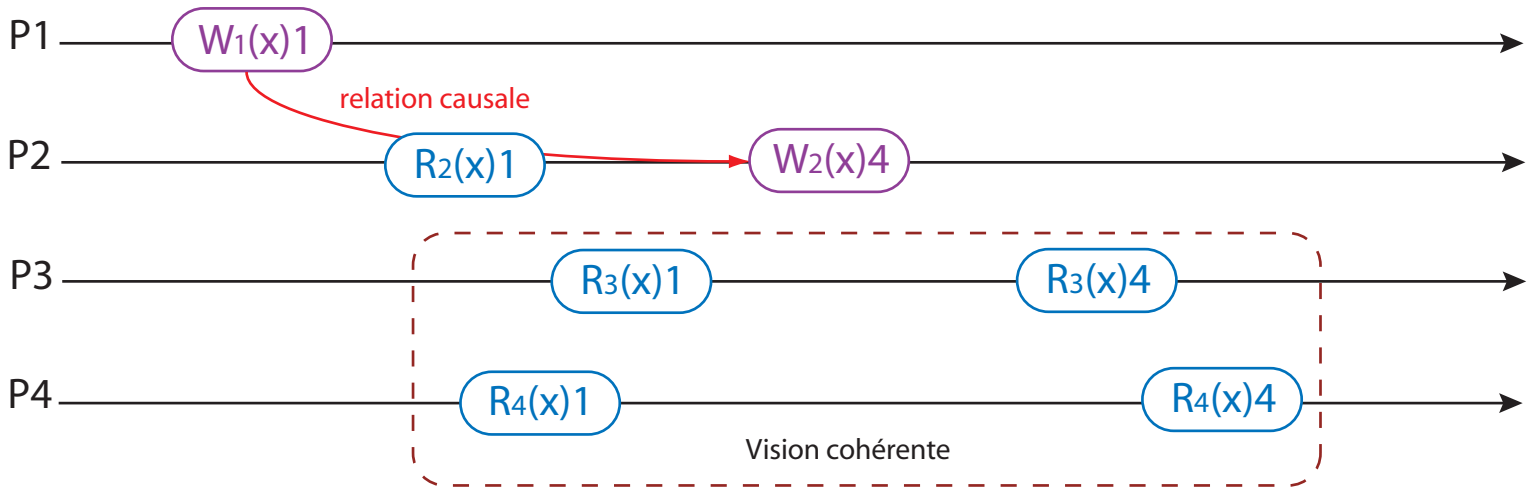
- $h_1 = w_1(x)1 \prec w_1(y)2 \prec r_1(x)4$
- $h_2 = w_2(y)3 \prec w_2(x)4 \prec r_2(y)2$
- $H \in h_1 \Psi h_2$ : $H \equiv WR$ avec :

$$WR = w_1(x)1 \prec w_2(y)3 \prec w_2(x)4 \prec w_1(y)2 \prec r_2(y)2 \prec r_1(x)4$$

# Cohérence causale

## Hutto et Ahamad (1990)

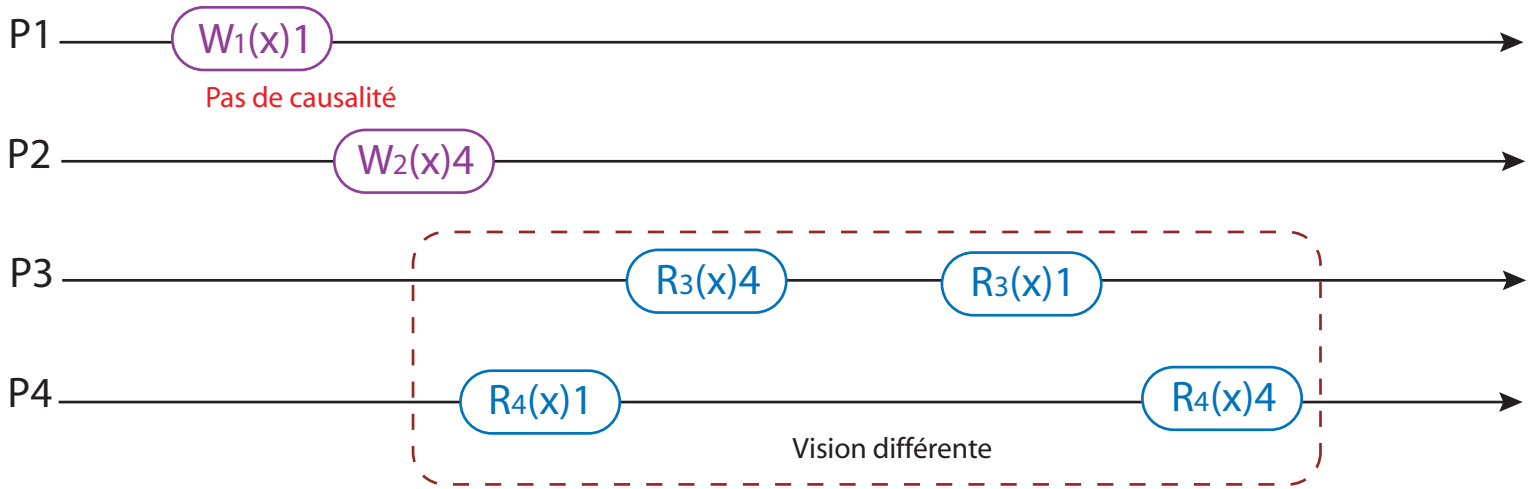
- Les processus perçoivent correctement les écritures causalement liées ;
- Les écritures non causalement liées peuvent être vues dans un ordre différent par des processus distincts.



# Cohérence causale (suite)

## Hutto et Ahamad (1990)

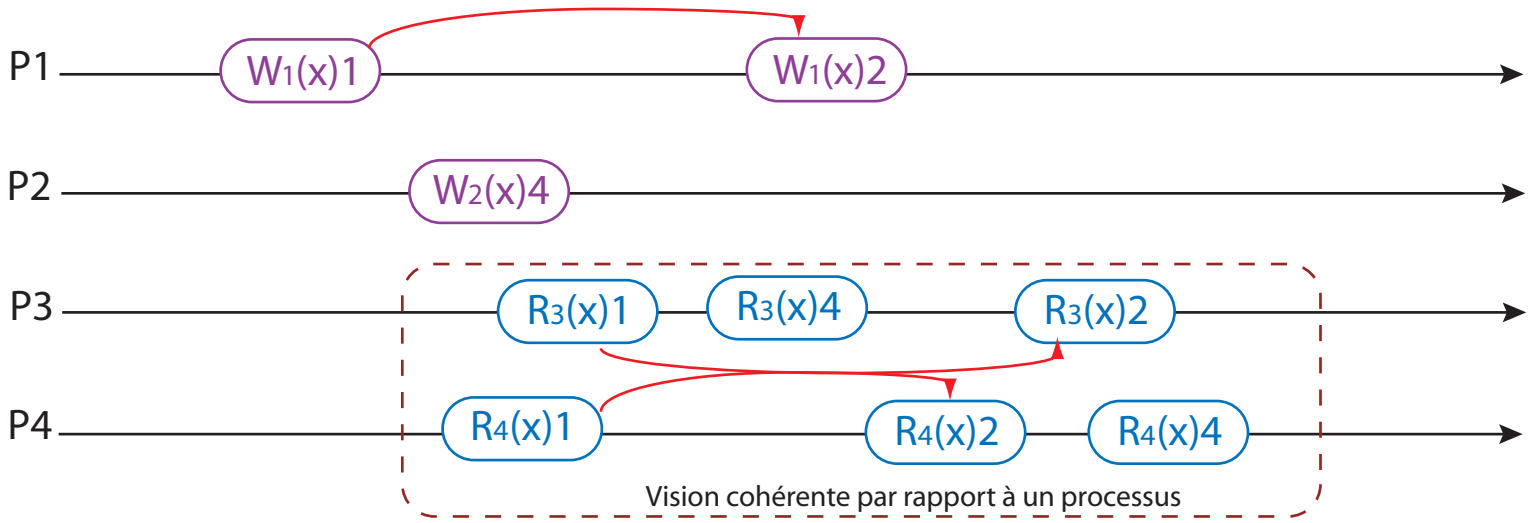
- Les processus perçoivent correctement les écritures causalement liées ;
- Les écritures non causalement liées peuvent être vues dans un ordre différent par des processus distincts.



# Cohérence PRAM (Pipelined RAM)

Lipton et Sandberg (1988)

- Les écritures issues d'un processus sont vues dans le même ordre par TOUS les autres processus ;
- Les écritures issues de processus distincts peuvent être vues dans un ordre différent ;
- Avantage : mise en œuvre simple.

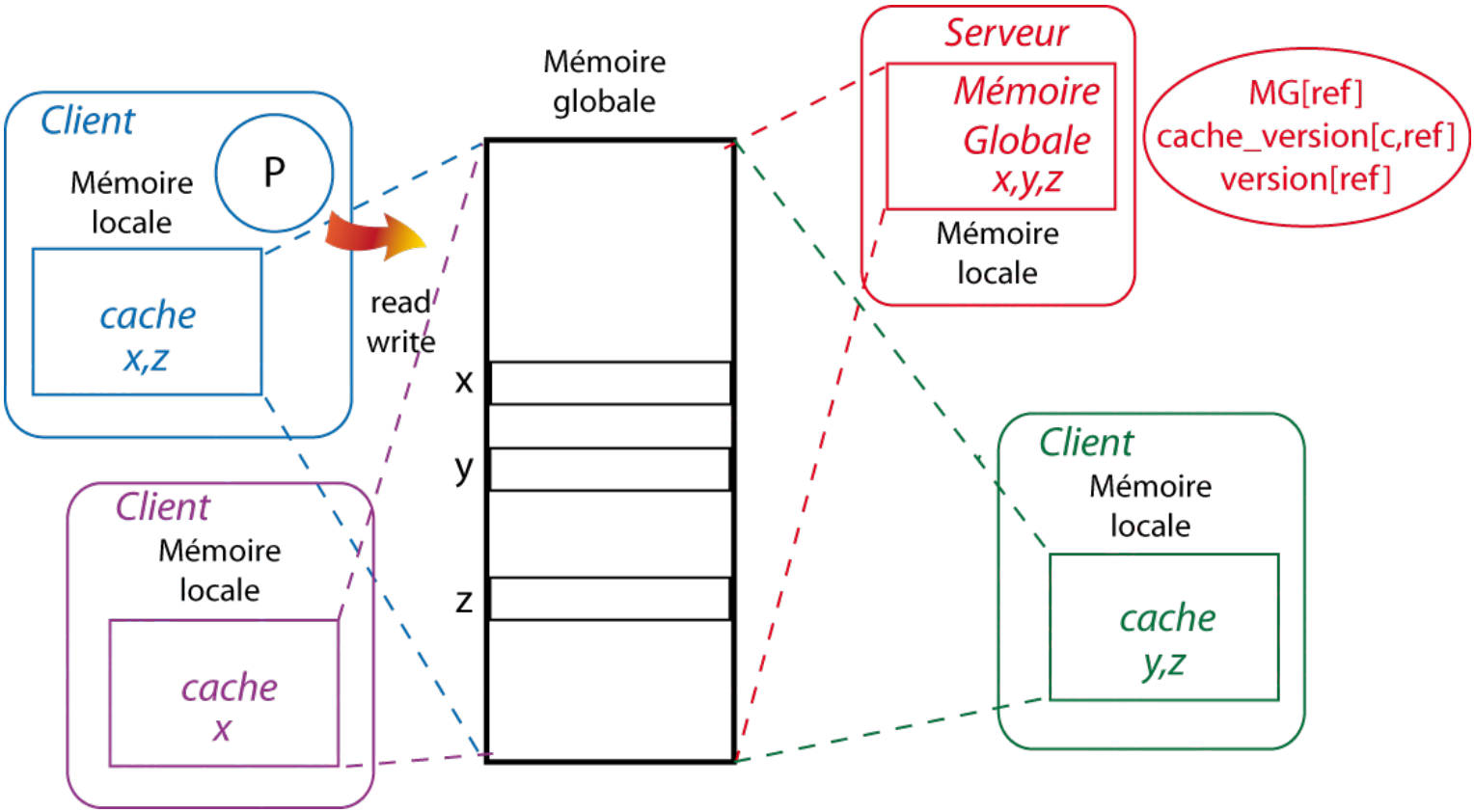


# Plan



# Une solution de type client/serveur

## Usage de caches mais aussi d'un serveur unique « ordonnanceur »



# L'algorithme

## Principes

☞ L'algorithme garantit la cohérence mémoire causale

---

- Chaque site gère un cache de données ;
- Une lecture se fait dans le cache local si possible sinon à distance (auprès du site serveur) ;
- Toute écriture est transmise au site serveur ;
- Les données en cache sont éventuellement invalidées en réponse à des écritures ou lectures distantes ;
- L'invalidation utilise un mécanisme de version sur les écritures.

# Structures de données de l'algorithme

## sur le serveur :

- `MG[ref]` : la mémoire globale ;
- `version[ref]` : les numéros de version courante de chaque mot référencé ;
- `version_cache[c,ref]` : les numéros de version des références en cache chez chaque client.

## pour un client $c$ :

- $\texttt{cache}_c\texttt{[ref]}$ : cache du client $c$ pour toute variable référencée `ref` ;
- $\texttt{valide}_c$ : l'ensemble des références de variables valides dans le cache du client $c$.

# Algorithmes des opérations : site client

## Lecture sur un site Client $c$

```
value read(ref x) {
  if (x ∉ valide_c) { /* x n'est pas en cache local */
   invalide,v = Serveur.read(x,c) ;  /* - RPC - */
   valide_c = (valide_c-invalide)+{x} ; cache_c[x] = v ;
  }
  return cache_c[x]
}
```

## Écriture sur un site Client $c$

```
write(ref x, value v) {
  invalide = Serveur.write(x,v,c) ; /* - RPC - */
  valide_c = (valide_c-invalide)+{x} ; cache_c[x] = v ;
}
```

# Algorithmes des opérations : site serveur

## Lecture sur le serveur : appel d'un client *c*

```
⟨set<ref>,v⟩ read(ref x, client c) {
  cache_version[c,x]=version[x];
  return ⟨invalider(c,x),MG[x]⟩
}
```

## Écriture sur le site Serveur : appel d'un client *c*

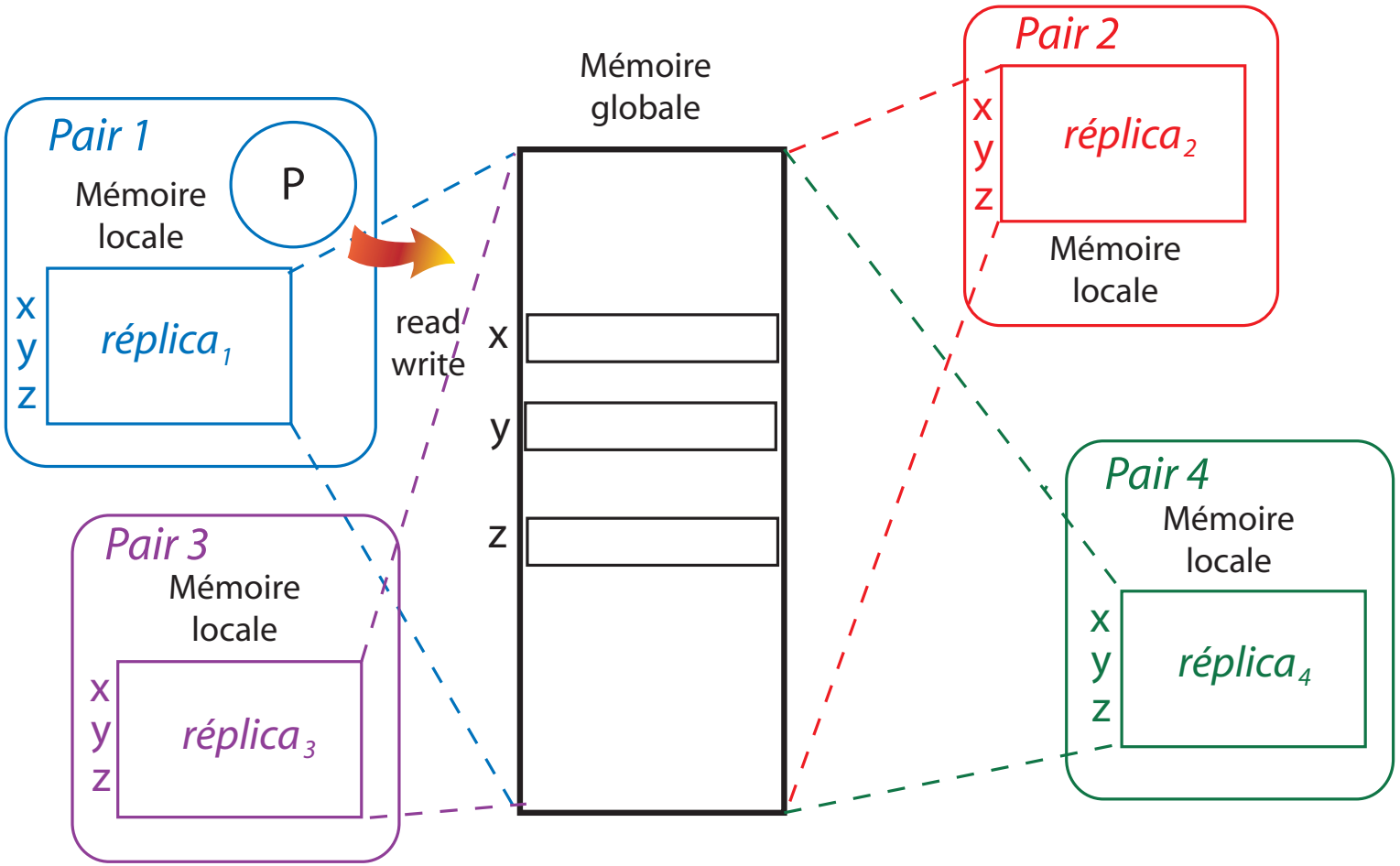
```
set<ref> write(ref x, value v, client c) {
  MG[x]=v; version[x]++; /* nouvelle version */
  cache_version[c,x]=version[x];
  return invalider(c,x)
}
```

# Algorithmes des opérations : site serveur

## Algorithme d'invalidation

```
set<ref> invalider(client c, ref x) {
  set<ref> invalide = {};
  for (ref y ∈ MG && y ≠ x) {
   if (cache_version[c,y]≠0 // c possède y en cache
   && cache_version[c,y]<version[y]){ // cache obsolète
    invalide = invalide + {y};
    cache_version[c,y] = 0; // y effacé du cache de c
   }
  }
  return invalide
}
```

# Une solution de type répliquée



# Principes de l'algorithme

## Principes

☞ L'algorithme garantit la cohérence mémoire séquentielle

---

- Chaque site gère une copie de la mémoire globale répliquée ;
- Une lecture se fait toujours sur le réplica local ;
- Toute écriture est diffusée à tous les sites ;
- Pour obtenir la cohérence séquentielle :
  $\Rightarrow$ utiliser un protocole de diffusion totalement ordonnée ;
- Lecture optimale (locale) ;
- Écriture coûteuse (diffusion totalement ordonnée)

# Algorithme des opérations

## Lecture sur le site *i*

```
value read(ref x) { return réplica_i[x] }
```

## Écriture sur le site *i*

```
write(ref x, value v) {
  abroadcast(x,v,i) ; /* diffusion tot. ordonnée */
  écrit_i[x].wait() ;
}
```

## Réception d'une requête d'écriture issue du site *i* sur le site *j*

```
upon receive(ref x, value v, site i) {
  réplica_j[x]= v ;
  if (j==i) écrit_i[x].notify() ;
}
```

# Une solution de type répliquée

## Un compromis

### Partage des lectures et écritures distantes

- Un lecteur lit $r$ copies ;
- Un rédacteur écrit la valeur dans $w$ copies.

### Conditions nécessaires de cohérence

- $w + r > N$ : on lit toujours la dernière version
- $w > \frac{N}{2}$
- Gestion d'estampilles fixant un ordre total sur les écritures.

☞ Notion de quorum