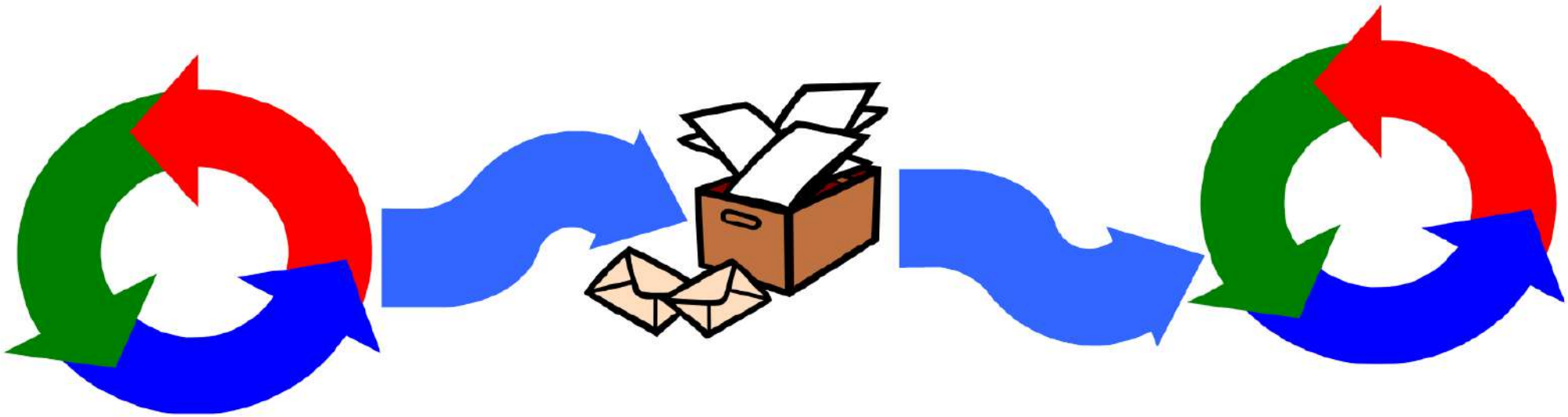
Systèmes et Applications Distribués
Communication

Synchronisation par échange de messages : application sur les Sockets



Plan du cours

- Introduction
  - Définitions
  - Problématique
  - Architectures de distribution
- Distribution intra-applications
  - Notion de processus
  - Programmation multi-thread
- Distribution inter-applications et inter-machines
  - Les Sockets
  - middlewares par appel de procédures distantes (RPC)
  - middlewares par objets distribués (Java RMI)
  - middlewares par objets distribués hétérogènes (CORBA/GRPC)
- Conclusion

Introduction : pourquoi la communication?

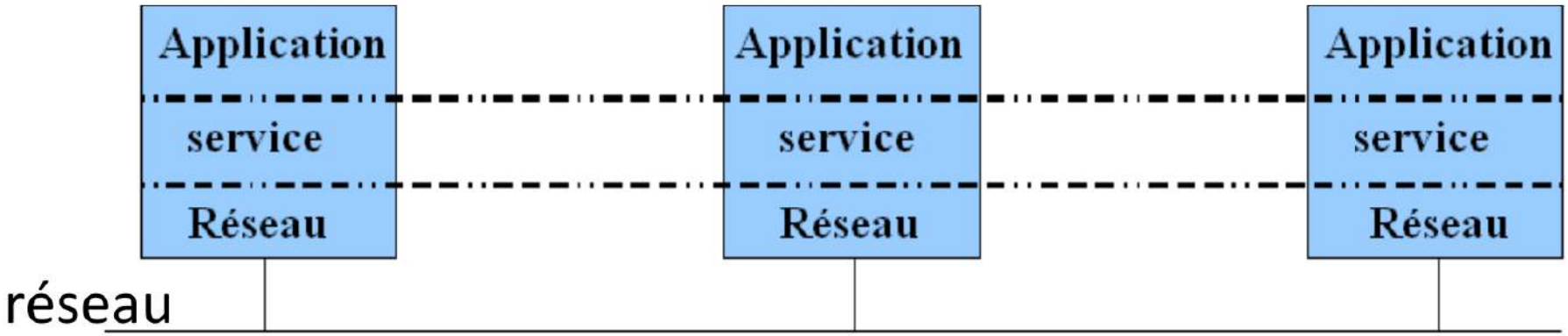
- Nous avons vu la synchronisation entre processus par partage de variables (mémoire commune).
  - La cohérence des états par exclusion mutuelle;
  - Détection de blocage.
- Mais quand les processus se trouvent sur deux sites distincts, la synchronisation se fait par envoi et réception de messages.
- Les messages contiennent des valeurs qui influencent le déroulement de l'exécution du récepteur.

Modèle de Distribution

- Le modèle de distribution considéré dans ce cours est composé de :
- Un ensemble de sites
  - Chaque site possède sa propre mémoire non accessible aux autres sites;
  - Chaque site dispose d'un identifiant unique (IP ou adresse MAC).
- Des lignes de communication
  - Bi-point : reliant deux sites;
  - Bi-directionnelle : l'échange est possible dans les deux directions;
  - Chaque direction est appelée "canal";
  - On considère le graphe résultant comme une clique : chaque deux sites peuvent physiquement échanger des données.

Les sites : modèle en couches

- Nous nous intéressons à chaque site en tant que composante d'une application distribuée.
- Conçue selon un modèle en couches.



- **Chaque couche fournit un ensemble de services aux couches supérieures.**
- **Objectif : masquer les difficultés d'implémentation**

Les couches

- La couche réseau et le réseau
  - Un canal de communication entre deux sites a les propriétés :
    - Les données ne sont pas altérées ;
    - Les messages ne sont pas perdus (pas toujours vrai, on va la relaxer) ;
    - Le canal est FIFO : les messages arrivent dans l'ordre de leurs envois.
  - Le réseau est considéré comme :
    - Asynchrone : le délai de transit est indéfini (le cas considéré ici) ;
    - Synchrone : le délai est borné et connu par le concepteur.
- La couche services
  - Est une API qui offre un certain nombre de primitives sous forme d'API à la couche application;
  - Offre les primitives d'envoi et de réception de messages ;
  - Utilise les primitives de l'API et de la couche réseau pour offrir ses services.
- La couche application
  - => c'est la couche qui nous intéresse dans ce cours avec celle du service.

La communication

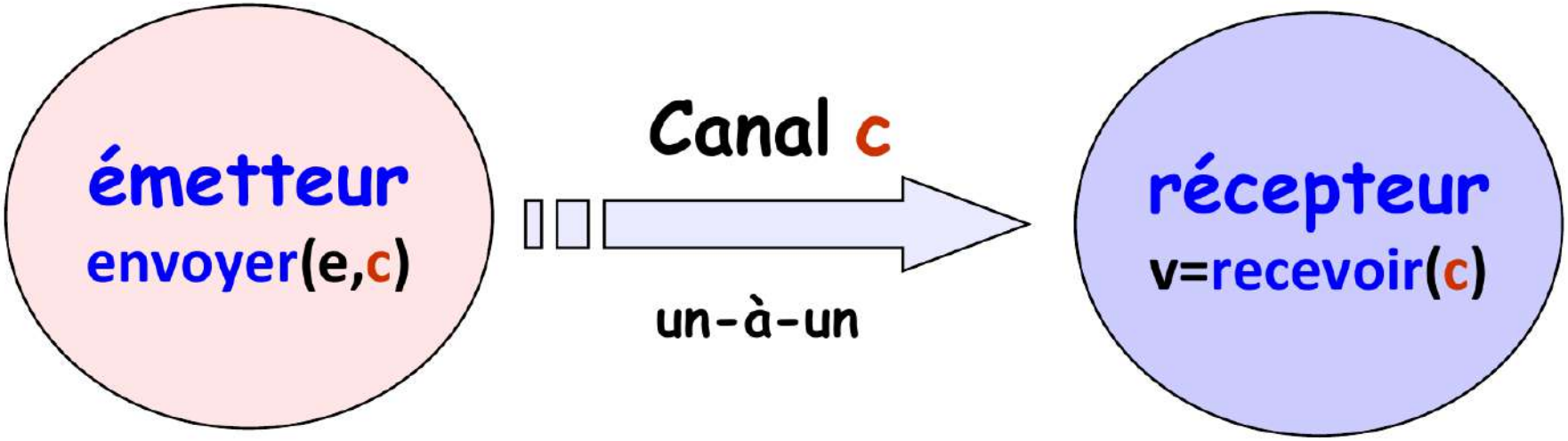
- Plusieurs définitions, mais on va garder une définition du point de vue de la couche application.
- Une communication est une suite de trois actions
  - L'envoi ;
  - Le transport (ne nous intéresse pas dans ce cours) ;
  - La réception .
- On s'intéresse à la sémantique des deux actions de communication sur chaque site et sur l'application distribuée.
- Ici on va voir les modèles les plus connus et utilisés
  - Communication synchrone ;
  - Communication asynchrone ;
  - Communication par rendez-vous.
- Attention à ne pas confondre avec le synchrone et asynchrone du réseau

Communication synchrone

- La communication est dite synchrone quand les actions d'envoi et de réception ne sont possibles que si :
  - L'émetteur se trouve dans un état d'envoi;
  - Et le récepteur dans un état de reception.
- La communication orale doit être synchrone, car celui qui parle ne parle que si il sait que son interlocuteur est dans un état d'écoute.
- Une autre façon de modéliser la communication synchrone est de la considérer comme une émulation de l'opération d'affectation distribuée.
- Mettre une valeur locale dans une variable distante ou mettre un valeur distante dans une variable locale[2].

  => D'où la modélisation en utilisant la notion de **canal** de communication.

Communication synchrone – notion de canal



envoyer(**e**,**c**) - envoie la valeur de l'expression e sur le canal c. Le processus appelle l'opération envoyer et se **bloque** jusqu'à réception du message par le récepteur.

v **= recevoir(*c*)** - recevoir une valeur dans la variable v à partir du canal c. le processus qui appelle **recevoir** se **bloque** jusqu'à ce qu'une valeur soit envoyée sur le canal.

affectation distribuée ***v = e***

Enrichissons notre langage avec les actions d'envoi et de réception

- On va rajouter dans notre langage :

EXPR ::= CONSTANTE | **CANAL** | VARIABLE
| EXPR+EXPR | EXPR*EXPR
| EXPR/EXPR | EXPR-EXPR

TEST ::= EXPR==EXPR | EXPR < EXPR |
EXPR > EXPR | TEST & TEST | TEST OU
TEST | ! TEST

INSTR ::= VARIABLE=EXPR | **CANAL!EXP** |
CANAL?VARIABLE

BLOCK ::= ε | INSTR;BLOCK
| if TEST
then BLOCK
else BLOCK
| while TEST
BLOCK ;

P1

```
int x=1;
c!x;
x+=1;
print(x);
```

x=

P2

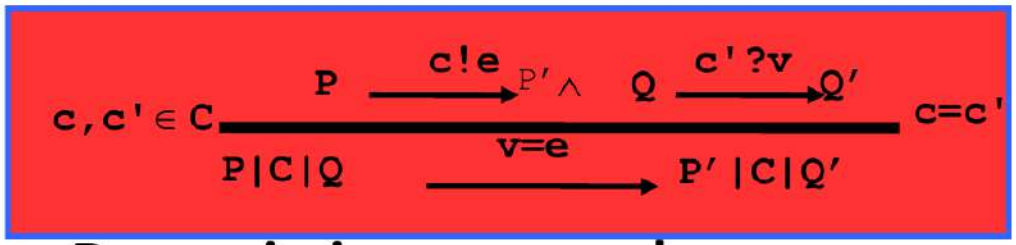
```
int y=0;
c?y;
print(y)
```

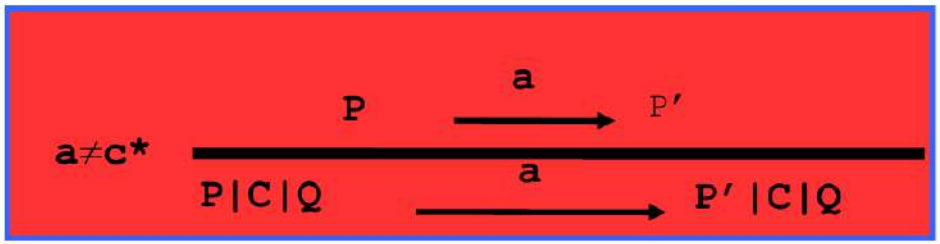
y=

```
Canal c;
new P1(c).start();
new P2(c).start();
```

La sémantique des actions de communication

- Localement d'abord
  - Envoi : $c!e;P \xrightarrow{c!e} P$
  - Réception $c?v;P \xrightarrow{c?v} P$
- Sémantique dans le cas de la concurrence
  - On introduit un opérateur de composition $P|C|Q$
    - P et Q deux processus de notre nouveau langage et C un ensemble de noms de canaux
  - Voici la sémantique opérationnelle de composition par des canaux





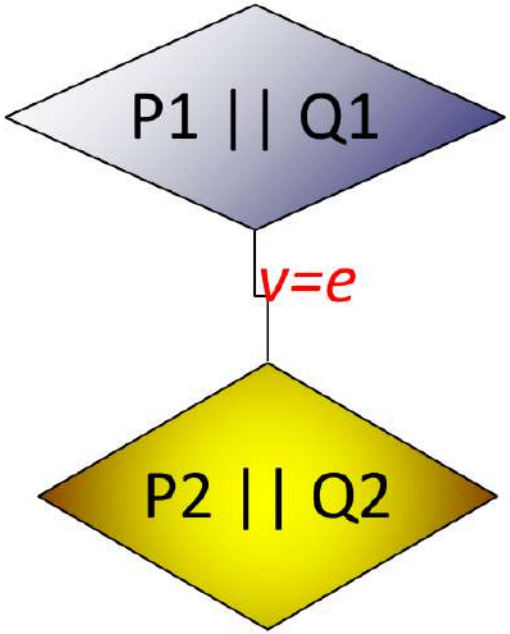
- **Par omission on conclue:**
  - Un processus qui est prêt a émettre/recevoir sur un canal est bloqué tant qu'aucun autre processus n'est capable de faire une action complémentaire sur ce même canal.
- **La synchronisation d'envoi/réception sur un canal est alors exécutée entre seulement deux processus à la fois (*un-à-un*).**
- **L'action synchrone résultante est équivalente à une affectation de la valeur envoyée vers la variable de réception.**

La communication synchrone synchronise les applications

- *ev* précède causalement *ev' (ev --> ev')* si [lamport 78]:
  - *ev* précède localement *ev'* (sur 1 site, dans 1 processus), ou
  - ∃ un message *m* tel que *ev = émission(m)*, *ev' = réception(m)*, ou
  - ∃ *ev"* tel que (*ev --> ev"*) et (*ev" --> ev'*)

P=P1;c!e;P2

Q=Q1;c?v;Q2



Émulation java d'une communication synchrone

- Cas producteur / consommateur :
- Canal : Mémoire tampon de taille 1
- Émetteur :
  - Le producteur
- Récepteur:
  - Le consommateur
- Mais il y a quelque chose qui change.
  - Le producteur ne finit de produire que
    - S'il a fini de mettre l'objet dans le tampon
    - ET que le consommateur a consommé ce qui a été produit.
  - Le consommateur reste inchangé.

Émulation du canal

```
public class Canal<E>  {
   private E message = null;

   public synchronized void envoyer(E v)
      throws InterruptedException {
      message = v;
      notify(); // notifyAll();
      wait(); // while(message != null)
  }

   public synchronized E recevoir()
      throws InterruptedException {

      if(message == null) wait(); // while()
      E tmp = message; message = null;
      notify();           // notifyAll()
      return(tmp);
  }}
```

Émulation de l'émetteur

```
public class Emetteur implements Runnable {
  private Canal<Integer> canal;
  private SlotCanvas display;
  public Emetteur(Canal<Integer> c, SlotCanvas d){
   canal=c; display=d;}

  public void run() {
    try {   int ei = 0;
      while(true) {
        display.enter(String.valueOf(ei));
        ThreadPanel.rotate(12);
        canal.envoyer(Integer.valueOf(ei));
        display.leave(String.valueOf(ei));
        ei=(ei+1)%10;
       ThreadPanel.rotate(348);
      }
    } catch (InterruptedException e){}
  }}
```

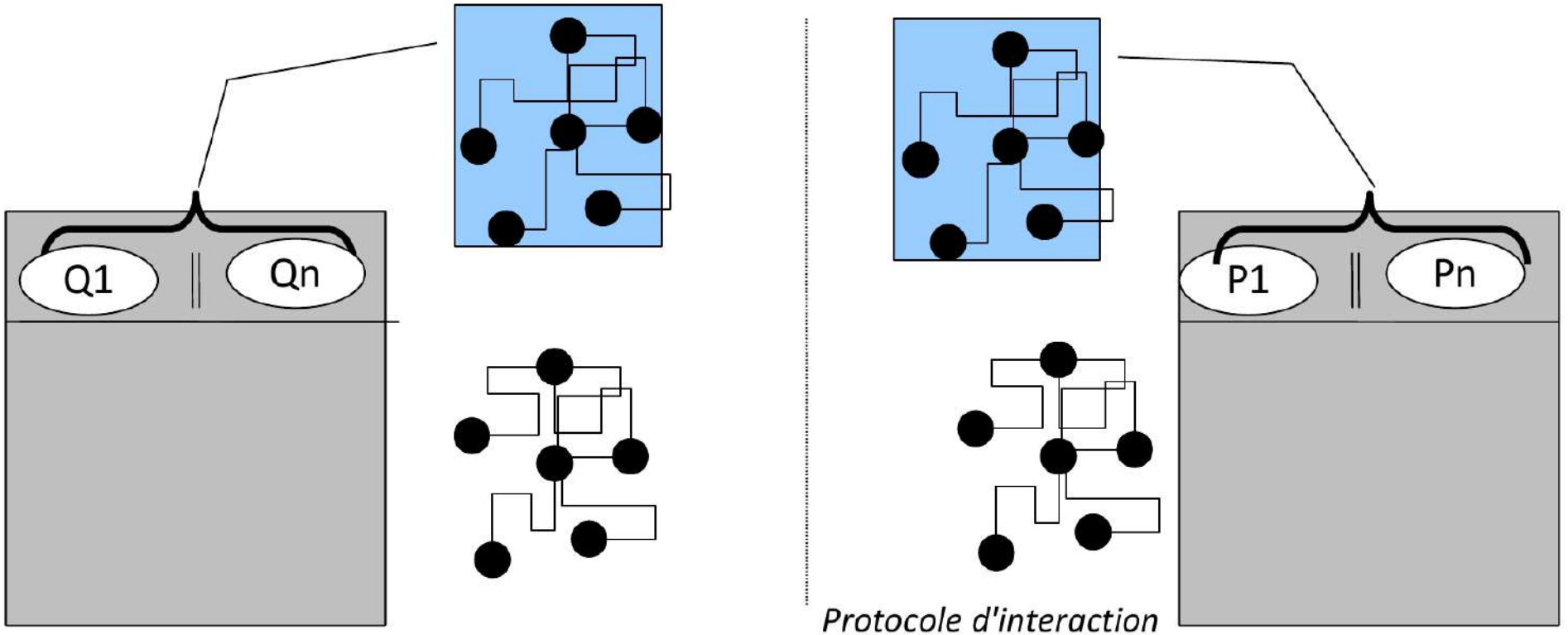
Émulation du récepteur

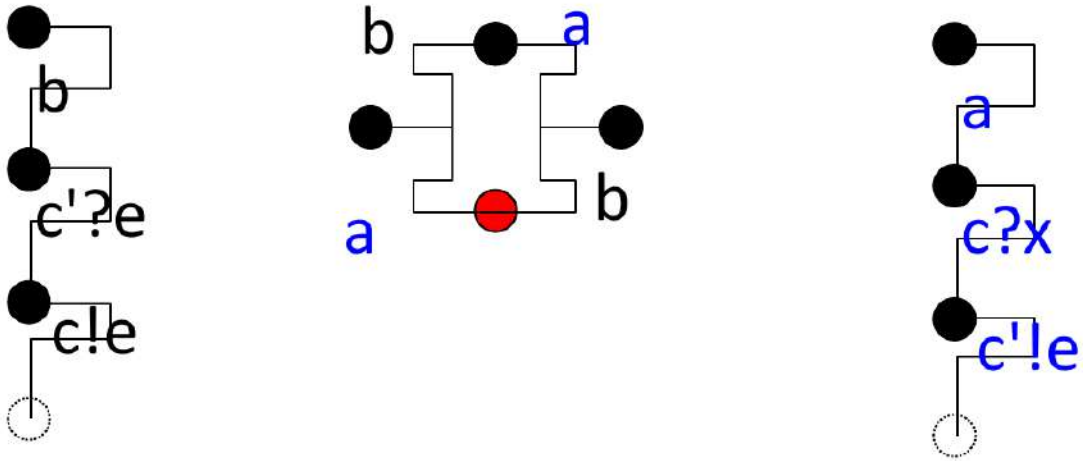
```
public class Recepteur extends Thread {
  private Canal<Integer> canal;
  private SlotCanvas display;
  public Recepteur(Canal<Integer> c, SlotCanvas d){
   canal=c; display=d;}

  public void run() {
    try { Integer v = null;
      while(true) {
        ThreadPanel.rotate(180);
        if(v!=null) display.leave(v.toString());
        v = canal.recevoir();
        display.enter(v.toString());
        ThreadPanel.rotate(180);
      }
    } catch (InterruptedException e){}
  }
}
```

Émulation java- communication synchrone

Caractérisation d'une application distribuée notion de deadlock





Avantage et inconvénient de la communication synchrone.

- Communiquer est une action de synchronisation.
- Un moyen efficace pour le contrôle d'exécution des applications distribuées
  - => des applications simples
- Contraignant quand la synchronisation n'est pas nécessaire, mais juste l'échange de données.
  - Beaucoup de synchronisation tue la concurrence (synonyme d'efficacité).
- Une application mal conçue => risque de blocages.
- Permet seulement une communication un-à-un.

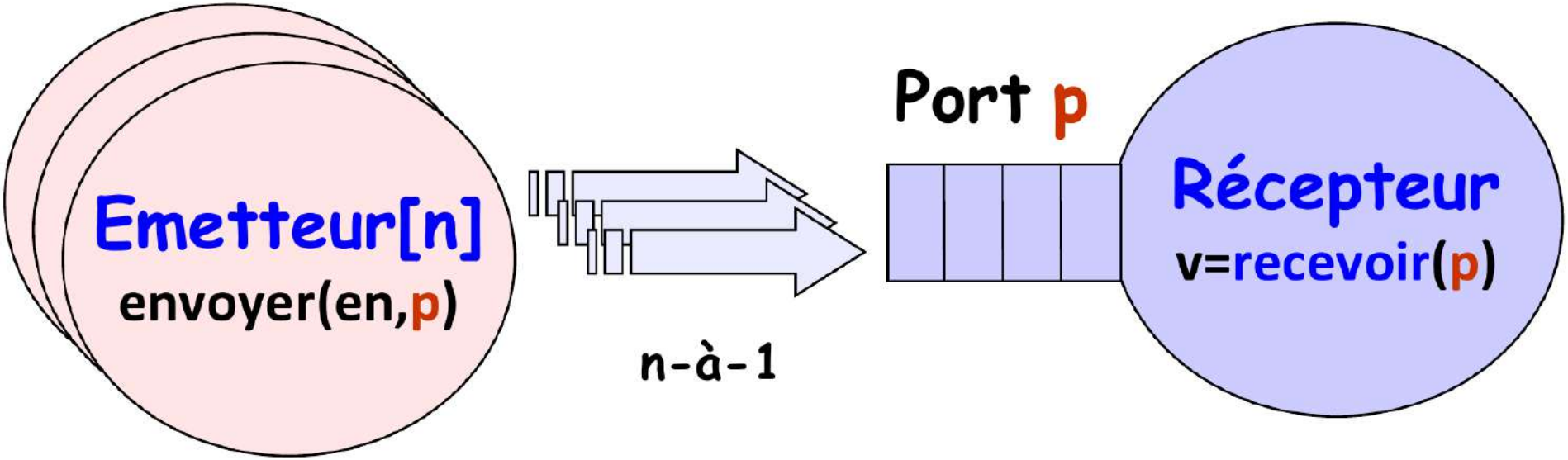
- Solution :

  - => **Communication asynchrone**.

Communication asynchrone

- C'est très contraignant : pour parler avec quelqu'un au téléphone, par exemple, il faut qu'il soit disponible pour écouter.
- Idée => utiliser les répondeurs (ou SMS)
  - Celui qui appelle peut déposer son message et continue à faire tout ce qu'il a à faire et qui ne dépend pas de la réponse.
  - Celui qui est appelé n'est pas obligé de définir son comportement en fonction de celui qui appelle. Il consulte son répondeur quand il a besoin de l'information.
- Propriété recherchée :
  - Action d'envoi n'est pas bloquante (peut l'être si le répondeur est plein)
  - L'action de réception est bloquante ssi il n'y a pas de message en attente.
  - Communication n-à-1
- D'où l'utilisation de la notion de **port**
  - Adresse d'écoute ;
  - Doté d'une mémoire tampon.

Communication asynchrone

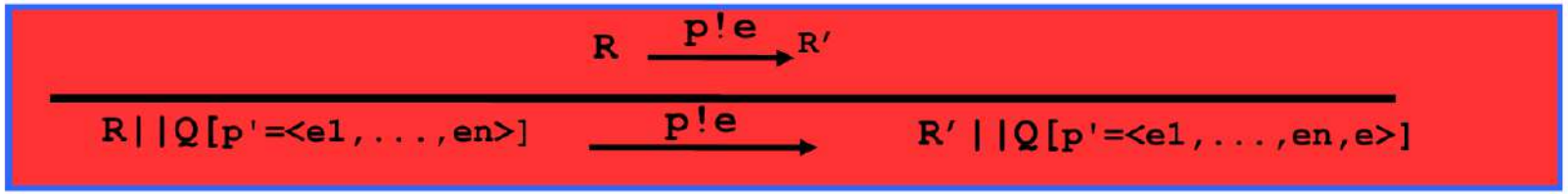


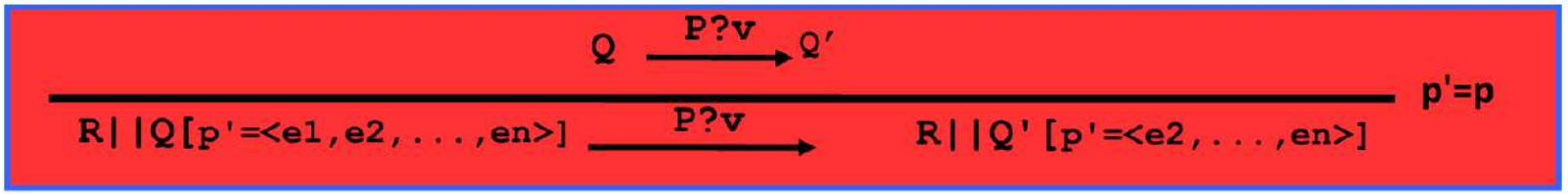
envoyer(e,p) - envoie la valeur de l'expression e au port p. le processus ne se bloque pas. Le message est stocké dans le tampon du port si celui-ci est non plein et que le récepteur n'est pas en attente.

v = recevoir(p) - recoit une valeur du port p et la stocke dans une variable v . le processus se bloque si le tampon est vide.

Enrichissons notre langage et notre sémantique

- Un processus peut donc avoir un ensemble de ports
- modélisé par une suite d'expressions p=<e1,...., en>.
- On garde la même syntaxe mais on change les canaux par des ports.
- Q [p_1=<....>,...p_n]
- Sémantique:
  - *(p!e;Q)—$^{p!e}$-->Q*
  - *(p?v;Q)[p=<e1,....,en>]—$^{v=e1}$-->Q[p=<....,en>] si p≠∅*
- Sémantique de la concurrence:





Émulation java- communication asynchrone

- Plusieurs producteurs - un consommateur
- Une zone tampon avec taille illimitée (ou limitée, bornée)
- Les émetteurs
  - producteurs
- Le port
  - la zone tampon (taille n, bornée)
- Le Récepteur
  - consommateur + port.

- Modifiez le code de canal pour qu'il devienne un port

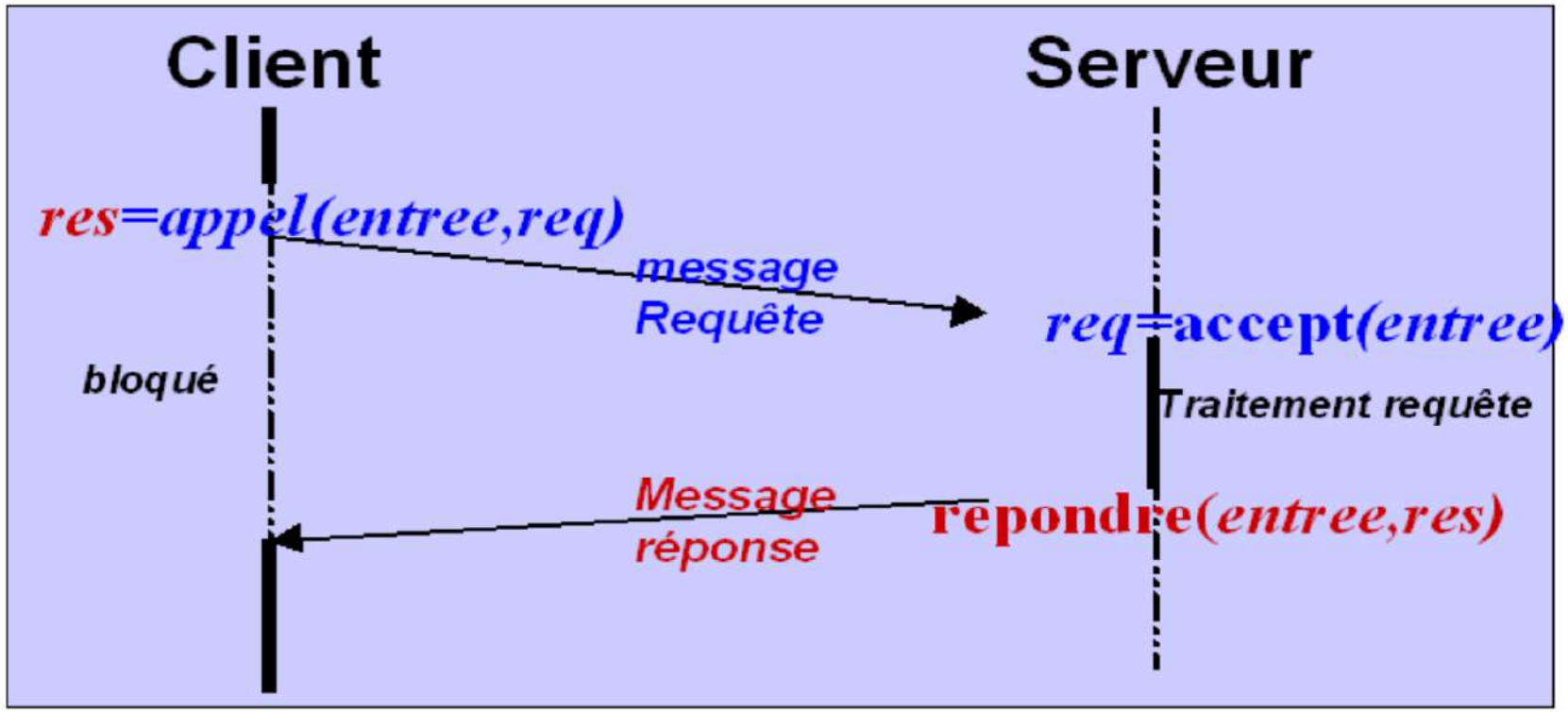
Émulation java- le port

```
public class Port<E> {
   private List<E> messages = new ArrayList<E>();
   public synchronized void envoyer(E v){
       messages.add(v);
       notify(); // un seul receveur
   }
   public synchronized E recevoir()
       throws InterruptedException {
          if(messages.size()==0) wait(); // while()
          E tmp = messages.get(0);
          messages.remove(0);
          return(tmp);
   }
}
```

Émulation java- communication asynchrone

Communication par rendez-vous

- La forme utilisée pour réaliser les systèmes *Requête-Réponse* pour supporter la communication *client-serveur*
- C'est une forme qui mélange les deux premières formes
  - Les clients envoient les demandes de *RDV* pour le traitement de requêtes.
  - Le serveur traite de manière asynchrone les requêtes. Une à la fois.
  - Les réponses sont envoyées au client sur un canal qui lui est spécifique



Communication par rendez-vous – point d'entrée

res=appel(e,req) - envoie la valeur ***req*** comme message de requête stocké dans le point d'entrée *e*.

Le processus est bloqué jusqu'à l'arrivée de la réponse res.

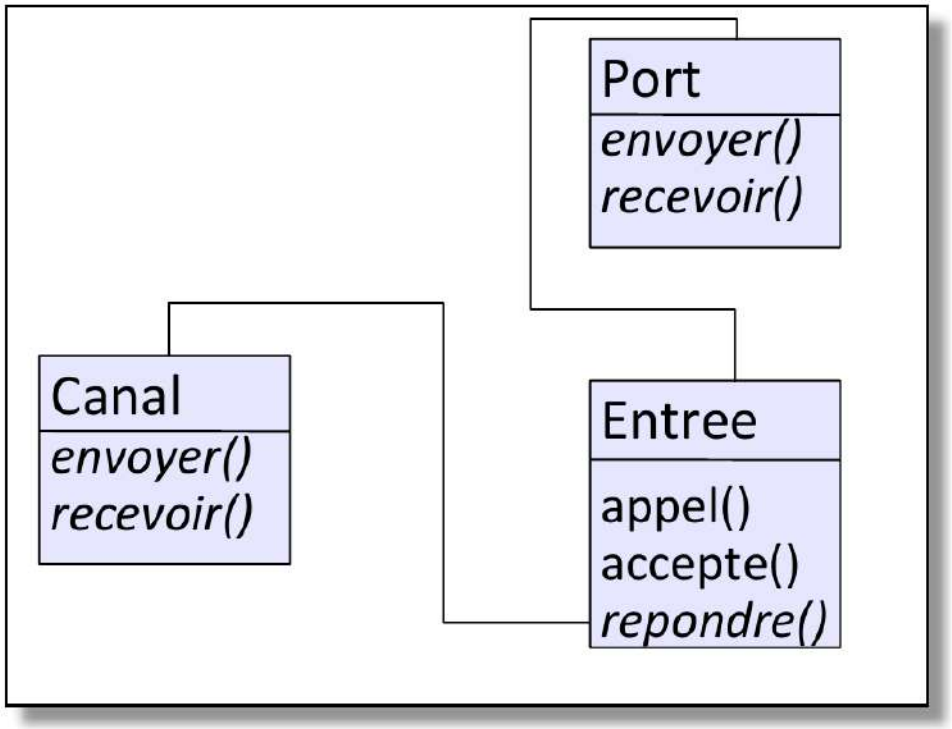
req=accept(e) - Reçoit la valeur de la requête envoyée sur l'entrée ***e*** dans la variable req. L'appel est bloquant quand l'entrée est vide.

répondre***(e,res)*** - envoit de ***res*** comme réponse à l'entrée ***e***.

Pour comprendre ce qu'est qu'un point d'entrée

Les entrées sont des ports, ils stockent les requêtes client pour un traitement asynchrone en plus d'autres fonctionnalités...

La méthode **appel** crée un canal et met le client en attente de la réponse sur ce canal. Le client à l'appel de **appel** se bloque tant qu'il n'a pas reçu ni la réponse ni un refus. La méthode **appel** envoit un message au port du serveur composé d'une référence au canal du client et la requête.



La méthode **accepte** récupère un message du tampon et récupère la requête ainsi que le canal du client. La méthode **repondre** envoie la réponse sur le canal du client.

Émulation java – communication par RDV

```
public class Entree<M, R>{
    private Port<CallMsg> port=new Port<>();  private CallMsg cm;

    public synchronized R appel(M req) throws InterruptedException {
        Canal<R> clientCan = new Canal<>();
        port.envoyer(new CallMsg(req, clientCan));
        return clientCan.recevoir();
    }

    public synchronized M accepte() throws InterruptedException {
        cm = port.recevoir();
        return cm.req;
    }

 public synchronized void reponse(R res) throws InterruptedException {
        cm.repCan.envoyer(res);
    }

    private class CallMsg{
        M req; Canal<R> repCan;
        CallMsg(M m, Canal<R> c){
            req = m;  repCan = c;
        }
    }
}
```

Émulation java – communication par RDV

```
public class Entree<M, R> extends Port<Object>{
    private CallMsg cm;

    public synchronized R appel(M req) throws InterruptedException {
        Canal<R> clientCan = new Canal<>();
        envoyer(new CallMsg(req, clientCan));
        return clientCan.recevoir();
    }

    public synchronized M accepte() throws InterruptedException {
        cm = (CallMsg) recevoir();
        return cm.req;
    }

 public synchronized void reponse(R res) throws InterruptedException {
        cm.repCan.envoyer(res);
    }

    private class CallMsg{
        M req; Canal<R> repCan;
        CallMsg(M m, Canal<R> c) {
            req = m;  repCan = c;
        }
    }
}
```

Émulation java – communication par RDV

```
public class Entree extends Port{
  private CallMsg cm;
  public synchronized Object appel(Object req) throws
InterruptedException {
    Canal clientCan = new Canal();
    envoyer(new CallMsg(req,clientCan));
    return clientCan.recevoir();
  }
  public synchronized Object accepte()throws InterruptedException {
    cm = (CallMsg) recevoir();
    return cm.req;
  }
  public synchronized void reponse(Object res) throws
InterruptedException {
    cm.repCan.envoyer(res);
  }
  private class CallMsg {
    Object  req; Canal repCan;
    CallMsg(Object m, Canal c)
      {req=m; repCan=c;}
  }
}
```

Émulation java – communication par RDV