Schooner: Une encapsulation orientée objet de supports d'exécution pour applications réparties

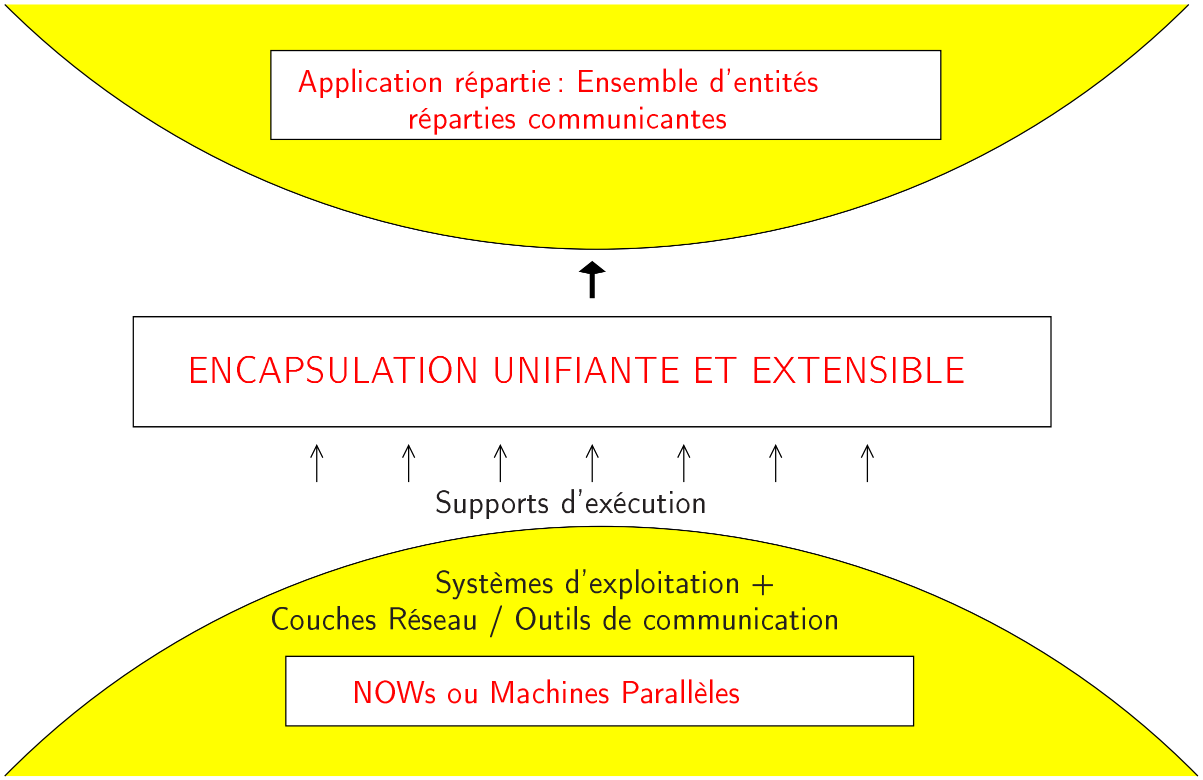
Nathalie Furmento

Thèse de Doctorat en Sciences
École Doctorale SPI (Discipline : Informatique)
Université de Nice – Sophia Antipolis

Email: Nathalie.Furmento@sophia.inria.fr

Lundi 10 Mai 1999

Motivations principales



Motivations principales *(suite)*

➢ Distinction claire entre le code lié au support d'exécution et le code propre à l'application.

➢ Portabilité vis-à-vis de la plate-forme d'exécution.

➨ Conception d'un modèle minimal, complet et extensible permettant de supporter une grande variété d'applications réparties.

PLAN

- Les applications réparties
- Schooner
- Utilisation de Schooner : Prosit
- Utilisation de Schooner : C++//
- Conclusion et perspectives

CONTENU

☞ • Les applications réparties
  – De quoi a-t-on besoin ?
  – Les supports d'exécution élémentaires
  – Les solutions comparables à Schooner
• Schooner
• Utilisation de Schooner : Prosit
• Utilisation de Schooner : C++//
• Conclusion et perspectives

De quoi a-t-on besoin ?

- ➢ Organisation d'une application répartie en un graphe dynamique de tâches de granularité différente.
- ➢ Interaction entre ces tâches via de l'échange de données.
- ➢ Gestion d'activités concurrentes avec un modèle d'ordonnancement redéfinissable.
- ➢ Portabilité de l'application via un support minimal.

Les supports d'exécution élémentaires

	Echange de messages	RPC	Processus légers
Pvm	O	N	N
MPICH	O	N	N
Pt-pvm	O	O	O
TPVM	O	O	O
PM^2	N	O	O
Nexus	N	O	O

- Solutions difficilement extensibles.
- Difficile de clairement séparer le code propre à l'application de celui gérant le support d'exécution.

Les solutions comparables à Schooner

➢ Corba, Panda, . . . , Converse, . . .

➢ Mais :

❏ Paradigmes proposés non compatibles ou directement utilisables par nos 2 applications finales (Prosit et C++//).

❏ Systèmes non extensibles (difficile voire impossible de modifier le comportement des entités fournies).

CONTENU

✓ • Les applications réparties

☞ • Schooner
  - Principaux objectifs
  - Schooner = Ensemble de briques
  - Les computers et les clusters
  - Les objets communicants
  - Communication : Messages actifs
  - La multi-activité
  - Synthèse
  - Implémentations de Schooner
  - L'environnement Schooner
- Utilisation de Schooner : Prosit
- Utilisation de Schooner : C++//
- Conclusion et perspectives

Principaux objectifs

➢ Fournir une interface portable vers les concepts de base nécessaires au support d'applications réparties.

➢ Vision structurée et abstraite du parc de machines et des entités de l'application.

➢ Utilisation du paradigme objet.

Schooner = Ensemble de briques

Objets Communicants

Computers & Clusters

Interfaces de la Librairie de Communication
[et de la Librairie de Processus Légers]

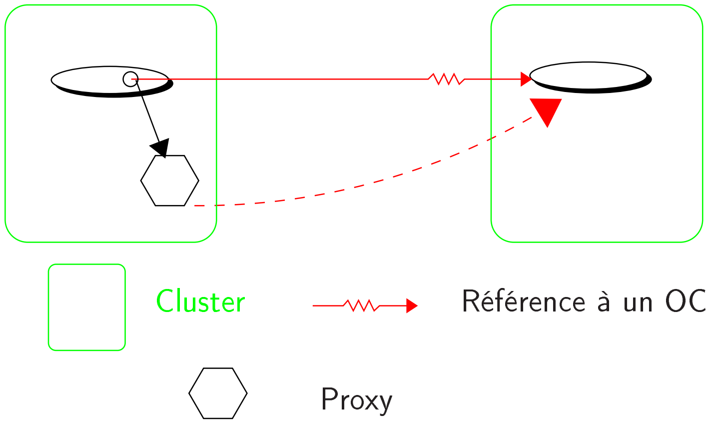
Les computers et les clusters

- Permettent de représenter et manipuler la machine virtuelle.
- Computer = Une des machines de cette machine virtuelle. Manipulé par un nom symbolique.
- Cluster = Support de calcul créé sur un computer.

Les objets communicants

- Entité de calcul de granularité plus fine qu'un cluster.

- Utilisation de l'approche *proxy* : représentant local d'un objet communicant sur n'importe quel cluster de l'application.



Communication : Messages actifs

- Opération à sens unique $\neq$ Modèle classique “send-receive”

- Nouveau type de message actif défini par une nouvelle classe dérivant soit de la classe `Message` (pour les clusters) soit de la classe `Data_Comm_Object` (pour les objets communicants).

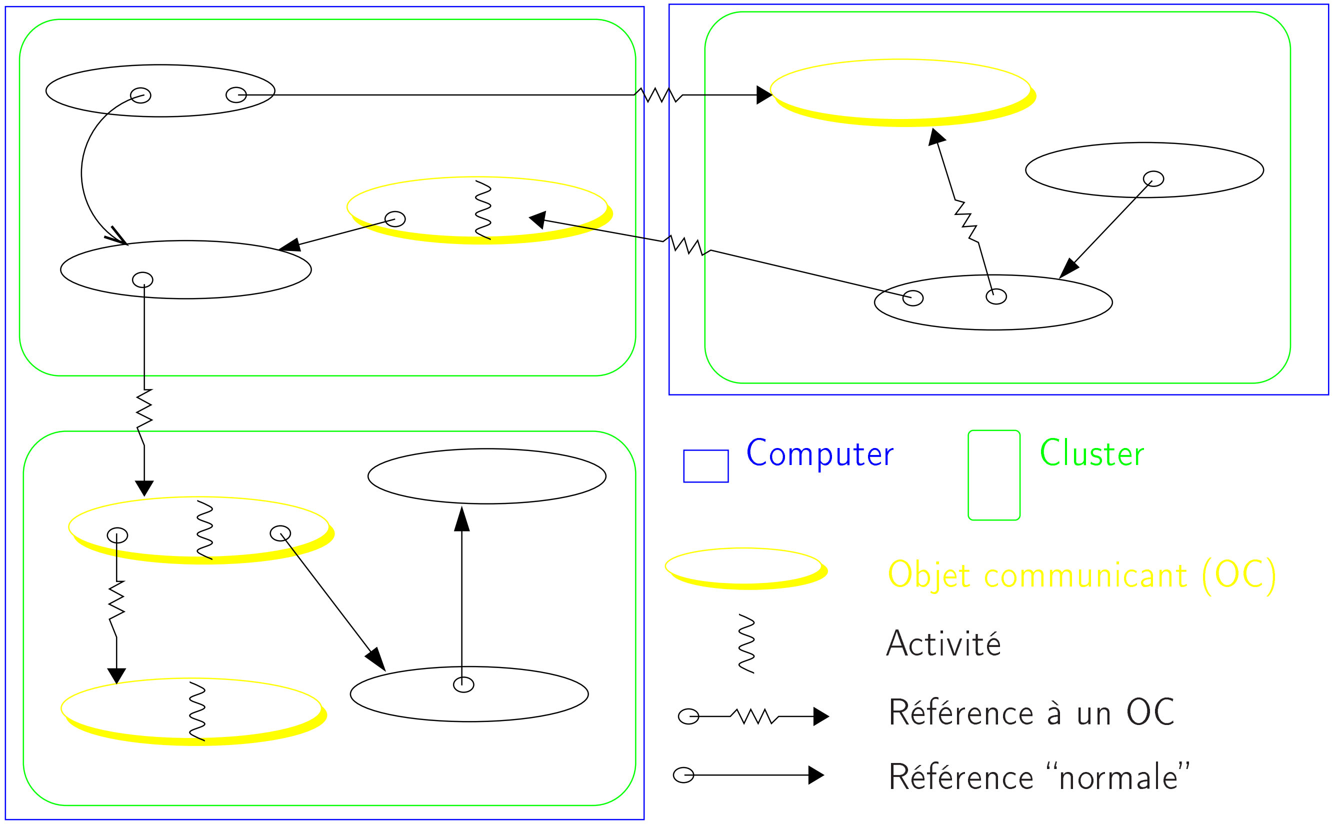
- Méthode de traitement associée à chaque type de message et automatiquement appelée lors de la réception d’un message de ce type.

- Des fonctions de linéarisation et de reconstruction sont également associées à toute classe de messages actifs.

La multi-activité

- *Pourquoi ?* Offrir un grain de parallélisme plus fin par l'introduction d'entités d'exécution autonome ⇒ objet communicant actif.
- *Comment ?*
  - Création automatique d'un processus léger à la création d'un objet communicant actif.
  - Exécution d'une méthode spécifique de la classe de cet objet.

Synthèse


Computer
Cluster
Objet communicant (OC)
Activité
Référence à un OC
Référence "normale"

Implémentations de Schooner

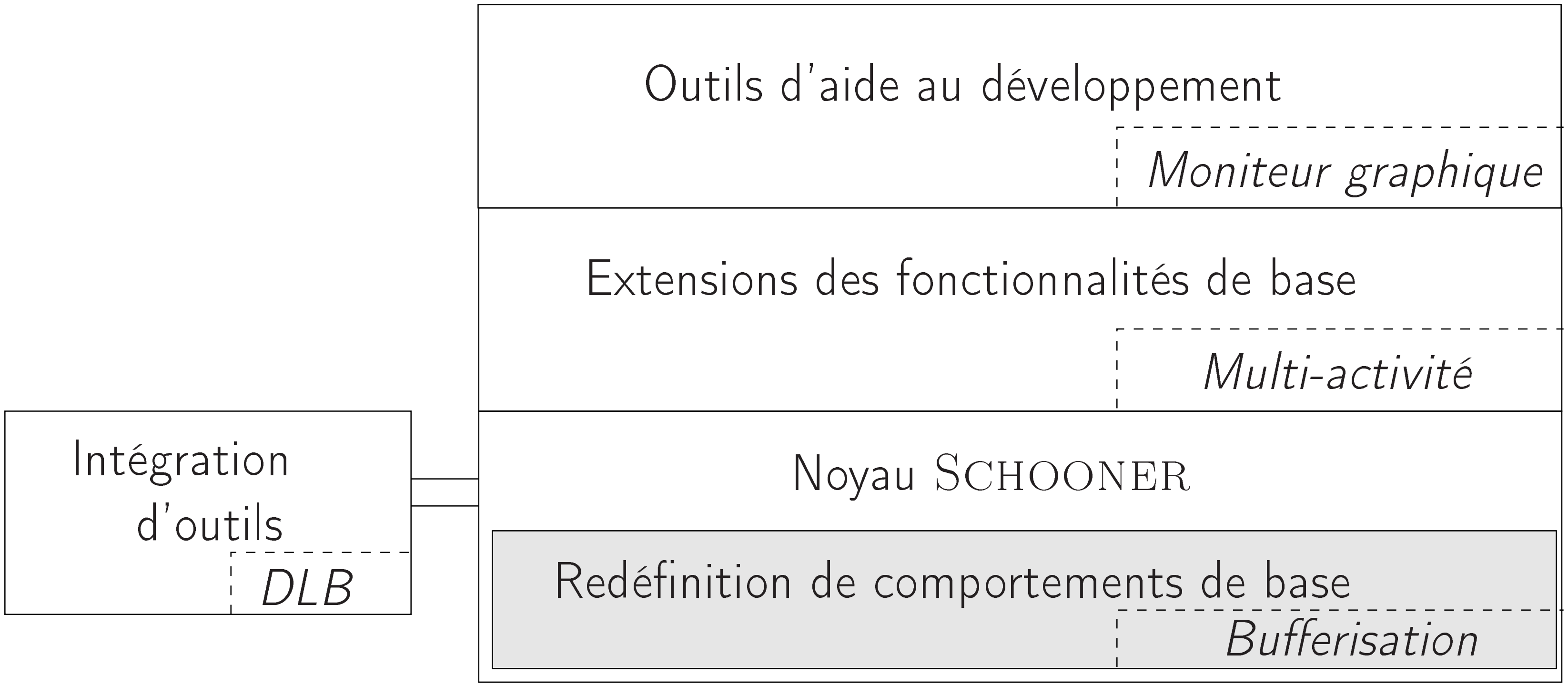
➢ Modèle de base : Pvm.

➢ Extension multi-active :

❏ PM^2 + Marcel.

❏ Nexus.

L'environnement Schooner

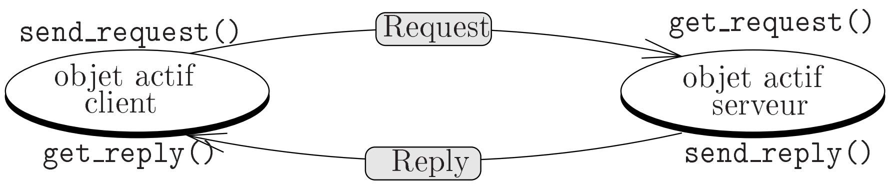


CONTENU

✓ • Les applications réparties
✓ • Schooner
☞ • Utilisation de Schooner : Prosit
  – Qu'est-ce-que Prosit ?
  – Besoins pour une version répartie
  – Prosit réparti au dessus de Schooner
  – Conclusion
• Utilisation de Schooner : C++//
• Conclusion et perspectives

Qu'est-ce-que Prosit ?

- Environnement orienté objet pour la simulation à évènements discrets.
- Simulation Prosit = Ensemble d'objets actifs interagissant par le biais de demande de services.
- Mécanisme de réification des appels de méthode (requête et réponse) :



Besoins pour une version répartie

- Vision structurée et abstraite du parc de machines.
- Mécanisme de placement des objets actifs et primitives de localisation.
- Migration des objets actifs.
- Synchronisation temporelle des objets actifs.

Prosit réparti au dessus de Schooner

- Modèle réparti de Prosit identique à celui proposé par Schooner.
- Simple utilisation ou spécialisation des classes Schooner servant à la modélisation des objets communicants.
- Utilisation de la version de Schooner sans processus légers : Prosit séquentiel utilise déjà une bibliothèque de processus légers pour la gestion de ses activités concurrentes.

Conclusion

- Mise en évidence de l'intérêt de bien isoler les mécanismes de support d'exécution de l'application elle-même :
  - Réutilisation maximum du code séquentiel.
  - Ajout minimal de code pour le passage à la version répartie.
- A permis de tester les outils d'équilibrage de charge, le moniteur graphique, la détection de propriétés stables et la bufferisation de messages.

CONTENU

✓ • Les applications réparties

✓ • Schooner

✓ • Utilisation de Schooner : Prosit

☞ • Utilisation de Schooner : C++//

  - Le modèle de C++//
  - Les protocoles de communication
  - Détails sur le protocole réactif
  - Recouvrement calcul / communications
  - Conclusion sur la technique

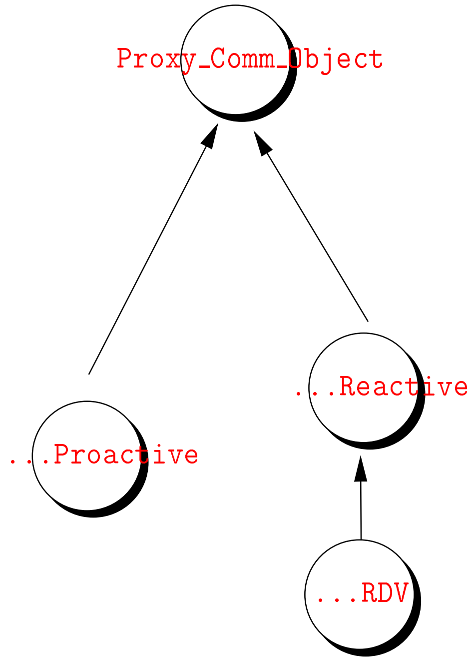
• Conclusion et perspectives

Le modèle de C++//

- Modèle hétérogène composé d'objets actifs et passifs.
- Polymorphisme entre objets passifs et actifs.
- Communication asynchrone vers les objets actifs (réification appel).
- Mécanisme d'attente par nécessité (futurs automatiques et transparents).
- Contrôle centralisé et explicite.

Les protocoles de communication

- *Protocole proactif* : communication non-bloquante pour l'émetteur et le récepteur n'est pas interrompu. Nécessité de s'enquérir de la présence de nouvelles requêtes.
- *Protocole réactif* : communication non-bloquante pour l'émetteur, le récepteur est interrompu dès réception d'une nouvelle requête.
- *Protocole réactif par rendez-vous* : phase de rendez-vous établi avec le récepteur + protocole réactif.



Détails sur le protocole réactif

- Nécessité de pouvoir interrompre l'activité principale de l'objet actif pour lui faire traîter la nouvelle requête.
- La librairie Marcel (PM^2) fournit une primitive pour dévier l'exécution d'un processus léger vers une autre fonction.
- Si ce mécanisme est indisponible $\Rightarrow$ Protocole transparent : dépot de la nouvelle requête à l'insu de l'objet actif sans interrompre son activité normale.

Recouvrement calcul / communications

- *Objectif principal* : Améliorer les performances d'exécution d'applications orientées objet basées sur des services distants en :

  RECOUVRANT

  la transmission de la requête associée au service distant

  par

  l'exécution distante de ce service

- Mécanisme transparent pour l'utilisateur.

Recouvrement calcul / communications *(suite)*

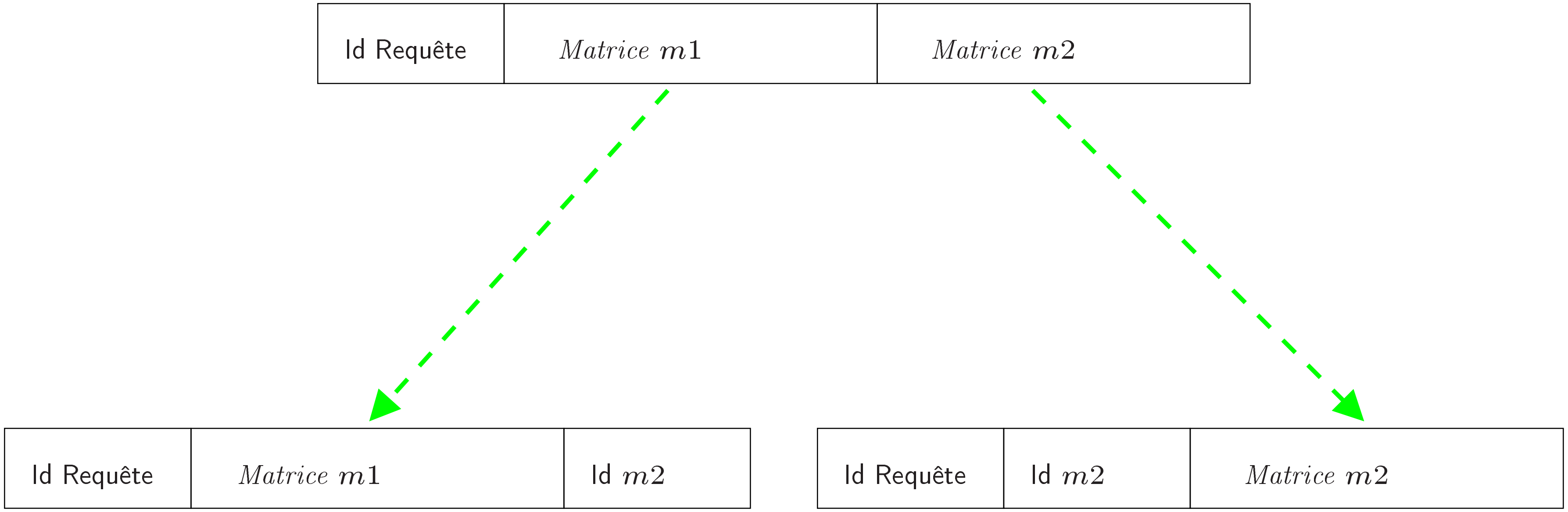
- *Idée directrice* : Être capable de démarrer le service distant avant la réception complète de l'ensemble des paramètres.

- *Comment ?* En décomposant les phases de communication et de calcul sur des plus petits volumes de données.

  $\Longrightarrow$ Modification de la transmission et de la réception d'une requête.

Recouvrement calcul / communications *(suite)*

$$objet \rightarrow service(Matrice\ *m1,\ Matrice\ *m2)$$



Recouvrement calcul / communications *(suite)*

➢ Création et transmission de la requête :

- ❑ Dissocier les phases de découpage et de mise à plat de celles d'envoi.

➢ Reconstruction de la requête :

- ❑ Identifier les données manquantes,
- ❑ Les mettre à jour de manière transparente dès leur réception.

➢ Exécution du service :

- ❑ Etre bloqué en cas de données manquantes.

Recouvrement calcul / communications *(suite)*

Comment découper une requête ?

- ➢ Solution de base : découpage fait au niveau du support d'exécution.
- ➢ Solution plus adaptée : découpage guidé par le programmeur.
  - ❑ Il peut facilement indiquer quels objets doivent être envoyés ultérieurement (héritage).
  - ❑ Les fonctions de mise à plat et de reconstruction génériques sont modifiées pour prendre en compte ces indications.

Recouvrement calcul / communications *(suite)*
Implémentation du mécanisme

➢ C++// :

❑ Objets à envoyer ultérieurement sont des objets de type `Future`.

➢ Schooner :

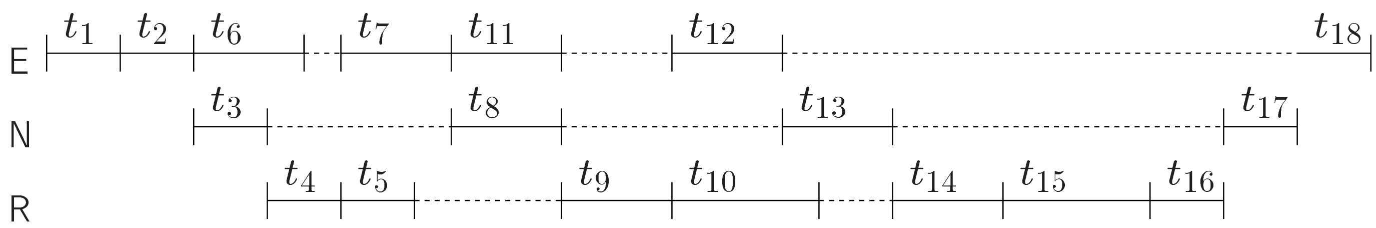
❑ Redéfinition de la méthode d'envoi de données vers un objet communicant afin d'envoyer une requête en plusieurs messages.

❑ Chaque sous-requête est un message actif ⇒ reçue et traitée de manière transparente.

Recouvrement calcul / communications *(suite)*

```
class OpMatrix : public Process {
    virtual int service(Matrice *m1, Matrice *m2) {
        m1->bar();
        m2->plus(m1);
        int res = m2->result();
        return (res);
    }
};
OpMatrix *objet = CppLL_new(("host"),OpMatrix,());
Matrice *m1 = new Matrice(COLUMN, LINE);
Matrice *m2 = CppLL_new (Matrice_Future, (COLUMN, LINE));
// initialiser les valeurs de m1 et m2
Démarrer_Horloge;
int res = objet->service(m1, m2);
Stopper_Horloge;
```

Recouvrement calcul / communications *(suite)*

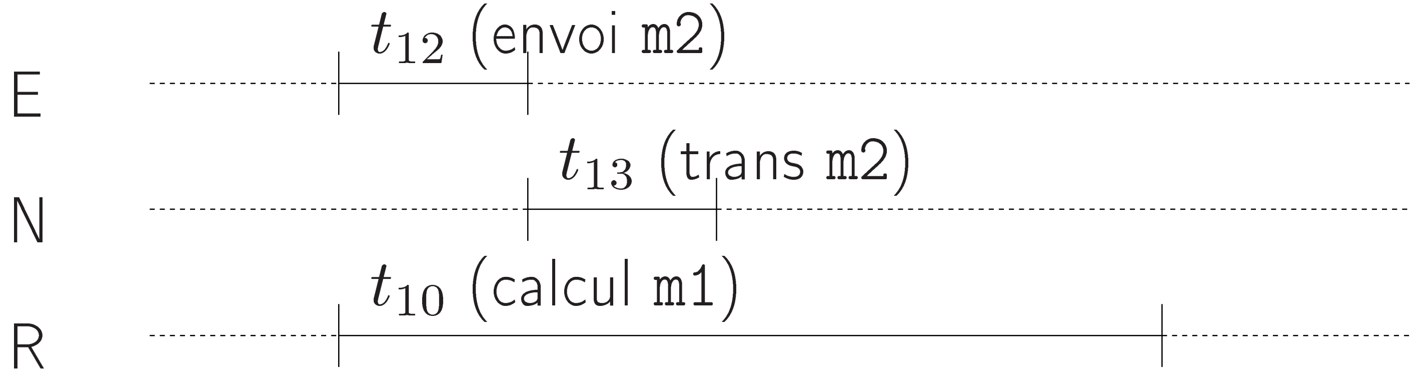


E Emetteur, N Réseau, R Récepteur

t_1	**Préparation envoi** `mid(service)`	t_{10}	**Calcul** `m1`
t_2	**Envoi réseau** `mid(service)`	t_{11}	**Préparation envoi** `m2`
t_3	**Transmission** `mid(service)`	t_{12}	**Envoi réseau** `m2`
t_4	**Réception** `mid(service)`	t_{13}	**Transmission** `m2`
t_5	**Calcul** `mid(service)`	t_{14}	**Réception** `m2`
t_6	**Préparation envoi** `m1`	t_{15}	**Calcul** `m2`
t_7	**Envoi réseau** `m1`	t_{16}	**Préparation envoi** `res`
t_8	**Transmission** `m1`	t_{17}	**Transmission** `res`
t_9	**Réception** `m1`	t_{18}	**Réception** `res`

Recouvrement calcul / communications *(suite)*

Cas d'un L.A.N

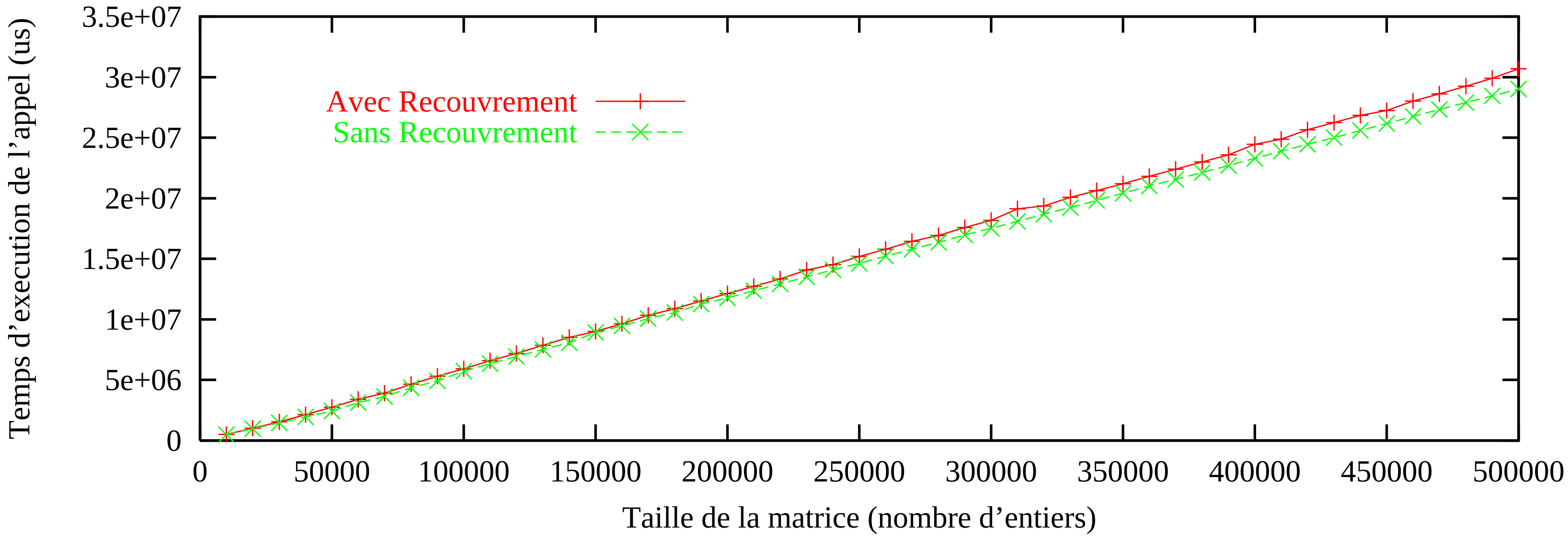


Phase de transmission (t_{13}) courte car débit élevé et latence faible :

- ➡ Recouvrement minime.
- ➡ Surcoût peut être plus élevé que le gain.

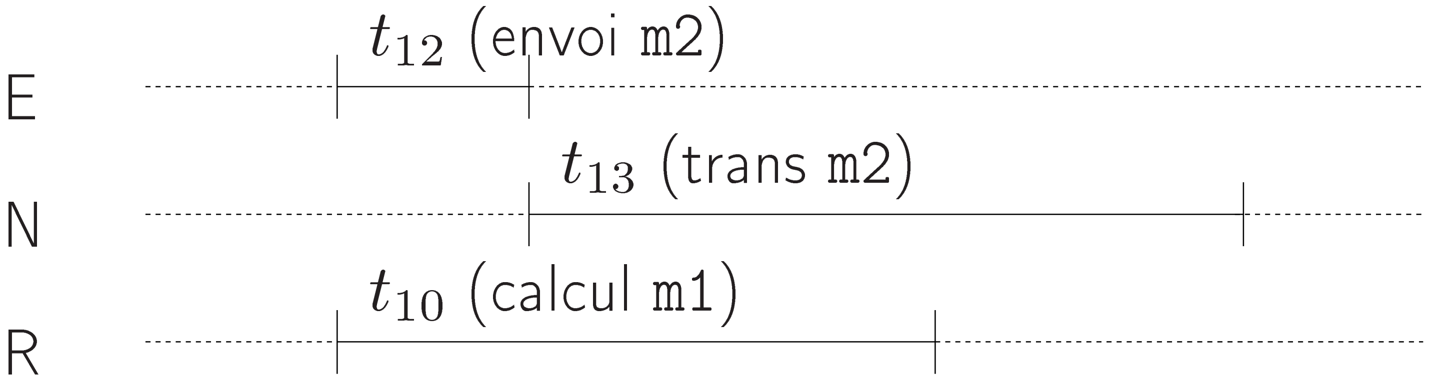
Recouvrement calcul / communications *(suite)*

Cas d'un L.A.N *(suite)*



Recouvrement calcul / communications *(suite)*

Cas d'un W.A.N

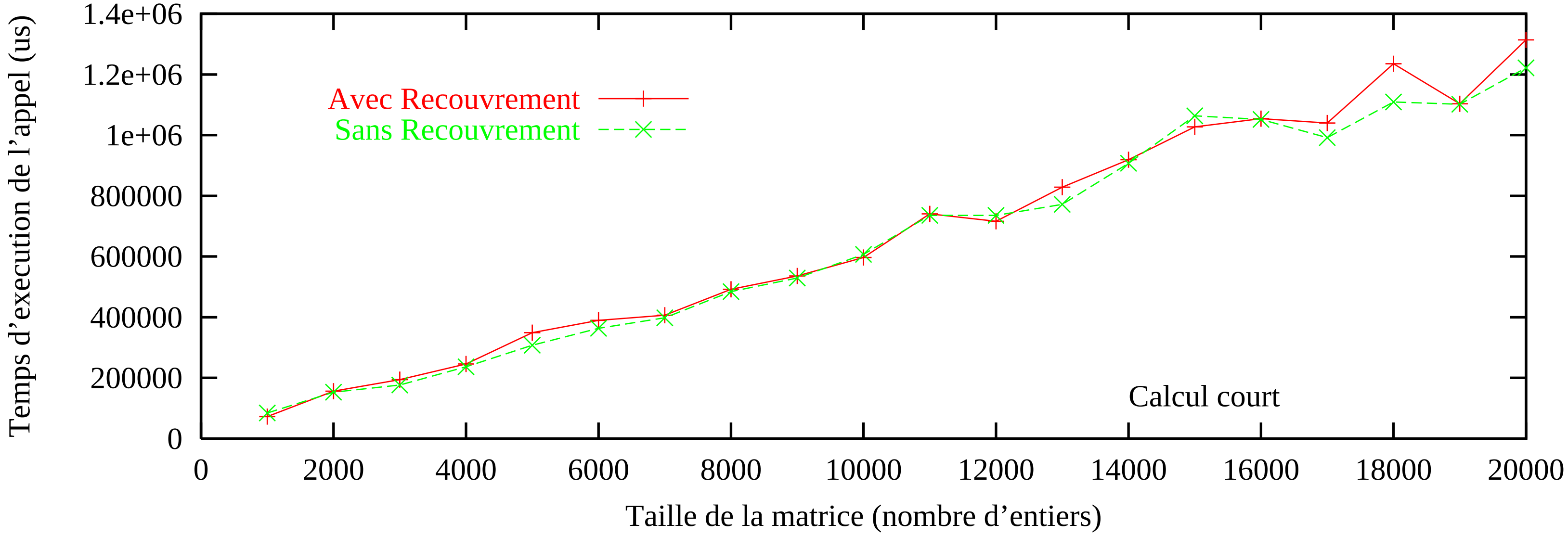


Recouvrement borné par t_{10} :

➡ Plus la phase de calcul (t_{10}) est grande, plus le recouvrement peut être important.

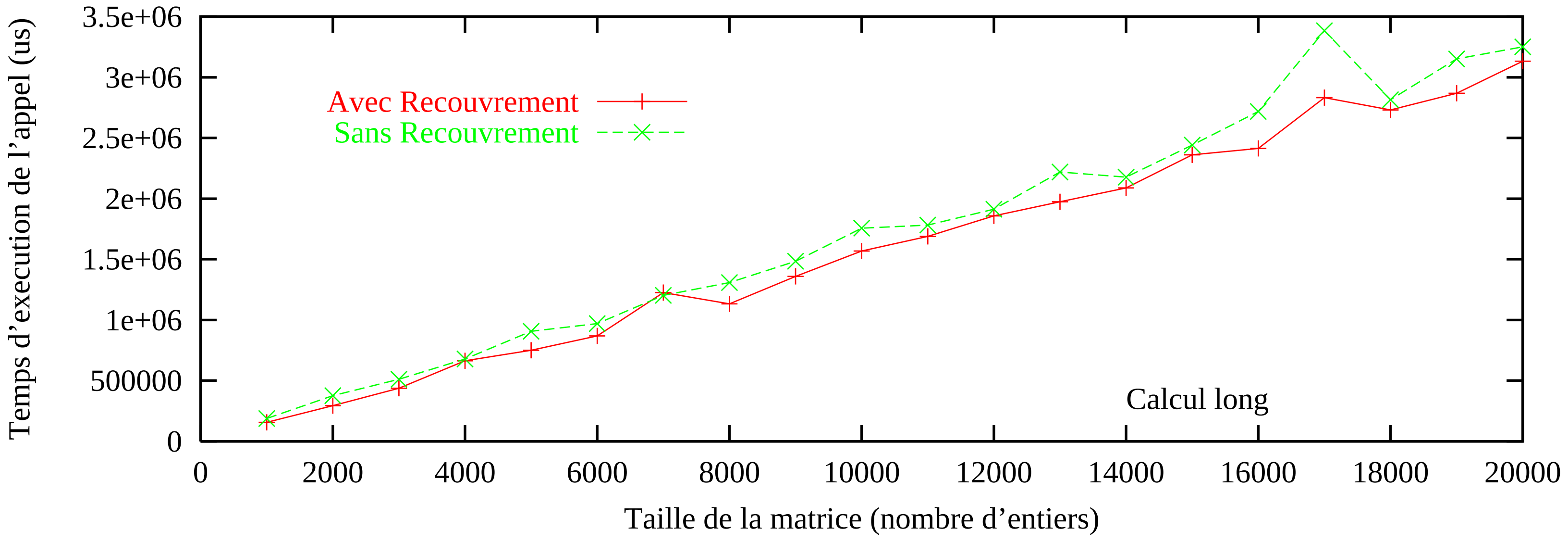
Recouvrement calcul / communications *(suite)*

Cas d'un W.A.N *(suite)*



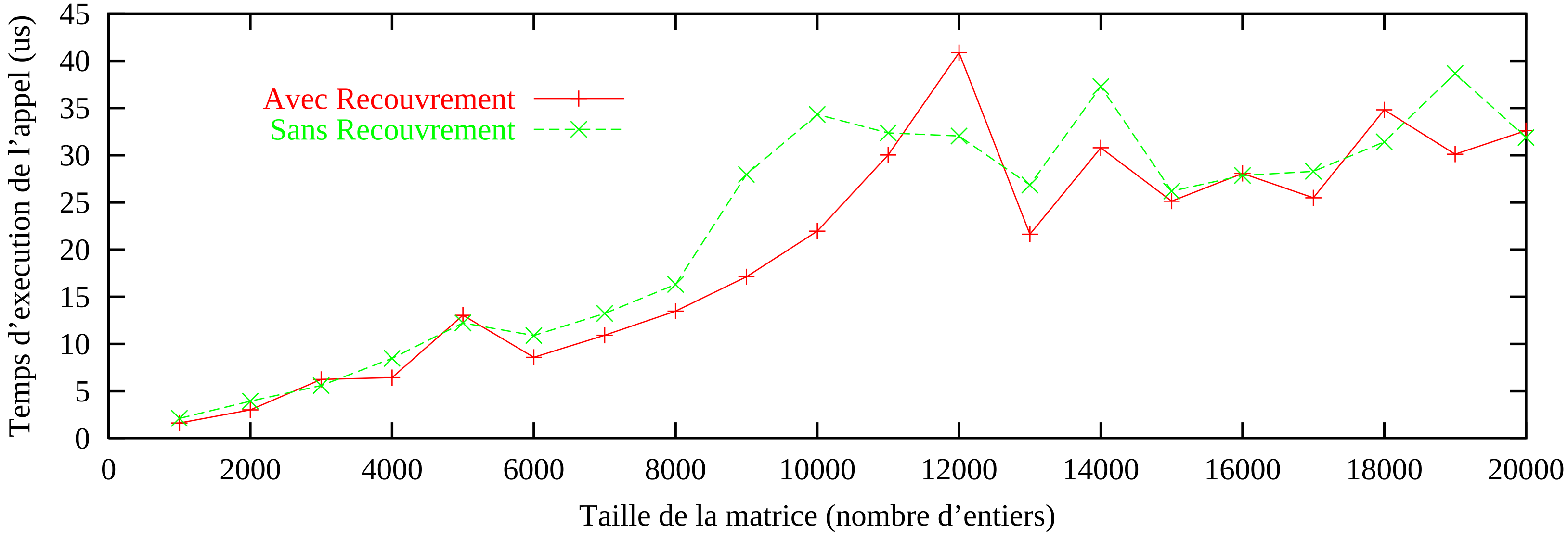
Recouvrement calcul / communications *(suite)*

Cas d'un W.A.N *(suite)*



Recouvrement calcul / communications *(suite)*

Variations de charge



Recouvrement calcul / communications *(suite)*

Mode de communication utilisé

➢ Communication via le démon Pvm :

❏ Comportement attendu : le calcul sur $m1$ est perturbé par la réception des paquets sur le démon.

➢ Communication directe entre clusters via un socket TCP :

❏ L'émission va être stoppée dès que le socket de réception est plein (taille par défaut de 16K).

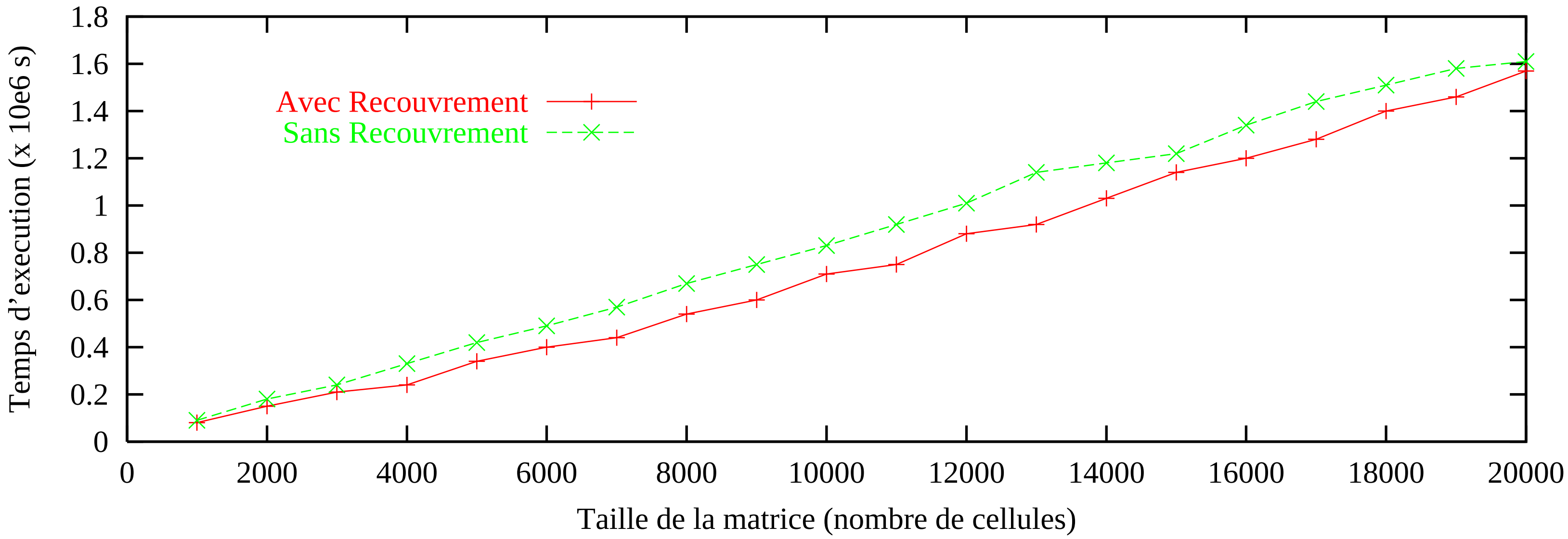
Recouvrement calcul / communications *(suite)*

- Ce socket n'est pas vidé tant que la fonction de réception n'est pas appelée.
  $\Longrightarrow$ Le recouvrement se limite à la transmission de données de taille égale à la taille du socket.

- L'utilisation de processus légers permettrait d'effectuer la réception pendant le calcul.

Recouvrement calcul / communications *(suite)*

Gain d'environ 15 % sur une exécution de service



Conclusion sur la technique

- Mécanisme presque transparent.
- Selon la configuration cible et le comportement prévu, on peut facilement choisir d'utiliser ou non le mécanisme.
- Poursuite des expériences sur architecture multi-threadée.

CONTENU

- ✓ • Les applications réparties
- ✓ • Schooner
- ✓ • Utilisation de Schooner : Prosit
- ✓ • Utilisation de Schooner : C++//
- ☞ • Conclusion et perspectives
  - Réalisations effectuées
  - Perspectives

Réalisations effectuées

- Conception d'un modèle unificateur constituant un support d'exécution pour la programmation d'applications réparties.
- Extensions de ce modèle :
  - Multi-activité [Barette, Baude, Caromel, Geib, Namyst, Méhaut, Tonin]
  - Bufferisation
  - Equilibrage de charge [Dalle, Siegel, Tanzy]

Réalisations effectuées *(suite)*

- Outils d'aide à la programmation :
  - Mécanisme de traces & Visualisation post-mortem [Clavaud]
  - Moniteur graphique [Siegel]
- Applications conçues au dessus de Schooner :
  - Prosit [Siegel]
  - C++// & Mécanisme de recouvrement calcul / communications [Baude, Caromel, Roudier, Sagnol]

Perspectives

- Mise au point de schémas de communication multipoints au niveau des objets communicants.
- Poursuite des expériences pour le mécanisme de recouvrement calcul / communications.
- Version répartie d'un simulateur de trafic routier dans le cadre du projet européen Hipertrans.