Université Joseph Fourier

U.F.R Informatique &
Mathématiques Appliquées

Institut National
Polytechnique de Grenoble

ENSIMAG

I . M . A . G .

ECOLE DOCTORALE
MATHEMATIQUES ET INFORMATIQUE

DEA D'INFORMATIQUE :
SYSTEMES ET COMMUNICATIONS

Projet présenté par :

Philippe Laumay

Déploiement d'un bus à messages sur un réseau à grande échelle

Effectué au laboratoire : SIRAC

Date : 20 juin 2000

Jury : Luc BELLISSARD,
Christian BOITET,
Yves DENNEULIN,
André FREYSSINET,
Sacha KRAKOWIAK.

Résumé

Ce projet de DEA se situe dans le cadre du laboratoire Sirac (Systèmes Informatiques Répartis pour Applications Coopératives), dont l'objectif est de concevoir et réaliser des services pour l'aide au développement, à l'exécution et à l'administration d'applications réparties.

On s'intéresse plus particulièrement aux applications réparties construites à base d'agents qui communiquent sur des réseaux mondiaux (Internet ou Intranet) à travers un bus à messages (MOM ou *Message-Oriented Middleware*). Dans le cadre d'une coopération avec le GIE Dyade, a été développé un bus de messages qui offre des propriétés particulières de fiabilité, de reprise après pannes, d'ordonnancement causal sur une architecture répartie. Ces propriétés permettent ainsi aux développeurs d'applications de s'affranchir des problèmes inhérents aux communications à distance.

Toutefois, ces propriétés posent un réel problème de passage à l'échelle car l'efficacité des algorithmes les mettant en oeuvre diminue avec l'augmentation du nombre de sites utilisés. La propriété d'ordonnancement causal est celle qui apporte le plus de difficulté, car l'algorithmique utilisée croît non linéairement avec l'augmentation des sites.

Dans ce rapport, nous proposons une solution qui consiste à utiliser une architecture de bus en cellules ou *domaines de causalité*. Mais interviennent alors des questions liées à la préservation des propriétés relatives à la transmission des messages sans dégrader les performances.

Nous avons expérimenté une solution à base de domaine de causalité par topologie réseau sur le bus à messages AAA développé au sein du laboratoire Sirac. Nos expérimentations nous montrent une nette amélioration des performances ; les coûts liés à l'ordonnancement passant d'une croissance quadratique à une croissance linéaire.

Remerciements

Je tiens à remercier les personnes qui m'ont permis par leur aide d'effectuer ce DEA dans les meilleures conditions possibles. Plus particulièrement je remercie,

Monsieur Yves Denneulin qui m'a fait l'honneur de participer à ce jury de DEA.

Messieurs Sacha Krakowiak et Christian Boitet pour leur participation au jury et pour l'attention qu'ils ont porté à ce travail.

Roland Balter pour m'avoir accueilli au sein du projet Sirac tout au long de cette année et pour les encouragements à la réalisation d'une thèse.

Luc Bellissard et André Freyssinet pour toutes les raisons précédentes, pour leurs conseils avisés sur le rapport et le DEA en général et surtout pour avoir supporté mes excès de paranoïa (ça c'est fini☺, ouf !).

David Feliot pour les taquineries de 16h qui me permettaient de me défouler (gueuler ça fait du bien !).

Nicolas Tachker pour ses discours qui rassurent sur les études, le boulot… la vie !

Remerciement général à ces quatre derniers pour leur soutien, leurs encouragements (et dieu sait qu'il m'en fallait), pour m'avoir aidé à prendre la difficile décision de continuer en thèse et surtout m'avoir fait comprendre l'expression "travail en équipe".

Eric Bruneton pour son immense aide pour la formalisation de la preuve de la causalité par transitivité.

Olivier Charra dit "Olive" pour les crises de fou rire relaxantes, les prises de têtes régulières (ça aussi c'est fini), les pauses yaourt, la découverte de la fromagerie Guillou et surtout sa motivation exemplaire pour aller courir le matin (ça j'espère que c'est pas fini !)…

Christophe Rippert dit "Bouboule" pour sa précision d'horloge atomique pour le repas de 11h30.

Mes parents, ma sœur et mon frère pour leur iiiiiiiiiiimmense soutien, leurs encouragements, les repas "pieds sous la table" lors des retours à onze heures du soir.

Et enfin à tous ceux-ci pour leur conseils, leur soutien ou tout simplement pour leur présence :
Raph, Rémi, Aline, Nono, Manu, Sara, Serge, Daniel, Fabienne, Michel, Céline, Cyril, Vania, Laurent, …

Table des matières

Résumé — ***2***

Remerciements — ***3***

Table des matières — ***4***

Table des Figures — ***6***

Introduction — ***7***

Partie 1 : Etat de l'art — ***9***

Chapitre 1 : Message-Oriented Middleware — **10**
- 1.1. Introduction — 10
  - 1.1.1. Le mode asynchrone — 10
  - 1.1.2. Le mode synchrone — 11
  - 1.1.3. Synthèse — 12
- 1.2. Modèles de communication — 13
  - 1.2.1. Message Passing — 14
  - 1.2.2. Message Queuing — 14
  - 1.2.3. Publish and Subscribe — 16
  - 1.2.4. Modèle événementiel — 18
  - 1.2.5. Comparaison du modèle événementiel et du modèle Publish/Subscribe — 19
- 1.3. Architectures des MOM — 19
  - 1.3.1. Hub & spoke — 20
  - 1.3.2. Snow flake — 21
  - 1.3.3. Bus — 22
- 1.4. Propriétés des MOM — 23
  - 1.4.1. Introduction — 23
  - 1.4.2. Qualité de service — 23
  - 1.4.3. Ordonnancement — 24
- 1.5. Synthèse — 26

Chapitre 2 : Passage à l'échelle — **27**
- 2.1. Problématique — 27
- 2.2. Architectures scalables — 27
  - 2.2.1. Hub & spoke — 27
  - 2.2.2. Snow flake — 28
  - 2.2.3. Bus — 28
- 2.3. Influence des propriétés — 29
  - 2.3.1. Routage efficace des messages — 29
  - 2.3.2. ordonnancement — 29

Partie 2 : L'environnement AAA (Agent Anytime Anywhere) — ***30***

Chapitre 3 : Le bus à messages AAA — **31**
- 3.1. Concepts du bus à agents — 31
- 3.2. Propriété de l'infrastructure — 32
  - 3.2.1. Propriété des agents — 32
  - 3.2.2. Propriétés des communications — 32
  - 3.2.3. Modèle de programmation — 33

3.2.4. Création et déploiement des agents ___ 34
3.3. L'infrastructure d'exécution AAA ___ 35
3.3.1. Une vue d'ensemble ___ 35
3.3.2. Les serveurs d'agents ___ 36
3.4. Synthèse ___ 38

Chapitre 4 : Les problèmes du passage à l'échelle du bus à messages AAA ___ 39
4.1. Introduction ___ 39
4.2. L'algorithme de causalité ___ 39
4.2.1. Protocole de base ___ 39
4.2.2. Première optimisation ___ 40
4.3. L'impact de la causalité ___ 41
4.3.1. Procédure de test ___ 41
4.3.2. Analyse des résultats ___ 42

Partie 3 : Proposition ___ 44

Chapitre 5 : Domaines de causalité ___ 45
5.1. Concepts ___ 45
5.2. Preuve de la causalité par transitivité ___ 45
5.2.1. Définitions ___ 45
5.2.2. Lemmes ___ 50
5.2.3. Théorème ___ 52
5.3. Synthèse ___ 55

Chapitre 6 : Utilisation des domaines de causalité pour le passage à l'échelle d'un bus à messages 56
6.1. Introduction ___ 56
6.2. Domaines de causalité par topologie applicative ___ 56
6.3. Domaines de causalité par topologie réseau ___ 58
6.4. Conclusion ___ 59

Chapitre 7 : Passage à l'échelle du bus à messages AAA : réalisation et évaluation 60
7.1. Implémentation des domaines de causalité ___ 60
7.1.1. Réalisation actuelle ___ 60
7.1.2. Perspectives de réalisation ___ 63
7.2. Expérimentation et résultats ___ 63

Conclusion ___ 67

Références ___ 69

ANNEXES ___ 72

Table des Figures

Figure 1: Abstraction d'un Message-Oriented Middleware 10
Figure 2: Exemple de communication asynchrone 11
Figure 3: Exemple de communication synchrone 12
Figure 4: Les quatre modèles de communication des MOM 13
Figure 5: Le Message Passing 14
Figure 6: Le Message Queuing 15
Figure 7: Le modèle Publish and Subscribe 16
Figure 8: Modèle événementiel. 19
Figure 9: Architecture Hub & spoke 20
Figure 10: Architecture Snow flake 21
Figure 11: Architecture d'un Bus logiciel 22
Figure 12: Modèle de base 31
Figure 13: Propriété de causalité 32
Figure 14: L'infrastructure AAA 36
Figure 15: Détail d'un serveur d'agents 36
Figure 16: Les queues de message du channel 37
Figure 17: Si un message est en avance (au sens causal), 40
Figure 18: Si un message est en avance (au sens causal), 40
Figure 19: Résultat de mesure d'envoi distant de notifications 42
Figure 20: Résultat de mesure de la diffusion de notifications 43
Figure 21: Représentation d'une trace 46
Figure 22: Une trace incorrecte 47
Figure 23: Violation de la causalité 47
Figure 24: Respect de la causalité dans un domaine 47
Figure 25: Abstraction invalide d'une trace 49
Figure 26: Les deux cas possibles du Lemme 4 51
Figure 27: Un contre-exemple au théorème 52
Figure 28: Trace correspondant à [p_1,...,p_c] 53
Figure 29: "Croisements" possibles de chaînes 54
Figure 30: La topologie applicative de NetWall et son horloge matricielle associée 57
Figure 31: Représentation de l'architecture de NetWall grâce à un ADL 57
Figure 32: Domaines de causalité par topologie réseau 58
Figure 33: Deux exemples de la structure des serveurs d'agents 60
Figure 34: La Class DomainItem 61
Figure 35: L'algorithme du Channel avec gestion des domaines de causalité 62
Figure 36: Exemple de la vision des domaines et de la structure associée 62
Figure 37: Résultat des mesures d'envoi local de notifications 63
Figure 38: Résultat des mesures d'envoi distant de notifications 64
Figure 39: Découpage en bus des domaines réalisé pour les tests 1 et 2 de l'Annexe 2 : 64
Figure 40: Résultat des mesures d'envoi local de notifications 65
Figure 41: Découpage en étoile des domaines réalisé pour les tests page 77 65
Figure 42: Découpage hiérarchique des domaines et résultat pour 90 serveurs répartis 66
Figure 43: Résultats comparés avec et sans gestion des domaines de causalité en étoile 66

Introduction

Il y a actuellement une nette orientation parmi les développeurs de logiciels à concevoir des systèmes distribués pour des composants autonomes fortement couplés[1]. Ces systèmes ont de plus en plus tendance à se déployer sur des réseaux à grande échelle. Les deux approches courantes pour réaliser un couplage fort sont l'invocation de méthode à distance ou la programmation événementielle. Dans les modèles événementiels les interactions entre les composants sont modélisées par des occurrences asynchrones d'événements. Afin d'informer les autres composants des occurrences d'événements internes (comme des changements d'état), les composants émettent des notifications qui comportent des informations sur ces événements. Lors de la réception de notifications, les autres composants peuvent réagir en exécutant des actions qui, en retour, peuvent provoquer d'autres occurrences d'événements et créer de nouvelles notifications. L'ordonnancement global des notifications a une grande importance car il permet de s'assurer que les composants réagiront selon l'organisation prévue par le programmeur.

Les réseaux à grande échelle (WAN) comme l'Internet, avec leur vaste potentiel de producteur et de consommateur de notifications, créent une opportunité pour le développement de nouvelles applications distribuées à base d'événements dans des domaines aussi divers que le commerce en ligne, l'analyse des marchés et de façon plus générale l'intégration d'applications. En général, l'asynchronisme, l'hétérogénéité et le haut degré de couplage qui caractérisent les applications à grande échelle conduisent à l'utilisation naturelle de systèmes distribués à base d'événements. Nous appelons de tels systèmes distribués des *middlewares* à messages (*Message-Oriented Middleware*, MOM). Toutefois, les solutions actuelles de type MOM ne répondent guère au problème du passage à l'échelle.

Un des points cruciaux de la problématique du passage à l'échelle est l'ordonnancement des notifications. L'ordonnancement est un principe essentiel car dans tout environnement distribué, le non-déterminisme survient inévitablement, à cause de l'asynchronisme et de l'hétérogénéité des environnements à grande échelle. Dans n'importe quel *Message-Oriented Middleware*, il est impossible de prédire précisément quand une information envoyée par un composant sera reçue par un autre. Cela est particulièrement vrai lorsqu'on utilise des réseaux à grande échelle (sur l'Internet par exemple). Un modèle réaliste suppose donc que l'ordre d'arrivée des informations envoyées est complètement arbitraire et entièrement inconnu. En particulier, les communications de différentes sources vers un destinataire donné (et dans certains cas, vers la source elle-même) peuvent arriver dans un ordre complètement aléatoire[2] et à des moments très différents. Lorsque cela se produit, il est nécessaire de fournir un mécanisme qui réordonne les messages à l'arrivée.

Un mécanisme d'ordonnancement couramment utilisé consiste à créer un *temps logique* afin d'ordonner les messages selon un *ordre causal.* Cet ordre causal est souvent géré par une horloge matricielle qui introduit des problèmes significatifs lors du passage à l'échelle. En effet, l'augmentation linéaire du nombre de participants accroît de façon quadratique[3] la taille de l'horloge et donc le coût de la gestion de la causalité. Une solution est d'utiliser une

[1] C'est à dire des composants applicatifs qui ont besoin l'un de l'autre.
[2] C'est à dire non corrélé avec l'ordre d'envoi.
[3] De l'ordre de n^2.

architecture de bus en cellules ou domaines de causalité mais il se pose alors le problème de préserver les propriétés relatives à la transmission des messages sans dégrader les performances.
Ce rapport propose une solution à base de domaine de causalité pour améliorer le coût de la causalité dans le bus à messages AAA développé au sein du laboratoire Sirac (INPG, INRIA, UJF) dans le cadre d'une coopération avec le GIE Dyade (Bull-INRIA). Il s'organise de la façon suivante : dans la première partie nous présentons un état de l'art sur les *Message-Oriented Middleware* et sur le passage à l'échelle. La deuxième partie présente plus précisément le contexte de travail à partir duquel les mécanismes de domaines de causalité ont été développés. Il s'agit de l'environnement d'exécution réparti AAA et de l'impact de la causalité sur ce dernier. Enfin la dernière partie apporte des propositions sur les domaines de causalité et présente l'algorithme que nous avons développé sur AAA afin d'améliorer les performances de passage à l'échelle du bus à messages. Cette dernière partie comprend une série d'expérimentations présentant l'amélioration des performances qu'apporte la solution des domaines de causalité.

Partie 1 : Etat de l'art

Chapitre 1 : Message-Oriented Middleware

1.1. Introduction

Il existe deux grandes techniques au sujet des *middlewares*[4] d'interconnexion distribués. Le plus ancien est le *message queuing* (asynchrone), qui date des années 70, et a été suivi dans les années 80 par le concept du RPC (synchrone), qui a évolué jusqu'à CORBA de nos jours.
Le *message queuing* s'est développé progressivement en toute une série de *middleware* à message (*Message-Oriented Middleware*, MOM) qui prennent à leur charge les problèmes transitoires de réseau, de connexion de machines, voire de disponibilité de machines. Ils sont utilisés pour interconnecter des applications de l'entreprise, chaque application interagissant localement avec le système de messagerie.
Le concept du RPC a cherché à répondre à un autre besoin, celui de construire plus facilement des applications communicantes. Le RPC reprend le modèle de programmation synchrone de l'appel procédural, utilisé majoritairement pour construire des applications centralisées, pour cacher la distribution effective des composants lors de leur programmation.

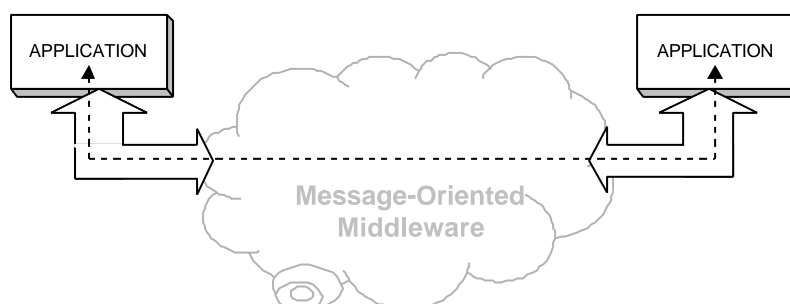


Figure 1: Abstraction d'un *Message-Oriented Middleware*

Il existe deux modes principaux de communication pour l'échange d'information, aussi appelés modes de synchronisation, le mode **synchrone** et le mode **asynchrone**.

1.1.1. Le mode asynchrone

Le mode asynchrone est le mode où « le récepteur n'a pas besoin d'être prêt pour accepter une communication quand l'émetteur fait un envoi » [Agh86]. Le système postal est un exemple simple de communication asynchrone. L'émetteur place une lettre dans une

[4] Le mot *middleware* a trouvé des traductions françaises par *intergiciel* ou *intersticiel*, mais ces traductions n'étant pas reconnues nous continuerons à employer le terme *middleware* dans la suite du rapport.

boîte postale et ne se préoccupe plus de son acheminement. Les services postaux sont responsables de la lettre et sont chargés de la mettre dans la boîte aux lettres du destinataire. Le destinataire n'a pas besoin d'être présent pour recevoir son message.

Cette comparaison nous montre bien que ce mode n'exige pas la disponibilité permanente du producteur et du consommateur entre l'envoi et la réception des messages. La communication par messages étant la base du fonctionnement des MOM, ce qui nous intéresse ici, c'est la possibilité d'échanger des messages entre des applications, que ces applications soient locales ou sur des ordinateurs distants, et que ces applications soient actives ou non. Nous voyons ainsi qu'à la notion d'échange de messages s'ajoute *l'asynchronisme* [Rib] qui est fondamental dans un contexte de traitement réparti : les applications qui échangent des messages ne sont pas nécessairement disponibles simultanément.

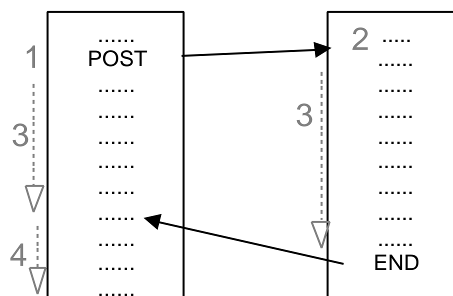


Figure 2: Exemple de communication asynchrone

Parmi les nombreuses implémentations de MOM qui existent actuellement, le mode asynchrone est le mode de communication qui est utilisé pour l'échange des messages dans tous les produits existants (AAA [DBFL99], FIELD [Reis90], IBM Gryphon [BCSS99], Vcom, IBM [Rib], Microsoft MQ [HS99], Peerlogic's Pipes [Pee], etc.)

1.1.2. Le mode synchrone

Le mode synchrone est le plus simple des modes de synchronisation, la définition donnée dans [Agh86] est la suivante : « l'émetteur et le récepteur de la communication doivent être prêts à communiquer avant qu'un envoi ne puisse être effectué ». L'exemple le plus souvent utilisé pour définir le mode synchrone est la communication téléphonique. L'émetteur ET le récepteur doivent établir une connexion afin d'être tous les deux prêts à communiquer, puis ils parlent chacun à leur tour, attendant que l'autre ait fini pour commencer.

Un autre exemple est l'appel de procédure (local ou distant). Lors d'un appel de procédure, l'appelant se **met en attente**; une fois la procédure appelante terminée, l'exécution se poursuit chez l'appelant (voir Figure 3). Le mode synchrone exige que les programmes qui communiquent entre eux soient disponibles au moment de l'échange sous peine de perdre purement et simplement la communication.

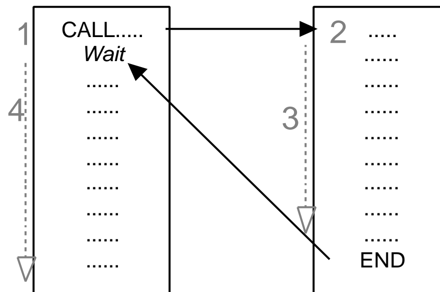


Figure 3: Exemple de communication synchrone

Il existe toute une série de variantes du mode synchrone par exemple le mode *oneway* de Corba [OMG98] qui consiste à réaliser une requête synchrone sans attendre de résultat ; si le destinataire n'est pas présent la requête échoue. Ou bien encore le mode synchrone différé que l'on retrouve au sein de Corba Messaging, dans ce mode, après une invocation à distance, l'émetteur ne se bloque pas mais continue son exécution et par la suite va régulièrement vérifier si les résultats sont arrivés (Polling).

L'intérêt du mode synchrone dans le cas d'un MOM est très limité, car la plupart des avantages du MOM reposent sur l'asynchronisme. Cependant, certaines applications qui communiquent grâce à un MOM ont besoin de réaliser des échanges synchrones, dans ce cas il est possible de simuler une communication synchrone sur des systèmes asynchrones [Reis90] [RR95] [Pee]. La construction d'un échange synchrone sur un système à communication asynchrone est simple, la communication synchrone signifie que le processus émetteur attend la réception d'un acquittement de fin de traitement de la part du récepteur avant de pouvoir continuer.

1.1.3. Synthèse

Un *Message-Oriented Middleware* est un système qui, de part sa nature (échange de messages), utilise une communication asynchrone. L'utilisation de ce mode de synchronisation est dictée par les nombreux avantages de la communication asynchrone. Tout d'abord, les applications qui échangent des messages ne sont plus liées l'une à l'autre, elles peuvent donc avoir des caractéristiques de disponibilité complètement différentes. Ensuite, l'asynchronisme permet de développer des applications qui traitent les informations "au fil de l'eau", c'est à dire au fur et à mesure que les données sont disponibles. Néanmoins les gros inconvénients de l'asynchronisme sont les problèmes de fiabilité et la perte de l'ordre des messages, mais un gestionnaire de messages doit fournir les moyens de limiter, voire d'annuler, ces inconvénients.

Très clairement les deux approches répondent à deux besoins différents, et sont pour l'instant appliquées à des domaines applicatifs distincts. L'approche prise par le monde CORBA consiste à offrir une interface standard, sous forme objet, au service de messagerie. Les MOM offrent une grande flexibilité dans l'intégration d'application, ils permettent la mise en place de solutions indépendantes du temps car ils opèrent en asynchrone ; Ils étendent la communication entre composant en permettant à un client de demander un service, puis de continuer son exécution sans avoir à attendre la réponse. Ils permettent d'intégrer des applications diverses dans des environnements hétérogènes (systèmes différents, réseaux et systèmes de communication différents).

Un Gestionnaire de message (MOM) doit donc suivre certaines règles de synchronisation, mais il doit aussi se conformer à un modèle de communication qui dicte la manière d'échanger des messages, c'est ce qui est abordé dans la section suivante.

1.2. Modèles de communication

D'après [ABDP99], tout modèle de communication (par message ou autre) repose sur l'un des deux principes de base suivants:

- Le mode de communication bidirectionnel, le plus souvent **orienté connexion**, correspond au mode le plus couramment implémenté pour les communications conversationnelles et question/réponse.
- Le mode de communication unidirectionnel souvent **orienté sans connexion**, qui correspond généralement aux implémentations de communication par messages.

Par définition la communication par message est complètement unidirectionnelle, seule l'utilisation d'une communication synchrone pourrait justifier l'utilisation du mode bidirectionnel. Mais les quatre modèles existants sont tous asynchrone ou créés un mode synchrone sur l'asynchrone (voir § 1.1.2), ils utilisent donc tous le mode unidirectionnel.

Le modèle de communication des MOM peut être représenté par une succession de couches qui symbolisent chacune une évolution historique et pragmatique de la communication par messages. Chacune des couches utilise les fonctionnalités fournies par la précédente, un peu comme les couches de communication réseau.

Il existe quatre "couches" de communication (voir Figure 4). Le modèle de **Message Passing** réalise l'envoi de message simples. Le **Message Queuing** ajoute la notion de persistance. Ces deux premiers modèles représentent la base de la communication des MOM. Au-dessus de ces deux modèles on trouve la plupart du temps le modèle par abonnement ou modèle **Publish/Subscribe**, qui utilise les fonctions du *Message Passing* ou du *Message Queuing* ; ce modèle ajoute la notion d'anonymat et d'abonnement. Au-dessus de tous ces modèles on trouve un modèle un peu à part : le modèle **événementiel**. Ce dernier étant plus considéré comme un modèle de programmation que comme un modèle de communication.

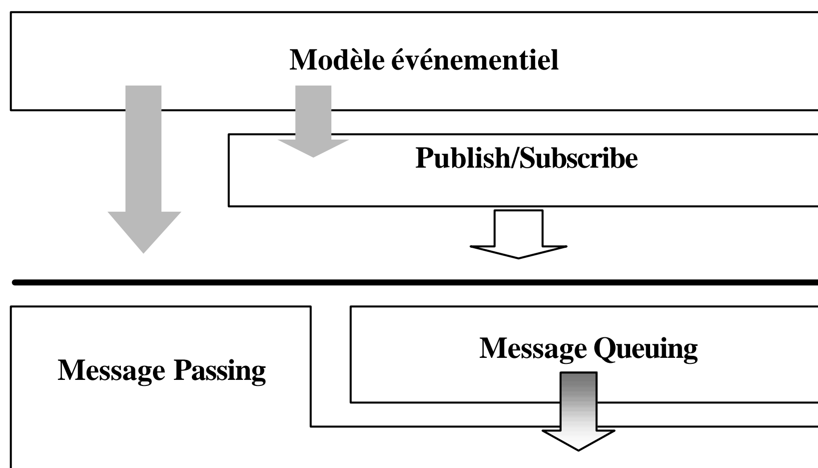


Figure 4: Les quatre modèles de communication des MOM

Il est important de se rappeler que tous ces modèles peuvent êtres utilisés comme modèle de communication des MOM bien qu'un grand nombre d'implémentation soient basé sur du *Message Queuing* ou *Publish/Subscribe*.

1.2.1. Message Passing

Ce mode d'échange est le mode de base de la communication par messages. La communication est unidirectionnelle et généralement non bloquante, elle prend la forme d'un échange de message, de la même manière que l'émission d'une lettre. Ce mode étant sans connexion, la disponibilité du destinataire à l'instant de l'émission n'est pas nécessaire. En effet, même si le destinataire est occupé, le message est stocké dans une **queue de messages** qui fait office de tampon chez le destinataire (représenté par la boite aux lettres). C'est cette file d'attente qui va permettre de répondre au besoin d'asynchronisme en jouant le rôle de tampon entre les applications s'échangeant des messages [Reis90]. L'ordre des messages dans cette queue est dicté par la politique d'ordonnancement du système utilisé.

Figure 5: Le *Message Passing*

Le fonctionnement de ce modèle est simple (voir Figure 5): l'émetteur qui désire envoyer un message fait un *Send* en donnant le nom de la "boîte aux lettres" du destinataire, c'est ensuite le système de *middleware* qui se charge de transférer le message dans la boîte aux lettres du destinataire. Ce dernier n'a plus qu'à faire un *Receive* (lorsqu'il est disponible) pour réceptionner le ou les messages se trouvant dans sa boîte aux lettres. L'exemple typique du système utilisant le *Message Passing* est le courrier électronique (*mail*).

1.2.2. Message Queuing

1.2.2.1. Principe de base

Le *Message Queuing* est souvent confondu, à tort, avec le *Message Passing.* En effet alors que dans le modèle précédent il n'y avait que la notion de queue de message simple (sans autre fonction supplémentaire que celle de tampon), dans le *Message Queuing*, il y a en plus la notion de **fiabilité**. Le rôle de la file d'attente, ici, est d'assurer une fonction de sécurisation des messages permettant de conserver leur intégrité quelle que soit la situation : arrêt de l'application destinataire, arrêt du système ou même destruction du support physique contenant la file d'attente [Rib]. De plus la représentation symbolique de la queue de message fiable n'est plus chez le destinataire mais au sein du gestionnaire de messages. On appelle cela la **persistance** des messages.

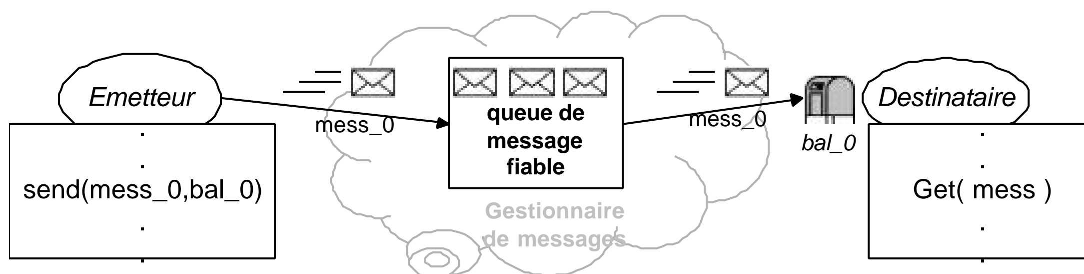


Figure 6: Le *Message Queuing*

Il est important de noter que ce n'est pas le rôle de l'application d'accéder directement aux files d'attente. Les applications font pour cela appel à un gestionnaire de files d'attente par l'intermédiaire d'une interface de programmation (API). Dans celle-ci seul les méthodes *send* et *get* sont disponibles et sont seules capables d'accéder aux files d'attente et d'envoyer ou recevoir un message.

Le *Message Queuing* est un modèle de communication très utilisé dans les MOM actuels (IBM Gryphon [BCSS99], Vcom, IBM MQSeries [Rib], Microsoft MQ [HS99], Peerlogic's Pipes [Pee], etc.), c'est d'ailleurs sur ce modèle qu'ont été implémentés les premiers MOM comme MQSeries d'IBM [Rib]. De plus c'est le modèle le plus facile à utiliser pour interconnecter des applications.

1.2.2.2. Communication de groupe

Le modèle de base du *message queuing* a ces limites: tout d'abord il est nécessaire que l'application émettrice connaisse le nom et l'adresse (i.e. la boîte aux lettres) du destinataire, ensuite si elle veut envoyer la même information à plusieurs destinataires, elle doit faire plusieurs envois (un pour chaque destinataire). Ces inconvénients peuvent être gênants dans certaines applications. Par exemple, on peut faire un parallèle avec la grande distribution: imaginons une grande surface qui est en rupture de stock sur des produits très demandés, le magasin doit lui-même faire la démarche de chercher les adresses de tous les grossistes qui sont susceptibles de posséder le produit. Ensuite le magasin doit réaliser, pour chaque grossiste, l'envoi d'une demande de produit afin de connaître les prix, la disponibilité... etc. Ce procédé n'est pas très efficace car lent et fastidieux et peut être amélioré avec une communication de groupe.

Une solution pour améliorer le *Message Queuing* est **la communication de groupe** [Bal00]. Un groupe est un ensemble de récepteurs identifiés par un nom unique, qui sont gérés de façon dynamique: les arrivées/départs peuvent se réaliser pendant le déroulement de l'application. Ce modèle utilise ensuite le système de communication sous-jacent c'est à dire le *Message Queuing* pour envoyer les messages. Les *middlewares* utilisant ce mode de communication sont Isis, Horus, Ensemble (Université de Cornell). Mais ce modèle ne répond pas encore a toutes les exigences qu'on peut demander à un MOM, comme la communication anonyme, c'est pourquoi beaucoup d'implémentations actuelles utilisent un modèle de communication *Publish/Subscribe*.

1.2.3. *Publish and Subscribe*

Le modèle *Publish/Subscribe*, aussi appelé modèle par abonnement en français, est le plus récent des modèles de communication utilisé dans les MOM [BCS+99] [DBFL99] [Tib99] [OPSS93] [CRW99], il utilise un mode de communication complètement unidirectionnel. C'est un un modèle qui est très proche du modèle événementiel, la principale différence se situe au niveau du critère d'envoi et de la notion d'anonymat. Dans le *Publish/Subscribe* l'événement est envoyé à un gestionnaire qui se charge de le répartir à ceux qui se sont **abonnés** à ce type d'événement (voir Figure 7).

L'avantage de ce modèle est appréciable : reprenons notre exemple de la grande distribution, notre magasin désire toujours rechercher un produit dont il est en rupture de stock. Pour cela, il peut faire comme précédemment et rechercher puis envoyer à chaque grossiste une demande de produit, mais cette façon de procéder est assez "lourde" en terme de temps et de réalisation. Une méthode qui serait plus simple pour lui, serait de signaler à un organisme spécifique sa demande de produit, charge à cet organisme de diffuser la demande aux seuls grossistes qui possèdent le dit produit ; les grossistes s'étant préalablement enregistrés auprès de l'organisme comme possédant le produit. Pour nous, l'organisme est représenté par le MOM.

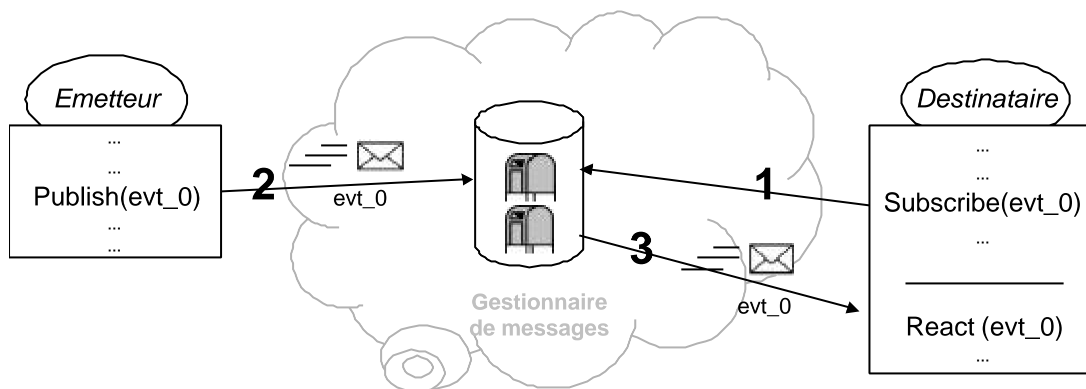


Figure 7: Le modèle Publish and Subscribe

Dans les systèmes de MOM par abonnement il y a deux types de participants. Premièrement des fournisseurs (*providers*) qui envoient (ou publient) des événements dans le système, et deuxièmement des consommateurs (*consumers*) qui s'abonnent (ou souscrivent) à certaines catégories d'événements du système sur certains critères. Le système assure que tout événement publié sera délivré à tout abonné intéressé. Typiquement un système *Publish/Subscribe* est constitué de gestionnaires d'abonnement (*Message Broker*) qui sont responsables du routage des messages entre les fournisseurs et les consommateurs.

1.2.3.1. Critères d'abonnement

Il existe deux politiques d'abonnement: l'abonnement basé sur le sujet de l'événement (*subject-based*), le plus ancien et le plus répandu ou bien une technique émergente basée sur le contenu de l'événement (*content-based*).

Subject-based

Les plus anciens et les plus répandus des systèmes par abonnement sont *subject-based* [OPSS93] [DBFL99] [Tib99], Sun© avec son interface JMS (Java Message Service [Sun99]) propose un service d'abonnement basé sur des critères d'abonnement sur le sujet (*Topic*). Dans ces systèmes, chaque événement est classifié comme appartenant à l'un des types de sujets prédéfinis (aussi appelés groupes, canaux ou intérêts). Les fournisseurs sont donc tenus de donner à leurs événements un des sujets prédéfinis, et les consommateurs doivent s'abonner à l'un de ces sujets fixés.
Continuons avec l'exemple de notre magasin qui ne s'en sort toujours pas de sa rupture de stock : imaginons que les sujets de base proposés par le système soient `[commande de produit / type de produit]`, `[annulation commande / type de produit]`. Certes cette vision est simpliste, on aurait pu créer des sujets plus typés comme par exemple `[ commande de petits pois en boite de 500g et de marque Bonduelle ]`, mais le nombre de sujet deviendrait exhaustif et les gestionnaires d'abonnement auraient beaucoup plus de mal à traiter puis router les événements. Notre magasin envoie donc un événement avec pour sujet [*commande de produit / petits pois*] et pour données la quantité de produit, le nom et le lieu du magasin. Auparavant tous les grossistes qui possèdent des petits pois se sont inscrits pour recevoir les événements qui ont ce type de sujet, le système envoie donc l'événement à tous les grossistes concernés par ce sujet. Le ou les grossistes qui reçoivent le message n'ont donc plus qu'à téléphoner au magasin pour lui faire une offre. Cette méthode est donc plus avantageuse pour le magasin car ce n'est plus à lui de faire la démarche longue et fastidieuse de rechercher et contacter les grossistes.

Content-based

Une alternative émergente à la technique *subject-based* est la technique basée sur le contenu (*content-based*).
Revenons au contexte de l'exemple précédent: des grossistes reçoivent la commande, mais en regardant les données du message certains s'aperçoivent qu'ils ne peuvent pas satisfaire la commande car le magasin ne se trouve pas dans leur secteur de livraison ou bien parce qu'ils n'ont pas la quantité de produit requise. Afin d'éviter d'envoyer inutilement des événements, la gestion des événements pourrait se baser sur le contenu du message ainsi, dans notre exemple, les grossistes ne recevraient que les événements réellement intéressants pour lui c'est à dire que les grossistes trop éloignés du magasin par exemple ne seraient pas notifiés. Un abonnement *content-based* est un prédicat de la forme `[type_message= "commande" & produit= "petits pois" & prix < 250000 & quantité = 30000]` dans notre exemple.

Ce type d'abonnement apporte une flexibilité supplémentaire pour les abonnés, le choix de filtrage parmi les différentes valeurs des attributs des événements permet de s'abstenir de la pré-définition des sujets [BCS+99]. Le principal avantage de l'abonnement *content-based* par rapport au traditionnel *subject-based*, est qu'il permet aux producteurs d'envoyer de l'information sur le réseau sans s'occuper ni de la connaissance des destinataires, ni de la localisation de ces destinataires. Dans notre exemple, l'abonné *subject-based* est forcé de choisir une demande parmi les sujets proposés contrairement à l'abonné *content-based* qui est libre de choisir un critère orthogonal selon la valeur des attributs. Enfin, le critère *content-based* est plus général car il peut être utilisé pour implémenter le *subject-based* alors que la réciproque est fausse.

De nombreux systèmes commencent à utiliser ce type d'abonnement ([BCSS99] [BCS+99]) mais ils doivent faire face aux deux problèmes majeurs que l'on rencontre en utilisant ce critère:
L'association efficace d'un événement envers un grand nombre d'abonnés.
La diffusion *multicast* efficace des événements à travers un réseau de gestionnaires de messages.
Nous n'abordons pas dans cette section les solutions à ces problèmes, celles-ci seront traitées ultérieurement dans le rapport.

Il est à noter qu'il existe une solution légèrement différente du critère *content-based* (notamment abordé dans les environnements de travail tel que Field [Reis90] ou le MOM Siena [CRW99]), il s'agit d'un critère basé sur un modèle d'événement (*pattern-based*). Ce modèle est très proche du critère d'abonnement basé sur le contenu, ici, on ne se préoccupe plus des valeurs des différents champs des événements. Le critère devient le modèle, c'est à dire le type de l'événement ainsi que le type de ses champs, par exemple `EVENT-[RECHERCHE, %produit, %quantité, GRENOBLE]`, dans ce cas on s'abonne à un événement de recherche de produit quelconque, de quantité quelconque mais sur la ville de Grenoble. On peut donc voir que cela se rapproche très fortement du critère basé sur le contenu en utilisant des prédicats.

1.2.4. Modèle événementiel

Le modèle événementiel existe depuis plus de d'une dizaine d'années dans les bases de données la notion de déclencheur (*Trigger*) faisait appel à un modèle par événement, puis sont apparues les règles active E-C-A (Evénement-Condition-Action) qui apportent une nouvelle façon de programmer. Le schéma de programmation typique du modèle événementiel est le suivant :

Lorsque	un événement du type **E** se produit.
Si	la condition **C** est satisfaite.
Alors	exécuter l'action **A**.

Il est important de comprendre qu'il se crée une association dynamique entre l'événement et la réaction[5], cette association porte sur le nom de l'événement. Une réaction est un traitement associé à l'occurrence d'un événement, autrement dit, une réaction peut être comparée à une procédure qui est appelée lorsque l'événement associé arrive chez le destinataire de l'événement. Une notion importante qui s'ajoute aux propriétés du modèle événementiel est la communication anonyme, en effet grâce à ce système il y a une totale indépendance entre l'émetteur et le (les) consommateur(s) d'un événement, cela permet de renforcer la sécurité.

[5] L'action exécutée à la suite d'un événement est très souvent appelée "Réaction", le mot "Action" étant plutôt destiné aux SGBD actives.

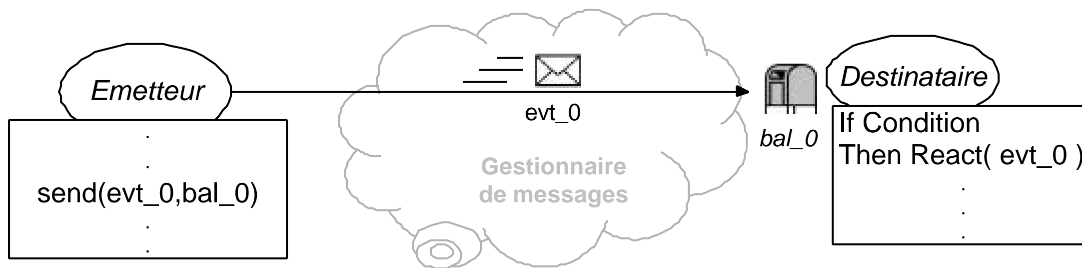


Figure 8: Modèle événementiel.

Le modèle événementiel utilise explicitement une communication par message pour envoyer les événements, il peut donc se servir aussi bien du *Message Passing* que du *Message Queuing* (voir Figure 8). C'est un modèle de communication qui se rapproche beaucoup des objectifs du MOM, et même si une majorité de MOM lui préfère le *Publish/Subscribe.* Il est souvent utilisé comme base de communication comme dans Information-Bus [OPSS93] ou bien comme mode de communication unique comme dans le MOM AAA [DBFL99] et dans un environnement de travail comme FIELD [Reis90].

1.2.5. Comparaison du modèle événementiel et du modèle Publish/Subscribe

On a souvent tendance à penser que le modèle événementiel est une couche de communication qui se branche sur le modèle *Publish/Subscribe* pour y apporter la notion d'abonnement. En fait si le modèle *Publish/Subscribe* est une évolution des autres modèles de communication, le modèle par événement est plus un modèle de programmation qu'un modèle de communication.
En effet, le modèle événementiel apporte la notion d'envoi d'événement à un (ou plusieurs) destinataire explicite. De plus, dans un système événementiel on a un modèle de réception complètement asynchrone appelé mode *Push,* qui consiste à réaliser une réception de manière implicite (la réception d'un événement entraîne l'exécution de la réaction associée).

Dans le modèle *Publish/Subscribe* il n'existe pas cette notion de destinataire explicite, l'envoi passe nécessairement par un intermédiaire appelé gestionnaire d'abonnement qui se charge de faire correspondre un événement à son (ses) abonné(s). Il n'y a donc pas de destinataire explicite. Comme dans le modèle événementiel la réception des événements se réalise selon un modèle *Push*, mais la réception peut aussi se réaliser selon un modèle synchrone *Pull* comme c'est le cas dans l'interface JMS [Sun99] (les clients viennent prendre périodiquement leurs messages sur le serveur).

Le modèle événementiel est donc un modèle à part entière qui peut se servir du modèle *Publish/Subscribe* comme base de communication mais qui peut tout aussi bien s'appuyer sur les couches inférieures (*Message Passing* ou *Message Queuing*).

1.3. Architectures des MOM

Comme on vient de le voir, les modes et modèles de communication des MOM sont assez différents, mais ils sont toujours basés sur un échange de messages. Les messages empruntent les canaux de communication qui interconnectent les machines sur lesquelles les

applications clientes s'exécutent. Ces réseaux de messages peuvent prendre plusieurs formes, du serveur centralisé au réseau complexe. Il existe trois types d'architectures de MOM, le mode centralisé (*hub & spoke*), le mode réparti en point à point (*snow flake*) et enfin le mode réparti en bus (*bus*).

1.3.1. Hub & spoke

L'architecture *hub & spoke* est appelée ainsi car elle se présente comme une roue de vélo, où tous les rayons (*spoke*) sont reliés au même moyeu central (*hub*) (voir Figure 9). Cette architecture est constituée d'un serveur principal unique qui centralise tous les messages (c'est le cas du MOM FioranoMQ [Fio00]). L'envoi de message est très simple à réaliser, l'émetteur lance son message au serveur central (*broker*) puis celui-ci le transmet dans la boîte aux lettres du destinataire [Ope98].

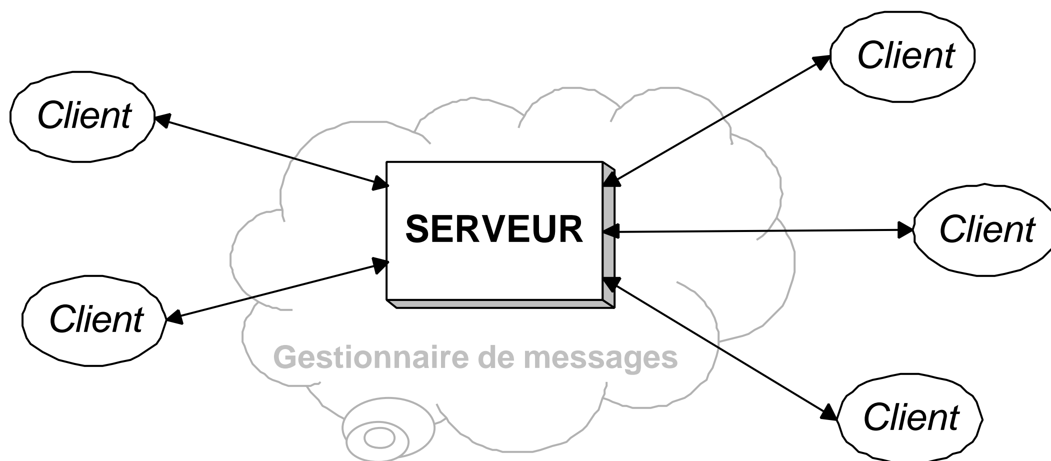


Figure 9: Architecture *Hub & spoke*

On peut comparer cette architecture à l'architecture Client-Serveur où tous les clients seraient rattachés au même serveur. C'est cette centralisation qui rend l'implémentation de ce type d'architecture assez simple, de plus la mise en place de certaines propriétés comme l'ordonnancement des messages ou la synchronisation devient très facile. Néanmoins la centralisation du traitement apporte de nombreux désavantages, tout d'abord cela engendre des problèmes de fiabilité, si le gestionnaire tombe en panne, tout le *middleware* est en panne et aucun messages ne peut circuler, toutes les applications qui interagissent sont donc bloquées jusqu'à rétablissement du MOM. Ensuite cette centralisation pose de gros problèmes de passage à l'échelle (la scalabilité[6]), en effet, il est assez simple de gérer une trentaine de clients sur le même gestionnaire, mais à partir de plusieurs centaines de clients le "serveur" central ne peut plus supporter la charge globale de tous les échanges de messages (voir 2.2). Cette architecture n'est donc pas recommandée pour réaliser un *middleware* qui nécessiterait une grande sécurité des échanges et/ou qui devrait être étendu sur de nombreuses machines.

[6] le terme anglais couramment utilisé pour décrire le passage à l'échelle est: **the scalability**, qui a donné l'anglicisme: **la scalabilité**, ainsi que l'adjectif: **scalable**.

1.3.2. Snow flake

Le deuxième type d'architecture est appelée *snow flake* à cause de sa topologie en flocon de neige (voir Figure 10). Contrairement à l'architecture précédente, dans celle-ci le gestionnaire de messages est distribué sur plusieurs sites. Le MOM est donc représenté par un ensemble de serveur qui constituent le gestionnaire global, chaque serveur étant une partie (un sous-module, une subdivision, etc.) de l'ensemble. Lorsqu'un client (une application) s'adresse au gestionnaire de messages il s'adresse en fait à l'un des "serveurs de messages". L'envoi de message se réalise de façon transparente pour le client, tout se passe comme s'il n'y avait qu'une seule entité gestionnaire. L'émetteur envoi un message au gestionnaire de messages via un serveur appelé point d'accès, puis le message est transmis par le serveur soit directement au destinataire si celui-ci est associé au même serveur, soit à un autre serveur auquel est associé le destinataire.

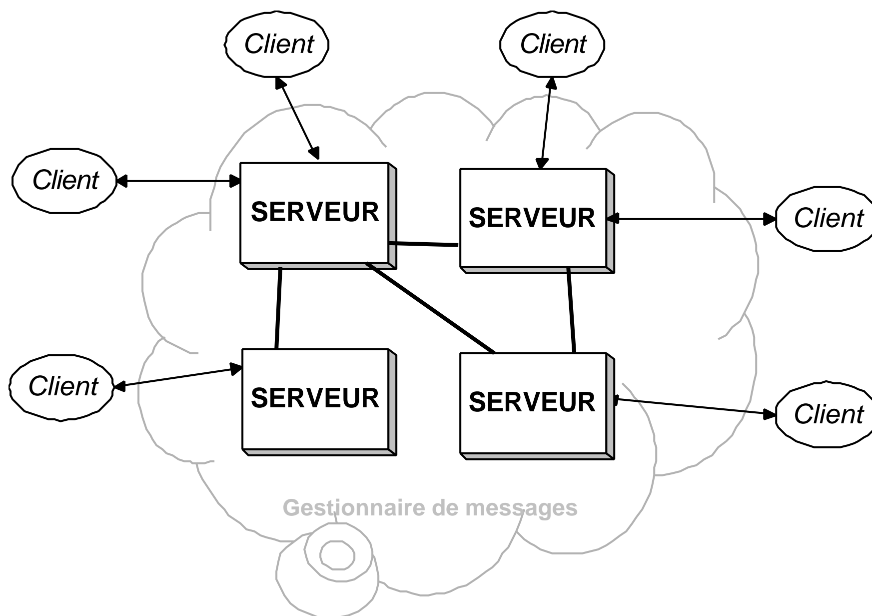


Figure 10: Architecture *Snow flake*

On peut comparer cette architecture à l'Internet, les serveurs de messages faisant office de routeur de message, avec des propriétés de fiabilité et ordonnancement en plus. Le gros avantage de cette architecture est la propriété de distribution du service de messages qui faisait défaut au modèle précédent. Grâce à cette répartition, il devient plus aisé de déployer des applications sur une large zone car la charge générale du gestionnaire de message est répartie sur l'ensemble des serveurs que ce soit la charge CPU ou la charge réseau. De plus, une architecture répartie apporte une fiabilité au système, car si un serveur tombe en panne, le reste des serveurs interconnectés peut continuer à fonctionner presque normalement. L'arrêt de l'un des serveurs ne pénalise donc pas l'ensemble des applications reliées au MOM.

La topologie de cette architecture permet de réaliser plus facilement une application à grande échelle. Dans l'architecture précédente, le passage à l'échelle était limité à cause du nombre restreint de clients associés à l'unique serveur gestionnaire de messages. Dans cette configuration, il est aisé de relier un grand nombre de serveur et de déployer les clients sur ces différents sites. Ainsi tous les clients ne s'adresseront pas toujours au même serveur. Ce

type d'architecture est aujourd'hui la plus répandue parmi les MOM industriels [BCSS99] [Pee] [HS99].

1.3.3. Bus

L’architecture en Bus est la plus récente des architectures répartie, elle est employée dans les MOM les plus récents [DBFL99],[Reis90],[OPSS93].
Contrairement aux deux modèles précédents, le modèle de bus à messages n'est pas forcément une architecture définie au niveau physique. Lorsque l'on parle de bus à messages on fait référence à un **bus logiciel** dans lequel les applications puisent les informations [OPSS93] ou par lequel les messages s’échangent [DBFL99]. Ce bus logiciel est en fait une vision logique d’une architecture distribuée dont le but est d’acheminer les messages à leurs destinataires. La vision réelle de l’architecture d’un bus est une architecture *snow flake* qui serait complètement maillée. La vision logique d’un Bus est la même que la représentation d’un LAN, c’est à dire un canal de communication reliant toutes les machines

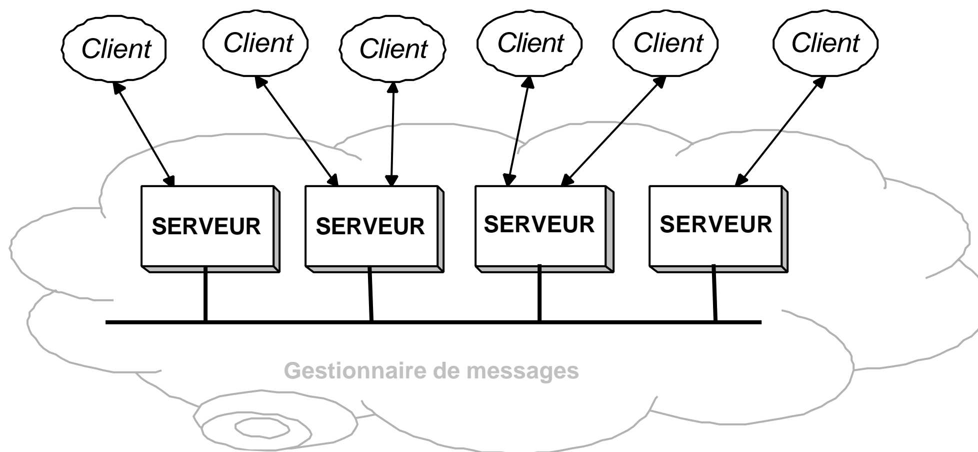


Figure 11: Architecture d’un Bus logiciel

Un bus logiciel est mis en œuvre de façon distribué, cela implique qu’il existe sur chaque site une entité représentant le bus (un peu comme dans l’architecture *snow flake*). Cette entité appelée bus local est présente dans tous les serveurs du gestionnaire de messages. Il est responsable de ses communications locales (au sein d’un même serveur) mais aussi de l’acheminement d’un message à un client distant. Comme dans l’architecture *snow flake*, l’envoi d’un message à un client distant est complètement transparent pour le client, c’est le bus local qui se charge de transférer le message au bus distant qui lui-même le transmet à son client directement.
L’architecture en Bus garde les mêmes avantages que l’architecture *Snow flake* au niveau de la distribution des serveurs, c’est à dire diminution de la charge CPU des serveurs, diminution de la charge réseau…etc. Elle apporte en plus la notion de communication de groupe qui est très adapté au modèle de communication *Publish/Subscribe*, de plus c’est une architecture qui est très adapté au réseau LAN. Néanmoins on garde toujours les problèmes de cohérence des données dupliquées, et le fait d’être très adapté au LAN n’est pas forcément compatible avec la topologie réseau disponible.

1.4. Propriétés des MOM

1.4.1. Introduction

On associe souvent les MOM à leurs caractéristiques de communication ; l'asynchronisme des envois de messages, l'architecture distribuée, la communication par abonnement…etc. Mais les MOM possèdent aussi des propriétés qui assurent une certaine qualité de service. La tolérance aux pannes, la tolérances aux fautes, la sécurité et l'ordonnancement sont les quatre propriétés essentielles des MOM [BCSS99]:

- La propriété de tolérance aux pannes permet aux MOM de garantir une fiabilité des communications en cas de crash de la machine d'exécution.
- Le modèle de tolérances aux fautes garantit l'atomicité des réactions et permet d'assurer la cohérence de l'exécution des MOM.
- La sécurité permet d'assurer la protection et l'intégrité des informations échangées.
- L'ordonnancement des messages permet d'assurer une cohérence de l'exécution globale des applications clientes.

Dans notre recherche d'optimisation des MOM pour le passage à l'échelle, nous ne nous attarderons pas sur la propriété de sécurité qui nous concerne moins, dans cette optique nous ne développerons pas cette propriété ici, néanmoins des points de repères peuvent êtres trouvés dans [BCSS99].
Nous nous employons dans la suite à étudier les propriétés de fiabilité et d'ordonnancement et leur implantation dans les différents MOM.

1.4.2. Qualité de service

Les modèles de tolérances aux fautes et de tolérances aux pannes sont des propriétés qui se regroupent dans les MOM sous la *qualité de service*.
Le modèle de tolérance aux fautes et aux pannes qui est typiquement implémenté dans les systèmes de communication de groupe traditionnels[7] est inapproprié pour les applications basées sur un MOM [BCSS99]. Dans les MOM, l'entité gestionnaire de message doit être vue comme une entité abstraite fiable qui permet de garantir la délivrance des messages.
Cette garantie est implémentée grâce à des propriétés de persistance et d'atomicité. Les abonnements sont persistants, les messages ne peuvent pas être perdus, permutés ou dupliqués, et des messages incorrects ne peuvent être générés. Une réaction ne doit pas s'arrêter en pleine exécution. L'implémentation doit donc préserver l'apparence de persistance et d'atomicité même si le système distribué peut contenir des composants logiciels ou matériels byzantins. Dans le cadre d'un modèle de communication par abonnement, ceci signifie que lorsqu'un abonné défectueux se reconnecte, il doit être possible de, soit délivrer tous les messages qu'il a manqués, soit de recevoir une petite série de messages qui recréera son état. De plus, contrairement aux systèmes traditionnels, il n'est pas suffisant de dire à un abonné défectueux ou déconnecté que son abonnement a été abandonné.
Si on regarde la norme *Corba Notification Service*, on peut voir qu'ils définissent quatre type de politiques de délivrance : *Best-effort*, *au-plus-une-fois*, *au-moins-une-fois* et *exactement-une-fois*. Ces politiques de délivrances sont basées sur l'association de la fiabilité des événements et de la fiabilité des connexions. A chacune des ces propriétés on

[7] Qui consiste à enlever automatiquement du groupe un processus défaillant ou lent donc à contourner le problème.

peut associer un niveau de fiabilité qui peut être soit persistant soit non-persistant. Une politique de délivrance *Best-effort* consiste à avoir une fiabilité non-persistante des événements et des connexions, c'est le cas de SIENA [CRW99]. *MQSeries* [Rib] utilise lui une fiabilité des événements (*Message Queuing*) mais une non-persistance des connexions, c'est une politique *au-moins-une-fois*. Une politique *au-plus-une-fois* consiste à maintenir une fiabilité des connexions (persistance) mais pas de événements. Lorsque l'on applique une fiabilité de persistance des connexions et des événements, on obtient une politique de délivrance *exactement-une-fois*, c'est le cas des MOM Fiorano [Fio00], MSMQ [HS99], et AAA [BDFHS99]. Certains MOM comme *Tib/Rendez-vous* [Tib99] offre même plusieurs niveaux de politique de délivrance que l'utilisateur choisit..

La propriété d'atomicité est une autre composante très importante de la qualité de service car elle permet de garantir une cohérence de l'exécution. La propriété d'atomicité utilise une gestion de transactions, celles-ci permettent des opérations d'extraction et d'insertion de messages dans une même unité logique de traitement. Toutes les opérations d'une séquence sont réalisées au sein d'une transaction, en cas de terminaison correcte de la séquence, il y a garantie de la conservation des changements liés à ces opérations, sinon annulation de tous les changements effectués. Ce type de gestion de la fiabilité et de l'atomicité est utilisée dans les MOM suivant [DBFL99] [Tib99] [Ope98] [OPSS93].
L'atomicité et la fiabilité permettent donc de garantir la délivrance des messages et donc d'assurer la une qualité de service sur le MOM.

1.4.3. Ordonnancement

1.4.3.1. Concepts

Dans les environnements distribués comme les MOM, le non-determinisme survient inévitablement, à cause de l'asynchronisme et du contexte d'environnement distribué. Dans n'importe quel gestionnaire de message, il est impossible de prédire précisément quand une information envoyée par une application sera reçue par une autre. Ceci est particulièrement vrai lorsqu'on utilise des réseaux à grande échelle (sur L'Internet par exemple), un modèle réaliste suppose donc que l'ordre d'arrivée des informations envoyées est complètement arbitraire et entièrement inconnu. En particulier, les communications de différentes sources à un destinataire donné (et dans certains cas, la source elle-même) peuvent arriver dans un ordre complètement aléatoire[8] et à des moments très différents. Lorsque cela se produit, il est nécessaire de fournir un mécanisme qui ordonne les messages à l'arrivée.

La plupart des implémentations actuelles de MOM ne proposent pas de politique d'ordonnancement des messages, au mieux ils utilisent des canaux d'échange FIFO afin de garantir un ordonnancement entre deux sites [OPSS93]. La plupart des MOM qui nécessitent une réelle politique d'ordonnancement dû aux applications qui s'appuient dessus (par exemple des applications de gestion d'ordres boursier, comme c'est le cas de TIB/Rendez-vous [Tib99]), utilisent une architecture spécifique centralisée[9]. L'utilisation d'une telle architecture permet de garantir un ordonnancement de facto car toutes les informations passent par le serveur central et sont redistribuées au(x) destinataire(s) dans l'ordre d'arrivée.

[8] C'est à dire complètement différent de l'ordre d'envoi.
[9] *hub & spoke* (§ Partie 1 :1.3.1), avec canaux FIFO entre les clients.

Il existe aussi une notion de priorité de message que l'on retrouve dans MSMQ [HS99] : dans ce MOM on peut définir des messages comme étant urgents afin qu'ils arrivent avant (plus rapidement) les autres. Au niveau de l'implémentation cela revient à les mettre en tête de la file d'envoi et à l'arrivée en tête de la file de réception. Mais cela ne répond pas au réel besoin d'ordonnancement.
Obtenir un réel ordonnancement sur des implémentations distribuées est beaucoup plus difficile car les MOM utilisent un mode de communication asynchrone. Dans les systèmes distribués asynchrones il n'existe pas de bornes sur la vitesse relative des processus ni sur le délai de propagation des messages. La communication est le seul mécanisme possible pour la synchronisation de tels systèmes [BM93]. Mais l'absence d'une horloge globale temps-réel oblige à créer un *temps logique* afin d'ordonner les messages selon un *ordre causal.*

1.4.3.2. La causalité

L'ordre causal (ou relation de précédence causale) parmi les événements d'un système est un concept fondamental, qui permet d'aider à résoudre les problèmes d'ordonnancement dans les systèmes distribués. L'ordre causal utilise le principe des horloges logiques introduit par [Lam78].
Dans un système à horloges logiques, chaque processus a sa propre horloge logique et chaque événement se voit affecter une estampille de temps (égale à l'horloge logique du processus émetteur). On obtient la propriété suivante :
« Si un événement *a* affecte causalement un événement *b*, alors l'estampille de *a* est plus petite que celle de *b* ». Cette propriété essentielle permet d'obtenir un ordre sur les événements en fonction de leur estampille.

Remarque :

Pour la suite nous avons besoin de définir deux propriétés ([BM93][RS95]), avec pour convention : e_i = événement[10] numéro i et $C(e_i)$ = estampille de l'événement i, la relation $\rightarrow$ signifie "*précède causalement*" et la relation < définit la notion d'ordre sur les estampilles[11]:

- Propriété *d'horloge cohérente* : $e_1 \rightarrow e_2 \Rightarrow C(e_1) < C(e_2)$
- Propriété *d'horloge fortement cohérente* : $e_1 \rightarrow e_2 \Leftrightarrow C(e_1) < C(e_2)$

Il existe plusieurs type d'horloge logique, les horloges logiques simples (temps scalaire), les horloges logiques vectorielles (temps vectoriel) et enfin les horloges logiques matricielles (temps matriciel).
L'implémentation d'une horloge logique simple ne permet d'obtenir que la propriété *d'horloge cohérente* qui ne permet de connaître l'ordre des événements qu'à posteriori et ne permet qu'à un processus de mesurer sa propre progression.
L'utilisation d'une horloge vectorielle permet d'avoir la propriété *d'horloge fortement cohérente*, on peut donc facilement reconstituer une trace d'exécution globale (réalisation d'état global) mais toujours et seulement a posteriori[12].

---

[10] Un événement étant caractérisé par l'envoi d'un message (*send(mes), subscribe(mess), publish(mess),…*) ou la réception d'un message (*recv(mess), react(mess),…*)

[11] pour une horloge scalaire $HL_1 < HL_2$ signifie que la valeur de HL_1 est plus petite que celle de HL_2 et pour les horloges matricielles et vectorielles que chaque élément de HL_1 est plus petit ou égal à l'élément homologue de HL_2.

[12] On peut tester l'ordre causal d'événement au moment de leur réception avec des horloges vectorielles à condition d'utiliser une diffusion fiable et causale des événements, mais à ce principe on préfère l'utilisation d'horloge matricielle.

La plus difficile mais la plus utile des horloges logiques dans le cadre d'ordonnancement de messages asynchrones est l'horloge matricielle. L'utilisation d'une horloge matricielle respecte la propriété *d'horloge fortement cohérente*, et en plus, permet de savoir au moment de la réception d'un événement si celui-ci dépend causalement d'un autre événement qui n'est pas encore arrivé sur le site. Dans ce cas on retarde la délivrance du message jusqu'à la réception de l'événement précédent (au sens causal du terme). L'implémentation[13] d'une telle politique d'ordonnancement pose de nombreux problèmes qui seront abordés plus tard (voir 2.3.2).

1.5. Synthèse

Ce chapitre nous a permis d'obtenir une vision global des MOM, leurs différents modèles de communication, les architectures et les propriétés disponibles.
Il existe quatre modèles de communications, tous basés sur l'envoi asynchrone de messages, Ils permettent de faire communiquer des applications fortement découplées dans des environnements hétérogènes :

- Le **Message Passing** permet l'envoi de messages unidirectionnel dans des queues de messages non sécurisées.
- Le **Message Queuing** comporte les même propriétés que le *Message Passing* mais apporte en plus la notion de fiabilité des queues de messages (propriété de persistance).
- Le **modèle Publish/Subscribe** ou modèle par abonnement permet de réaliser un envoi de 1 vers N selon des critères d'abonnement.
- Le **modèle événementiel**, qui se rapproche plus d'un modèle de programmation, suit le modèle règles actives E-C-A des SGBD.

Il existe trois types d'architectures qui assurent une modularité de déploiement des MOM sur n'importe quel type de topologie réseau et applicative :

- La topologie **hub&spoke** est une architecture centralisée représentée par un seul *serveur* représentant le MOM.
- La topologie ***snow flake*** est une architecture distribuée représentée par un ensemble de serveurs interconnectés (mais pas complètement maillés).
- La topologie en **Bus** est une architecture distribué représentée par un ensemble de serveurs interconnectés à la manière d'un LAN (c'est à dire un graphe de serveurs complètement maillés).

Les propriétés assurent la fiabilité et la cohérence des envois de messages. Il existe plusieurs politiques de qualité de services permettant de garantir certaines propriétés sur les MOM. La fiabilité, l'atomicité et l'ordonnancement sont les plus importantes. La fiabilité sur les connexions et sur les queues de messages permet d'assurer la garantie de délivrance des messages et de tolérance aux pannes. L'atomicité permet de garantir une certaine tolérances au fautes, et une politique d'ordonnancement assure une cohérence des échanges de messages.
Grâce à toutes ces caractéristiques, les MOM sont souvent comparés à une "technologie de collage" des applications distribuées ("*glue technology*", [BCSS99]). Cette expression exprime clairement l'intérêt des MOM, c'est à dire l'intégration d'applications en environnements hétérogènes distribués. Et le développement actuel des larges réseaux augmente la degré d'utilisation des MOM, mais pose de nombreux problèmes lors du passage à l'échelle que nous allons essayer de détailler dans le chapitre suivant.

[13] Pour plus de détail sur l'implémentation d'une horloge matricielle se référer à la Partie 1 :4.1)

Chapitre 2 : Passage à l'échelle

2.1. Problématique

Le développement des larges réseaux (WAN, Wide-Area Networks), tel que l'Internet, offrent des nouvelles motivations pour adopter l'utilisation des middleware à messages. Le nombre important de clients (générateurs de messages) créé une opportunité pour le développement d'applications originales qui peuvent effectivement communiquer par messages. Les exemples les plus flagrants sont l'analyse des marchés, le commerce en ligne, l'indexation, la disponibilité et la sécurité des bases de données et de façon plus générale : l'intégration d'applications. De plus de nombreuses applications existantes qui sont déjà conçues autour de la notion d'interactions de messages (d'événements) peuvent augmenter leur scalabilité (ou passage à l'échelle) à travers la connectivité globale des WAN. Par exemple, les systèmes de workflow à base d'événements et prévus généralement sur des réseaux locaux peuvent être associés à travers les multiples localisations physiques d'une grande entreprise.
Lorsqu'on parle de passage à l'échelle, on ne se réfère pas seulement au nombre de clients émetteurs, au nombre de clients destinataires ou au nombre des messages échangés mais aussi au bouleversement des nombreuses suppositions faites pour les réseaux locaux (LAN) tel que la faible latence, l'abondante bande passante, les plates-formes homogènes, les connexions fiables et continues, le contrôle centralisé…etc.
Il y a de nombreuses dimensions dans le passage à l'échelle, la plupart des analyses se sont portées sur le problème de l'association efficace entre les abonnements et les messages dans les MOM à base de communication publish/subscribe (*message matching* [BCS+99][CRW99]) et du routage efficace de ces messages.
Ce chapitre s'étend sur l'impact du passage à l'échelle sur la fiabilité et les performances des différents types d'architectures et sur les problèmes qui en découlent sur les propriétés des MOM.

2.2. Architectures scalables

2.2.1. Hub & spoke

De nombreux MOM ont été développés pour des réseaux locaux, et généralement basés sur un serveur centralisé. L'utilisation de cette architecture a pour conséquence l'incapacité à s'adapter aux WAN, où le nombre et la distribution des services aux clients peut rapidement submerger une solution centralisée. Ce type d'architecture est un véritable goulot d'étranglement, et cela à deux niveaux : pour la charge CPU, et pour les communications sur le réseau.
Tout d'abord, le CPU est sollicité pour gérer le serveur central c'est à dire gérer la correspondance entre les messages et les abonnements et pour cela il est nécessaire d'exécuter des algorithmes coûteux. De façon plus générale le traitement des messages venant des clients devient considérablement plus coûteux pour le CPU lorsque ce nombre

augmente, et gérer des centaines de milliers de messages devient impossible pour un serveur unique.
Ensuite, des problèmes réseaux viennent se greffer sur la surcharge CPU car les clients qui veulent s'échanger des informations passent forcément par le serveur et donc surcharge les canaux de communication. De plus, la distance des clients au serveur joue un rôle non négligeable. L'emplacement du serveur ne correspond pas forcément à l'emplacement des clients. Si le serveur qui regroupe toutes les informations se situe à une grande distance des clients qui l'utilisent, le temps de latence devient énorme et les communications seront d'autant plus lentes que les clients seront loin et nombreux.
Le passage à l'échelle rime aussi avec disponibilité, et un autre inconvénient majeur de ce type d'architecture est la disponibilité, en effet si le serveur lâche, plus aucun client ne peut se connecter et communiquer. Cette architecture est donc la moins adaptée à la scalabilité des MOM.

2.2.2. Snow flake

Dans l'architecture *snow flake*, les principaux problèmes qui se posaient précédemment sont résolus grâce à la distribution des serveurs. Cette distribution permet d'une part de réduire la charge CPU sur chaque serveur et d'autre part de réduire la concentration des communications réseau sur une même machine. Cela se réalise par le rapprochement des serveurs vers les clients (et vice et versa).
Néanmoins cette architecture n'est pas si idéale que cela, la large distribution imposée par le passage à l'échelle, inflige des contraintes au niveau de la cohérence de la base d'abonnement, du routage efficace des messages et des problèmes de scissions du MOM.
Les problèmes de cohérence sont important dans l'architecture *snow flake*, par exemple si on utilise le modèle *Publish/Subscribe* comme modèle de communication sur cette architecture, lorsqu'un client réalise un désabonnement (unsubscribe) il faut garantir qu'il ne recevra plus de message de ce type et donc faire circuler l'information sur toutes les bases d'abonnement répliquées sur l'ensemble des serveurs. A ce problème vient s'en greffer un autre intrinsèquement lié : le problème de routage efficace des messages vers leurs abonnés. Ces deux problèmes sont connus sous la nom anglais "*message matching*".
Le problème de scission est un problème inhérent à la répartition, car si une liaison se brise entre deux parties du MOM, celui-ci peut être coupé en deux. Cela devient très gênant et engendre les problèmes de cohérence et de routage des messages.

2.2.3. Bus

Les avantages de l'architecture en Bus pour la scalabilité sont nombreux. Tout d'abord, la forte distribution des serveurs permet, comme pour l'architecture *snow flake* de répartir la charge CPU et la charge réseau, ensuite ce modèle est très adapté au mode publish/subscribe. Contrairement au modèle précédent, le graphe des serveurs étant complètement maillé, les problèmes de cohérence, de routage efficace et de scission sont quasiment nuls car toutes les informations peuvent être envoyées à tous serveurs et si une connexion se brise les serveurs peuvent toujours communiquer avec les autres.
Néanmoins ce type d'architecture est très, voire trop, adapté pour les LAN, et la topologie de l'application que l'on souhaite déployer sur une architecture en Bus ne correspond pas forcément à la topologie d'un LAN. C'est à dire que les clients ne se trouvent pas forcément rassemblés sur un même réseau local. L'utilisation de routeurs s'avère souvent nécessaire et on se retrouve dans les même problèmes que dans l'architecture *snow flake*, et il devient

encore plus difficile de régler les problèmes intrinsèques à la répartition (routage efficace, cohérence de la base,… etc.)

2.3. Influence des propriétés

2.3.1. Routage efficace des messages

Un des principaux problèmes lors du passage à l'échelle se situe au niveau du routage efficace des messages, en fait ce problème se découpe en deux. Premièrement, le problème de la correspondance efficace des messages avec un grand nombre d'abonnés sur des serveurs répartis (problème appelé *message matching*).
Deuxièmement, le problème d'envoi efficace en multicast des messages à travers le réseau de serveurs. Ce dernier est crucial sur deux points, tout d'abord quand le système est géographiquement très éloigné et que les serveurs sont connectés via un WAN relativement lent (par rapport aux connexions rapides des LAN). Ensuite la scalabilité oblige le système à s'élargir afin de supporter le nombre important de clients (émetteurs et consommateurs dans le cadre de *Publish/Subscribe*). Dans les deux cas il est nécessaire de n'envoyer les événements qu'aux serveurs qui ont des clients intéressés.
Le premier problème est souvent résolu avec l'aide d'une table de correspondance. Et le deuxième en définissant des groupes multicast. Mais ces solutions ont du mal à passer à l'échelle, et nécessite l'application d'algorithmes de traitement spécifiques, des solutions ont été trouvées dans [CRW99] [BCS+99] [BCSS99].

2.3.2. ordonnancement

Comme on a pu le voir dans le chapitre précédent, l'ordonnancement des messages peut prendre plusieurs formes, l'ordonnancement FIFO est le type d'ordonnancement qui est le moins coûteux car souvent il découle de l'utilisation de couche réseau comme TCP/IP qui intègre une gestion optimale de cette propriété. Par contre l'utilisation d'ordonnancement causal introduit des problèmes significatifs lors du passage à l'échelle parce que l'augmentation linéaire du nombre de participant accroît de façon quadratique le coût de la gestion de la causalité.
Ce coût est dû au problème de *buffering* qui est le facteur principal de perte de performance [CS93]. Par *buffering* il faut entendre tout ce qui se rapproche aux coûts de stockage (en mémoire et sur support persistant) des horloges logiques et des messages en attente. Lorsque le nombre de participants augmente, le nombre de messages causalement dépendants augmente lui aussi à peu près proportionnellement. Le nombre de messages référencés dans des dépendances causales et qui n'ont pas été délivrés augmente proportionnellement en fonction des références, de plus il faut prendre en compte les taux d'erreurs (délais d'arrivée) qui représente un pourcentage non négligeable du nombre de messages (une classique mais réelle supposition). Donc les besoins (donc les coûts) de *buffering* augmente de façon exponentielle avec le nombre de client.

De nombreuses optimisations existent pour réduire le coût lié à l'ordonnancement comme l'envoi de messages par paquet, le débrayage de la gestion de l'ordonnancement du niveau système au niveau applicatif, le partitionnement des participants en groupes distincts suivant la topologie de l'application ou du réseau. Nous nous attarderons à proposer certaines solutions dans la Partie 3.

Partie 2 : L'environnement AAA (Agent Anytime Anywhere)

Chapitre 3 : Le bus à messages AAA

Nous avons vu dans la partie précédente une classifications des différents modèles de communication, des architectures et des propriétés des MOM. Nous présentons dans celle-ci le bus à messages AAA (Agent Anytime Anywhere) appelé aussi bus à agent développé au sein du laboratoire Sirac en collaboration avec le GIE Dyade. C'est sur ce MOM que nous chercherons à résoudre le problème du passage à l'échelle.
Dans ce premier chapitre nous présentons le MOM AAA et ses caractéristiques, puis dans le chapitre suivant nous montrerons l'impact de ces caractéristiques sur le passage à l'échelle.

3.1. Concepts du bus à agents

Les agents sont des objets "réactifs" qui se comportent conformément au modèle "événement → réaction" : un événement est une transition d'état significative à laquelle un ou plusieurs agents vont réagir. Dans notre modèle, un événement est représenté par un message typé appelé *notification*. Une notification contient toutes les informations nécessaires décrivant l'événement associé. Lorsqu'un agent reçoit une notification, il doit exécuter la réaction appropriée. L'envoi de notifications s'effectue au travers d'un bus logiciel qui assure certaines propriétés sur les communications. L'envoi de notifications représente le seul moyen de communication et de synchronisation entre agents.
Un agent est composé d'un état et d'un ensemble de règles de type condition=>réaction qui décrivent comment un agent réagit aux notifications qu'il reçoit.

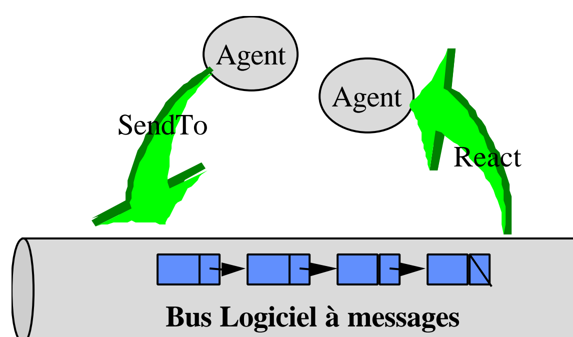


Figure 12: Modèle de base

3.2. Propriété de l'infrastructure

3.2.1. Propriété des agents

Les agents AAA sont *persistants* : La duré de vie d'un agent n'est pas liée à la durée de vie d'une exécution. L'état d'un agent n'est pas perdu lorsque la machine sur laquelle il s'exécute s'arrête ou tombe en panne. L'état est sauvegardé sur un support persistant.

La réaction des agents est *atomique* : Cette propriété assure qu'une réaction est, soit complètement exécutée soit annulée. Lorsqu'une réaction doit être exécutée, le bus démarre une transaction qui sera validée à la fin de la réaction :
Si une réaction parvient à son terme, le nouvel état de l'agent est sauvegardé et toutes les notifications envoyées durant la réaction sont effectivement envoyées. En d'autres termes, le bus n'achemine effectivement toutes les notifications envoyées durant une réaction que lorsque cette dernière est validée.

Si une faute survient au cours d'une réaction, le système effectue un retour arrière. Ce retour arrière restaure l'état précédent de l'agent et supprime les notifications envoyées durant la réaction.

3.2.2. Propriétés des communications

Au niveau du modèle de programmation, les agents communiquent via des notifications. Au niveau de l'infrastructure de communication, ces notifications sont transformées en messages. Cette infrastructure de communication est basée sur un bus logiciel (Message Oriented Middleware, MOM) qui assure des propriétés d'asynchronisme, de fiabilité et d'ordonnancement sur les communications.

- *Asynchronisme* : Cette propriété permet aux producteurs et aux consommateurs d'événement de s'exécuter de façon indépendante (fonctionnement en **mode déconnecté**). En d'autres termes, un agent A1 peut envoyer une notification n1 à un agent A2 même si ce dernier n'est pas en cours d'exécution. La propriété d'asynchronisme fournie par le bus associé à la persistance des messages assure que l'agent A2 recevra bien la notification n1 lorsqu'il sera en cours d'exécution

- *Fiabilité* : Cette propriété assure que lorsqu'une notification est envoyée, son acheminement est garanti malgré l'occurrence de panne (machine ou réseau). Ce type de fiabilité doit bien être différencié de celle fournie par TCP (TCP traite seulement le cas de perte de message, de doublons et de dé-ordonnancement. En aucun cas, TCP ne traite le cas de panne machine).

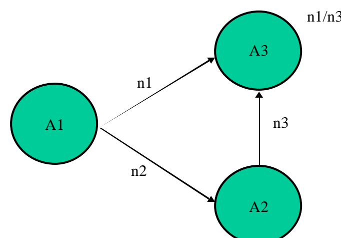


Figure 13: Propriété de causalité

➢ *Ordonnancement causal* : la Figure 13 illustre la propriété de causalité.

Dans cet exemple, l'agent A1 envoie une notification n1 à l'agent A3, puis une notification n2 à A2. Sur réception de la notification n2, l'agent A2, envoie une notification n3 à l'agent A3. La propriété de causalité assure que la notification n1 sera délivrée à A3 avant la notification n3.

3.2.3. Modèle de programmation

Tous les agents sont des objets Java qui doivent hériter de la classe de base *Agent*. La classe *Agent* fournit le comportement de base de tous les agents A3 (voir en annexe une liste non exhaustive des méthodes fournie par Agent).

3.2.3.1. Identification des agents

Tous les agents sont identifiés de manière unique à l'aide d'un *AgentId*. Ce dernier peut être obtenu en appelant la méthode getId() définie dans *Agent*.

3.2.3.2. Définition des notifications

Une notification est mise en œuvre sous la forme d'une classe java qui doit hériter de la classe *Notification*. Une notification va contenir des attributs représentant une information compréhensible par l'agent destinataire. Toutes les notifications doivent être sérialisables. Ceci vient du fait que les notifications sont acheminées sur le réseau sous forme sérialisée.

L'exemple suivant montre la programmation d'une classe de notification qui contient une chaîne de caractère :

```
package work.is.tp1.v1;
import aaa.agent.*;

public class HelloWorldNot extends Notification {

  public String msg = "Hello world";

  public HelloWorldNot(String  msg) {
this.msg = msg;
  }
}
```

3.2.3.3. Envoi de notification

Un agent peut envoyer une notification à un autre agent via la méthode *sendTo* définie dans la classe *Agent*. La signature de la méthode sendTo est la suivante :

SendTo(AgentId ag, Notification n)
ag : id de l'agent destinataire
n : la notification qui doit être envoyée

Exemple : sendTo(ag, new HelloWorldNot("hi everybody")) envoie une notification HelloWorldNot à l'agent identifié par ag.

3.2.3.4. Réaction aux notifications

Lorsqu'un agent reçoit une notification, il doit exécuter la réaction correspondante (le code traitant la notification). En pratique la programmation de la réaction d'un agent consiste principalement dans la mise en œuvre de la méthode *react* de l'agent. Cette méthode a pour but de définir le code qui sera exécuté lors de l'arrivée d'une notification. Le bus délivre une notification en appelant la méthode *react* du destinataire. L'exemple suivant montre un exemple d'agent qui réagit à la réception d'une notification de type *HelloWorldNot* en imprimant la chaîne contenue dans cette notification à l'écran.

```
package work.is.tp1.v1;
import java.io.*;
import aaa.agent.*;

public class HelloWorld extends Agent {

  public HelloWorld(short to) {
   super(to);
  }

  public void react(AgentId from, Notification not) {
     try {
              if (n instanceof HelloWorldNot) {
         // on a reçu une HelloWorldNot
                System.out.println(((HelloWorldNot)n).msg);
              }
              else super.react(from,not);
     }
     catch (Exception e){
              e.printStackTrace();
     }
  }
}
```

3.2.4. Création et déploiement des agents

Tous les agents sont créés et s'exécutent dans des serveurs d'agent. Un serveur d'agent est un hôte logiciel pouvant héberger des agents et les faire communiquer. Chaque serveur d'agent est identifié par un numéro. Dans la version de la plate-forme fournie actuellement, tous les serveurs d'agents sont interconnectés entre eux de façon statique. En d'autres termes, l'ensemble des serveurs d'agents participant à une application est connu

avant son lancement. Tous les serveurs d'agents sont définis statiquement dans un fichier de configuration (l'annexe contient un exemple de fichier de configuration).

La création d'un agent comporte les deux phases suivantes :
Instanciation locale de l'agent
Déploiement de l'agent sur le serveur destinataire.

Dans l'exemple suivant, la classe *Launch* va lancer le serveur d'agentés numéro 1, va instancier un agent *helloworld* et le déployer sur le serveur d'agents 0. Lors de l'instanciation de *helloworld*, on spécifie le site sur lequel doit être déployé l'agent (ici 0).

```
package work.is.tp1.v1;
import java.io.*;
import aaa.agent.*;

public class Launch {
  public static void main (String args[]) {
    try {
      // lancement du serveur  0
      // arg[0] : num du serveur
            // arg[1] : répertoire représentant la racine de persistance
      Server.init(args);
      // instanciation locale de helloworld
      HelloWorld ag0 = new HelloWorld(0);  // création de l'agent ag0
      // déploiement de ag0
      ag0.deploy();
      /* ici on peut envoyer une notif de démarrage à ag0*/
      Server.start(); // démarrage du serveur
      System.out.println("server started");
    } catch (Exception exc) {
      exc.printStackTrace(System.out);
```

3.3. L'infrastructure d'exécution AAA

3.3.1. Une vue d'ensemble

L'infrastructure AAA est basée sur un bus à messages qui a pour rôle d'acheminer les notifications et de provoquer la réaction de l'agent destinataire. Ce bus à messages est mis en œuvre de façon distribuée. Cela implique qu'il existe sur chaque site une entité représentant le bus. Cette entité, appelée *bus local*, est présente dans tous les serveurs d'agents.

Lors de l'émission d'une notification, le bus local encapsule la notification dans un message et l'achemine de manière fiable grâce à un système de queues persistantes. A l'arrivée, le message est stocké dans une queue de notification entrante. Le bus local du serveur d'agents destinataire prend le premier message de la queue et appelle la réaction de l'agent destinataire. La Figure 14 illustre l'envoi d'une notification de l'agent A1 à A3.

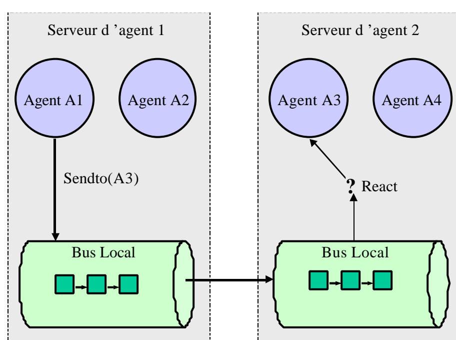


Figure 14: L'infrastructure AAA

3.3.2. Les serveurs d'agents

Un serveur d'agents est une machine virtuelle servant d'hôte aux agents. Cette machine virtuelle a en charge la création des agents, leur exécution et leur communication. Pour cela, chaque serveur d'agents comporte un bus local et une fabrique d'agents. En pratique, un serveur d'agents est inclus dans un processus.

Le bus local

Le bus local a la charge de transmettre les notifications et d'exécuter la réaction de l'agent destinataire. C'est le bus local qui assure les propriétés d'atomicité et de fiabilité des agents. Le bus local gère lui-même les communications locales (communication au sein d'un même serveur d'agent). Cependant lorsque des notifications sont envoyées à des agents résidents sur un serveur d'agent distant, le bus local transmet l'événement au bus du serveur d'agent destinataire comme montré à la Figure 14. Tous les bus locaux sont fortement interconnectés.

Tous les bus locaux sont composés d'un *channel* et d'un *engine*. Le *channel* est responsable de la localisation des notifications et de leur transport. *L'engine* représente le moteur d'exécution du serveur d'agent, il a pour rôle de délivrer les notifications en exécutant la méthode *react* des agents destinataires. En pratique, deux *threads* sont associés respectivement au channel et à l'engine. Cela implique qu'une seule réaction à la fois s'exécute dans un même serveur d'agent.

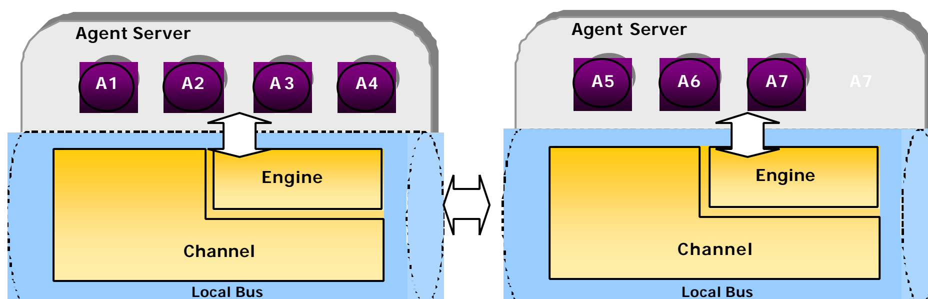


Figure 15: Détail d'un serveur d'agents

Le channel

Le composant *channel* assure la transmission des notifications de manière fiable, asynchrone et maintient la propriété d'ordre causal. Pour cela, le channel utilise un système de queue de messages et estampille tous les messages à l'aide d'une horloge logique. Toutes les notifications envoyées au cours d'une réaction sont stockées dans une queue de message transitoire non persistante appelé *tmp-out*. Lorsque la réaction est validée, toutes les notifications contenues dans *tmp-out* sont transférées dans une queue de messages persistante appelée *qout*. Dès lors, toute notification envoyée durant la réaction est assurée d'être délivrée. Le channel n'est qu'une boucle qui récupère les messages de *qout* et les transmet un à un au channel destinataire au sein d'une transaction. Lors de l'arrivée d'un message, le channel distant démarre une transaction pour stoker le message de façon persistante dans une queue de message entrant appelée *qin*. La Figure 16 montre le système de queues de messages utilisées par le channel.

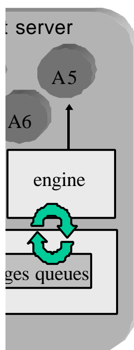
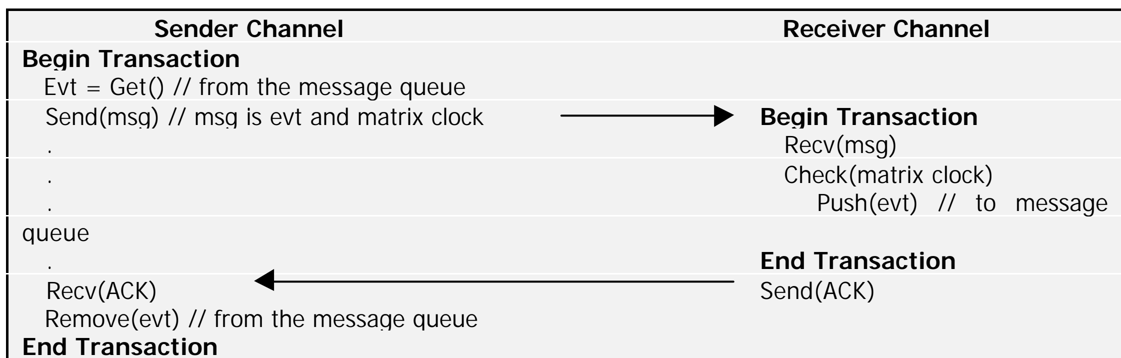


Figure 16: Les queues de message du channel

Un message n'est enlevé de *qout* que lorsqu'il stocké dans *qin* au sein du channel destinataire. La fiabilité des communications et l'asynchronisme sont assurés par ce système de queues persistantes et les transactions associées. Lorsqu'un message ne peut être envoyé à son destinataire, une faute est suspectée, le channel réessaye périodiquement d'envoyer le message. L'algorithme mis en œuvre par le channel est le suivant :



Sender Channel		Receiver Channel
Begin Transaction		
Evt = Get() // from the message queue		
Send(msg) // msg is evt and matrix clock	⟶	**Begin Transaction**
.		Recv(msg)
.		Check(matrix clock)
.		Push(evt) // to message
queue		
.		**End Transaction**
Recv(ACK)	⟵	Send(ACK)
Remove(evt) // from the message queue		
End Transaction		

Le moteur d'exécution

Comme nous l'avons vu, l'*engine* représente le moteur d'exécution du serveur d'agent. Il gère l'atomicité de la réaction et assure la sauvegarde de l'état de l'agent entre chaque réaction. *L'engine* est une boucle qui attend l'arrivée de notification dans *qin*, exécute la réaction appropriée et sauvegarde l'état de l'agent. Tous ces traitements sont effectués au sein d'une transaction. Toutes les notifications émises durant une réaction sont stockées dans la queue *tmp-qout*. Lorsque la réaction est validée, *l'engine* transfert le contenu de *tmp-qout* dans la queue persistante *qout* pour être traité par le channel. L'algorithme de *l'engine* est le suivant :

```
Class Engine {
    While (true) {
          (BEGIN TRANSACTION)
          evt = Channel.get();// get next message in channel
          ag = Agent.load(evt.dest); // get the agent to process event
          ag.react(evt);
          // execute reaction to event, all agent sent during this reaction is inserted into persistant
queue in order to processed by the channel.
          ag.save(); // save changes, then commit.
          (END TRANSACTION)
    }
}
```

3.4. Synthèse

On a donc pu voir que le bus à messages AAA est un MOM qui utilise un modèle événementiel, ceci est réalisé selon le modèle E-C-A. les événements sont représentés par des notifications et les actions sont des réactions à ces notifications.
L'architecture de AAA est une topologie en Bus qui permet d'obtenir tous les avantages du Bus, à savoir, la répartition de la charge CPU et réseau et l'abstraction des problèmes de *message matching* (voir Partie 1 :2.3.1). AAA possède aussi des propriétés de fiabilité (persistance des messages et des agents) et d'atomicité des réactions. De plus le Bus dispose d'une propriété d'ordonnancement causal implémenté grâce à une horloge matricielle elle-même persistante.
Toutes ces caractéristiques chargent le Bus et posent des problèmes lors du passage à l'échelle.

Chapitre 4 : Les problèmes du passage à l'échelle du bus à messages AAA

4.1. Introduction

Comme nous l'avons vu dans la Partie précédente, le passage à l'échelle d'un bus à messages pose de nombreux problèmes. L'ajout de nombreux clients et le déploiement sur des zones très étendues du gestionnaire de message engendre des problèmes de fiabilité, d'efficacité mais surtout de performances. Cette baisse de performances peut être due à de nombreux facteurs comme la baisse des performances réseaux (à cause de l'élargissement du MOM sur des WAN), la mauvaise architecture du MOM ou encore la gestion de l'ordonnancement.
Dans le MOM AAA les messages sont ordonnancés causalement, et pour assurer le respect de cet ordre causal, les serveurs d'agents utilisent une horloge logique matricielle. Nous avons pu identifier que la gestion de la causalité entraînait une baisse des performances et empêchait le passage à l'échelle du bus. Et cela pour une raison principale : la taille de la matrice. La taille de la matrice augmente de façon quadratique avec le nombre de serveurs, cela n'est pas gênant tant que le nombre de serveurs reste petit (de l'ordre de la dizaine) mais lorsqu'on déploie le Bus à Agent sur des centaines ou des milliers de machines le coût de stockage de la matrice en mémoire et surtout sur disque (pour la persistance) devient trop important et ralentit les performances globales du MOM.
Ce chapitre décrit la façon dont est gérée la causalité dans le MOM AAA et analyse une série de mesures qui permettent d'appréhender le coût réel de l'horloge matricielle et le besoin d'optimisation pour permettre le passage à l'échelle.

4.2. L'algorithme de causalité

4.2.1. Protocole de base

Soit n le nombre de sites[14], sur chaque site S_i on définit l'horloge matricielle comme une matrice $H_i[1..n, 1..n]$ initialisée à 0. Soit une composante quelconque de H_i : $H_i[j,k]$ est la vision de S_i de l'état du canal de communication de S_j vers S_k (c'est à dire le nombre de messages envoyés sur le canal du site j au site k). $H_i[i,*]$ et $H_i[*,i]$ sont évidement à jour sur S_i, les autres composantes représentent la connaissance par S_i de l'état global.
Lors d'un envoi de message *m* par S_i, de S_i vers S_j, H_i définit l'ensemble des envois de messages dont le message *m* dépend causalement. Le protocole d'envoi de message fonctionne comme suit :

A. Sur le site S_i, lorsqu'un message est envoyé sur le canal de communication $C_{i \to j}$ (canal de S_i à S_j) on exécute une incrémentation de l'horloge :
| $H_i[i,j] := H_i[i,j] + 1$

B. Chaque message M porte comme estampille H_M l'horloge matricielle H_i du site émetteur après la mise à jour. A la réception d'un message (M, H_M) envoyé par le

[14] c'est à dire le nombre de serveurs d'agent

site S_i, le site récepteur S_j retarde la délivrance de M jusqu'à ce que la condition suivante soit satisfaite :

$\mid \quad k \in [1..n], k \neq i \; H_M[k, j] \leq H_j[k, j]$ **et** $H_M[i, j] \equiv H_j[i, j] + 1$

La première condition ($\forall$ k $\in$ [1..n], k$\neq$i $H_M[k, j] \leq H_j[k, j]$) permet de vérifier qu'aucun site ne nous a envoyé un message qu'on aurait dû recevoir avant celui-ci (au sens causal) (voir Figure 17).

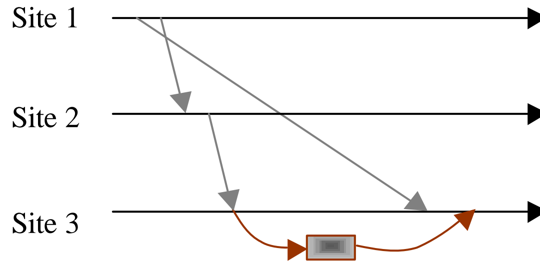


Figure 17: Si un message est en avance (au sens causal),

il est stocké puis délivré lorsque la condition est vérifiée.

La deuxième condition de délivrance ($H_M[i, j] \equiv H_j[i, j] + 1$) nous permet de vérifier la propriété de canaux FIFO (voir Figure 18).

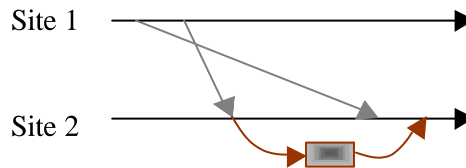


Figure 18: Si un message est en avance (au sens causal),

il est stocké puis délivré lorsque la condition est vérifiée.

C. Après délivrance, l'horloge matricielle est mise à jour afin d'actualiser notre connaissance des autres canaux en se servant de la connaissance du site émetteur (valeur de H_M) :

$\mid$ Pour k=1..n et l=1..n $H_j[k, l] := Max(Hj[k, l], H_M[k, l])$

4.2.2. Première optimisation

Un des inconvénients de ce protocole est la taille des estampilles des messages[15]. Une première optimisation du protocole a donc été réalisée afin de réduire la taille des estampilles de messages. L'idée est donc de réduire cette information en ne transmettant que les modifications de cette horloge, en contre-partie chaque site doit conserver la matrice de chaque autre site (n-1 matrices) pour savoir quelles modifications il est nécessaire d'envoyer. A partir de là deux nouveaux problèmes se posent qui sont la taille de

[15] (n $\times$ n-1) puisque la diagonale de la matrice est inutile pour la mise en œuvre du protocole.

l'information conservée sur chaque site et le coût du calcul des "diff" sur les matrices[16]. La solution pour éviter ces problèmes est de mémoriser pour chaque valeur de la matrice le dernier état dans lequel elle a été modifiée et de conserver pour chaque canal d'émission la dernière valeur de l'état lors de la dernière émission.
Cette optimisation à base de compteurs d'état change l'algorithme de gestion de la matrice, lors de l'émission on envoie la valeur de la matrice SI son état est plus récent que lors du dernier envoi, puis on incrémente le compteur d'état. A la réception on met à jour les éléments reçus ainsi que leur état courant.

4.3. L'impact de la causalité

4.3.1. Procédure de test

Nous avons réalisé une série de mesures sur le bus à agent afin d'analyser l'impact de la causalité. Le protocole de test que nous avons utilisé consiste à calculer le temps d'aller-retour d'une notification en fonction du nombre de serveur d'agents. L'envoi et la réception d'une notification prennent en compte de nombreux facteurs :

- La sérialisation/désérialisation du message.
- Le temps de transfert sur le réseau (en cas d'envoi distant).
- La sauvegarde de l'agent (pour la persistance)
- Et surtout le traitement de la **causalité** :
  - Test d'envoi et de réception (mise à jour de la matrice).
  - Sauvegarde de la matrice sur disque (persistance).

Le coût de la sérialisation est constant car nous utilisons toujours les même notifications (même taille mémoire), idem pour le coût de sauvegarde des agents sur disque lors de la transaction. Le temps de transfert sur le réseau est considéré comme constant ici car les tests ont été effectués sur cinq machines en réseau local fermé, et les perturbations du réseau sont donc minimes. Nous pouvons donc considérer que la variation du temps d'envoi et de réception de notifications se base principalement sur la variation du traitement de l'horloge matricielle (coût de la sauvegarde sur disque essentiellement).

Pour les tests nous avons déployé sur chaque serveur un agent réactif à nos notifications qui les renvoie dès qu'il les reçoit (ping-pong). Chacun des envois est lancé par un agent principal de test déployé sur le serveur 0 et qui se charge de calculer le temps d'aller-retour des notifications, il réalise pour chaque test une série de cent envois et affiche une moyenne des valeurs obtenues.
Dans chacun de nos tests nous avons abordé trois aspects du coût de la matrice. Premièrement le temps d'aller-retour pour l'envoi d'un agent local à un agent local ; ce test permet d'éviter le coût de la sérialisation et d'envoi sur le réseau. Deuxièmement le temps d'aller-retour pour l'envoi à un agent résidant sur un serveur distant (le serveur ayant le plus grand numéro), et troisièmement le temps pour d'aller-retour de notification pour une diffusion sur l'ensemble des serveurs afin de générer un trafic réseau fort et surtout un goulot d'étranglement sur le serveur principal.

Nous avons réalisé dans un premier temps des tests en exécutant un certain nombre de serveur d'agent sur la même machine physique (tests en environnement centralisé). Puis nous avons réparti les serveurs sur cinq machines afin d'augmenter de nombre total de

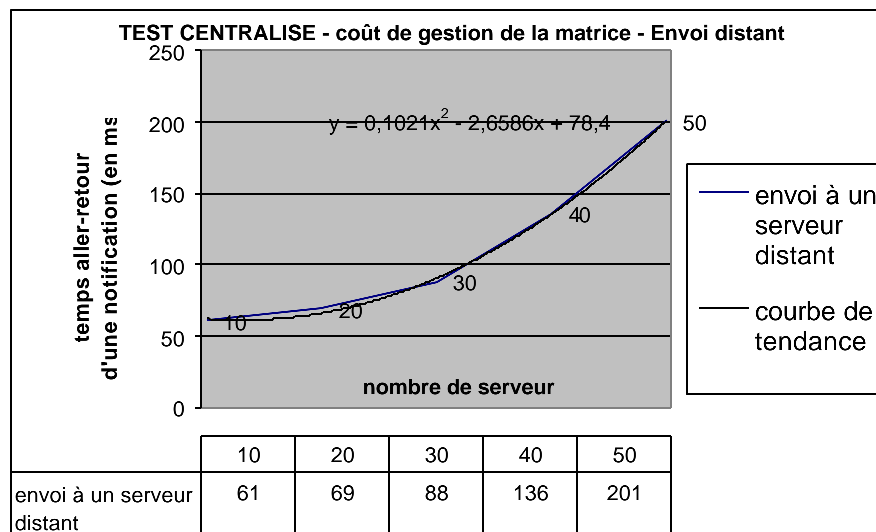
[16] Le "diff" consiste à faire la différence entre deux matrices (celle du site et celle de l'estampille).

serveurs (tests en environnement réparti), les serveurs ont été répartis au prorata du nombre de machine[17]. Les machines de tests sont des PC Bi-Pentium II avec 512Mo de RAM et 9Go de capacité disque dur, toutes sont reliées par des cartes réseau Ethernet 100Mbits, le système d'exploitation utilisé est LINUX noyau 2.0 et 2.2 suivant les machines.
L'ensemble des résultats des tests est disponible dans les Annexes à la fin de ce rapport.

4.3.2. Analyse des résultats

Remarque générale sur l'ensemble des tests :
Nous aurions souhaité faire des tests de plus grande envergure afin de déployer plusieurs centaines de serveur mais les conditions de tests ne nous permettaient pas de dépasser les 150 serveurs répartis sur 5 machines. Néanmoins ces tests nous permettent d'avoir une juste vision des coûts qui sont engendrés lors du passage à l'échelle.

Lorsqu'on regarde les mesures de l'Annexe 1 :(page 73 et page 74) dans leur ensemble on s'aperçoit immédiatement que les courbes[18] de tendances ont une croissance en n^2. Avant d'effectuer les tests nous pensions que les courbes auraient une croissance plus rapide car l'augmentation en n^2 de la taille de la matrice engendre en coût encore plus grand pour la sauvegarde sur disque et pour les envois d'estampilles. En réalité, le coût observé est bien celui provoqué par la taille de la matrice.
On remarque ceci immédiatement sur le test centralisé d'envoi à distance (Figure 19).

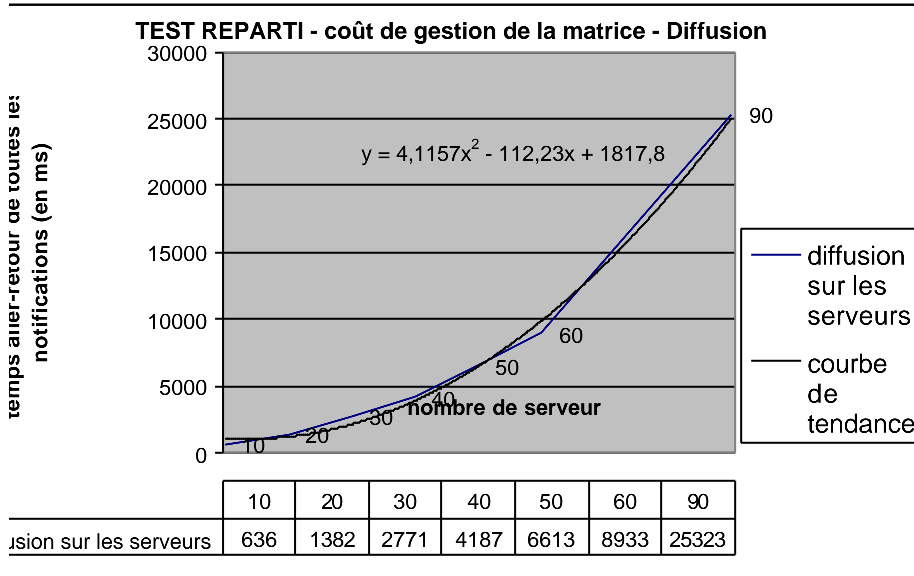


	10	20	30	40	50
envoi à un serveur distant	61	69	88	136	201

Figure 19: Résultat de mesure d'envoi distant de notifications

Pour les tests de diffusion nous n'avons pas pu dépasser les 90 serveurs d'agents, car les accès disques devenaient trop coûteux et faisait augmenter les temps de façon anormal, néanmoins on peut constater que le coût associé à la matrice augmente en n^2, ceci est particulièrement vrai pour les tests en environnement réparti (voir Figure 20).

[17] 10 serveurs sur 5 machines ➔ 2 serveurs par machines, 20 serveurs sur 5 machines ➔ 4 serveurs par machines…etc.
[18] Les équations exactes sont disponibles sur les graphes.



	10	20	30	40	50	60	90
usion sur les serveurs	636	1382	2771	4187	6613	8933	25323

Figure 20: Résultat de mesure de la diffusion de notifications

L'ensemble des ces courbes nous donne un bon aperçu du coût engendré par l'ordonnancement des messages, l'augmentation quadratique de la matrice apporte une augmentation similaire des coûts associés.

Partie 3 : Proposition

Chapitre 5 : Domaines de causalité

5.1. Concepts

La gestion de l'ordonnancement et par conséquent de la causalité est très coûteuse pour les MOM et en particulier pour le MOM AAA, la croissance en n² des coûts lors du passage à l'échelle entraîne une baisse de performance qui n'est pas acceptable. Il est donc nécessaire de trouver des moyens de limiter ces effets.
Il existe plusieurs solutions pour restreindre ces coûts, chacun des ces moyens porte sur la gestion de *domaines de causalité*. Un domaine de causalité est un ensemble d'entités dans lequel on conserve la propriété d'ordonnancement, chacun de ces domaines étant relié aux autres. L'avantage de cette solution est d'alléger dans chacun de ses domaines le coût lié à la causalité (donc à l'ordonnancement). Néanmoins il est nécessaire de vérifier que si la causalité est respectée dans chacun des domaines alors elle est vérifiée sur l'ensemble de ces domaines interconnectés. Intuitivement il semble naturel de penser que le respect de la causalité dans chaque domaine respecte la causalité de façon globale. Il est donc nécessaire de faire la preuve de ce respect de la *causalité par transitivité*. Cette preuve est réalisée dans la section suivante, elle nous démontre que cette propriété est vraie à la condition qu'il **n'existe pas de cycle** dans l'interconnexion des domaines.

5.2. Preuve de la causalité par transitivité

5.2.1. Définitions

Soit $\mathbf{P}=\{P_1,\ldots,P_U\}$ un ensemble de processus, $\mathbf{D}=\{D_1,\ldots,D_U\}$ un ensemble de domaines, et $\mathbf{R}$ un sous-ensemble quelconque de $\mathbf{P} \times \mathbf{D}$. La relation $\in$ définie par $p \in d \Leftrightarrow (p,d) \in \mathbf{R}$ décrit la répartition des processus dans les différents domaines.

5.2.1.1. Traces

Définition 1

Une trace d'exécution est définie par la donnée de :
Un ensemble $\mathbf{M} = \{m_1,\ldots,m_n\}$ de messages.
Deux fonctions *src* et *dst* de $\mathbf{M}$ dans $\mathbf{P}$, telles que $\forall m \in \mathbf{M}$, $\mathrm{src}(m) \neq \mathrm{dst}(m)$.
Pour chaque processus p, un ordre total strict $<_p$ défini par le sous-ensemble des messages émis ou reçus par p (défini par $\mathbf{M}_p = \{m \in \mathbf{M} \mid \mathrm{src}(m) = p \vee \mathrm{dst}(m) = p\}$).

src(m) désigne naturellement le processus émetteur de m, et dst(m) le processus récepteur de m. $m <_p m'$ signifie que l'émission (ou la réception) de m par p a lieu avant l'émission (ou la réception) de m' par p. Cette définition exclue volontairement la diffusion de messages, l'envoi simultané de plusieurs messages, la réception simultanée de plusieurs messages... Une trace peut être représentée par un diagramme temporel comme celui de la Figure 21.

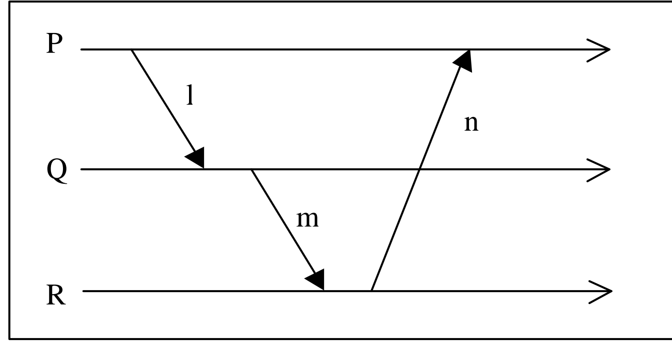


Figure 21: Représentation d'une trace

Définition 2

On dit qu'une trace est valide si elle vérifie : $\forall m \in \mathbf{M}, \exists d \in \mathbf{D}, \mathrm{src}(m) \in d \wedge \mathrm{dst}(m) \in d$.

Définition 3

La restriction d'une trace à un domaine d est la trace définie par :
L'ensemble des messages dont la source et la destination appartiennent à d: $\mathbf{M}_d = \{m \in \mathbf{M} \mid \mathrm{src}(m) \in d\ \mathrm{dst}(m) \in d\}$.
Les fonctions src_d et dst_d sont obtenues par restrictions des fonctions src et dst à $\mathbf{M}_d$.
Pour chaque processus p, l'ordre total $<_{d,p}$ est obtenu par restriction de l'ordre total $<_p$ à $\mathbf{M}_d$.

5.2.1.2. Causalité

Définition 4

On dit que m dépend causalement de n, et on note $n \prec m$, si l'une des conditions suivantes est vérifiée :
m et n sont émis par un même processus p, et m est émis après n: $\exists p \in \mathbf{P}, \mathrm{src}(n)=p \wedge \mathrm{src}(m)=p \wedge n <_p m$.
m est émis par un processus ayant précédemment reçu n: $\exists p \in \mathbf{P}, \mathrm{dest}(n) = p \wedge \mathrm{src}(m) = p \wedge n <_p m$.
il existe un message l tel que $n \prec l$ et $l \prec m$.

Définition 5

On dit qu'une trace est correcte si elle est valide et si la relation $\prec$ définit une relation d'ordre partiel sur les messages.

Cette définition permet d'exclure les traces qui ne peuvent pas se produire dans la réalité, comme dans la Figure 22, où un message "remonte dans le temps".

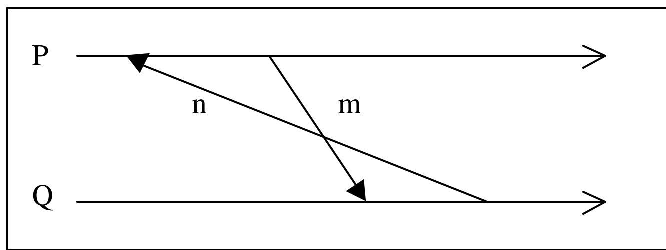


Figure 22: Une trace incorrecte

Définition 6

Une trace correcte respecte la causalité si, pour chaque processus *p*, l'ordre de réception des messages est compatible avec l'ordre causal : $\forall p, m, n \in \mathbf{P} \times \mathbf{M} \times \mathbf{M}$, $(\mathrm{dst}(m) = p \wedge \mathrm{dst}(n) = p \wedge n \prec m) \Rightarrow n <_p m$.

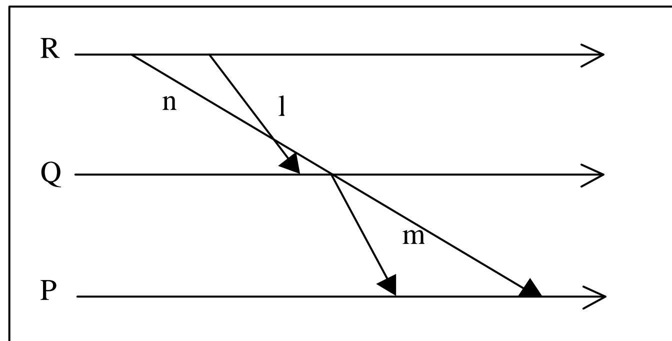


Figure 23: Violation de la causalité

Définition 7

On dit qu'une trace correcte respecte la causalité dans le domaine *d* si la restriction de cette trace au domaine *d* respecte la causalité.

Par exemple, la trace de la Figure 24 respecte la causalité dans D_1 mais pas dans D_2.

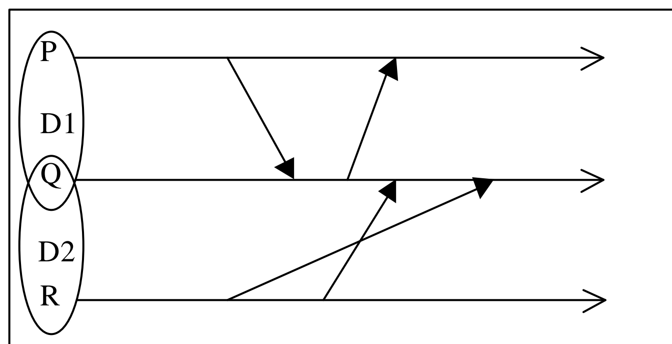


Figure 24: Respect de la causalité dans un domaine

5.2.1.3. Chemins

Définition 8

Un chemin de p à p_c est une suite non vide $[p_1,...,p_c]$ de processus tels qu'il existe, pour chaque couple de processus consécutifs, un domaine contenant ces deux processus : $\forall i < c$, $\exists d \in \mathbf{D}, p_i \in d \wedge p_{i+1} \in d$.

Définition 9

Soit $C = [p_1,...,p_c]$. Alors, par définition, la longueur de C est égale à c, sa source est égale à p_1, et sa destination est égale à p_c.

Définition 10

Un chemin direct de p_1 à p_c est un chemin $[p_1,...,p_c]$ tel que tous les processus soient 2 à 2 distincts : $i \neq j \Rightarrow p_i \neq p_j$.

Définition 11

Un chemin minimal de p_1 à p_c est un chemin direct $[p_1,...,p_c]$ tel que $i+1< j \Rightarrow \neg(\exists d \mid p_i \in d \wedge p_j \in d)$.

Définition 12

Un cycle est un chemin direct dans le graphe des domaines tel qu'il existe un domaine contenant la source et la destination du chemin, et tel qu'il n'existe pas de domaine contenant tous les processus du chemin.

5.2.1.4. Chaînes

Définition 13

Etant donné une trace, une chaîne est une suite non vide $(m_1,\dots,m_k)$ de messages de cette trace, telle que chaque message est émis après réception du précédent :$\forall i<k, \exists p \in \mathbf{P}$, $\mathrm{dst}(m_i) = p \wedge \mathrm{src}(m_{i+1}) = p \wedge m_i <_p m_{i+1}$.

Définition 14

Soit $C = (m_1,\dots,m_k)$. Alors, par définition, la longueur de C est égale à k, sa source est égale à $\mathrm{src}(m_1)$, et sa destination est égale à $\mathrm{dst}(m_k)$.

Définition 15

On définit le chemin associé à une chaîne $(m_1,\dots,m_k)$ d'une trace valide par :
$[\mathrm{src}(m_1),\mathrm{src}(m_2),...,\mathrm{src}(m_k),\mathrm{dst}(m_k)]$

Le chemin ainsi défini est bien un chemin : deux processus consécutifs correspondent en effet à la source et à la destination d'un même message, qui appartiennent par conséquent à un même domaine (puisque la trace est supposée valide).

Définition 16

Une chaîne directe est une chaîne dont le chemin associé est direct. Une chaîne minimale est chaîne dont le chemin associé est minimal.

5.2.1.5. Traces abstraites

La répartition des processus en domaines n'est visible qu'au niveau système. Au niveau applicatif, les processus s'échangent des messages "abstraits" qui ne sont pas contraints par les domaines (un message "abstrait" peut aller d'un processus à un autre directement, même s'ils ne font pas partie d'un même domaine). Par contre, ces messages "abstraits" sont représentés, au niveau système, par des chaînes de messages "concrets". Les définitions suivantes précisent les conditions dans lesquelles une trace peut être vue comme une trace de messages abstraits correspondant à une trace concrète. Elles définissent en fait comment les messages abstraits sont implémentés par des messages concrets.

Définition 17

Une abstraction d'une trace est un ensemble de chaînes minimales $C = \{c_1,\ldots,c_k\}$ tel que $\forall(m_1,\ldots,m_k) \in C, \forall\, i < k, p = \mathrm{dst}(m_i) \Rightarrow \neg(\exists m, m_i <_p m <_p m_{i+1})$.

Définition 18

On dit qu'une trace *T'* est une trace abstraite associée à une trace *T* s'il existe une abstraction $C = \{c_1,\ldots,c_k\}$ de *T* telle que *T'* soit égale à la trace déduite[19] de *T* en considérant chaque chaîne $(m_1,\ldots,m_k)$ de *C* comme un message direct de $\mathrm{src}(m_1)$ à $\mathrm{dst}(m_k)$.

Remarques :

ces définitions signifient que les messages abstraits sont représentés par des chaînes de messages concrets "directes" (sans boucles, sans détours inutiles…) et qui ne se "croisent" pas. Elles excluent les situations du genre de celles de la Figure 25 (où l'abstraction - invalide - est constituée des 2 chaînes entourées).

toute trace peut être vue comme une trace abstraite associée à elle-même (en utilisant l'abstraction $C = \{(m_1),..,(m_k)\}$.

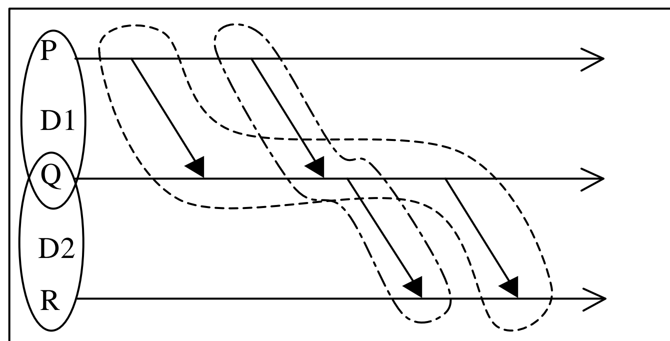


Figure 25: Abstraction invalide d'une trace

[19] la définition précise est trop laborieuse à détailler.

5.2.2. Lemmes

Lemme 1

Aucun domaine ne contient la source et la destination d'un chemin minimal de longueur supérieure ou égale à 3.

Preuve: c'est une conséquence immédiate de la condition $i+1< j \Rightarrow \neg(\exists d \mid p_i \in d \wedge p_j \in d)$ de la Définition 11, qui peut s'appliquer dès que le chemin contient au moins 3 processus.

Lemme 2

Si $(m_1,\dots,m_k)$ est une chaîne dont la source p et la destination q sont différentes, et si la trace est correcte, alors il existe une chaîne directe $\forall(n_1,\dots,n_L)$ ayant même source et même destination, et telle que $m_1 \leq_p n_1$ et $n_L \leq_p m_k$.

Preuve: Si k=1, le résultat est évident. Supposons par récurrence que le résultat est vrai pour les chaînes de longueur k < K, et considérons une chaîne $(m_1,\dots,m_K)$ de longueur K. Si cette chaîne est directe, le résultat est démontré. Si par contre elle ne l'est pas, alors, par définition, le chemin associé $(p_1,\dots,p_{K+1})$ n'est pas direct, c'est à dire qu'il existe i < j tel que $p_i = p_j$. Considérons alors la chaîne $(n_1,\dots,n_L)$ définie par[20]:

$(m_j,\dots,m_K)$ si $i = 1 \wedge j < K+1$.
$(m_1,\dots,m_{i-1})$ si $i > 1 \wedge j = K+1$.
$(m_1,\dots,m_{i-1},m_j,\dots,m_K)$ si $i >1 \wedge j<K+1$.

C'est bien une chaîne. De plus, elle a même source et même destination que la chaîne de départ. Enfin, elle est de longueur strictement inférieure à K. On peut donc, en utilisant l'hypothèse de récurrence, en déduire l'existence d'une chaîne directe $(n'_1,\dots,n'_h)$ ayant même source et même destination, et telle que $n_1 \leq_p n'_1$ et $n'_h \leq_q n_L$.
De plus, $m_1 \leq_p n_1$ et $n_L \leq_q m_k$. En effet, dans le premier cas, $m_1 <_p m_j$ (sinon la chaîne définie par $(m_1,...,m_{j-1})$, "reçue" par p avant d'être "émise" par p, violerait l'hypothèse que la trace est correcte). De même, dans le second cas, $m_{i-1} <_q m_K$. Donc, dans tous les cas, $m_1 \leq_p n_1$ et $n_L \leq_q m_k$.
On en déduit que $m_1 \leq p\, n'_1$ et $n'_h \leq q\, m_K$. Le lemme est donc démontré pour les chaînes de longueur K, et donc, par récurrence, pour toutes les chaînes.

Lemme 3

Si m dépend causalement de n, alors soit il existe une chaîne $(n,...,m)$, soit il existe une chaîne $(l,...,m)$ où l est un message émis après n par le processus ayant émis n ($l <_p n$).

La preuve de ce lemme se fait par induction sur les trois cas possibles de dépendance causale entre deux messages (voir 5.2.1.2) :
soit m et n sont émis par un même processus p, et m est émis après n. Dans ce cas, il existe une chaîne de type $(l,...,m)$ où l est un message émis après n (il suffit de prendre la chaîne (m)).

---

[20] le cas $i = 1 \wedge j = K+1$ est impossible (car $p \neq q$).

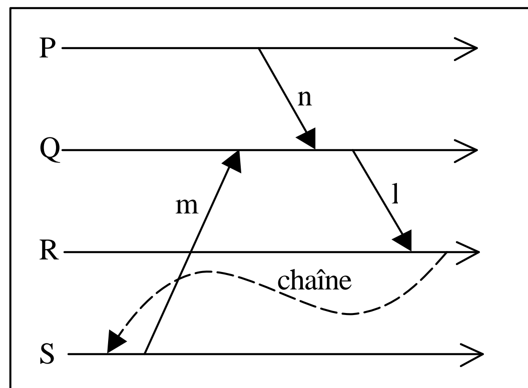
soit m est émis par un processus ayant reçu précédemment n. Il existe alors une chaîne de type $(n,...,m)$: il suffit de prendre la chaîne (n,m).
soit enfin il existe un message k tel que $n \prec k$ et $k \prec m$. En appliquant l'hypothèse d'induction, on obtient quatre cas possibles, et on trouve dans tous les cas qu'il existe soit une chaine $(n,...,m)$, soit une chaîne $(l,...,m)$ où l est un message émis après n par le processus ayant émis n.

Lemme 4

Si une trace correcte ne respecte pas la causalité, alors il existe deux processus p et q, un message n de p vers q, et une chaîne $(m_1,...,m_n)$ de p à q telle que $n <_p m_1$ et $m_n <_q n$.

Preuve: si une trace correcte ne respecte pas la causalité alors, par définition, il existe un processus q recevant un message m avant un message n, alors que m dépend causalement de n. D'après le Lemme 3, soit il existe une chaîne $(n,...,m)$, soit il existe une chaîne $(l,...,m)$ où l est un message émis après n par le processus ayant émis n.

Dans le premier cas, la destination de n étant différente de la source de m, la chaîne $(n,...,m)$ est nécessairement de la forme $(n,l,...,m)$. Mais alors, la chaîne $(l,...,m)$, "émise" par q après être "reçue" par q (puisque $m <_q n$ et $n <_q l$), viole l'hypothèse que la trace est correcte. Ce cas est donc impossible (Figure 26, trace du haut).



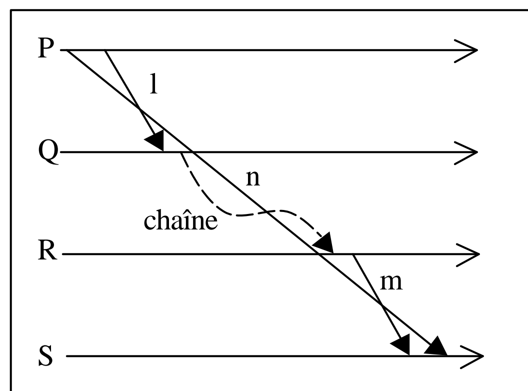


Figure 26: Les deux cas possibles du Lemme 4

Dans le second cas, on a un message n de $p = \mathrm{src}(n)$ vers q, ainsi qu'une chaîne $(l,...,m)$ de p à q telle que $n <_p l$ et $m <_p n$ (Figure 26, trace du bas). Le lemme est donc démontré.

5.2.3. Théorème

On souhaite montrer que si une trace correcte respecte la causalité dans chaque domaine, alors toute trace abstraite associée respecte la causalité de façon globale. En fait, c'est faux dans le cas général, comme le montre l'exemple de la Figure 27 (en prenant comme trace abstraite la trace elle-même).

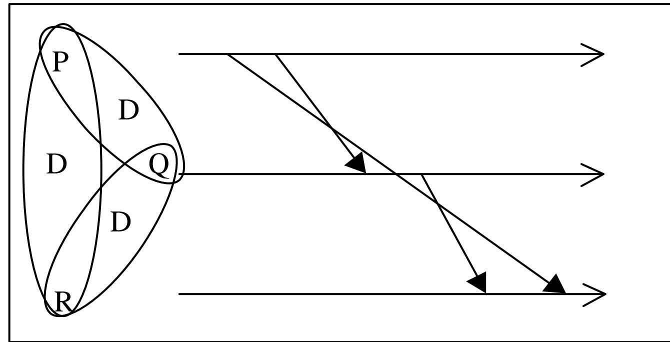


Figure 27: Un contre-exemple au théorème

5.2.3.1. Enoncé

Théorème

Les deux propositions suivantes sont équivalentes:

P1 : toute trace abstraite, associée à une trace correcte respectant la causalité dans chaque domaine, respecte la causalité de façon globale.

P2 : Le graphe d'interconnexion des domaines est sans cycles.

Un corollaire immédiat, en utilisant le fait qu'une trace peut être vue comme une trace abstraite associée à elle même, est que s'il n'existe pas de cycles alors toute trace correcte respectant la causalité dans chaque domaine respecte la causalité globale. D'autre part, la preuve de P1 $\Rightarrow$ P2 ci-dessous montre aussi l'implication inverse : si toute trace correcte respectant la causalité dans chaque domaine respecte la causalité globale, alors il n'existe pas de cycles.

5.2.3.2. Preuve

Preuve de P1 $\Rightarrow$ P2

Pour prouver que de P1 $\Rightarrow$ P2, il est équivalent de prouver la contraposée, qui s'énonce ainsi: s'il existe un cycle, alors il existe une trace abstraite, associée à une trace correcte respectant la causalité dans chaque domaine, qui ne respecte pas la causalité globale.

Puisqu'il existe un cycle, il existe, par définition, un chemin direct $[p_1,...,p_c]$ tel qu'il existe un domaine contenant p_1 et p_c, et tel qu'il n'existe pas de domaine contenant tous les p_i. Considérons alors la trace représentée par la Figure 28.

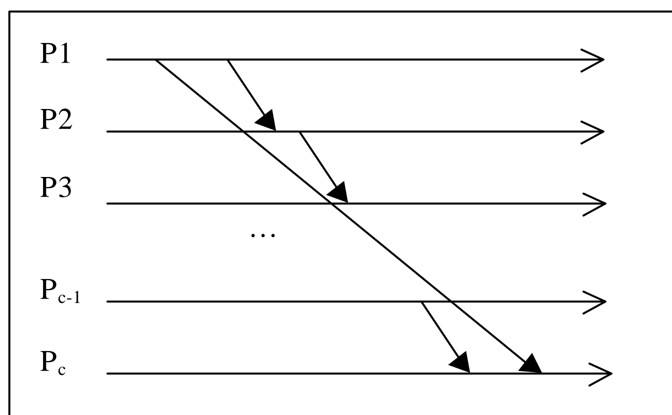


Figure 28: Trace correspondant à [p_1,...,p_c]

Cette trace est correcte (car [p1,...,pc] est un chemin, et car il existe un domaine contenant p1 et pc). Par contre, elle ne respecte pas la causalité globale. Enfin, elle respecte la causalité dans chaque domaine. En effet, aucun domaine ne contient à la fois tous p_i. Donc aucune trace restreinte à un domaine ne peut contenir tous les messages de la trace globale. Or il suffit de retirer un seul message (n'importe lequel) de la trace globale pour que la violation de la causalité disparaisse. On a donc construit une trace correcte qui respecte la causalité dans chaque domaine, ainsi qu'une trace abstraite associée (la trace elle-même) qui ne respecte pas la causalité de façon globale. Donc ¬P2 ⇒ ¬P1.

Preuve de P2 ⇒ P1

Pour montrer que P2 ⇒ P1, il suffit, d'après la contraposée du Lemme 4, de montrer que:

P: pour toute trace abstraite, associée à une trace correcte respectant la causalité dans chaque domaine, il n'existe pas deux processus *p* et q, un message abstrait *n'* de *p* vers *q*, et une chaîne de messages abstraits ($m'_1,\dots m'_n$) de *p* à *q* tels que $n' <_p m'_1$ et $m'_n <_q n'$.

Pour cela, on montre par récurrence généralisée que P(x) est vrai pour tout x, avec P(x) définie par:

P(x) : pour toute trace abstraite, associée à une trace correcte respectant la causalité dans chaque domaine, il n'existe pas deux processus *p* et *q*, un message abstrait *n'* de *p* vers *q* correspondant à une chaîne minimale de longueur x, et une chaîne abstraite ($m'_1,\dots m'_n$) de *p* à *q* tels que $n' <_p m'_1$ et $m'_n <_q n'$.

Preuve de P(1)

Par l'absurde, supposons qu'il existe, pour une trace abstraite associée à une trace correcte respectant la causalité dans chaque domaine, deux processus *p* et *q*, un message abstrait *n'* de *p* vers *q* correspondant à une chaîne (*n*) de longueur *1*, et une chaîne abstraite ($m'_1,\dots m'_n$) de *p* à *q* telle que $n' <_p m'_1$ et $m'_n <_q n'$ (trace concrète similaire à celle du bas de la Figure 26).

Soit alors $(m_1,...,m_k)$ la chaîne concrète correspondant à la chaîne $(m'_1,...,m'_n)$. D'après le Lemme 2, il existe une chaîne directe $(n_1,...,n_h)$ telle que $m <_p n_1$ et $n_h <_q m$. Cette chaîne ne peut pas être de longueur 1, sinon la causalité serait violée dans le domaine contenant p et q (qui existe puisqu'il existe un message de p vers q). Soit donc $(p,r,...,q)$ le chemin, direct par définition, associé à cette chaîne. Aucun domaine ne peut contenir tous les processus de ce chemin (sinon la causalité serait violée dans ce domaine). Par contre, il existe un domaine contenant p et q. $(p,r,...,q)$ est donc un cycle, ce qui est contradictoire avec les hypothèses. Donc P(1) est vrai.

Preuve de P(x)

Supposons que P(y) est vrai pour tout y < x, et montrons que P(x) est vrai. Par l'absurde, supposons qu'il existe, pour une trace abstraite associée à une trace correcte respectant la causalité dans chaque domaine, deux processus p et q, un message abstrait n' de p vers q correspondant à une chaîne minimale $(n_1,...,n_x)$ de longueur $x \geq 2$, et une chaîne abstraite $(m'_1,...,m'_n)$ de p à q telle que $n' <_p m'_1$ et $m'_n <_q n'$ (trace abstraite similaire à celle du bas de la Figure 26).

Soit $(p,a_1,...,a_{x-1},q)$ le chemin (minimal) associé à la chaîne $(n_1,...,n_x)$. Soit $(m_1,...,m_u)$ la chaîne concrète correspondant à la chaîne de messages abstraits $(m'_1,...,m'_n)$. D'après le Lemme 2, il existe une chaîne directe $(L_1,...,L_v)$ telle que $m_1 <_p L_1$ et $L_v <_q m_u$. Soit alors $(p,b_1,...,b_{v-1},q)$ le chemin direct associé[21].

Premier cas possible : les processus du chemin $(a_{x-1},...,a_1,p,b_1,...,b_{v-1},q)$ sont 2 à 2 distincts. Dans ce cas, ce chemin est un cycle. En effet, il existe un domaine contenant a_{x-1} et q, et il n'existe pas de domaine contenant p et q (Lemme 1 appliqué au chemin minimal $(p,a_1,...,a_{x-1},q)$), et donc, a fortiori, il n'existe pas de domaine contenant tous les processus du chemin. Or l'existence d'un cycle est contradictoire avec les hypothèses, donc ce cas est impossible.

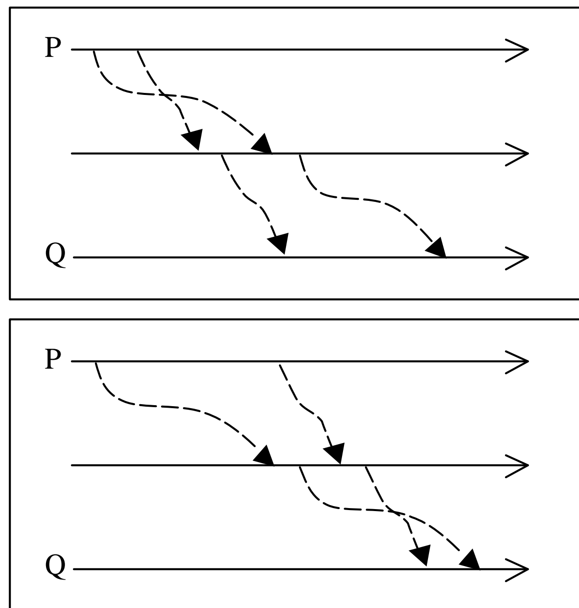


Figure 29: "Croisements" possibles de chaînes

[21] $v \geq 2$, sinon $(n_1,...,n_x)$ ne serait pas minimale.

Deuxième cas possible : les processus du chemin $(a_{x-1},...,a_1,p,b_1,...,b_{v-1},q)$ ne sont pas 2 à 2 distincts. Les a_i étant 2 à 2 distincts et distincts de p et q, et les b_i étant 2 à 2 distincts et distincts de p et q, c'est donc qu'il existe i et j tels que $a_i = b_j$. Les chaînes $(n_1,...,n_x)$ et $(L_1,...,L_v)$ ne peuvent pas se "croiser" en a_i (comme dans la Figure 25). Donc elles se croisent soit "avant", soit "après" ce processus (Figure 29). Dans le premier cas, la trace abstraite correspondant à l'abstraction $\{(n_1,...,n_i),(L_1),...,(L_i)\}$[22] ne vérifie pas P(i), i < x. Dans le second cas, la trace abstraite correspondant à l'abstraction $\{(n_{i+1},...,n_x),(L_{i+1}),...,(L_v)\}$ ne vérifie pas P(x-i), x-i < x. Donc, dans les deux cas, on peut construire une trace abstraite qui ne vérifie pas P(y) avec y < x, ce qui est contradictoire avec les hypothèses.

Dans tous les cas, on aboutit à une contradiction. Donc P(x) est vrai et, par récurrence, P(n) est vrai pour tout n. Donc P est vrai, ce qui revient à dire que P2 $\Rightarrow$ P1.

5.3. Synthèse

Nous venons de voir dans la section précédente que la causalité par transitivité au travers des domaines était respectée si et seulement si il n'existe pas de cycle dans l'interconnexion des domaines. C'est à dire que l'on doit garantir qu'il existe un chemin unique pour aller d'un domaine à un autre. Ce résultat nous permet donc d'assurer que l'ordre causal global sera respecté si on le respecte dans chaque domaine de causalité. On peut donc prédire que le coût de 10 domaines de causalité de 10 entités sera bien inférieur au coût d'un domaine de 100 entités. Cette propriété de baisse des coûts est capitale pour le passage à l'échelle, nous proposons dans le chapitre suivant différents critères de répartition des domaines de causalité.

[22] C'est bien une abstraction, car $(n_1,...,n_i)$, extraite d'une chaîne minimale, est elle aussi minimale. D'autre part, elle vérifie le critère principal des abstractions (Définition 17) puisque la chaîne $(n_1,...,n_x)$ le vérifie par hypothèse.

Chapitre 6 : Utilisation des domaines de causalité pour le passage à l'échelle d'un bus à messages

6.1. Introduction

Nous avons vu dans le chapitre précédent qu'un domaine de causalité est un ensemble d'entités dans lequel on conserve la propriété d'ordonnancement. La causalité sur l'ensemble de ses domaines interconnectés est respectée s'il n'existe pas de cycle de domaines.
Il est possible d'utiliser plusieurs critères pour créer les domaines de causalité, des critères physiques (machines, topologie réseau, architecture du MOM…etc.) ou des critères applicatifs (topologie de l'application, besoins de l'application).
Cette séparation en domaines peut donc se réaliser à deux niveaux. Au niveau de l'architecture de l'application tout d'abord, dans ce cas les entités sont représentées par les composants de l'application. La gestion de l'ordonnancement intervient donc au niveau des échanges de messages entre ces composants, c'est ce qu'on appelle l'ordonnancement par topologie applicative.
La deuxième approche se situe au niveau du système lui-même suivant l'architecture du réseau, dans ce cas les entités sont représentées par les serveurs du MOM. La gestion de l'ordonnancement intervient donc au niveau des échanges de messages entre les serveurs, c'est ce qu'on appelle l'ordonnancement par topologie réseau.
Nous nous attardons dans ce chapitre à développer ces deux solutions, nous verrons dans le chapitre suivant une implémentation de la deuxième solution sur le MOM AAA.

6.2. Domaines de causalité par topologie applicative

L'idée de domaines de causalité par topologie applicative consiste à dire que certaines applications peuvent ou non avoir besoin d'un ordonnancement fort mais que c'est au MOM de se soucier de la gestion la plus efficace possible de l'ordonnancement suivant les applications.
Par exemple, dans un MOM à communication par abonnement, si un consommateur décide de s'abonner pour l'achat en ligne d'une série d'objet à un certain prix mais n'est intéressé que par un objet. Le consommateur peut être sensible à l'ordre des deux dernières mise à jour du prix d'un même objet afin de connaître la tendance. Mais celui-ci peut ne pas être sensible à l'ordre des mises à jour plus anciennes de cet objet ou de l'ordre de mise à jour des autres objets.

Un autre exemple qui touche directement l'horloge logique qui gère la causalité : l'application NetWall, un Pare-Feu développé par Bull et qui se sert du MOM AAA pour gérer ses fichiers de log. Cette application a une topologie applicative très singulière (voir Figure 30), tous les pare-feu sont reliés à un poste d'administration central unique. Chacun utilisant un serveur d'agent pour communiquer.

Les seules informations qui transitent se font entre le serveur central et les serveurs des pare-feu. Ce type de topologie possède des propriétés spécifiques qui agissent sur la forme de l'horloge matricielle, ainsi seules la première ligne et la première colonne sont utilisées pour gérer l'ordonnancement le reste de la matrice n'étant jamais mis à jour et reste à la valeur 0.

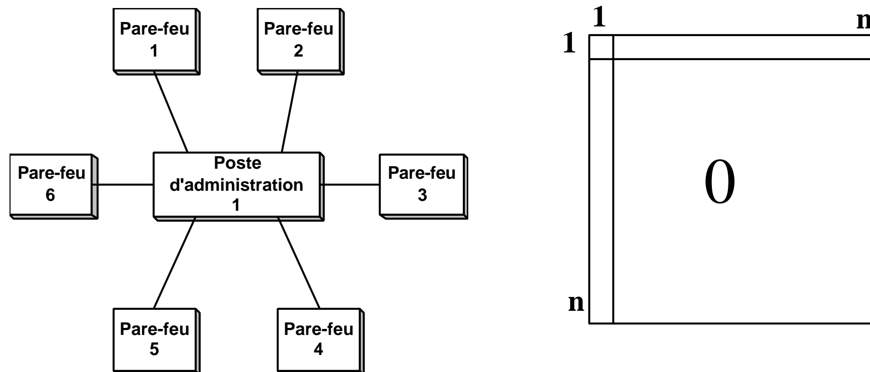


Figure 30: La topologie applicative de NetWall et son horloge matricielle associée

Avec cette propriété le coût[23] de la gestion d'une matrice complète paraît superflu. L'utilisation de langage de description d'architecture (*Architecture Description Language*, ADL) pour NetWall nous permet de savoir exactement quels sont les canaux de communication et comment ils sont utilisés (voir Figure 31).

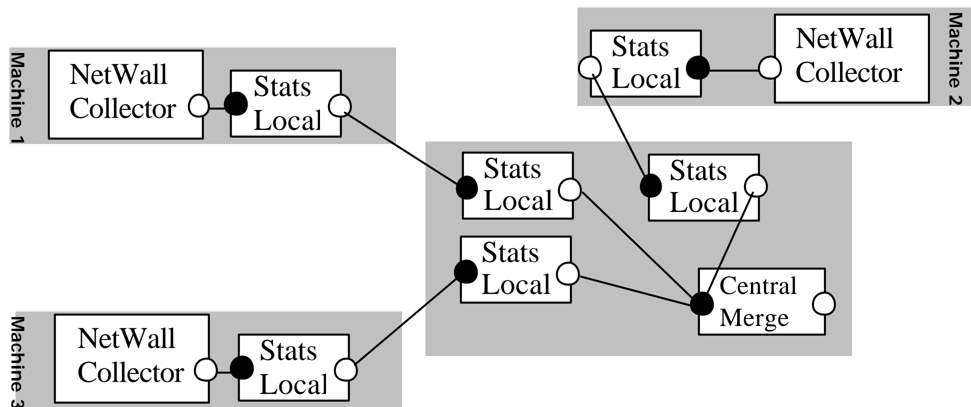


Figure 31: Représentation de l'architecture de NetWall grâce à un ADL

Grâce à cette représentation on voit immédiatement les canaux de communication, il est donc aisé de savoir (dans ce cas) que les messages transitant toujours dans la même direction, la matrice aura certaines parties dites creuses, c'est à dire des parties qui ne seront jamais utilisées. On peut donc ne conserver la matrice que sous la forme de vecteur qui sont beaucoup moins coûteux et qui permettraient d'améliorer les performances de la gestion de l'ordonnancement et donc du MOM.

[23] Place mémoire utilisée, sauvegarde sur disque de toute la matrice, test de réception sur l'ensemble de la matrice…etc.

Ces deux exemples montrent bien la nécessité de flexibilité de la politique d'ordonnancement du système afin d'alléger le besoin d'ordonnancement quand cela est nécessaire et justifié (premier exemple) et de le préserver de façon optimale grâce à des ADL pour alléger le traitement (deuxième exemple).

6.3. Domaines de causalité par topologie réseau

Le principe des domaines de causalité par topologie réseau consiste séparer le MOM en sous-domaines dans lesquels la causalité serait respectée et de relier ces domaines par des serveurs spécifiques qui sont à cheval sur les domaines (voir Figure 32).
Il est important de noter qu'on ne parle pas ici de la topologie réseau réelle du MOM mais bien de la topologie selon laquelle sera découpée la gestion de la causalité. C'est une vision logique du découpage de la causalité en sous-domaines[24]. Dans la Figure ci-dessous, le MOM est représenté par l'ensemble des 7 serveurs interconnectés mais en réalité la découpe en domaine restreins les communications entre les serveurs ainsi les groupes de serveurs pouvant communiquer entre eux sont (S1,S2,S3) dans le domaine A, (S4,S5) dans le domaine B, (S6,S7) dans le domaine C et (S3,S5,S6) dans le domaine D.

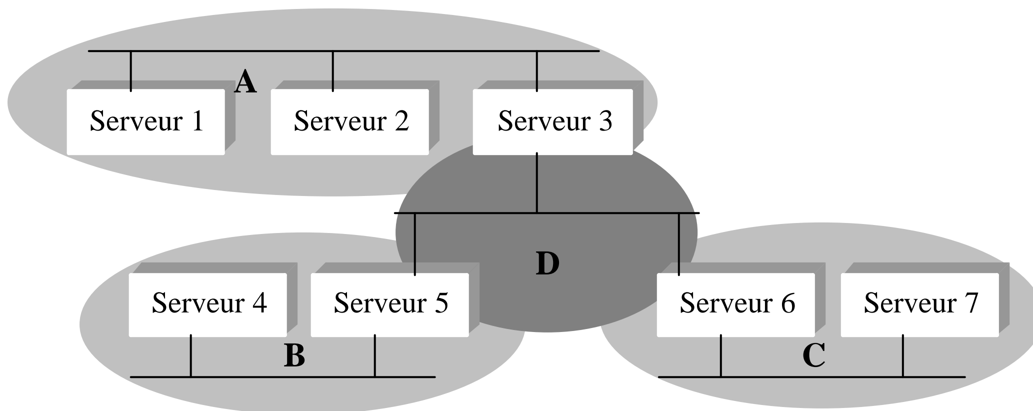


Figure 32: Domaines de causalité par topologie réseau

L'ordonnancement des messages n'est respecté **que** dans les domaines A,B,C,D, c'est à dire que la gestion de la causalité est respectée uniquement dans chaque domaine. Lorsqu'un client connecté au serveur 4 par exemple désire communiquer avec le serveur 7 il est obligé d'être router à la manière d'un paquet IP sur un réseau, c'est à dire qu'il empruntera le chemin S4→S5 puis S5→S6 puis S6→S7. Ce routage est assuré par le système et est complètement invisible aux clients qui n'ont pas la vision découpée du MOM mais une vue globale.
Le respect de la causalité uniquement dans chacun des sous-domaines permet de garantir son respect sur l'ensemble du MOM. La propriété de *causalité par transitivité* permet d'assurer que le routage d'un message au niveau système, à travers des domaines où la causalité est respectée, respecte la causalité de façon globale.

---

[24] une idée assez proche avait été rapidement abordée dans [Rug95] qui parlait de la notion de "voisins" ou l'on ne mettait à jour que les éléments de la matrice qui correspondaient à un canal voisin (du même sous-réseau).

6.4. Conclusion

Dans ce chapitre, nous avons proposé deux solutions afin de réduire le coût de l'ordonnancement des messages basées sur le principe de domaines de causalité. La première, les domaines de causalité par topologie applicative consiste à utiliser la description de l'architecture de l'application afin d'optimiser l'utilisation de l'ordonnancement. L'application étant décrite à l'aide d'un ADL, il devient possible d'utiliser cette description afin d'en dégager la topologie de l'application et d'optimiser la gestion de l'ordonnancement à cette topologie.
La deuxième, les domaines de causalité par topologie réseau consiste à découper la gestion de la causalité dans des domaines et à relier ces domaines grâce à des serveurs-routeurs. Ce découpage permet d'alléger le coût de la causalité dans chaque domaine et garantit la conservation de la causalité de façon globale sur l'ensemble des domaines.

Il existe une troisième solution qui permet de limiter le coût de l'ordonnancement, cette solution est basée sur l'ordonnancement applicatif. Le principe de l'ordonnancement applicatif est de délocaliser la propriété d'ordonnancement au niveau de l'application. Le système (le gestionnaire de message) fournit des primitives afin de permettre à l'application de gérer ou non l'ordonnancement de ses envois de messages. Cette méthode à l'avantage d'être très modulaire car le système laisse libre choix à telle ou telle application de mettre en place une politique d'ordonnancement. Ainsi les applications qui ont réellement besoin d'un ordonnancement fort (comme les applications de gestion d'ordre de bourse ou d'enchère en ligne) peuvent intégrer ce service afin d'assurer le fonctionnement valide de leur application. Et dans le même temps d'autres applications peuvent utiliser le MOM sans se servir de l'ordonnancement trop coûteux pour elles. Grâce à ce principe le MOM n'a plus à gérer la propriété d'ordonnancement pour tous les messages et du coup gagne en performance. Evidement l'ordonnancement applicatif montre vite ses limitations, car si toutes les applications se mettent à utiliser ce service on se retrouve dans le cas du MOM avec l'ordonnancement sur tous les messages et les problèmes de scalabilité ressurgissent.
Néanmoins cette piste doit rester ouverte car combinée avec les autres propositions elle peut devenir intéressante.

Chapitre 7 : Passage à l'échelle du bus à messages AAA : réalisation et évaluation

7.1. Implémentation des domaines de causalité

7.1.1. Réalisation actuelle

Réaliser le passage à l'échelle du Bus à Agent AAA nécessite l'optimisation de la gestion de l'ordonnancement réalisée à l'aide d'horloges logiques matricielles très coûteuses en ressources. Pour cela nous avons réalisé une implémentation de domaines de causalité par topologie réseau[25]. Nous rappelons que le principe est de découper le MOM en sous-domaines dans lesquels la causalité est respectée et de les relier par des serveurs à cheval sur plusieurs domaines (à la manière d'un routeur). Il faut s'assurer qu'il n'existe aucun cycle de domaines afin de vérifier la condition de validité de la causalité par transitivité.

Cette transformation du bus à agent devrait être réalisée de façon transparente vis à vis des Agents, c'est à dire qu'aucune modification ne devait se voir au niveau applicatif, et la désignation des destinataires au niveau applicatif devrait rester la même. Tout a donc été réalisé au niveau des serveurs d'agents. Deux problèmes se présentaient : la gestion de la ou des matrices en cas d'appartenance à plusieurs domaines de causalité et le routage des messages.
Pour le premier problème, nous avons créé sur chaque serveur une liste d'items pour chaque domaine, ainsi chaque item rassemble l'ensemble des informations sur le domaine auquel appartient le serveur ; il y a sur chaque serveur autant d'items que de domaine d'appartenance. Pour le problème de routage, nous nous sommes inspirés de ce qui se fait dans les protocoles réseau actuels ou chaque machine possède une table de routage.
Chaque serveur possède donc une liste d'information pour chaque domaine auquel il appartient et une table de routage (voir Figure 33).

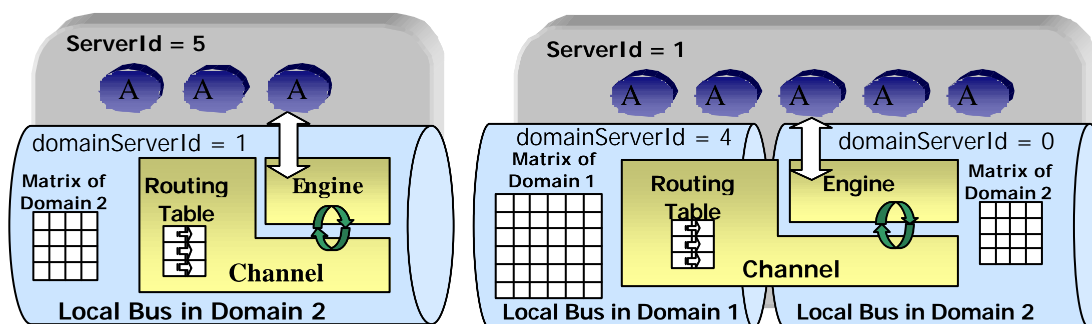


Figure 33: Deux exemples de la structure des serveurs d'agents

[25] Dans la suite nous utiliserons le terme "domaines de causalité" pour exprimer les domaines de causalité par topologie réseau avec gestion de la *causalité par transitivité*.

Il était nécessaire de pouvoir distinguer l'identifiant d'un serveur sur l'ensemble du MOM et l'identifiant du même serveur dans un domaine. L'identifiant global a été conservé afin de cacher l'implémentation des domaines de causalité aux agents et l'identifiant du domaine a servit à la gestion de l'horloge matricielle.
Les informations d'un domaine sont représentées par un objet *DomainItem* qui contient l'ensemble des informations d'un domaine, c'est à dire l'identifiant de ce domaine, l'identifiant du serveur dans ce domaine, une table de tous les serveurs du domaine et leur identifiant respectif dans le domaine, la matrice associé à ce domaine, et un pointeur vers le *DomainItem* suivant (c'est à dire un autre domaine d'appartenance du serveur) (voir Figure 34).

```
Class DomainItem {
    short domainId;
            // l'identifiant du domaine
    short domainServerId;
            // l'identifiant du serveur dans ce domaine.
    short[] idTable;
            // la table des identifiants de chaque serveur du domaine.
            // correspondance entre un ServerId et un domainServerId
    MatrixClock mclock;
            // l'horloge matricielle du domaine
    DomaintItem next;
            // un pointeur vers un autre domaine d'appartenance du serveur
}
```

Figure 34: La Class DomainItem

Grâce à ce regroupement un Serveur peut appartenir à autant de domaine que possible, il lui suffit de rajouter un *DomainItem* à sa liste. Cette liste a été conçue afin de gérer l'ajout dynamique d'un serveur à un domaine mais cette fonctionnalité n'a pas encore été implémentée.

Le deuxième composant important de la gestion des domaines et la table de routage. Elle donne pour l'ensemble des serveurs du MOM l'identifiant du serveur auquel envoyer effectivement les messages. Cette table est construite de façon statique au démarrage du serveur. Pour cela l'utilisateur doit remplir un fichier en donnant la liste des domaines du MOM et les serveurs y appartenant. Ensuite à l'initialisation le serveur construit le graphe d'interconnexion des serveurs puis grâce à un algorithme "du plus court chemin" il trouve pour chacun des autres serveurs du MOM s'il doit ou non passer par un routeur et si oui il inscrit dans sa table de routage l'identifiant de ce routeur.

Les changements qu'il a fallu réaliser dans le noyau d'exécution pour l'utilisation des domaines se situent sur les deux classes qui gère l'ordonnancement à savoir : Channel et Network. Nous simplifions l'explication de l'implémentation en ne décrivant que le Channel.

Le Channel

Nous l'avons vu dans la présentation du bus, le composant *channel* assure la transmission des notifications de manière fiable, asynchrone et maintient la propriété d'ordre causal. Avec la gestion des domaines de causalité le Channel doit en plus réaliser une translation d'adressage grâce à la table de routage puis affecter au message l'estampille correspondant

au domaine dans lequel le message est envoyé. A la réception le Channel destinataire vérifie l'estampille du message dans le domaine de réception puis il vérifie le réel destinataire du message afin de savoir s'il lui est directement destiné ou s'il n'est que le routeur du message. S'il lui est directement destiné, il le transfère dans la queue de message d'entrée, sinon il le transfère dans la queue de message de sortie, puis il renvoi un acquittement à l'émetteur. La Figure 35 montre l'algorithme simplifié du Channel.

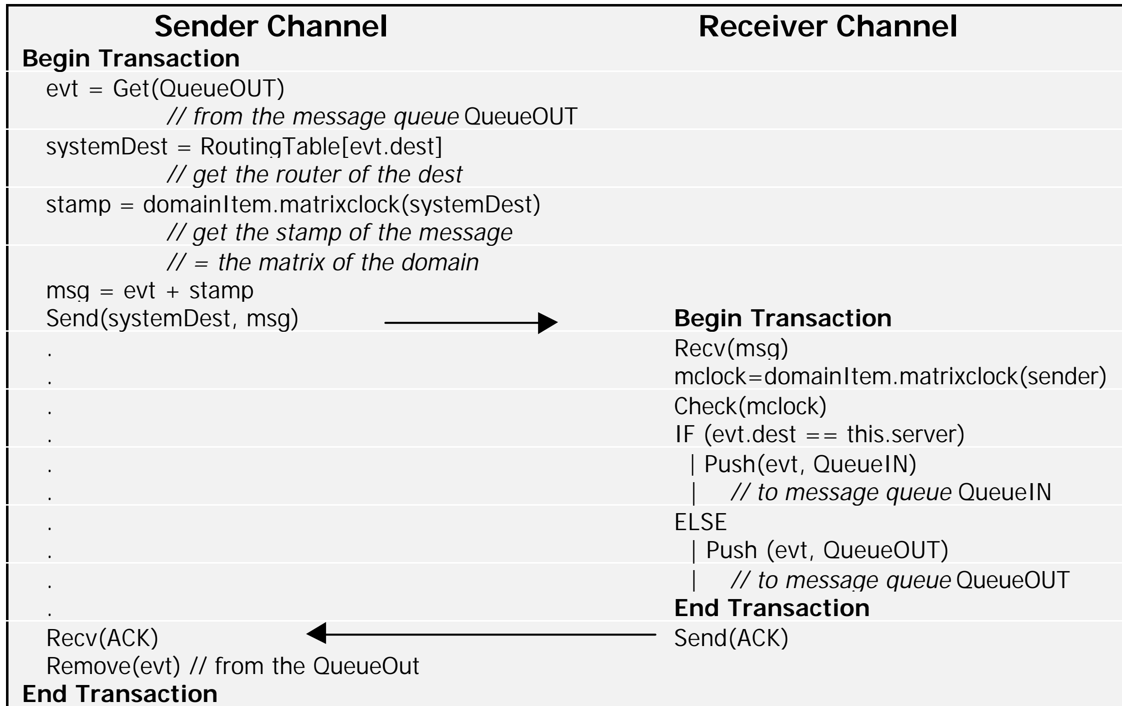


Figure 35: L'algorithme du Channel avec gestion des domaines de causalité

Nous n'avons apporté aucune modification au moteur d'exécution (*l'engine*) car la gestion des domaines de causalité est invisible aux agents et tout le traitement de routage et de vérification de la causalité se gère au niveau du *Channel.*

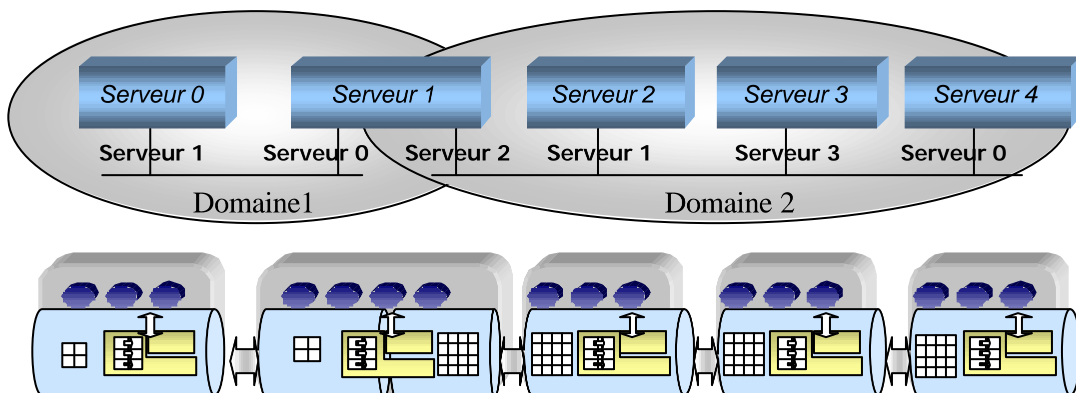


Figure 36: Exemple de la vision des domaines et de la structure associée

7.1.2. Perspectives de réalisation

La priorité pour la réalisation est la vérification de l'architecture des domaines créés par l'utilisateur. Il faut vérifier que l'ensemble de ce découpage respecte la condition de validité des domaines, c'est à dire qu'il ne doit pas y avoir de cycle de domaines.

Actuellement la configuration des domaines se fait par l'intermédiaire d'un fichier dans lequel il faut écrire la liste des domaines et les serveurs qui y sont rattachés. Les serveurs d'agents possèdent eux aussi un fichier de configuration qui leur donnent toutes les informations necessaires à leur initialisation (identifiant, machine d'exécution, services associés), il serait donc souhaitable d'intégrer les informations sur les domaines dans ce fichier de configuration en donnant pour chaque serveur le(s) domaine(s) auquel(s) il appartient.
On pourrait aussi imaginer que les serveurs changent de domaines dynamiquement afin de s'adapter seuls aux problèmes de permormance. Par exemple, si un serveur detecte que le domaine auquel il est rattaché devient surchargé, il pourrait changer dynamiquement de domaine afin de répartir la charge. Bien sûr cela pose de nombreux problèmes dont celui de la vérification de respect de la condition de validité.

7.2. Expérimentation et résultats

Afin de vérifier l'amélioration des performances avec la gestion des domaines de causalité nous avons réalisé une série de mesures. Ces mesures suivent exactement le même protocole que les premières mesures sur l'impact de la causalité. Il y a 3 séries de mesures réalisées avec les domaines de causalité qui se trouve dans l'Annexe 2 :.

Tests centralisés (Annexe 2 :: page 75)

Pour les tests centralisés de la première série nous avons réalisé une configuration de domaines de causalité en bus et nous avons limité le nombre de serveur par domaine à 10.
Lors des tests centralisés nous voyons clairement qu'il subsiste quelques fluctuations des mesures dans les courbes d'envois locaux et distants. Ceci est essentiellement dû à la faible variation des valeurs qui paraissent amplifiées sur la courbe.

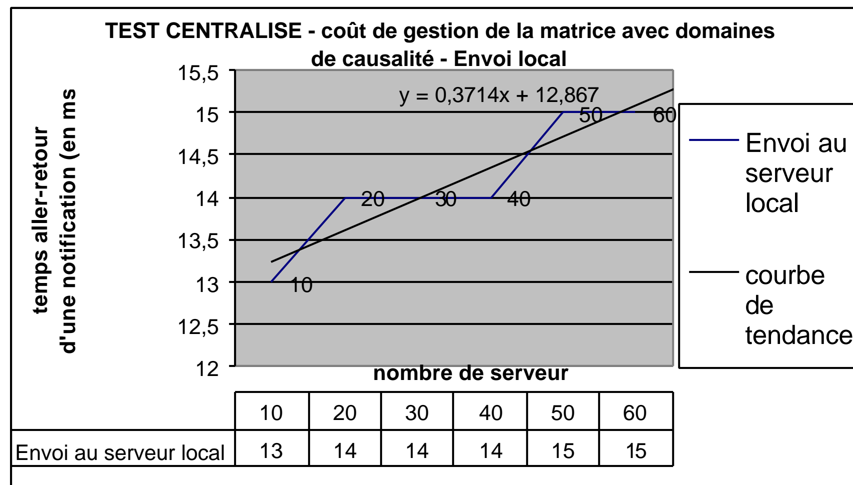
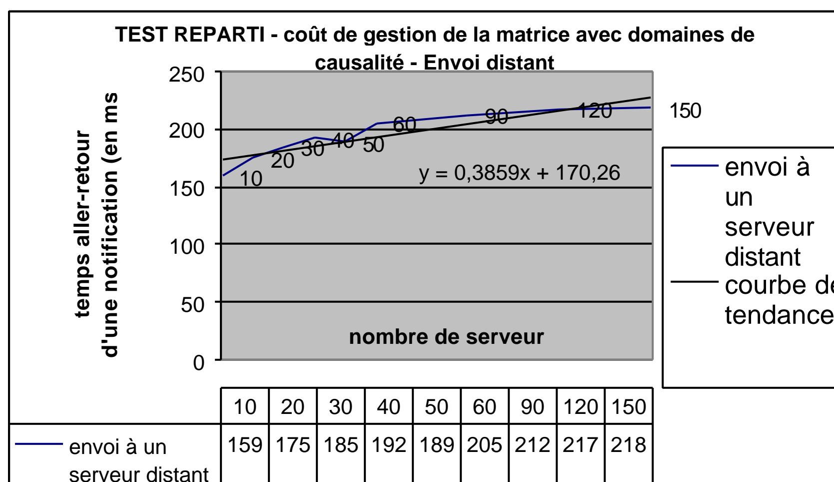


Figure 37: Résultat des mesures d'envoi local de notifications

Un détail dans les tests centralisés qui ne saute pas aux yeux est la possibilité d'exécuter jusqu'à 60 serveurs d'agents sur une seule machine alors que les tests réalisés sans les domaines de causalité limitaient se nombre à 50 (au dessus les démons NFS des machines plantaient et entraînaient le crash des machines). Ceci prouve le coût moins important de la charge disque de l'implémentation avec domaines de causalité.

Tests répartis en bus (Annexe 2 :: page 76)



	10	20	30	40	50	60	90	120	150
envoi à un serveur distant	159	175	185	192	189	205	212	217	218

Figure 38: Résultat des mesures d'envoi distant de notifications

Dans la deuxième série de test (*Test en environnement réparti et domaine de causalité en bus*) nous avons réalisé une configuration de domaines de causalité selon la configuration des machines, c'est à dire que nous avons reproduit le découpage de la répartition des serveurs selon les machines en domaines en les reliant selon une méthode de bus. Par exemple pour le test réparti d'envoi distant avec 120 serveurs répartis à raison de 24 par machines, nous avons découpé les serveurs en 5 domaines de 24. Dans cette série nous avons relié les domaines selon un modèle de bus (voir Figure 39).

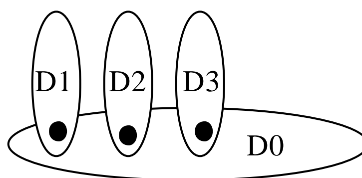
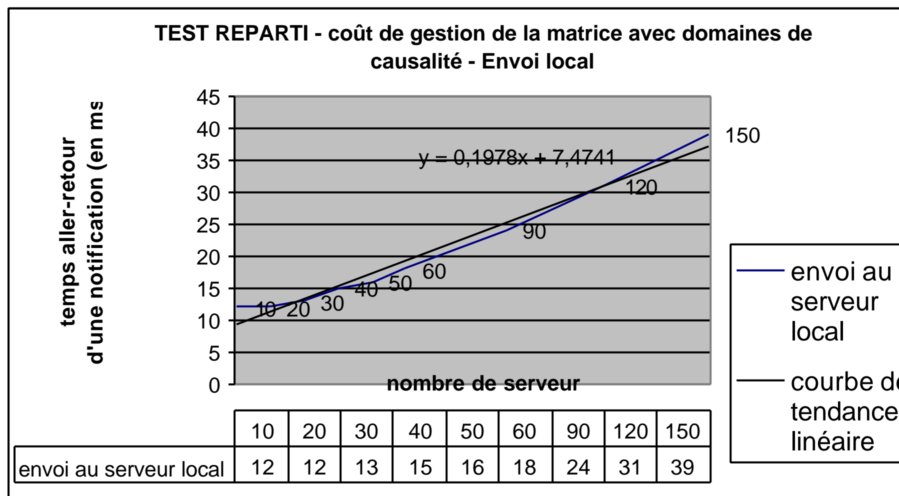


Figure 39: Découpage en bus des domaines réalisé pour les tests 1 et 2 de l'Annexe 2 :

Ces premiers tests nous montrent immédiatement les avantages des domaines de causalité, lorsque l'envoi reste dans le même domaine (comme c'est le cas avec les tests d'envoi local et distant en centralisé et réparti) les coût de la gestion de l'ordonnancement est complètement linéaire et la pente des courbes de tendance est très faible ce qui laisse présager de très bon résultat lors du déploiement de centaines ou milliers de serveurs. Ceci est dû au fait que la taille de la matrice dans un même domaine est très réduite par rapport à une matrice globale et tous les coût associés à cette matrice (stockage mémoire, sauvegarde,

traitement) sont réduits. Les tests de diffusions aussi sont très prometteurs. Il subsiste néanmoins un pic lors de la diffusion sur 90 serveurs en répartis qui est certainement dû à la surcharge inopinée du serveur principal lors de l'envoi des notifications (ce test n'étant réalisé qu'une seul fois contrairement à la moyenne de cent envois des autres tests).

Tests repartis en étoile (Annexe 2 :: page 77)



	10	20	30	40	50	60	90	120	150
envoi au serveur local	12	12	13	15	16	18	24	31	39

Figure 40: Résultat des mesures d'envoi local de notifications

La dernière série de test (*Test en environnement réparti et domaine de causalité en étoile*) a été réalisée avec une configuration de domaines en étoile. C'est à dire que nous avons limité le nombre de serveurs par domaine à 10 puis nous avons relié tous les domaines au serveur d'agent principal (voir Figure 41).

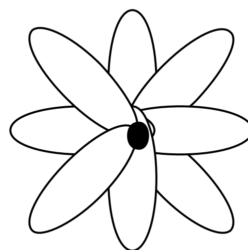
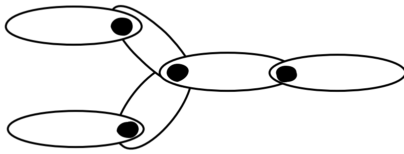


Figure 41: Découpage en étoile des domaines réalisé pour les tests page 77

Ce type de topologie montre immédiatement son avantage pour la gestion de la diffusion, grâce à ce découpage nous pouvons réaliser un tests de diffusion sans que les coût excessifs des accès disque ne viennent gêner les performances des envois.

Tests repartis hiérarchique

Nous avons aussi réalisé un test qui n'est pas disponible dans les annexes avec une configuration très spécifique des domaines de causalité. Cette configuration consiste à gérer une hiérarchie de domaine comme montré sur la Figure 42, avec des domaines de 10 serveurs d'agents.



Envoi local	Envoi distant	Diffusion
19 ms	201 ms	19146 ms

Figure 42: Découpage hiérarchique des domaines et résultat pour 90 serveurs répartis

Les résultats de ce test réalisés avec 90 serveur répartis nous montre encore une fois que l'envoi dans un même domaine est peu coûteux mais que le routage à travers de nombreux serveurs ralenti les performances du bus mais reste toujours plus avantageux qu'une solution sans domaines de causalité. Néanmoins la diffusion reste très coûteuse car elle nécessite un routage important des messages.

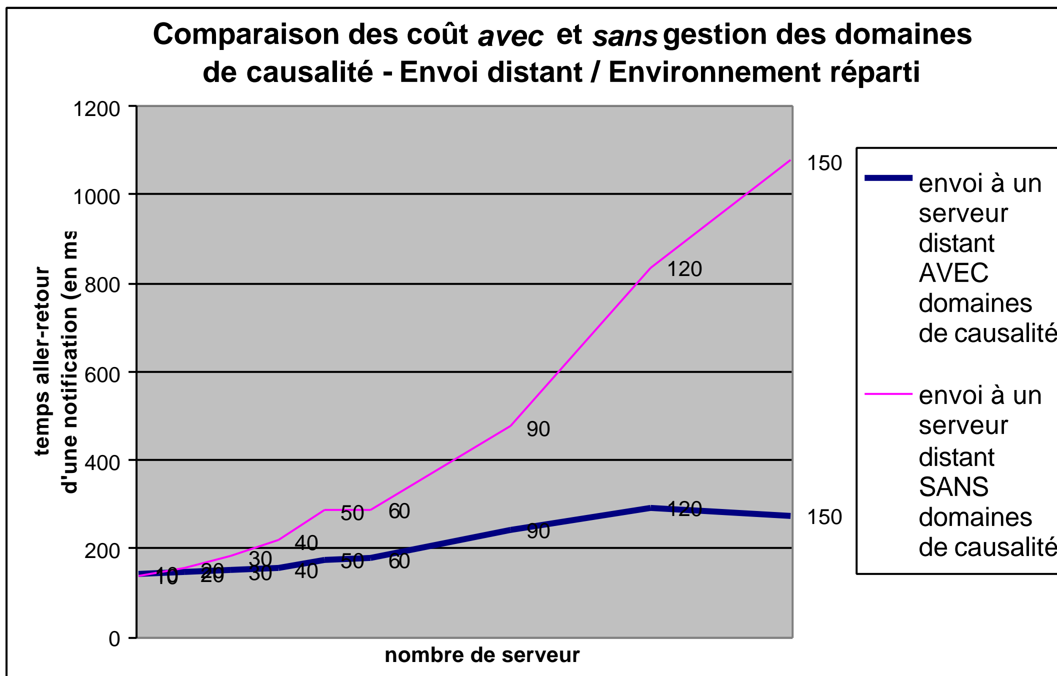


Figure 43: Résultats comparés *avec* et *sans* gestion des domaines de causalité en étoile

Pour conclure sur ces tests, nous pouvons dire que les mesures que nous avons réalisées nous montre clairement l'avantage qu'apporte les domaines de causalité sur les performances (voir Figure 43), ils permettent de réduire considérablement les coût lors des envois dans le même domaine. La topologie en étoile est la plus avantageuse mais risque de surcharger le serveur central de la gestion de nombreux domaine. L'intérêt du découpage hiérarchique (ou en bus) serait d'ajuster des applications qui communiquent souvent localement (avec un petit nombre de composant) et de temps en temps aux domaines distants.

Conclusion

Le problème auquel nous avons cherché une solution est celui du passage à l'échelle d'un *middleware* à messages (limitation de la dégradation des performances lors de l'augmentation du nombre de sites connectés).
Les *middlewares* à messages ou MOM (*Message-Oriented Middleware*) sont une infrastructure logicielle de communication asynchrone qui prend à sa charge les problèmes transitoires de réseau, de connexion de machine, voire de disponibilité de machines. Ils sont utilisés pour interconnecter des applications d'entreprise, chaque application interagissant localement avec le système de messagerie. Les MOM possèdent des propriétés de fiabilité, de sécurité et d'ordonnancement.
Toutefois, ces propriétés posent un réel problème de passage à l'échelle car l'efficacité des algorithmes les mettant en oeuvre diminue avec l'augmentation du nombre de sites utilisés. La propriété d'ordonnancement causal est celle qui apporte le plus de difficulté, car les temps d 'exécution des algorithmes utilisés croissent non linéairement avec l'augmentation des sites.

Nous avons proposé deux solutions afin de réduire le coût de l'ordonnancement des messages. Ces solutions sont basées sur le principe de domaines de causalité. Un domaine de causalité est un ensemble d'entités dans lequel on conserve la propriété d'ordonnancement. Nous avons définis deux critères de domaines de causalité :

- **Les domaines de causalité par topologie applicative** consistent à utiliser la description de l'architecture de l'application afin d'optimiser l'utilisation de l'ordonnancement. L'application étant décrite à l'aide d'un ADL, il devient possible d'utiliser cette description afin d'en dégager la topologie de l'application et d'optimiser la gestion de l'ordonnancement à cette topologie.
- **Les domaines de causalité par topologie réseau** consistent à découper la gestion de la causalité dans des domaines et à relier ces domaines grâce à des serveurs-routeurs. Ce découpage permet d'alléger le coût de la causalité dans chaque domaine et garantit la conservation de la causalité de façon globale sur l'ensemble des domaines.

Une troisième solution que nous n'avons pas approfondie consiste à réaliser un ordonnancement applicatif. Ce dernier consiste à délocaliser la propriété d'ordonnancement au niveau applicatif afin de laisser le programmeur libre d'utiliser ou non les services d'ordonnancement fournis par le système de MOM.

Nous avons expérimenté la gestion des domaines de causalité par topologie réseau sur le bus à messages AAA. Les tests de mesures sur l'implémentation réalisée nous montrent clairement les avantages fournis par cette méthode : une augmentation des performances du bus à messages, la transparence pour les applications clientes du bus et la modularité des configurations des domaines de causalité. Nous avons transformé une croissance quadratique des coûts en une croissance linéaire.
Cependant nous ne considérons pas notre effort d'évaluation comme achevé. Nous allons dans un premier temps améliorer l'implémentation actuelle afin de l'intégrer complètement dans le bus AAA. Puis nous allons essayer de développer une implémentation pour chacun des domaines de causalité avec une priorité pour les domaines de causalité par topologie applicative. Une fois l'ensemble des solutions implémenter et tester nous les intégrerons dans le bus à messages AAA.

Les perspectives de ce travail seraient de pouvoir généraliser cette amélioration de la qualité de service à l'adaptation spontanée du MOM. Celui-ci pourrait *s'adapter* aux diverses complications auxquelles il doit faire face (baisse des performances résea. Ainsi on peut imaginer qu'en fonction des applications clientes du MOM celui-ci décide de regrouper ou déployer les domaines de causalité pour améliorer les performances globales et tout ceci en se servant d'une part des informations sur la topologie de l'application (par l'intermédiaire de son ADL) et d'autre part de la topologie du réseau disponible. Cette flexibilité de l'ordonnancement pourrait donc à terme être reproduite pour les autres propriétés des MOM comme la persistance ou la sécurité afin d'en améliorer les performances lors du passage à l'échelle.

Références

[ABDP99] L. Avignon, T. Brethes, C. Devaux, and P. Pezziardi. « Le livre blanc de l'EAI » OCTO technology, Octobre 1999.

[Agh86] Gul A. Agha. « Actors: A Model of Concurrent Computation in Distributed Systems ». *The MIT Press*, ISBN 0-262-01092-5, Cambridge, MA, 1986.

[Bal00] Roland Balter, « Modes de structuration d'applications réparties ». Cours de DEA "Informatique Système et Communication". Université Joseph Fourier, Grenoble, France 2000.
http://sirac.inrialpes.fr

[BCS+99] G. Banavar, T. Chandra, D. Sturman, B. Mukherjee, J. Nagarajaro and J. Storm : « An efficient multicast protocol for content-based Publish-Subscribe systems » . *Proceedings of the 19th International Conference on Distributed Computing Systems*, Austin, TX, USA. IEEE Computer Society: 262-272, 31 May - 4 June 1999.
http://www.research.ibm.com/gryphon/

[BCSS99] G. Banavar, T. Chandra, R Strom and D. Sturman. « A case for Message Oriented Middleware ». *Distributed Computing 13th International Symposium*, Bratislava, Slovak Republic, September 27-29, 1999, Proceedings. Lecture Notes in Computer Science, Vol. 1693 : 1-18, Springer, 1999, ISBN 3-540-66531-5
http://www.research.ibm.com/gryphon/

[BDFHS99] L. Bellissard, N. De Palma, A. Freyssinet, M. Herrmann, S. Lacourte, « An Agent Platform for Reliable Asynchronous Distributed Programming », *Symposium on Reliable Distributed Systems (SRDS'99)* (short paper), Lausanne - Suisse, 20-22 October 1999

[Bel00] Luc Bellissard, « Modèle de bus à message ». Cours de DEA "Informatique Systèmes et Communication". Institut National Polytechnique INPG, Grenoble, France, 2000.
http://sirac.inrialpes.fr

[BM93] Ö. Babaõglu, K. Marzullo, « Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms » , in [Mul93].

[CRW99] Antonio Carzaniga, David S. Rosenblum and Alexander L. Wolf, « Interfaces and Algorithms for a Wide-Area Event Notification Service ». Technical Report CU-CS-888-99, University of Colorado, Department of Computer Science, Boulder 1999.
http://www.cs.colorado.edu/~carzanig/papers/

[CS93] David R. Cheriton and Dale Skeen, « Understanding the Limitations of Causally and Totally Ordered Communication », *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Décembre 1993.
http://www.acm.org/sigmod/dblp/db/conf/sosp/sosp93.html#CheritonS93

[DBFL99] Noël De Palma, Luc Bellissard, André Freyssinet and Serge Lacourte. « An Agent Based Message Oriented Middleware ». Technical Report, SIRAC Project and AAA-DYADE, INRIA Rhône-Alpes, 1999.
http://www.dyade.fr/fr/action/aaa/aaa.html

[Fio00] Fiorano software Inc. « FioranoMQ 4.1 Technical Overview », Avril 2000
http://www.fiorano.com

[HS99] Alex Hormer and David Sussman. In book : « *MTS & MSMQ* », Chapitre 5, Eyrolles, Paris, France, février 1999. ISBN 1-8610046-0.

[Kor97] M. Korhonen. « Message Oriented Middleware (MOM) ». Technical Report, Department of Computer Science, Helsinki University of Technology, avril 1997.

[Lam78] L. Lamport. « Time clocks, and the ordering of events in a distributed system ». *Communications of the AC*M, 21(7):558-565,1978.

[Mul93] S. Mullenderr (ed.), *Distributed Systems* (2nd edition), Addison-Wesley, 93.

[OMG98] Object Management Group, «CORBA Messaging » , White paper, Joint Revised Submission 98-05-05, mai 1998.
http://www.omg.org

[Ope98] Open Horizon. « Ambrosia 3.0 : A Event Management System », Technical Overview. Mars 1998.
http://www.openhorizon.com

[OPSS93] B. Oki, M. Pflueg, A. Siegel and D. Skeen. « The Information Bus - An architecture for extensible Distributed Systems ». *Operating Systems Review*, 27(5), pp 58-68, décembre 1993.

[Pee] PeerLogic. « LiveContent PIPES », Technical Overview, 2000.
http://www.peerlogic.com

[Pin98] Jonathan Pinnock. « Professional DCOM, Application Development » Chapitre 6, Wrox Press, USA, 1998. ISBN 1-861001-31-2.

[Reis90] P. Reiss. « Connecting Tools Using Message Passing in the Field Environment ». *IEEE Software*, pages 57-66, juillet 1990.

[Rib] Gérard Ribière. « Communication et traitement en mode message avec MQSeries ». *Techniques de l'ingénieur*, Rapport technique H2-768, IBM France.

[RR95] B. Rao Rama. « Making the most of Middleware ». Technical Report, Integrated Solution Inc, septembre 1995.

[RS95] Michel Raynal and M. Singhal, « Logical Time : A way to capture causality in distributed systems », *IEEE Computer* (2): 49-56, 1995.

[Rug95] Frederic Ruget, « Cheaper Matrix Clock », In *Proc. Of the 8th Int. WorkShop on distributed Algorithms* (WDAG-8), Terschelling, the Netherlands, September 1994.

[Ste95] S. Steinke. «Middleware meets the network», *LAN: The Network Solutions Magazine*, volume 10, number 13, p. 56-61, décembre 1995.

[Sun99] Sun Microsystems, Inc. "JMS Specifications", version 1.0.2a, Californie, novembre 1999.
http://java.sun.com/jms

[Tib99] TIBCO. TIB/Rendezvous, White Paper. 1999
http://www.rv.tibco.com

ANNEXES

Annexe 1 : Résultats de mesures du bus à agent avant gestion des domaines de causalité. *73*

Test en environnement centralisé ________________ **73**

Test en environnement réparti ________________ **74**

Annexe 2 : Résultats de mesures du bus à agent avec gestion des domaines de causalité *75*

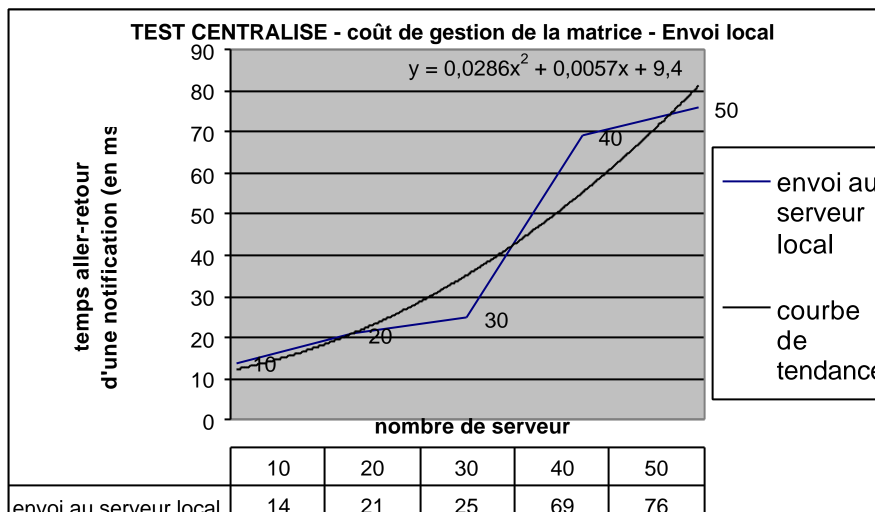
Test en environnement centralisé et domaine de causalité en bus ________________ **75**

Test en environnement réparti et domaine de causalité en bus ________________ **76**

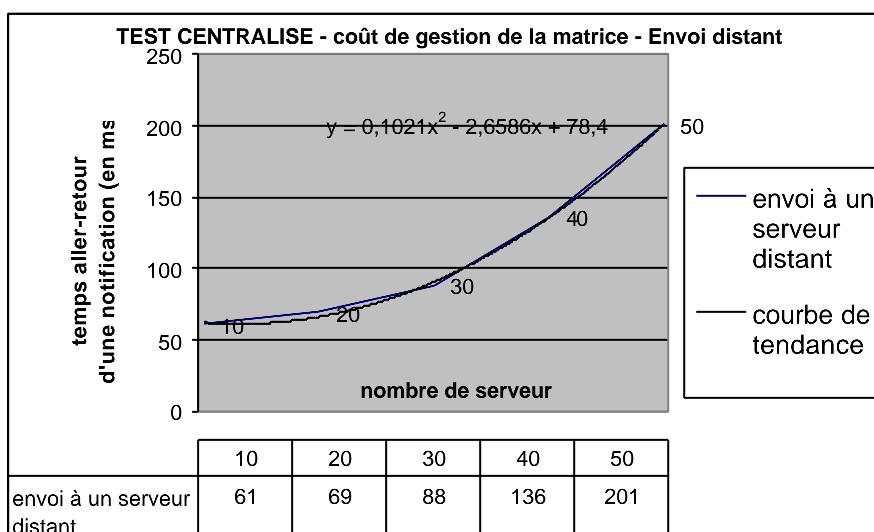
Test en environnement réparti et domaine de causalité en étoile ________________ **77**

Annexe 1 : Résultats de mesures du bus à agent avant gestion des domaines de causalité.

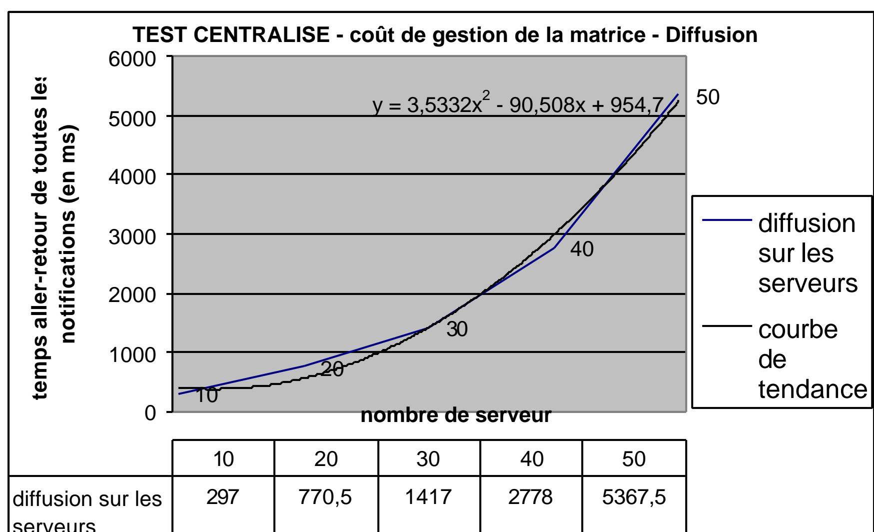
Test en environnement centralisé



	10	20	30	40	50
envoi au serveur local	14	21	25	69	76

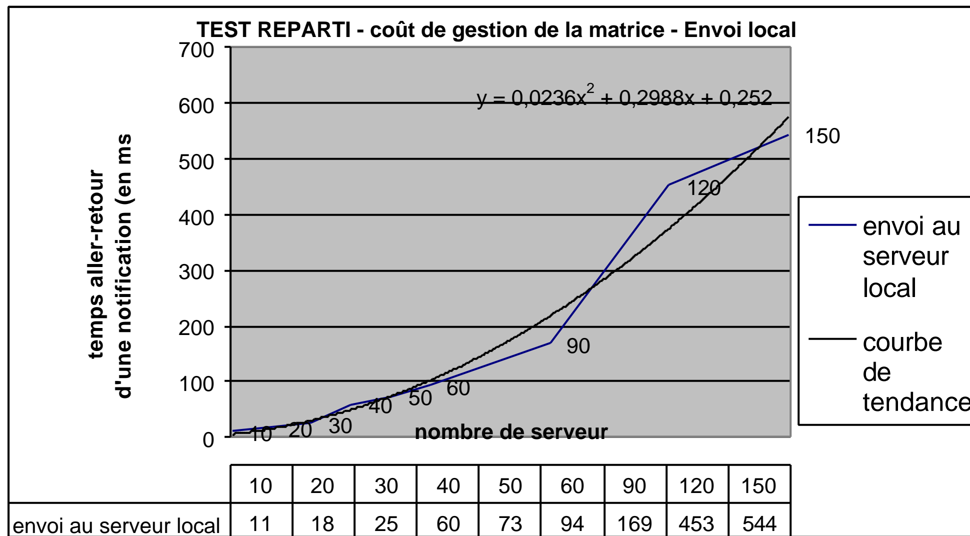


	10	20	30	40	50
envoi à un serveur distant	61	69	88	136	201

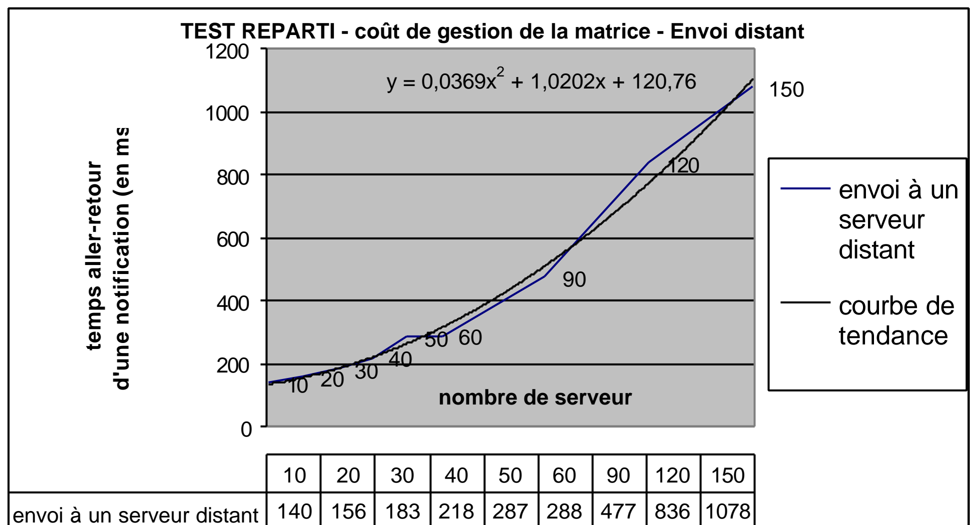


	10	20	30	40	50
diffusion sur les serveurs	297	770,5	1417	2778	5367,5

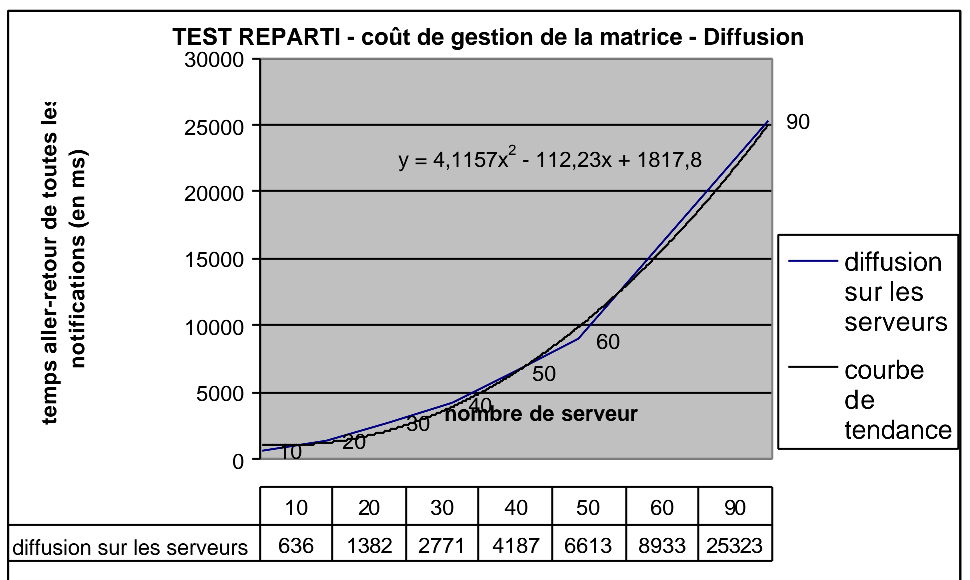
Test en environnement réparti



	10	20	30	40	50	60	90	120	150
envoi au serveur local	11	18	25	60	73	94	169	453	544



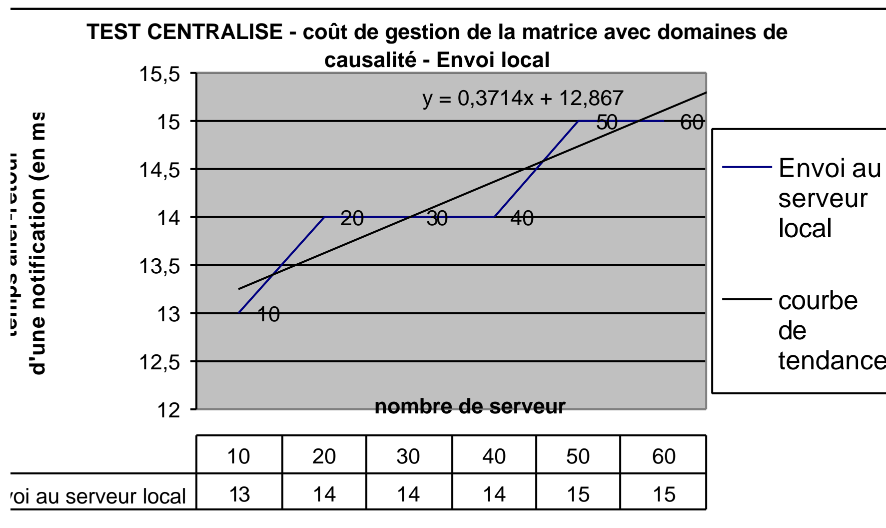
	10	20	30	40	50	60	90	120	150
envoi à un serveur distant	140	156	183	218	287	288	477	836	1078

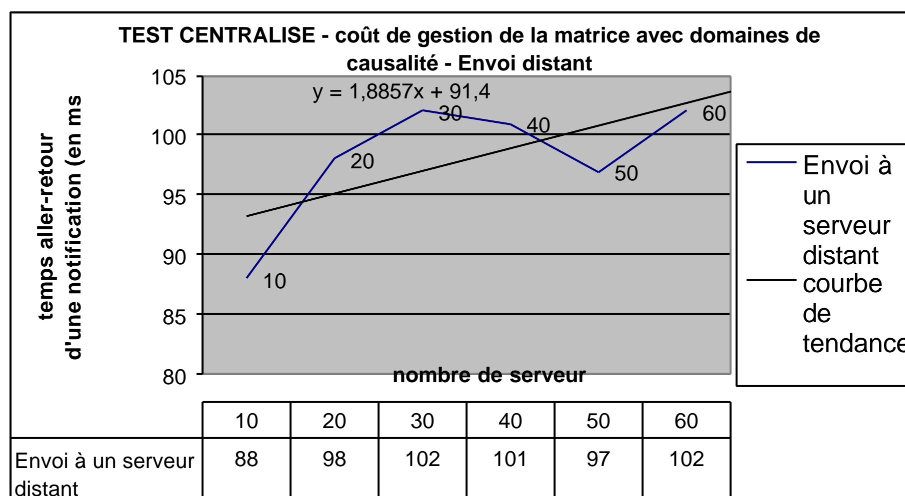


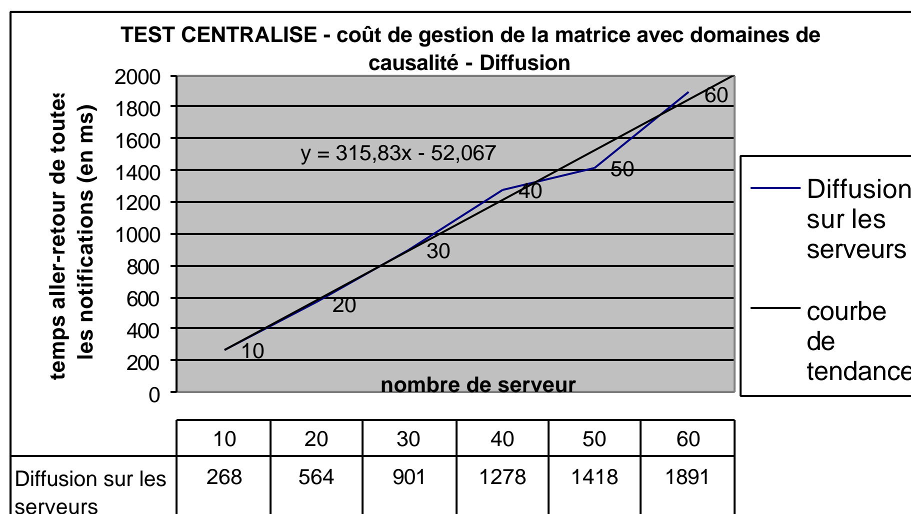
	10	20	30	40	50	60	90
diffusion sur les serveurs	636	1382	2771	4187	6613	8933	25323

Annexe 2 : Résultats de mesures du bus à agent avec gestion des domaines de causalité

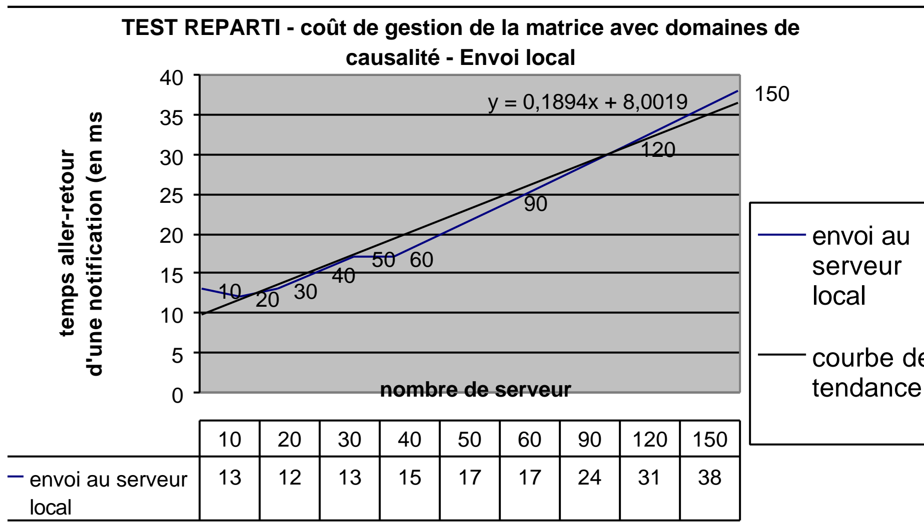
Test en environnement centralisé et domaine de causalité en bus



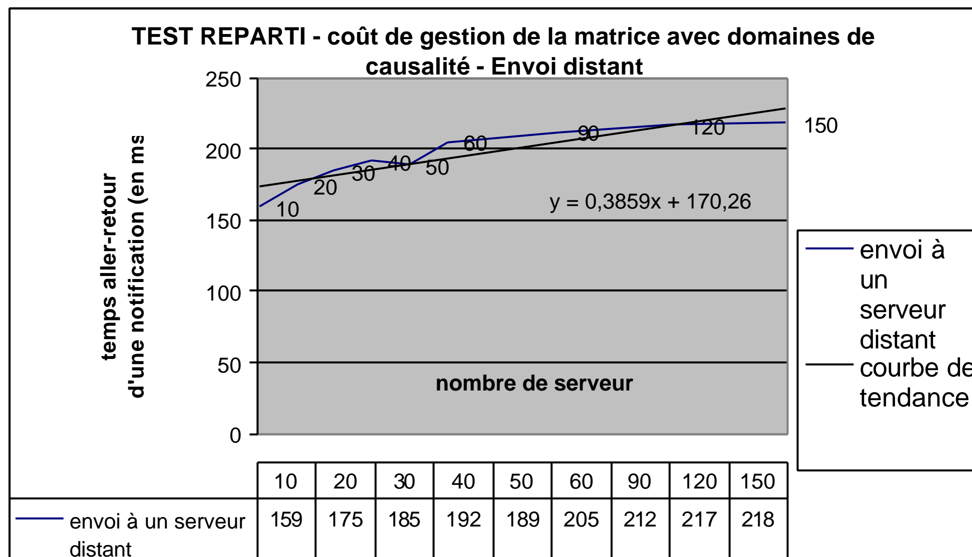




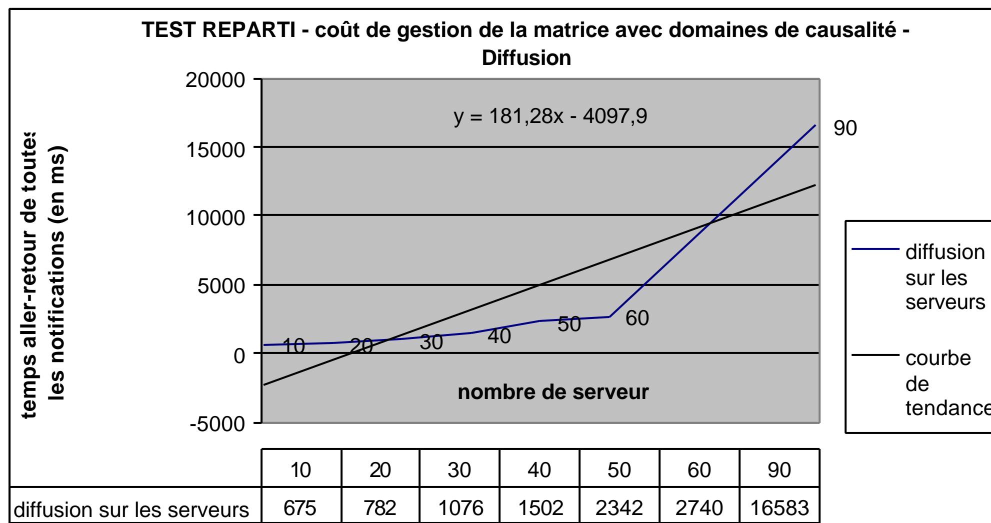
Test en environnement réparti et domaine de causalité en bus



	10	20	30	40	50	60	90	120	150
envoi au serveur local	13	12	13	15	17	17	24	31	38

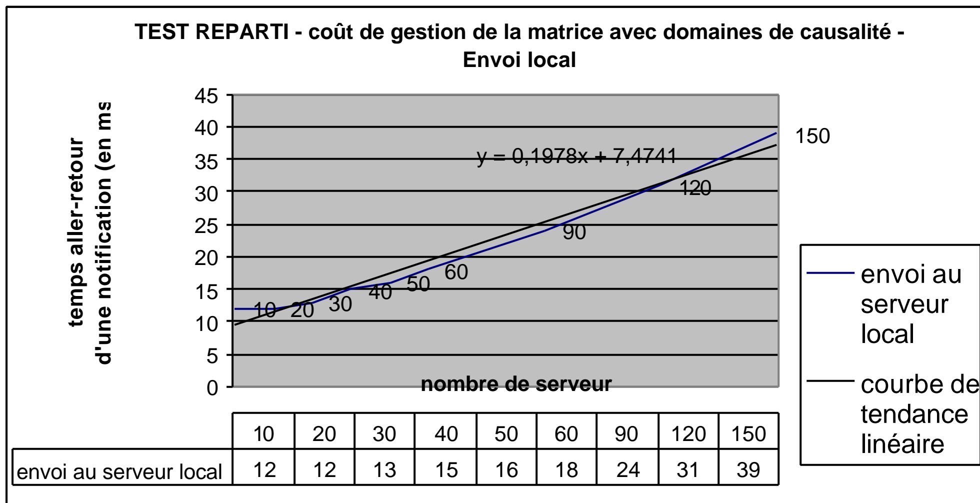


	10	20	30	40	50	60	90	120	150
envoi à un serveur distant	159	175	185	192	189	205	212	217	218

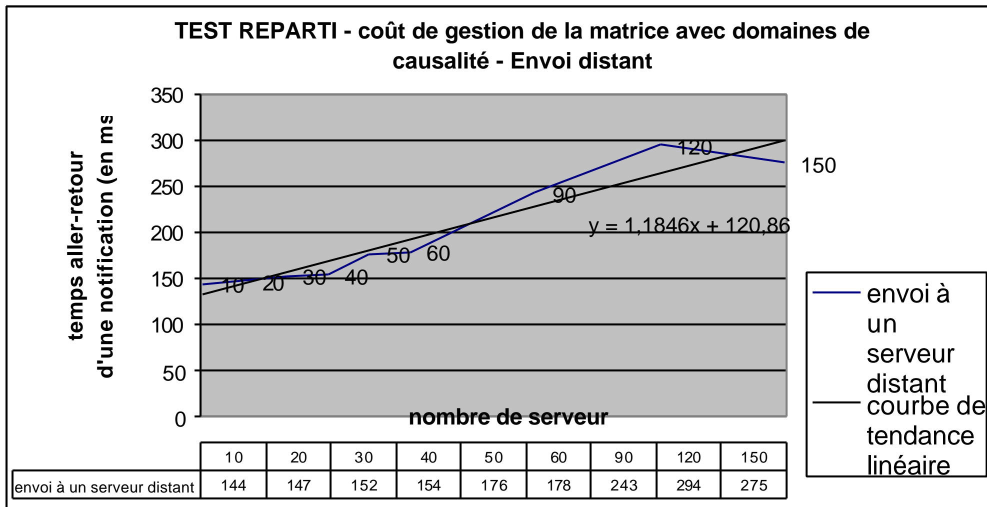


	10	20	30	40	50	60	90
diffusion sur les serveurs	675	782	1076	1502	2342	2740	16583

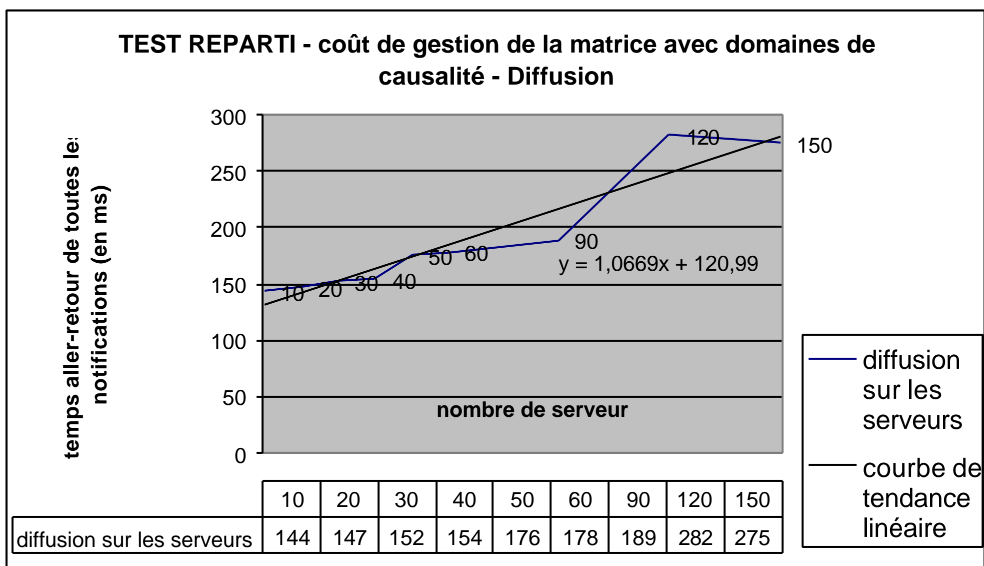
Test en environnement réparti et domaine de causalité en étoile



	10	20	30	40	50	60	90	120	150
envoi au serveur local	12	12	13	15	16	18	24	31	39



	10	20	30	40	50	60	90	120	150
envoi à un serveur distant	144	147	152	154	176	178	243	294	275



	10	20	30	40	50	60	90	120	150
diffusion sur les serveurs	144	147	152	154	176	178	189	282	275