

Modélisation, conception et réalisation de systèmes répartis

Andrzej Duda

▶ **To cite this version:**

Andrzej Duda. Modélisation, conception et réalisation de systèmes répartis. Réseaux et télécommunications [cs.NI]. Université Joseph-Fourier - Grenoble I, 1994. tel-00005088

HAL Id: tel-00005088
https://theses.hal.science/tel-00005088

Submitted on 25 Feb 2004

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Chapitre I

Évaluation de performances des systèmes répartis

I.1 Introduction

Dans ce chapitre, nous présentons notre contribution dans le domaine de l'évaluation de performances des systèmes répartis. La démarche scientifique consistait d'abord à étudier les caractéristiques des systèmes répartis qui ont un impact sur les performances, à les analyser, modéliser et obtenir les indices de performances. Pour ce but, nous avons utilisé la modélisation analytique qui a été validée par la simulation. Ensuite, pour pouvoir comparer les conclusions de cette modélisation avec la réalité, nous avons construit une maquette de système réparti dans le projet *Epsilon* et nous avons proposé une méthode expérimentale de mesure dans les systèmes répartis.

I.2 Modélisation des systèmes répartis

La modélisation analytique fait usage de la théorie de files d'attente, une discipline mathématique qui a connu un grand essor dans les années 50 en apportant un formalisme pour l'analyse des réseaux téléphoniques. Ensuite, la théorie de files d'attente a servi dans les années 70 à la modélisation des systèmes informatiques centralisés et des réseaux de transmission de données. Une file d'attente représente une ressource comme l'unité centrale ou un disque et les clients de la file sont des entités actives comme des programmes ou des transactions qui sont en compétition pour l'utilisation de la ressource. Un modèle de file d'attente peut prévoir le temps d'attente nécessaire pour l'utilisation de la ressource, la longueur de la file et le taux d'utilisation de la ressource. Les recherches dans ce domaine ont abouti dans les années 70 au modèle *BCMP* qui permet de modéliser analytiquement un réseau général de files d'attente [Baskett 75]. La méthode *MVA* a été également un progrès dans la résolution analytique des files d'attente [Reiser 80]. Plusieurs outils de modélisation basés sur ce modèle ont été développés (*QNAP2*, *RESQ*, *BEST/1*) et certains d'entre eux fournissent également un langage de simulation et un support pour dérouler une simulation, collecter des statistiques et calculer les indices de performances.

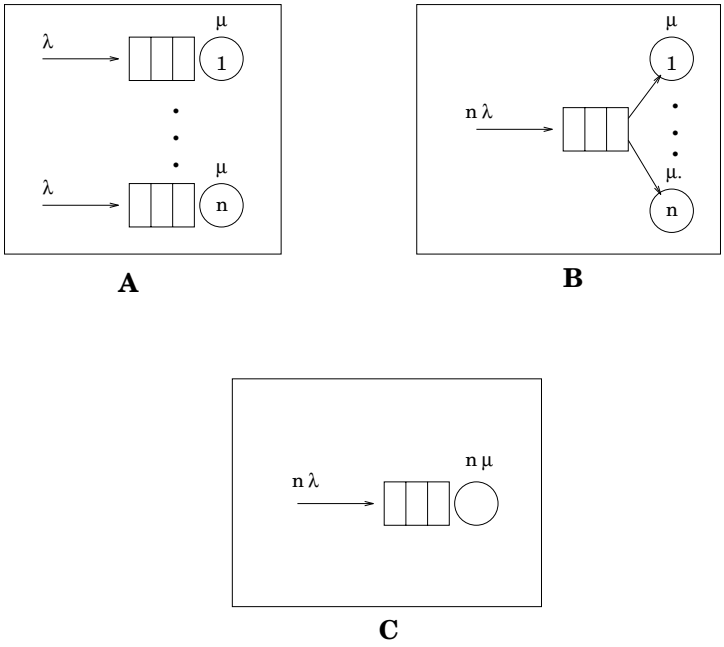


Fig. 1.1 : Modèles simples d'un système réparti

Pour montrer l'utilité que peut avoir la modélisation analytique, considérons un modèle simple d'un système réparti composé de *n* stations (Fig. 1.1). Le modèle *A* représente un système réparti où les stations travaillent de façon indépendante : les demandes d'exécution arrivent à chaque station avec le taux λ et chaque station traite ces demandes avec la vitesse μ. Dans cette configuration, si une station est libre, elle ne peut pas traiter les demandes qui sont en attente dans les stations occupées. Le modèle *B* correspond à un système qui dispose de mécanismes de répartition de la charge. Cette technique permet l'amélioration des performances d'un système réparti. Elle est fondée sur le fait que dans un tel système, il peut y avoir des sites oisifs pendant que d'autres sont surchargés. Les sites surchargés peuvent alors exécuter des programmes sur les sites oisifs, ce qui améliore les performances. La répartition de la charge est prise en compte dans le modèle par la file commune qui est conceptuellement réalisée par le mécanisme de répartition de la charge. Les deux configurations précédentes peuvent être comparées avec un système centralisé, mais *n* fois plus rapide.

Comparons les performances de ces trois configurations en supposant que les temps entre deux arrivées et les temps de traitements sont distribués exponentiellement (ces derniers ont la moyenne 1) et le nombre de stations est 2. La Fig. 1.2 présente le temps moyen entre l'arrivée d'une demande d'exécution et la fin de son traitement dans les trois systèmes en fonction de la charge. On peut constater que les performances sont meilleures dans la configuration *C*, puis *B* et enfin *A*. Les mêmes relations de performances existent dans ces systèmes pour *n* quelconque. Pour comprendre pourquoi *B* est meilleur que *A*, il faut observer que dans la

configuration *A* les capacités de traitement ne sont pas utilisées quand il y a des demandes d'exécution en attente dans les stations occupées et qu'il existent en même temps des stations libres. Cette situation est de faible importance quand la charge est légère, mais elle devient importante avec l'accroissement de la charge. L'amélioration de performance dans la configuration *B* montre les gains potentiels que peut apporter un mécanisme de répartition de la charge. Pour comprendre la supériorité de la configuration *C* sur *B*, il faut remarquer que si la charge est légère, le système *B* ne peut traiter les demandes qu'avec la vitesse μ, tandis que le système *C* les traite *n* fois plus rapidement. On voit bien ce phénomène sur la Fig. 1.2 où pour la charge nulle, le temps moyen de traitement dans le système *C* est deux fois plus petit que dans le système *B*.

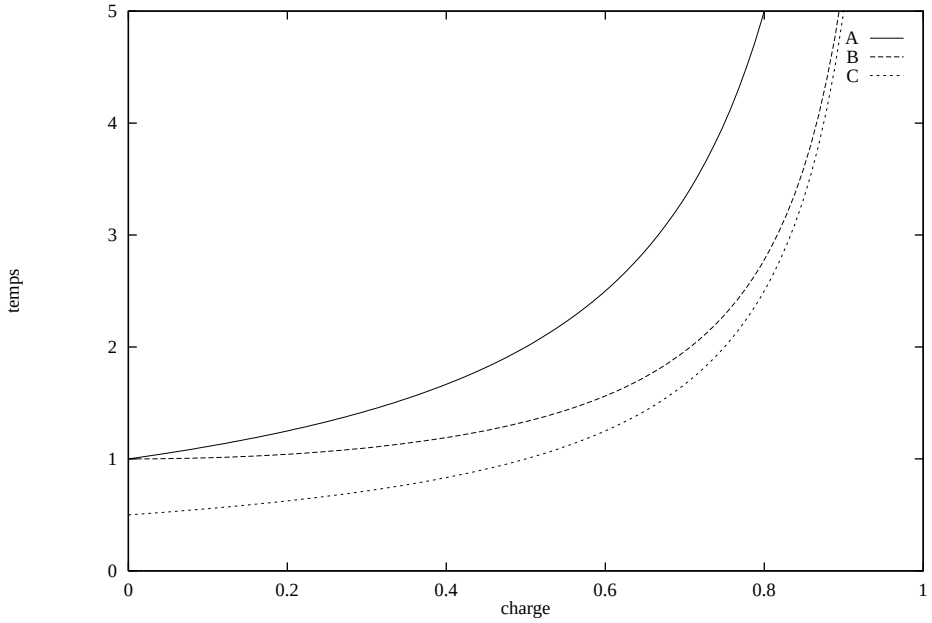


Fig. 1.2 : Performances de trois systèmes

On peut se demander où est la place du parallélisme dans ces modèles. En fait, cette comparaison de performances nous apprend qu'il est plus avantageux d'avoir une seule machine très rapide que plusieurs machines même si elles disposent de mécanismes de répartition de la charge. Mais comment augmenter la vitesse de traitement d'une seule machine au−delà des limites imposées par la technologie? La solution vient du coté du parallélisme − la vitesse supérieure de traitement peut être obtenue par un traitement de demandes d'exécution en parallèle. La configuration *C* est un modèle correspondant à un point de vue de clients d'une machine parallèle − si elle comprend *n* processeurs et si chaque demande d'exécution peut être divisée en *n* parties pouvant être traitées en parallèle, elle sera vue par les utilisateurs comme un seul serveur ayant une vitesse de traitement *n* fois plus rapide que la vitesse des

processeurs. Aussi, le même modèle indiquera les gains potentiels dans un système réparti avec répartition de la charge (file commune) où les demandes d'exécution sont divisées en tâches pouvant être exécutées en parallèle (vitesse de traitement n fois plus rapide).

La modélisation analytique présentée ci−dessus fait usage de la théorie de files d'attente conventionnelles. Ce modèle a été conçu pour l'analyse des effets de conflits d'accès aux ressources et ne prend pas en compte les phénomènes du parallélisme et des contraintes de la synchronisation. Par exemple, ce modèle ne permet pas de modéliser la duplication de clients ou l'attente d'un client pour un autre. L'étude de ces phénomènes n'est possible qu'à l'aide de la simulation et pourtant leur analyse est importante parce que par exemple la synchronisation de plusieurs activités parallèles peut entraîner des retards importants qui réduisent les gains de vitesse dus au parallélisme.

I.2.1 Primitives de synchronisation de type *fork−join*

Dans le travail [Duda 87a], [Duda 88a], il s'agissait de trouver de nouveaux modèles pour prendre en compte les phénomènes du parallélisme et de la synchronisation ainsi que les conflits d'accès aux ressources. Nous avons développé des modèles de files d'attente qui caractérisent l'exécution de programmes répartis et parallèles écrits à l'aide des primitives *fork et join*. Un programme est composé de tâches avec des contraintes de précédence représentées par un graphe de précédence. Un système réparti qui traite les demandes d'exécution de ces programmes peut être modélisé comme un réseau de file d'attente avec la duplication des clients et leur synchronisation. Le modèle de base est une file *fork−join* composée de n stations en parallèle. Un client qui arrive à la file est dupliqué en n exemplaires (primitive *fork*) et traité en parallèle par les stations. Les clients qui terminent leur traitement attendent la fin du dernier client pour quitter le système (primitive *join*).

La file *fork−join* est difficile à analyser, parce qu'il n'existe pas de solutions en forme produit. Très peu de résultats existaient à l'époque (1984). Pour la file *fork−join* composée de deux serveurs exponentiels, Flatto et Hahn ont donné la fonction génératrice des probabilités d'état [Flatto 84]. Baccelli et Makowski ont proposé des bornes sur le temps de réponse de la file *fork−join* générale [Baccelli 85a]. Des réseaux plus complexes − *série−parallèles* ont été analysés par Baccelli et Massey qui ont calculé les bornes sur les moments du temps de traversée (moyenne et variance) [Baccelli 85b]. Nelson et Tantawi ont donné une approximation simple du temps de réponse de la file *fork−join* [Nelson 85]. Un de nos résultats est identique à leur borne supérieure sur les temps de réponse dans une file à deux stations. D'autres modèles d'exécution des programmes parallèles ont été aussi proposés. Heidelberger et Trivedi ont analysé un système parallèle où les programmes sont divisés en

plusieurs tâches qui s'exécutent de manière indépendante et qui ne nécessitent pas de synchronisation [Heidelberger 82], [Heidelberger 83].

Nous avons proposé une solution approchée de la file *fork−join* et des réseaux complexes série−parallèles fondée sur la décomposition et l'équivalence de réseaux de files d'attente. Notre approche consiste à représenter la file *fork−join* par un serveur équivalent dont le taux de service dépend de l'état. Les paramètres de ce serveur sont obtenus en analysant la file *fork−join* en isolation comme un réseau fermé avec un nombre de tâches allant de un jusqu'à l'infini. Les indices de performances de la file *fork−join* peuvent être ensuite obtenus en analysant le serveur équivalent. La même approche peut être aussi appliquée à l'analyse des performances des systèmes où les programmes ont un graphe de précédence plus complexe par exemple comprenant des tâches en série et parallèles. Dans ce cas on procède récursivement à chaque niveau d'imbrication des primitives *fork−join* à la décomposition et au remplacement d'une file *fork−join* par un serveur équivalent.

Pour plus de détails, nous renvoyons le lecteur à l'article [Duda 87a] et ici nous allons présenter quelques conclusions intéressantes. Pour une exécution parallèle sur deux stations on pouvait espérer dans le cas idéal une accélération de traitement de l'ordre de 2. Si l'on considère une file *fork−join* avec des paramètres distribués exponentiellement (les inter−arrivées et le temps de service), on obtient une accélération de 4/3, ce qui est seulement 2/3 de l'accélération idéale. Cette valeur diminue encore avec l'augmentation du nombre de stations. Ce résultat montre que le coût de la synchronisation peut être très élévé.

Le cas de la file *fork−join* avec deux stations parallèles est un cas simple où nous avons pu obtenir les formules directement comparables avec la file M/M/1. Pour plus de stations ou pour des réseaux représentant des contraintes de précédence complexe, nous avons utilisé l'outil de résolution des modèles markoviens *QNAP2* [Veran 85]. Ce même outil nous a aussi permis de valider les résultats analytiques en les comparant avec la simulation. La précision des résultats est d'environ 10% pour une charge élevée et meilleure pour une charge légère.

I.2.2 Analyse approchée des primitives de synchronisation

La méthode d'analyse des performances de programmes parallèles complexes ayant un graphe de précédence de type série−parallèle décrite dans la section précédente n'est possible que pour un petit nombre de processeurs parallèles. Ceci vient de l'explosion combinatoire du nombre d'états d'un processus de Markov utilisé pour la résolution. Pour un nombre de processeurs plus important nous avons développé une méthode d'analyse approchée basée sur l'équivalence d'espaces d'états [Duda 88a].

Notre raisonnement commence avec la constatation que la file *fork−join* à deux stations parallèles est isomorphe avec la file M/M/1/K (le plus simple réseau fermé à

forme produit) et avec un réseau fermé à deux stations avec blocage (les files des stations ont un nombre de places limité) [Akyildiz 87]. L'équivalence isomorphique de ces trois systèmes veut dire que les espaces d'états de ces systèmes sont identiques, donc les indices de performances le sont aussi. Pour les files à trois stations on peut montrer l'équivalence isomorphique avec un réseau fermé à trois stations avec blocage. Mais, ces deux systèmes n'ont pas de solutions en forme produit. Pour un nombre de stations encore plus élevé, l'équivalence entre la file *fork−join* et un systèmes avec blocage n'est pas vérifiée. Néanmoins, ces équivalences des espaces d'états suggèrent une méthode d'analyse approchée de la file *fork−join*. L'idée consiste à résoudre un réseau de file d'attente fermé dont l'espace d'états est "similaire" à l'espace d'états de la file *fork−join,* au lieu de chercher à trouver les probabilités d'états du processus de Markov caractérisant le serveur équivalent à une file *fork−join*. Le réseau fermé possède une solution en forme produit qui donc peut être calculée efficacement. La similarité signifie que les espaces d'états des réseaux ont des nombres d'états et des taux de transitions presque identiques. Cette similarité des espaces d'états implique que les performances, et en particulier le débit de la file qui nous intéresse pour trouver les paramètres du serveur équivalent, sont similaires. On a utilisé un réseau fermé à forme produit qui est composé de même stations que la file *fork−join* et qui contient le nombre de client choisi de telle façon que le nombre d'états dans les espaces d'états soit proche. L'analyse du réseau fermé est ensuite effectuée à l'aide de l'outil *QNAP2*. La précision de l'approximation est bonne pour la charge légère et se détériore avec l'augmentation de la charge (mais reste au niveau de 15%).

I.2.3 Performances de programmes parallèles

Les deux modèles que nous venons de présenter permettent d'analyser et de comprendre le comportement complexe des programmes parallèles qui sont en compétition pour obtenir des ressources comme des processeurs et qui en même temps subissent des délais de synchronisation. Ces modèles ne prennent pas en compte le coût des communications, ce qui reflète bien les caractéristiques des machines multiprocesseurs ou des systèmes répartis où l'on peut négliger ce coût.

Pour bien comprendre l'effet de la communication sur les performances de programmes parallèles, nous avons étudié le problème de partitionnement optimal d'un programme parallèle en plusieurs tâches pouvant être traitées en parallèle pour diminuer le temps d'exécution du programme [Duda 88b]. Théoriquement, on peut créer le plus grand nombre possible de tâches pour diminuer au maximum le temps d'exécution (ce nombre est néanmoins limité par le niveau de parallélisme inhérent au programme ou par une unité indivisible d'exécution, par exemple une procédure ou une instruction). Cependant, si la création d'une tâche, le transfert de données initiales et de résultats ne sont pas instantanés, ils induisent un coût de

communication qui varie selon le type de réseau d'interconnexion. Dans ce cas, l'augmentation du nombre de tâches peut avoir un effet pervers, par exemple la diminution du temps global d'un programme parallèle. Apparemment, il existe un compromis entre le parallélisme et la communication que nous voulions étudier. Nous avons développé des bornes sur le temps d'exécution des tâches parallèles et nous avons trouvé le nombre optimal de tâches qui maximise les indices de performances de programmes parallèles (*accélération* et *qualité*).

Le problème du partitionnement optimal des programmes apparaît dans les systèmes parallèles aussi bien que dans les systèmes répartis. Ces deux types de systèmes ont des caractéristiques différentes − le rapport vitesse de processeurs/ vitesse de communication est beaucoup plus grand dans les systèmes parallèles que dans les systèmes répartis. Ceci est utilisé pour négliger le coût de communications dans la modélisation des systèmes parallèles. Mais, dans les deux types de systèmes, la granularité des tâches peut être rendue si fine que les coûts de communication ne peuvent pas être négligés. C'est surtout le cas des systèmes à mémoire répartie comme l'hypercube ou les réseaux locaux de microprocesseurs. Le problème est de moindre importance pour les multiprocesseurs à mémoire partagée, mais il apparaît aussi dans ces systèmes pour lorsque le nombre de processeurs est important.

Le même problème a été déjà étudié dans des contextes différents. Robinson a analysé le temps d'exécution des algorithmes que l'on peut décomposer en k tâches parallèles sur un multiprocesseur asynchrone [Robinson 79]. Il a supposé que la mémoire partagée est utilisée pour la communication et il a analysé la décomposition optimale des algorithmes. Reed a considéré des structures différentes d'interconnexion de processeurs [Reed 83]. Les tâches peuvent être créées dynamiquement et le temps d'exécution dépend de la topologie du réseau et du coût des communications. Son étude de simulation fait apparaître un compromis entre le parallélisme et la communication. Lint et Agrawala ont discuté des problèmes de communication dans la conception d'algorithmes parallèles [Lint 88]. Ils ont présenté plusieurs exemples qui montrent l'influence du coût de communication sur les performances de programmes parallèles. Ils ont comparé le compromis entre le parallélisme et la communication au compromis entre le temps et l'espace mémoire qui caractérise des algorithmes séquentiels. Axelrod a analysé les performances d'algorithmes parallèles qui font usage de barrières de synchronisation [Axelrod 86]. Il a conclu que si les barrières ont un coût qui ne peut pas être négligé, il existe un nombre optimal de tâches. Kruatrachue et Lewis ont déterminé le grain optimal de tâches (c'est à dire, la taille de tâches) dans le cas de la connexion complète de processeurs et de temps d'exécution déterministes [Kruatrachue 88].

Pour comprendre le problème, considérons un modèle déterministe d'exécution d'un programme parallèle suivant. On va supposer que le temps d'exécution d'un programme sur un processeur est T. Le programme est décomposé en n tâches qui sont exécutées sur des processeurs différents (nous supposons que l'on dispose

d'autant de processeurs qu'il y a de tâches). Le coût de communication associé à l'exécution d'une tâche parallèle (le temps de sa création, le temps de transfert des paramètres initiaux et des résultats) sera dénoté par *C*. Nous supposons que les tâches ne communiquent pas entre elles à part la création et le passage des résultats. Le coût global de communication d'un programme dépendra de la structure du réseau d'interconnexion.

Le temps d'exécution de *n* tâches sur *n* processeurs t_n peut être exprimé comme une fonction du coût de communication de la façon suivante :

$$t_n = \frac{T + g(n)C}{n} \qquad (1)$$

où *g(n)* représente le nombre de communications qui dépend de la structure du réseau d'interconnexion donnée. Pour les quatre structures que nous avons considérées, cette fonction est la suivante :

anneau

$$g_{anneau}(n) = \frac{n(n-1)}{2}$$

arbre

$$g_{arbre}(n) = [(n+1)\log_2(n+1) - 2n]$$

hypercube d'ordre *k*

$$g_{hypercube}(n) = \frac{[n(k-1)+1]\log_k[n(k-1)+1] - kn}{k-1}$$

connexion complète

$$g_{c-c}(n) = n-1$$

Pour la dérivation de ces formules le lecteur peut consulter l'article [Duda 88a]. On se contentera de remarquer que le nombre de communications est $O(n^2)$ pour l'anneau, $O[n\log(n)]$ pour l'arbre et l'hypercube et $O(n)$ pour la connexion complète.

La Fig. 1.3 présente le temps d'exécution en fonction du nombre de tâches pour différentes structures d'interconnexion. On peut remarquer que seul le temps d'exécution pour l'anneau a un minimum bien distinct. Un minimum existe aussi pour l'arbre et l'hypercube, mais la fonction est très plate aux environs du minimum. Il n'existe pas de minimum pour la connexion complète. On peut observer que la diminution du temps d'exécution est d'abord très grande et devient ensuite légère.

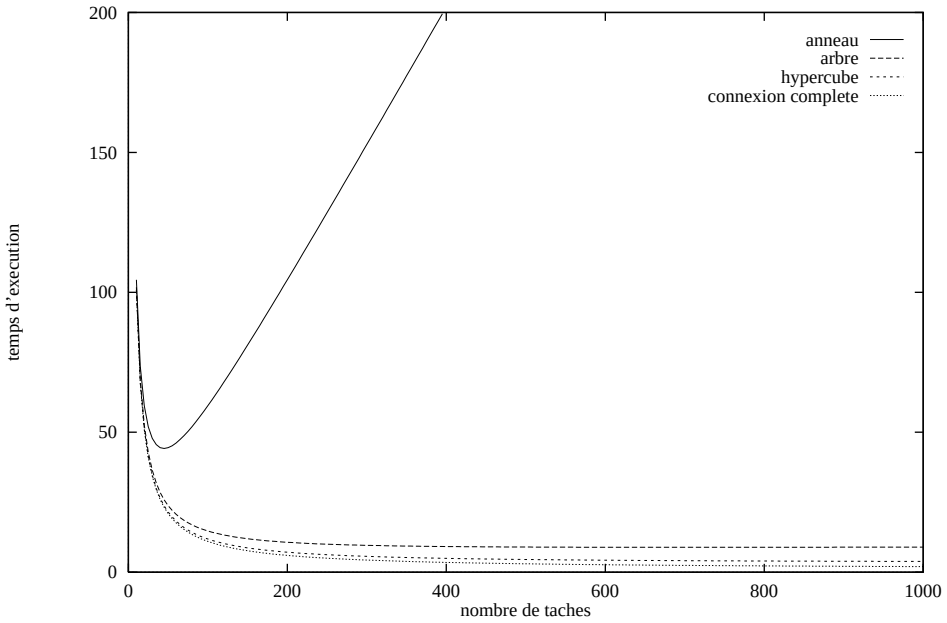


Fig. 1.3 : Le temps d'exécution en fonction du nombre de tâches

Pour quantifier les performances, nous allons considérer les indices suivants (P_n désigne la somme des temps de traitement de tâches et des temps de communications pendant l'exécution de n tâches) :

accélération

$$S_n = \frac{t_1}{t_n}$$

efficacité

$$E_n = \frac{t_1}{nt_n}$$

redondance

$$R_n = \frac{P_1}{P_1}$$

qualité

$$Q_n = \frac{S_n E_n}{R_n}$$

L’accélération est un indice les plus utilisés qui montre de combien une exécution parallèle est plus rapide qu’une exécution séquentielle.

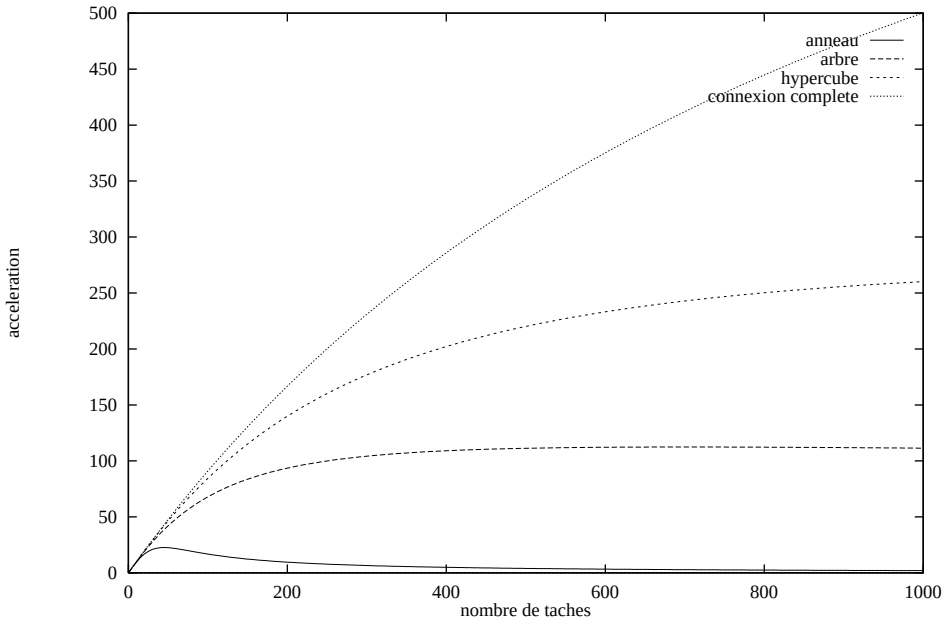


Fig. 1.4 : L’accélération en fonction du nombre de tâches

La Fig. 1.4 présente l’accélération en fonction du nombre de tâches pour des différentes structures d’interconnexion. On observe les mêmes phénomènes que sur la figure précédente − il existe un maximum distinct pour l’anneau, des maxima plats pour l’arbre et l’hypercube et pas de maximum pour la connexion complète. Ces figures suggèrent que dans le cas de l’arbre, de l’hypercube et de la connexion complète il serait préférable de limiter le nombre de tâches à une valeur plus petite que l’optimale parce que les indices de performance ne s’améliorent que légèrement au dépens de l’augmentation très grande du nombre de tâches. Un autre indice de performance permet d’exprimer quantitativement cette observation.

La Fig. 1.5 présente la qualité en fonction du nombre de tâches pour des différentes structures d’interconnexion. On peut remarquer que les maxima sont marqués plus nettement que dans le cas de l’accélération et qu’ils sont atteints pour un nombre de tâches plus petit. En outre, il existe un maximum pour la connexion complète. Les maxima sont placés aux endroits qui correspondent à la courbure en forme de "genou" visible sur les courbes dans la Fig. 1.3. Ils marquent l’endroit où la diminution du temps d’exécution devient plus petite en fonction du nombre de tâches. Ce résultat est important dans le cas de systèmes parallèles multiprogrammés où plusieurs programmes parallèles peuvent être simultanément en exécution. Dans ce cas, il peut être intéressant d’allouer un nombre restreint de processeurs à plusieurs programmes plutôt que d’allouer tous les processeurs à un seul.

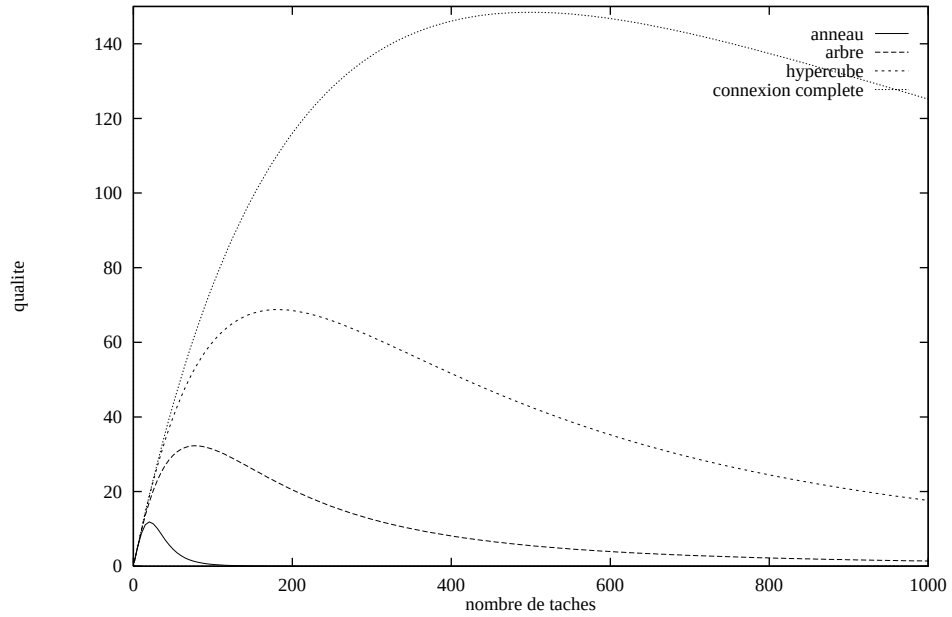


Fig. 1.5 : La qualité en fonction du nombre de tâches

Dans l'article [Duda 88a], nous avons donné des formules pour le nombre optimal de tâches quand ceci était possible (optimal au sens de maximalisation de l'accélération et de la qualité). Dans le cas contraire, nous avons donné des équations qu'il faut résoudre numériquement. On peut reprocher au modèle déterministe sa rigidité et le fait qu'il ne permet pas de prendre en compte la variabilité des temps de communications et de traitement de tâches. Pour pallier ces défauts, nous avons aussi considéré un modèle probabiliste où ces temps sont des variables aléatoires. Ce modèle permet de prendre en compte les délais de synchronisation. Du modèle probabiliste, nous avons dérivé une borne supérieure pour les temps d'exécution. Les résultats du modèle sont un peu différents de ceux du modèle déterministe − dans le cas des quatre structures d'interconnexion, il existe bien un minimum du temps d'exécution. Ce minimum est bien distinct pour l'anneau et très plat pour les autres structures. Comme pour le modèle déterministe, nous avons aussi considéré la qualité comme l'indice de performances. Pour les quatre structures d'interconnexion, les valeurs optimales du nombre de tâches sont plus petites que celles prévues par le modèle déterministe.

I.2.4 Stratégies de duplication d'objets

Récemment, dans le cadre du projet Guide décrit dans le Chapitre <sr>, nous avons été amené à étudier le problème de la duplication d'objets. Un objet est une unité d'encapsulation et de répartition qui peut être partagée par plusieurs programmes. Les systèmes à objets comme Guide cherchent à exploiter le parallélisme et

la répartition pour améliorer les performances, la fiabilité et la disponibilité. La technique de base pour atteindre ces objectifs est la duplication qui peut être mise en œuvre de différentes manière. Le principe consiste à avoir plusieurs copies d'un même objet qui sont gérées par un algorithme de maintien de la cohérence. La sémantique du maintient de la cohérence varie de la cohérence forte (toutes les copies sont identiques à tout instant) aux différentes variantes de la cohérence faible (les copies peuvent différer).

La duplication d'objets a deux objectifs. Le premier est d'améliorer les performances, ce qui peut être obtenu en exécutant des opérations sur une copie locale d'un objet. Le coût de communication nécessaire à une invocation distante est de ce fait éliminé. En outre, un objet dupliqué peut être utilisé en parallèle ce qui élimine le goulot d'étranglement que peut constituer un objet non dupliqué. Le deuxième objectif de la duplication est l'amélioration de la fiabilité et de la disponibilité. Si des copies multiples existent dans un système, un objet peut être utilisé même si certains sites ne sont pas opérationnels.

Le maintien de la cohérence de copies multiples d'un objet peut bénéficier de l'usage d'un protocole de diffusion fiable et ordonnée. Ce type de protocole garantit l'acheminement fiable de messages dans le même ordre vers plusieurs destinations. Plusieurs travaux consacrés à la duplication d'objets s'appuient sur une couche du protocole de diffusion fiable et ordonnée [Bal 89]. On peut trouver dans la littérature l'argument suivant contre l'emploi des protocoles de diffusion pour la duplication d'objets : ces protocoles sont coûteux parce qu'ils ont besoin de plusieurs tours d'échanges de messages ou de long délais avant de délivrer un message à destination. Il existe aussi une opinion répandue affirmant que ces protocoles engendrent une charge importante pour le système − chaque message doit être traité par tous les sites du système, même si certains sites ne sont pas concernés. Néanmoins, les protocoles de diffusion développés récemment, comme celui d'Amoeba [Kaashoek 91], présentent de bonnes performances de sorte qu'il devient intéressant de les utiliser pour la duplication d'objets. Le travail de Bal, Kaashoek et Tanenbaum présente une discussion de stratégies différentes de duplication d'objets [Bal 89]. Leur but est d'améliorer les performances des applications parallèles. Ils discutent des avantages et des inconvénients de chaque stratégie et présentent la mise en œuvre et les mesures d'une stratégie particulière − la *mise−à−jour parallèle* qui emploie un protocole de diffusion pour maintenir la cohérence forte entre des copies d'objet. Dans le travail [Duda 93], nous avons présenté une analyse théorique des performances de différentes stratégies de duplication d'objets qui font usage des protocoles de diffusion. Ce travail étend les résultats de Bal, Kaashoek et Tanenbaum en analysant et quantifiant les effets de facteurs différents comme la taille des objets, le coût des communications et le taux de lecture sur les performances des stratégies.

Dans la suite de la présentation, nous allons supposer que les opérations sur les objets dupliqués sont à exécuter de manière atomique. Ceci garantit la sérialisation

des exécutions : si deux opérations sont exécutées simultanément, le résultat est le même que si une opération est exécutée avant l'autre, mais l'ordre n'est pas déterminé. Nous ne considérons pas ici les modèles de transactions atomiques qui permettent de rendre indivisibles des séquences d'opérations. Pour la plupart des applications réparties que nous considérons pour Guide, cette propriété est suffisante.

L'exécution atomique des opérations sur des objets dupliqués peut être réalisée à l'aide d'un protocole de diffusion fiable et ordonné qui garantit les trois propriétés suivantes :

ordre
: les messages sont délivrés à toutes les destinations dans le même ordre,

atomicité
: soit tous les membres d'un groupe reçoivent un message, soit aucun ne le reçoit,

fiabilité
: les deux propriétés précédentes sont garanties même en présence de pannes de sites.

Ces propriétés facilitent le maintient de la cohérence de copies multiples − les mises−à−jour peuvent être diffusées à toutes les copies qui les reçoivent dans le même ordre. Si les copies sont identiques au départ, l'application de mêmes opérations dans le même ordre permet de maintenir leur cohérence.

Le protocole de diffusion peut être utilisé de différentes manières pour la duplication d'objets. Nous avons identifié les trois stratégies de duplication suivantes :

- copie primaire avec l'invalidation des copies secondaires,
- copie primaire avec mises−à−jour des copies secondaires,
- mises−à−jour parallèles des copies cohérentes.

Parfois on utilise le terme *duplication passive* pour designer les deux premières méthodes et le terme *duplication active* pour la première. Nous présentons les détails de chaque technique ci−dessous.

Invalidation des copies secondaires

Dans cette stratégie, il existe une copie d'objet appelée *copie primaire* sur laquelle on applique toutes les opérations de *mise−à−jour*. Les opérations qui ne modifient pas l'état de l'objet (*lecture*) sont exécutées sur les copies locales, appelées *copies secondaires,* dont un exemplaire se trouve sur chaque site. La stratégie est illustrée dans la Fig. 1.6. Les opérations de mise−à−jour sont diffusées à tous les sites à l'aide du protocole de diffusion fiable et ordonné pour qu'une copie secondaire puisse devenir primaire si le site primaire tombe en panne. L'opération de mise−à−jour invalide une copie secondaire et la première opération de lecture entraîne un chargement du nouvel état de l'objet à partir de la copie primaire.

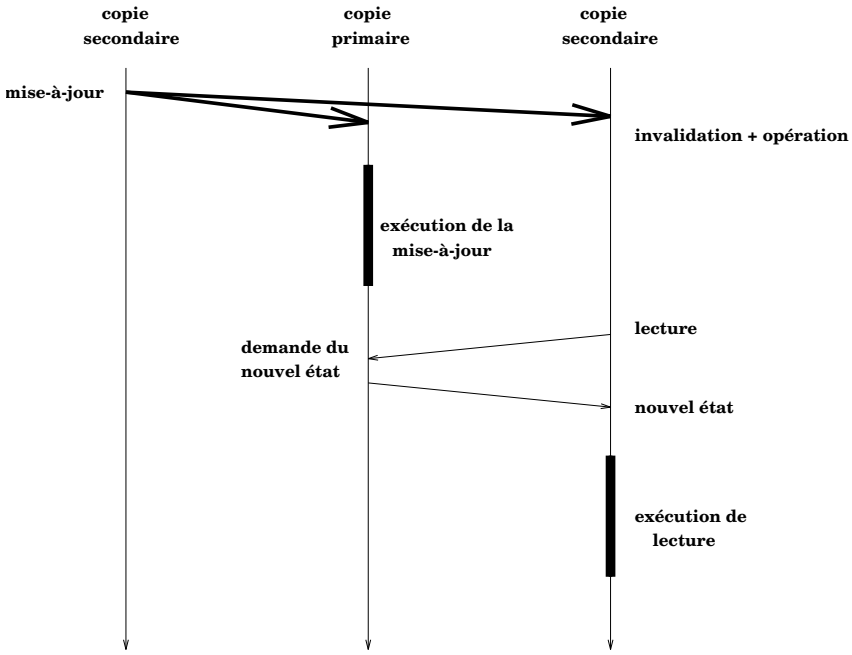


Fig. 1.6 : Invalidation des copies secondaires

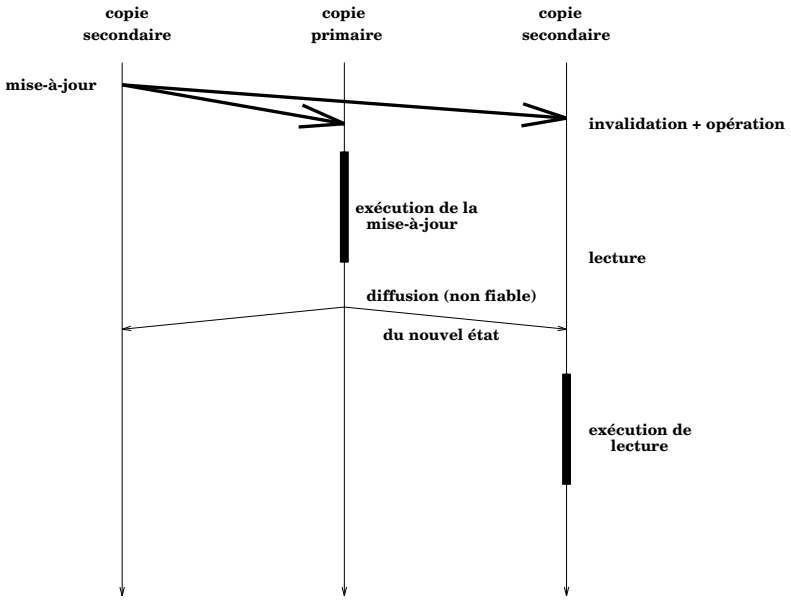


Fig. 1.7 : Mise−à−jour des copies secondaires

Mises−à−jour des copies secondaires

Cette stratégie est similaire à la précédente : il existe des copies secondaires et une copie primaire, et les opérations de mise−à−jour sont diffusées à tous les sites à l'aide du protocole de diffusion fiable et ordonné. La différence consiste à diffuser le nouvel état de l'objet à tous les sites après chaque opération de mise−à−jour. Pour cela, il suffit d'utiliser un protocole de diffusion sans les propriétés fortes de fiabilité et d'ordonnancement, parce qu'en cas d'erreur ou de panne de site, les sites secondaires peuvent toujours demander au site primaire le chargement du nouvel état de l'objet. Une opération de lecture qui arrive après l'invalidation d'une copie secondaire par une opération de mise−à−jour doit attendre la réception du nouvel état de l'objet. La stratégie est illustrée dans la Fig. 1.7.

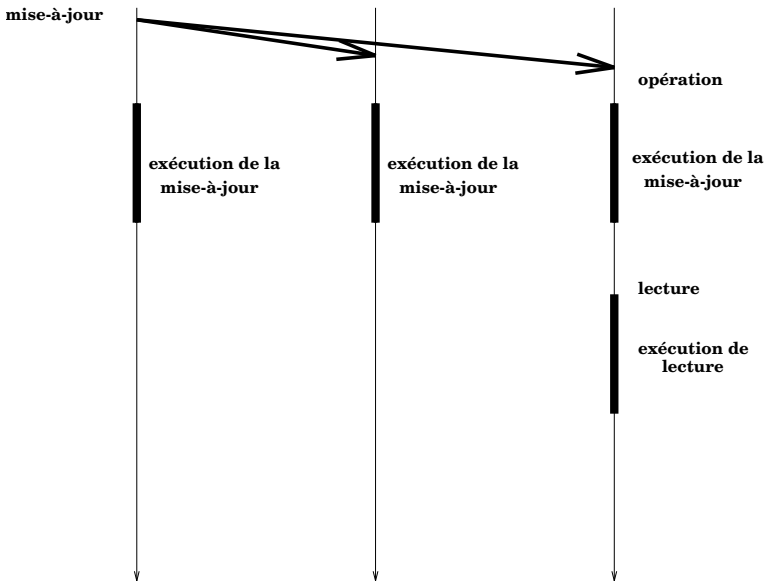


Fig. 1.8 : La stratégie des mises−à−jour parallèles

Mises−à−jour parallèles

Dans cette stratégie, il n'y a pas de copie primaire ni de copies secondaires. Toutes les copies sont traitées de la même façon − les opérations de mise−à−jour sont diffusées à tous les sites à l'aide du protocole de diffusion fiable et ordonné, et exécutées sur tous les sites. Au départ, toutes les copies sont identiques et l'application de mêmes opérations dans le même ordre maintient la cohérence de copies. La stratégie est illustrée dans la Fig. 1.8.

Cette stratégie est souvent considérée comme coûteuse dans la littérature, parce que toutes les opérations de mise−à−jour sont exécutées sur tous les sites. Mais, elle peut être performante dans le cas d'opérations courtes sur de gros objets − on ne transfère que de cours messages d'invocation et on évite de transférer l'état de l'objet. Intuitivement, la stratégie d'invalidation des copies secondaires parait la meilleure quand le taux de mises−à−jour est élevé. Dans ce cas, le transfert de l'état de l'objet n'a lieu qu'après plusieurs mises−à−jour consécutives suivies d'une lecture, tandis que dans la stratégie de la mise−à−jour des copies secondaires, l'état de l'objet est transféré après chaque mise−à−jour. En revanche, cette dernière stratégie peut donner de meilleurs résultats quand les opérations de lecture sont très fréquentes, parce que la diffusion du nouvel état après une mise−à−jour est moins coûteuse que le

chargement du nouvel état dans la stratégie de l'invalidation des copies secondaires. La stratégie de mise−à−jour parallèle a l'avantage de répartir la charge uniformément sur tous les sites, tandis que dans les deux autres stratégies le site où se trouve la copie primaire est plus chargé que les autres sites, ce qui peut limiter les performances globales du système. En cas de charge élevée, la stratégie de mise−à−jour parallèle peut avoir de meilleures performances, parce qu'il n'y a pas de coûts de communication supplémentaires pour maintenir les copies secondaires. En outre, dans cette stratégie, la disponibilité est obtenue sans coût supplémentaire − quand un site tombe en panne, les copies sur les autres sites sont disponibles immédiatement. Dans les autres stratégies, le système doit passer par une phase de reconfiguration, si le site primaire tombe en panne.

Pour comparer les performances des stratégies respectives, il nous faut des fonctions de coût pour chaque stratégie et un modèle qui permet d'obtenir les indices de performances en fonction de ces coûts. Dans le contexte des algorithmes de mémoire partagée répartie, Stumm et Zhou ont utilisé un modèle simple pour comparer des algorithmes différents [Stumm 90]. Ils ont choisi comme indice de performance le coût moyen associé à un accès aux données dans le système. Ce coût est calculé pour chaque algorithme en fonction du coût de communication et correspond à la durée du traitement de message C, le temps de transfert sur réseau étant négligé. Leurs résultats montrent que les algorithmes qui sont équivalents à nos stratégies d'invalidation et de mise−à−jour des copies secondaires ont de performances supérieures aux algorithmes où les mises−à−jour sont effectuées en parallèle. Cette conclusion est l'effet du modèle simpliste et de la fonction du coût choisis. En effet, leur modèle favorise les algorithmes qui n'utilisent pas de protocoles de diffusion. Par exemple dans le modèle de mises−à−jour parallèles, le coût d'une mise−à−jour sur N sites est $(N+2)C$ − une mise−à−jour est envoyée à un séquenceur $2C$ et ensuite la mise−à−jour est envoyée à tous les sites NC. Ce calcul pénalise un protocole de diffusion parce que le coût de la diffusion est cumulé pour tous les sites. Si le temps de réponse est choisi comme un indice de performance, alors dans les conditions de charge légère, les sites traitent le message diffusé en parallèle, donc le coût est seulement de $4C$. Si la charge est élevée, les phénomènes d'attente apparaissent et leurs effets ne peuvent être prédits qu'à l'aide d'un modèle de files d'attente. Pour ces raisons, nous avons choisi d'utiliser ce type de modèle pour comparer les stratégies de duplication d'objets.

Un modèle exact de stratégies de duplication comporterait un réseau de files d'attente avec des arrivées groupées et des transitions forcées correspondants à certaines actions (par exemple, le chargement du nouvel état d'un objet). Même si toutes les variables en question sont distribuées exponentiellement, le réseau n'a pas de solution analytique. Nous avons construit un modèle approché, basé sur un réseau ouvert avec plusieurs classes de clients où les demandes de service des clients sont spécifiques à chaque stratégie à comparer. Nous ne rentrons pas dans les détails du

modèle (on peut pour cela consulter [Duda 93]), et présentons seulement l'idée générale de la modélisation.

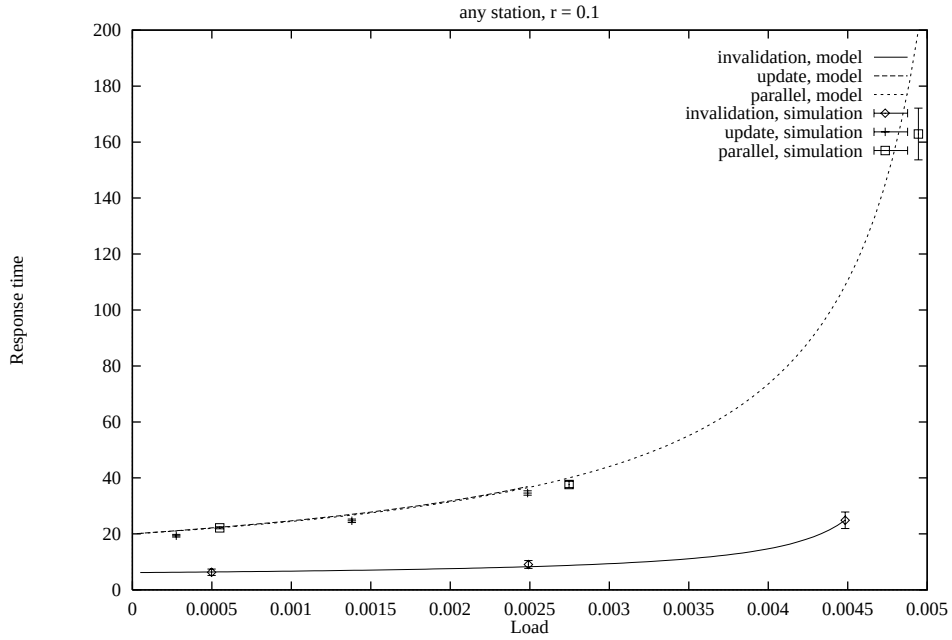


Fig. 1.9 : Temps de réponse pour le taux de lecture de 0.1, site secondaire

Nous supposons que les demandes d'exécution des opérations sur un objet arrivent sur un site selon un processus de Poisson. Une opération est une lecture avec une probabilité appelée le *taux de lectures*. Les temps des opérations sont distribuées exponentiellement et nous supposons que le coût de communication est dominé par le temps du traitement de messages (le temps de transfert sur réseaux est négligé). Comme nous nous intéressons à une comparaison des stratégies, la diffusion des opérations de mise−à−jour qui est présente dans les trois stratégies n'est pas prise en compte dans le modèle. Pour représenter les actions qui se déroulent dans les différentes stratégies, le modèle utilise des classes différentes de clients. Par exemple, dans le modèle de l'invalidation des copies secondaires, il y a trois classes de clients − *mises−à−jour*, *lectures* et *chargements*. Le site primaire exécute les mises−à−jour provenant de tous les sites, les lectures locales, et traite les demandes de chargement émanant des sites secondaires. Les paramètres du modèles sont : le taux d'arrivée des opérations, le taux de lecture, les temps d'exécution des opérations, le coût de communication et le nombre de sites.

Pour donner une idée sur les performances relatives des stratégies, nous allons présenter quelques résultats du modèle. Tout d'abord, comme le modèle est approché, nous avons comparé ses résultats avec la simulation. La simulation a été programmée à l'aide du logiciel *QNAP2* [Veran 85]. Les paramètres du modèle sont les suivants :

Paramètre	*Valeur*
traitement de message court	1 ms
traitement de message long	20 ms
opération de lecture	20 ms
opération de mise−à−jour	20 ms
nombre de sites	10

Les Fig. 1.9, Fig. 1.10, Fig. 1.11 et Fig. 1.12 présentent les performances des stratégies pour les différents taux de lectures. Les résultats de la simulation sont présentés avec les intervalles de confiance de 95%. La comparaison avec la simulation montre une bonne précision du modèle. Les différences les plus notables existent pour la charge élevée, la condition pour laquelle il est difficile d'obtenir les résultats de simulation stables. Pour les valeurs de paramètres choisis qui correspondent à un équilibre entre le coût d'exécution par rapport au coût de communication, on peut constater que la stratégie de mises−à−jour en parallèle présente les meilleures performances − elle a le temps de réponse le plus petit et elle peut supporter la charge la plus élevée. Les deux autres stratégies sont limitées par les performances du site primaire. Pour le taux de lecture élevé et la charge légère, la stratégie d'invalidation est meilleure que la mise−à−jour des copies secondaires. Quand la charge monte, le phénomène inverse peut être constaté.

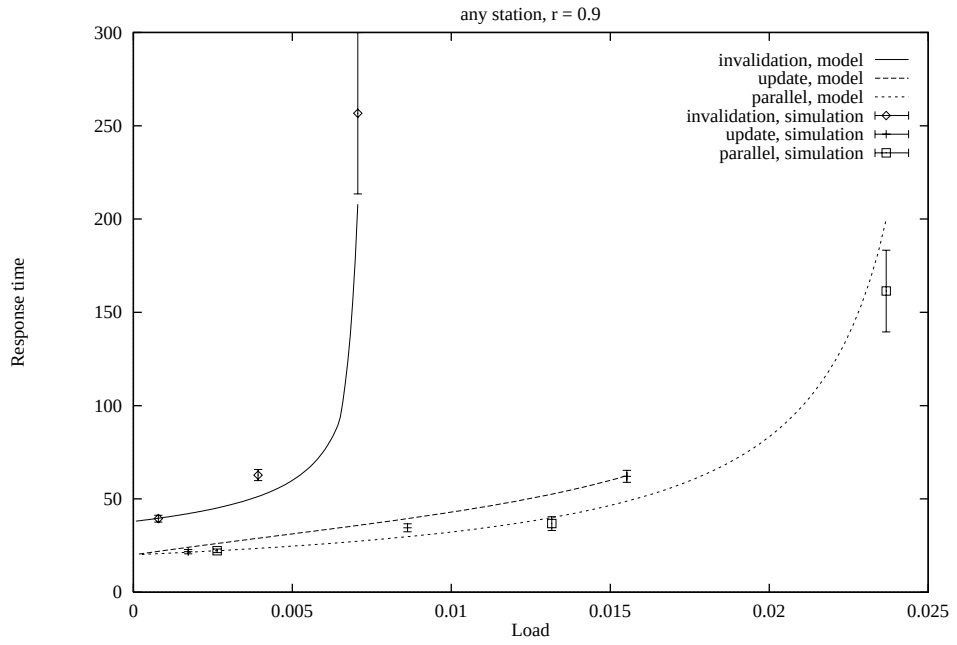


Fig. 1.10 : Temps de réponse pour le taux de lecture de 0.9, site secondaire

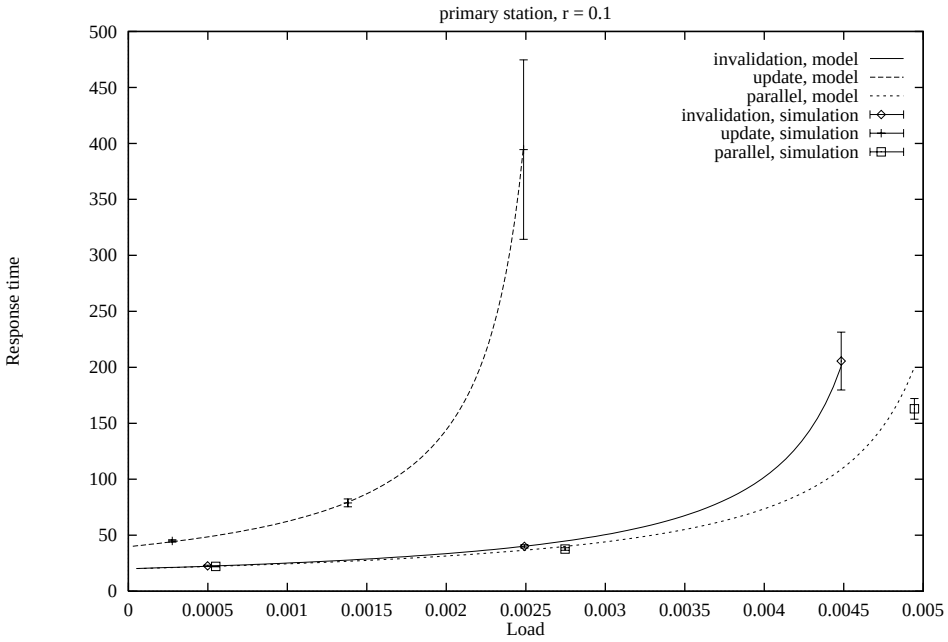


Fig. 1.11 : Temps de réponse pour le taux de lecture de 0.1, site primaire

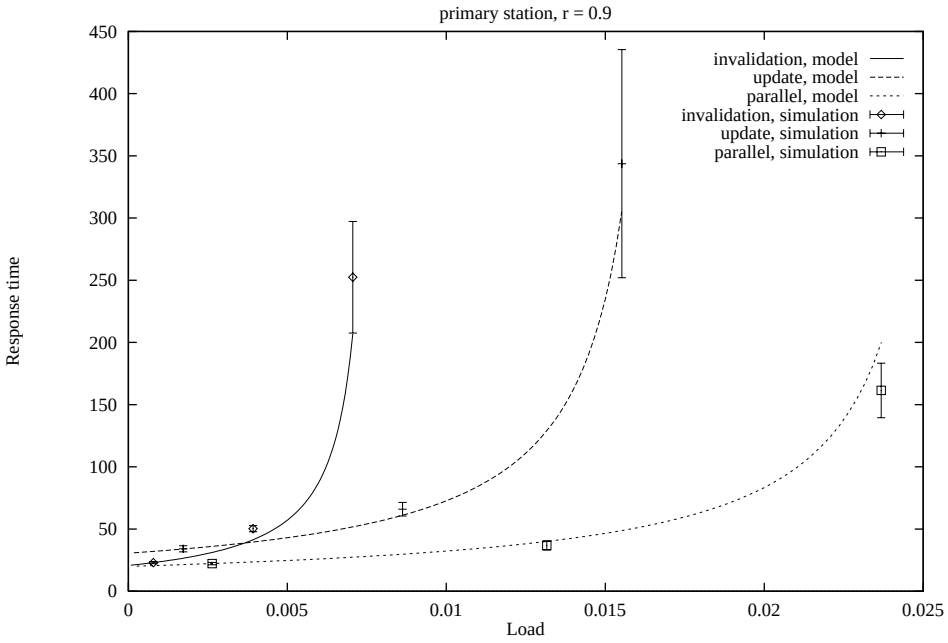


Fig. 1.12 : Temps de réponse pour le taux de lecture de 0.9, site primaire

On peut trouver d'autres comparaisons de performances dans l'article [Duda 93]. La stratégie de mises−à−jour en parallèle présente les meilleures performances pour la charge moyenne et élevée indépendamment des autres paramètres. Dans les

conditions de la charge légère et pour le taux de lectures petit, la meilleure stratégie est l'invalidation des copies secondaires. Quand les temps d'exécution sont plus grands que les temps de communication, la stratégie de mise−à−jour des copies secondaires est la meilleure pour la charge légère. Les tables ci−dessous résument la comparaison.

exécution ≈ communication	*charge légère*	*charge élevée*
taux de lectures petit	Invalidation	Parallèle
taux de lectures grand	Parallèle ou Mise−à−jour	Parallèle

exécution > communication	*charge légère*	*charge élevée*
taux de lectures petit	Invalidation	Parallèle
taux de lectures grand	Mise−à−jour	Parallèle

Cette comparaison montre qu'il n'y a pas la meilleure stratégie pour toutes les conditions. Mais, pour les conditions les plus plausibles (taux de lectures élevé), la stratégie de mises−à−jour en parallèle est la meilleure si les temps d'exécution sont petits par rapport aux temps de communication. Pour des opérations longues, la mise−à−jour des copies secondaires est meilleure.

I.3 Mesures de performances des systèmes répartis

Les sections précédentes montrent la difficulté d'évaluation des performances d'un système réparti ou parallèle. Déjà dans un système centralisé, il est souvent difficile de prendre en compte dans un modèle toutes les interactions présentes dans le système. Un modèle analytique très détaillé devient insoluble et un modèle de simulation peut être long à mettre au point et à exécuter. Aussi, il faut connaître au moment de l'établissement du modèle tous les facteurs qui influencent les performances. Dans un système réparti, il est encore plus difficile d'établir, puis de résoudre un modèle satisfaisant − nous avons vu que les modèles présentés précédemment étaient partiels et ne se concentraient que sur un aspect de performances. Les interactions dans un système réparti sont plus nombreuses, souvent mal maîtrisées et même inconnues. Les mesures ont donc une place importante dans l'évaluation des performances. Les mesures de tels systèmes restent plus complexes que dans le cas centralisé pour plusieurs raisons [Joyce 87]:

- un système réparti possède plusieurs centres de contrôle, ce qui rend les techniques séquentielles habituelles non immédiatement applicables ;

- les délais de communication et l'absence d'horloge commune rendent difficile, voire impossible, la détermination de l'état global du système à un instant donné ;
- les systèmes répartis fondés sur la communication asynchrone entre processus sont par essence *non−déterministes* − il est donc difficile de reproduire des erreurs et de tester des situations possibles mais peu probables.

Les moyens de base pour l'instrumentation d'un système dans un but de mesures sont les *sondes matérielles* et *logicielles*. Les premières sont des équipements connectés physiquement au composant à mesurer et qui permettent d'en connaître l'état. On appelle *moniteur matériel* un outil de mesure qui utilise les sondes matérielles. Les secondes sont des instructions rajoutées au code du système à mesurer (système d'exploitation ou application) dans le but de recueillir les informations nécessaires, et l'on nomme *moniteur logiciel* un outil de mesure qui les utilise. Enfin, un *moniteur hybride* met en œuvre à la fois des sondes matérielles et logicielles.

Les moniteurs matériels ont l'avantage d'être rapides et précis. Ils ont été fréquemment utilisés pour l'évaluation des protocoles dans les réseaux [Cellary 84], en particulier sous forme d'"espions de ligne" dont CERBÈRE est un exemple [Ansart 82]. Les réseaux locaux à diffusion se prêtent également bien à l'utilisation de moniteurs matériels, sous la forme d'une station supplémentaire du réseau, puisque toutes les stations voient passer l'ensemble du trafic. Dans ce cas, l'observation n'en perturbe absolument pas le fonctionnement. Cette méthode a été appliquée par exemple au réseaux *Ethernet* [Shoch 80].

Les deux inconvénients des moniteurs matériels sont leur manque de souplesse et leur incapacité à observer des données reliées au logiciel (par exemple, le temps passé dans une partie d'un programme), inconvénients que ne présentent pas les moniteurs logiciels. Ceux−ci sont donc largement utilisés pour les mesures dans les système répartis, en particulier lorsque les configurations étudiées sont variables. Ils ont été appliqués aux domaines aussi variés que des architectures multi−processeurs [Gehringer 82], des noyaux d'exécution répartis [Cheriton 83] ou des programmes répartis [Miller 84]. L'enjeu ici est de minimiser la perturbation, inévitable puisque des instructions sont ajoutées au logiciel, due à la mesure.

Pour expérimenter des outils de mesure et pour faciliter la programmation des applications réparties à mesurer nous avons mis en place un projet de construction d'une maquette de système réparti − *Epsilon* [Bernard 87], [Bernard 89a], [Bernard 89b]. *Epsilon* est un ensemble matériel et logiciel permettant l'évaluation des performances, au moyen de mesures, d'algorithmes répartis (ou plus généralement d'application réparties), s'exécutant en parallèle sur des machines hétérogènes y compris des microprocesseurs ou stations de travail, reliées par un réseau local. Le projet a deux aspects : il comprend d'une part un ensemble de primitives de haut

niveau pour faciliter l'écriture des applications réparties, et d'autre part, un ensemble d'outils logiciels qui permettent l'observation et les mesures des applications. Le premier aspect est décrit dans le chapitre II et nous présentons ici l'aspect mesures.

Pour *Epsilon*, la spécification de l'outil de mesure a pris en considération les éléments suivants :

- le niveau d'observation visé est élevé − algorithme ou application réparties ; ainsi, si nous cherchons à mesurer le délai de transmission d'un message, il s'agit du délai de bout en bout (de niveau application à niveau application), puisque c'est ce délai qui va conditionner le comportement de l'algorithme ou de l'application.
- tout logiciel doit être portable, donc on exclut toute modification au noyau des systèmes d'exploitation des machines hôtes.
- de même que les systèmes d'exploitation et les mécanismes de communication entre processus distants sont cachés au programmeur d'application répartis, les détails des mécanismes d'observation et de mesures doivent lui être invisibles.

Notre choix pour l'outil de mesures s'est donc porté sur un moniteur logiciel. Les données recueillies par les moniteurs logiciels peuvent être exploitées en temps réel ou en temps différé. L'exploitation en temps réel est nécessaire si l'observateur veut agir sur le comportement du système pendant l'exécution, au vu de données immédiates, ou si, plus simplement, on souhaite visualiser certains indices de performances pendant l'exécution. C'est le cas du système Jade [Joyce 87] et des metteurs au point interactifs [Harter 85]. Cette approche souffre du problème de la perturbation du système observé et pour cette raison nous avons choisi pour *Epsilon* d'exploiter les données en temps différé.

Les données recueillies au moment de l'occurrence d'un événement constituent un enregistrement appelé *trace*. Les modifications à apporter au logiciel pour générer les traces peuvent se placer à différents niveaux : dans le noyau, dans les primitives de communication inter−processus ou dans les sources de l'application. Si l'on s'intéresse aux couches hautes du logiciel, et si le programmeur souhaite observer des événements à l'intérieur des frontières d'un processus, la méthode qui convient est le placement des modifications aux niveaux des sources de l'application. Pour résumer, la méthode de collecte de mesures dans *Epsilon* consiste en l'utilisation d'un moniteur logiciel dirigé par événements permettant de générer, dans les sources des programmes, des traces d'exécution destinées à être analysées en temps différé.

En général, les modification au code source des programmes d'application sont effectuées "manuellement" par le programmeur (cf. par exemple *METRIC* [MacDaniel 77]). Dans *Epsilon*, elles sont réalisées de façon invisible pour le programmeur

par un outil logiciel, *TRASS*, qui est construit à partir du mécanisme de *déviation* [Haddad 88]. Lorsqu'un appel de procédure a lieu, le code de la procédure n'est pas exécuté immédiatement, mais le contrôle passe d'abord à un module enregistreur, qui réalise lui−même l'appel de procédure, puis reprend le contrôle, avant de retourner. C'est le module enregistreur qui génère les traces et assure l'invisibilité de la déviation. Le programmeur dispose d'un *mini−langage* qui lui permet de décrire, dans les fichiers de spécifications, quelles fonctions doivent être déviées, dans quels modules, quels arguments sont passés etc. La collecte de mesures peut être activée ou désactivée avant, ou pendant l'exécution d'un module. Un module de remplacement génère automatiquement, à partir des fichiers source du programme et des fichiers de spécifications des observations, des fichiers source modifiés qui, une fois compilés, permettent l'exécution du programme avec génération de traces.

L'un de problèmes majeurs rencontrés a été la difficulté d'exploiter les traces d'une exécution, et plus particulièrement, d'obtenir les mesures de performances globales qui caractérisent un algorithme réparti, comme par exemple le temps d'attente de section critique d'un algorithme d'exclusion mutuelle. Dans un système réparti faiblement couplé, les sites possèdent une horloge locale et il n'y a pas d'horloge commune. Le temps global n'est donc pas immédiatement observable. Cependant, il est nécessaire de disposer d'un temps global, ou au moins d'une approximation de ce temps, pour effectuer des mesures de performances. Nous avons été confronté à cette difficulté dès nos premières implantations d'algorithmes répartis dans *Epsilon*, pour mesurer le temps de transmission d'un message d'application à application ou pour vérifier que l'exclusion mutuelle était bien réalisée.

Une première approche à ce problème consiste à synchroniser les horloges locales en temps réel [Lamport 78], [Srikanth 87], par l'échange explicite de messages datés. Cette classe de méthodes induit une surcharge de travail pour les applications et perturbe donc le fonctionnement du système que l'on est en train d'évaluer.

Lorsqu'il n'est pas nécessaire de disposer du temps global pendant le déroulement d'une application répartie, une autre approche peut être mise en œuvre : l'exploitation *a posteriori* de traces d'exécution locales enregistrées en temps réel. C'est l'approche utilisée dans *DPM* [Miller 84]. Cependant, la méthode proposée suppose que les délais de transmission sont constants pour chaque triplet *[origine−destination−taille du message]*, leur valeur étant mesurée système arrêté. Notre expérience nous a montré que cette hypothèse est loin d'être vérifiée en pratique.

Nous avons développé une méthode d'estimation du temps global à partir de traces locales d'exécution recueillies sur chaque site, en utilisant les horloges locales [Duda 87b]. Il s'agit donc d'une reconstitution *a posteriori* du temps global, et non pas d'un algorithme de décalage d'horloges en temps réel, mais aucune hypothèse n'est faite sur la distribution des délais de transmission. La base de la méthode est l'appariement des événements émission d'un message sur un site − réception du même message sur le site destinataire, et le temps global estimé doit avoir la propriété

de produire un délai positif entre ces paires d'événements, puisque le temps de transmission réel est évidemment positif. La méthode traite les paires d'instances d'émission et de réception comme des réalisations de deux variables aléatoires ayant une dépendance fonctionnelle linéaire entre elles. Une simple régression linéaire par la méthode des moindres carrés permet certes d'obtenir une estimation du décalage (*time offset*) et de la dérive (*time offset rate*) des horloges locales, mais elle présente l'inconvénient de conduire à des paires aberrantes pour lesquelles le délai de transmission serait négatif. Aussi, nous proposons une méthode d'estimation par enveloppe convexe, qui élimine par sa nature même les paires aberrantes. Cette méthode est applicable à plus de deux sites. Des résultats de simulation sont présentés pour la valider.

I.3.1 Estimation du temps global dans de systèmes répartis

Considérons un système réparti composé de N sites sur lesquels s'exécute un programme réparti. Les traces locales de son exécution sont enregistrées par le moniteur sur chaque site en utilisant des horloges locales C_i, $i=1,...,N$. On suppose que les horloges sont stables, c'est à dire, que leur deuxième dérivée temporelle est nulle. Dans ce cas, la différence entre une horloge locale et une horloge parfaite peut être exprimée comme :

$$\Delta(t) = \alpha(t_0) + \beta(t - t_0) \qquad (2)$$

où $\Delta(t)$ est le decalage au moment t, α est le décalage initial et β est la dérive de l'horloge. Cette relation correspond bien aux propriétés des horloges à base de quartz [Duda 87b].

Pour expliquer la méthode de l'estimation du temps global, prenons l'exemple du système composé de deux stations A et B. L'horloge du site A servira comme horloge de référence et pour simplifier on va supposer qu'elle est parfaite : $C_A(t) = t$. Donc, on peut exprimer l'horloge du site B comme $C_B(t) = \alpha + \beta t$, à cause du modèle linéaire (2). Nous ne considérons que deux types d'événements, à savoir l'émission et la réception de messages. Les traces recueillies sur les sites sont les paires des instants (t_i^E, θ_i^E), $i = 1,...,n_E$ et (t_i^R, θ_i^R), $i = 1,...,n_R$ où l'exposant E désigne l'événement émission et R l'événement réception. Cette notation signifie que le message i est émis par le site A à l'instant t_i^E et il est reçu sur le site B à l'instant θ_i^E ; de la même façon, le message j est émis par le site B à l'instant θ_j^R et il est reçu sur le site B à l'instant t_j^R. Les instants sont déterminés par la valeur de l'horloge locale sur un site donné. La Fig. 1.13 présente un exemple de traces locales qui illustrent cette notation. Si l'on dénote par τ_i^E, τ_i^R les délais de transmission de messages dans le sens A vers B et B vers A respectivement, les relations suivantes sont satisfaites par les instants de traces :

$$\theta_i^E = \alpha + \beta(t_i^E + \tau_i^E) \quad (3)$$

$$\theta_i^R = \alpha + \beta(t_i^R - \tau_i^R) \quad (4)$$

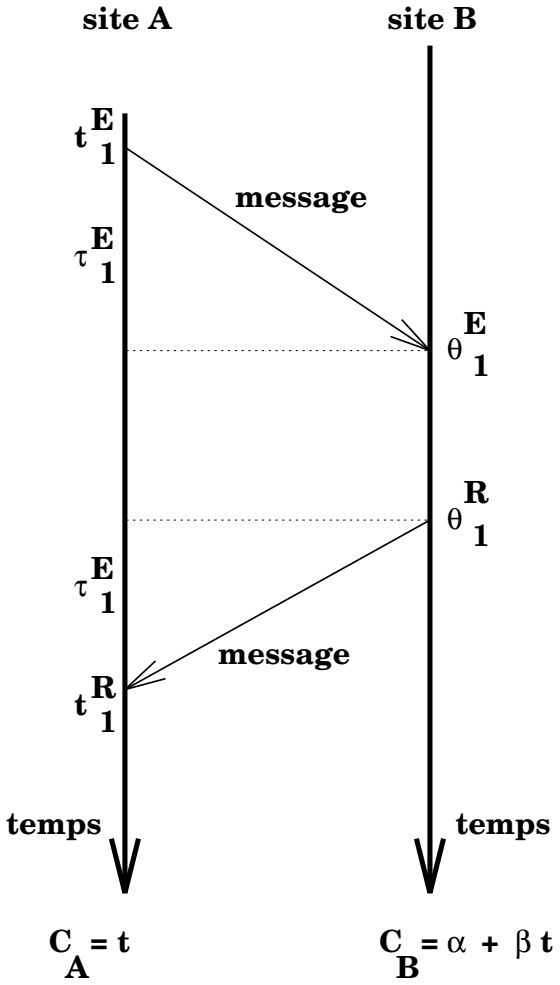


Fig. 1.13 : Exemple de traces locales

Le problème de l'estimation du temps global peut être formulé de manière suivante : à partir des traces locales, trouver les coefficient inconnus de l'horloge C_B : le décalage initial α et la dérive β. La Fig. 1.14 permet de bien comprendre le problème. Cette figure montre les traces sur un plan cartésien où l'axe *X* correspond au temps local *t* sur le site *A* et l'axe *Y* correspond au temps local θ observé sur le site *B*. Les paires de traces (t_i^E, θ_i^E) sont représentées comme des points carrés, tandis que les paires (t_i^R, θ_i^R) comme des points ronds. On peut constater que les points correspondants aux émissions se trouvent au−dessus (et respectivement, les points correspondants aux réceptions au−dessous) de la droite $\alpha+\beta t$ qui représente

l'horloge sur le site *B*. La différence entre un point *i* et la droite est soit $\beta\tau_i^E$ ou $-\beta\tau_i^R$. Cette figure suggère une solution simple pour trouver les inconnues α et β : une régression linéaire par la méthode des moindres carrés permet d'obtenir les estimations du décalage α et de la dérive β. Pour plus de détails sur les estimateurs de α et β obtenus par cette méthode et leur précision, le lecteur peut consulter l'article [Duda 87b].

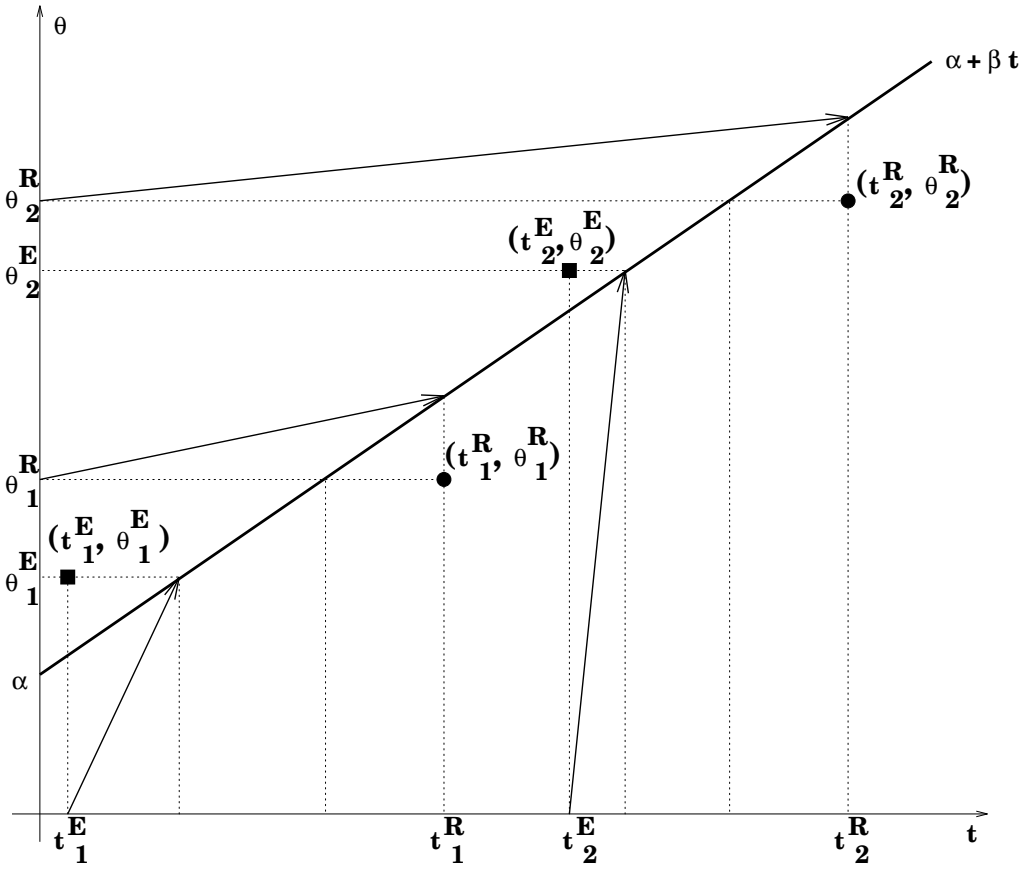


Fig. 1.14 : Traces locales comme des points dans un plan cartésien

La régression linéaire présente l'inconvénient de conduire à des paires de traces aberrantes pour lesquelles le délai de transmission serait négatif. La raison de ce probleme est que la méthode traite les paires de façon indistincte et la droite définie par les estimations du décalage α et de la dérive β peut passer au−dessus d'un point correspondant à l'émission ou au−dessous d'un point correspondant à la réception. Ceci revient à ce que certains délais de transmission soient négatifs.

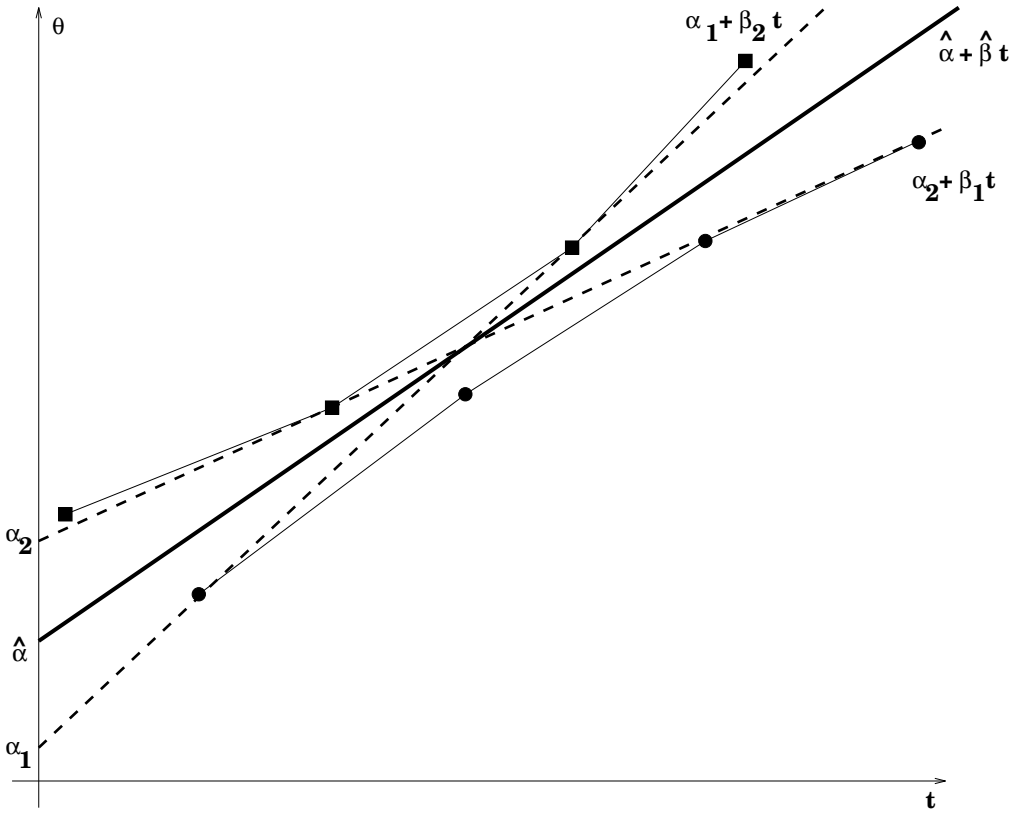


Fig. 1.15 : Méthode de l'enveloppe convexe

Nous avons proposé une autre méthode d'estimation du décalage α et de la dérive β qui ne présente pas cet inconvénient. La Fig. 1.15 présente le principe de cette méthode géométrique basée sur les enveloppes convexes des points correspondants aux traces [Sedgewick 84]. D'abord, on cherche deux enveloppes convexes formées par les points correspondants aux traces d'émissions (points carrés) et aux traces de réceptions (points ronds). Ensuite, on trouve deux droites limites $\alpha_1+\beta_2 t$ et $\alpha_2+\beta_1 t$ qui se placent de façon extrême entre les deux enveloppes convexes. Les estimateurs $\hat{\alpha}$ et $\hat{\beta}$ peuvent être facilement exprimés en fonction de α_1, α_2, β_1, et β_2 (cf. [Duda 87b]). Les valeurs de α_1, α_2, β_1, et β_2 donnent une évaluation de la précision d'estimation parce qu'elles constituent les bornes pour les valeurs exactes.

L'article [Duda 87b] donne plus de détails sur la méthode. On montre que l'estimateur de β est asymptotiquement sans biais, c'est à dire, il converge vers la valeur exacte avec le nombre grandissant de traces. L'estimateur de α est aussi asymptotiquement sans biais, mais à condition que les temps minimaux de transmission soient égaux dans deux directions. Cela veut dire que si cette dernière condition est vérifiée, il est possible d'estimer le temps global avec la précision désirée. La condition reste néanmoins relativement lâche − les distributions et les moyennes des temps de

transmission dans deux directions peuvent être différentes. Si la condition n'est pas vérifiée, l'erreur asymptotique de l'estimation est proportionnelle à la différence de ces temps minimaux de transmission.

La méthode reste valide pour *N* stations et l'article [Duda 87b] donne un algorithme qui traite les traces recueillies sur *N* sites et calcule une estimation du temps global.

Nous avons validé les deux méthodes par une simulation. Un échange de messages entre deux sites a été simulé et nous avons calculé les estimations du décalage α et de la dérive β. Ensuite, nous avons trouvé les temps de transmission. L'emploi de la simulation permet de vérifier la validité de la méthode et de mesurer sa précision (en ne disposant pas d'un système avec des horloges synchronisées, c'était le seul moyen de comparer les temps de transmission calculés à partir de nos estimations avec les valeurs réelles). La précision observée a été excellente : dans tous les cas considérés, l'erreur d'estimation du temps global a été inférieure à 0.4%.

Dans sa thèse, Haddad [Haddad 88] a approfondi et étendu la méthode. D'une part, la formalisation du problème lui a permis d'obtenir une solution générale et de prouver la validité statistique de notre méthode. Il a montré que la prise en compte de l'asymétrie des communications impose une limite théorique à la précision avec laquelle il est possible d'estimer le décalage entre horloges, et que la précision de notre méthode converge vers cette précision théorique. D'autre part, il a pris en compte la granularité des horloges (les horloges des ordinateurs ne sont pas continues et elles présentent des valeurs discrètes multiples d'un quantum de temps appelé *granularité* de l'horloge). La granularité n'est pas en général assez fine pour que les temps locaux puissent être assimilés à des fonctions continues. Si la granularité est du même ordre de grandeur que les délais de transmission des messages, on peut observer des "pseudo"−temps de transmission négatifs dus aux troncatures des horloges. Haddad a étendu notre méthode au cas des horloges à granularité forte. Enfin, il a montré que les algorithmes de calcul des estimateurs ont une complexité linéaire par rapport à la taille des échantillons, que les estimateurs convergent asymptotiquement et que la vitesse de convergence est satisfaisante.

La méthode a été aussi étendue dans la thèse de Jezequel [Jezequel 89b]. Il a proposé un algorithme réparti qui permet le calcul de l'estimation du temps global de manière répartie sur *N* sites. L'algorithme et le support pour générer des traces ont été implémentés sur des systèmes différents : un hypercube Intel iPSC/2 , un Paragon XP/S, un réseau de transputeurs et un réseau local de stations de travail SUN [Jezequel 89a]. Ses résultats expérimentaux de l'estimation s'avèrent très bons, surtout pour les systèmes parallèles qui disposent d'horloges de bonne précision.

I.4 Conclusions

Nous avons présenté au début de ce chapitre nos travaux dans le domaine de la modélisation des systèmes répartis. Un modèle simple de files d'attente a permis de modéliser le comportement des primitives de synchronisation du type *fork−join*. Ce modèle montre l'influence de la synchronisation sur les performances des systèmes répartis. Même si le modèle est simple, on peut en tirer une conclusion importante : dans un système où les tâches parallèles doivent se synchroniser et en même temps entrer en compétition pour des ressources comme des processeurs, les délais de synchronisation peuvent être importants à cause de la variabilité des temps de terminaison des tâches.

Le modèle du parallélisme de tâches présenté plus haut montre l'influence importante que peut avoir la communication sur les performances. Il est intéressant de noter qu'une étude expérimentale a montré le comportement prédit par notre modèle dans deux systèmes : un réseau de transputeurs et une *Connection Machine* [Ghosal 90]. En mesurant le temps d'exécution et l'accélération en fonction du nombre de processeurs sur le réseau de transputeurs interconnectés en maille, les auteurs ont obtenu les courbes similaires à celles de la Fig. 1.3 et Fig. 1.4 pour la structure d'interconnexion en forme d'arbre. Il ont aussi présenté les mesures d'un indice de performances appelée *efficacy* qui correspond à notre indice de *qualité*. La aussi, les courbes sont similaires aux nôtres pour l'arbre de la figure Fig. 1.5. Les mesures d'un algorithme de multiplication de matrices sur la *Connection Machine* montrent un comportement similaire à la courbe de l'anneau de la Fig. 1.3. Les auteurs définissent la notion d'*ensemble de travail de processeurs*, qui est analogue à la notion d'ensemble de travail des systèmes à mémoire virtuelle. L'ensemble de travail de processeurs correspond au nombre de processeurs qui maximalise *efficacy* (ou notre indice *qualité*) et peut être utilisé pour définir les politiques d'ordonnancement des programmes parallèles.

La modélisation théorique des performances s'est montrée récemment utile dans le projet Guide où nous nous sommes posés la question d'utiliser des protocoles de diffusion fiable pour la duplication d'objets. Comme nous n'avons pas trouvé de réponses dans la littérature, nous avons fait une étude dont les résultats sont présentés dans la section I.2.4. Ceci montre que la modélisation reste un outil indispensable pour évaluer différents choix de conception.

La modélisation analytique permet de prévoir le comportement d'un système dans la phase de sa conception avant même qu'il existe. Aussi, elle est utile dans la phase du réglage d'un système pour ajuster ses paramètres et obtenir de meilleures performances. L'un des problèmes de la modélisation est sa limitation − pour obtenir un modèle soluble, il faut des simplifications qui présentent une abstraction trop éloignée du système modélisé. En outre, les données d'entrée des modèles peuvent ne pas correspondre aux valeurs réelles. Par exemple, le temps de transmission des

messages est souvent supposé constant, ou distribué exponentiellement. Notre expérience dans *Epsilon* nous a montré que la réalité est beaucoup plus complexe. En plus, au moment du développement des algorithmes et systèmes répartis dans les années 80, il existait très peu de méthodes de modélisation et de résultats. La raison en est que les techniques de modélisation permettent difficilement la prise en compte du parallélisme et l'expression de contraintes de synchronisation.

Une simulation permet de considérer un modèle plus détaillé, mais elle est difficile à mettre au point. Obtenir des statistiques précises d'un système surchargé peut prendre un temps très long. Il n'est pas rare de présenter les résultats d'une simulation qui ensuite s'avère erronée [Livny 93]. En fait, il faut être très prudent avec les résultats d'une modélisation ou d'une simulation, parce que les conclusions que l'on tire des modèles peuvent s'avérer fausses dans la réalité. Aussi, ce domaine présente un autre danger si bien formulé par Ferrari [Ferrari 83]:

> "*the study of performance evaluation as an independent subject has sometimes caused researchers in the area to lose contact with reality*"

Il est difficile de passer de la modélisation aux mesures, mais c'est le seul moyen de ne pas rester à un niveau d'abstraction trop haut, souvent éloigné des réalités. L'exercice de l'évaluation de performances donne parfois une sensation de faim − on traite des problèmes dans l'abstrait et souvent on ne participe pas à l'élaboration des systèmes. Pour ces raisons, nous nous sommes impliqués dans le projet *Epsilon* qui permettait de reprendre contact avec la réalité et de valider les approches de modélisation précédemment développées. Le projet *Epsilon* a apporté beaucoup sur deux aspects : en développant une maquette d'expérimentation des algorithmes répartis, et en proposant la méthode originale d'estimation du temps global. Malheureusement, l'équipe s'est dissoute au moment le plus intéressant où la maquette était opérationnelle (sur trois IBM PC et un VAX), où on pouvait écrire des applications réparties à l'aide des primitives de communication, et où la méthode de l'estimation du temps global a été implémentée. La seule expérience que nous avons pu faire a été la mesure des indices de performances d'un algorithme réparti d'exclusion mutuelle (cf. thèse de Bernard [Bernard 89b], qui présente les fonctions de densité empiriques obtenues pour le temps de transmission de messages et le temps d'attente de section critique). *Epsilon* nous a donné la satisfaction d'accomplir un travail expérimental et théorique intéressant et fécond, et il est dommage que nous n'ayons pu le continuer. Mais, sa suite entreprise par Jezequel fait que les résultats d'*Epsilon* sont poursuivis.

I.5 Bibliographie

[Akyildiz 87] I.F. Akyildiz, ''Exact Product Form Solution for Queueing Networks with Blocking'', *IEEE Trans. on Computers*, C−36, pp. 122−125, 1987.

[Ansart 82] J.P. Ansart et J. Damidau, ''CERBÈRE, A Tool to Keep an Eye on High Level Protocols'', *Proc. Protocol Verification and Testing*, pp. 529−538, North−Holland, 1982.

[Axelrod 86] T.S. Axelrod, ''Effects of Synchronization Barriers on Multiprocessor Performance'', *Parallel Computing*, 3, pp. 129−140, 1986.

[Baccelli 85a] F. Baccelli et A. M. Makowski, ''Simple Computable Bounds for the Fork−Join Queue'', *Proc. John Hopkins Conf. Information Sciences and Systems*, John Hopkins University, 1985.

[Baccelli 85b] F. Baccelli et W. A. Massey, ''Series−Parallel, Fork−Join Queueing Networks and their Stochastic Ordering'', *Bell Labs. Technical Memorandum 11211−851101−36*, novembre 1985.

[Bal 89] H. Bal, F. Kaashoek, A. Tanenbaum, *Replication Techniques for Speeding Up Parallel Applications on Distributed Systems*, (IR−202), Vrije Universiteit, décembre 1989.

[Baskett 75] F. Baskett et al., ''Open, closed and mixed Networks of Queues with Different Classes of Customers'', *JACM*, 22(2), avril 1975.

[Bernard 87] G. Bernard, A. Duda, Y. Haddad, G. Harrus, ''Implementing Distributed Applications on a Heterogenous Local Area Network'', *Proc. 2th Int. Symposium on Computer and Information Sciences*, pp. 276−287, 1987.

[Bernard 89a] G. Bernard, A. Duda, Y. Haddad, G. Harrus, ''Primitives for Distributed Computing in a Heterogenous Local Area Network Environement'', *IEEE Trans. on Software Engineering*, 15(12), pp. 1567−1578, 1989.

[Bernard 89b] G. Bernard, *Performances et systèmes répartis : de la modélisation à la réalisation*, Thèse de Docteur ès Sciences, Université de Paris−Sud, Orsay, avril 1989.

[Cellary 84] W. Cellary et M. Stroiñski, ''Analysis of Methods of Computer Network Performance Measurment'', *Proc. Performance of Computer−Communication Systems*, pp. 465−480, North−Holland, 1984.

[Cheriton 83] D.R. Cheriton et W. Zwaenepoel, ''The Distributed V Kernel and Its Performance for Diskless Workstations'', *Proc. 9th Symp. Operating Systems Principles*, octobre 1983.

[Duda 87a] A. Duda et T. Czachórski, ''Performance Evaluation of the Fork and Join Synchronization Primitives'', *Acta Informatica*, 24, pp. 525−553, 1987.

[Duda 87b] A. Duda, G. Harrus, Y. Haddad, G. Bernard, ''Estimating Global Time in Distributed Systems'', *Proc. 7th Int. Conference on Distributed Computing Systems*, pp. 299−306, Berlin, 1987.

[Duda 88a] A. Duda, ''Approximate Performance Analysis of Parallel Systems'', *Proc. Computer Performance and Reliability*, édité par G.Iazeolla, P.J.Courtois, O.J.Boxma, pp. 189−202, North−Holland, 1988.

[Duda 88b] A. Duda, ''On the Tradeoff between Parallelism and Communication'', *Proc. 4th Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pp. 379−391, Palma de Mallorca, septembre 1988.

[Duda 93] A. Duda, ''Analysis of Multicast−based Object Replication Strategies in Distributed Systems'', *Proc. 13th Int. Conference on Distributed Computing Systems*, pp. 311−318, Pittsbourgh, mai 1993.

[Ellingson 73] C.E. Ellingson et R.J. Kulpinski, ''Dissemination of System Time'', *IEEE Trans. on Communications*, COM−21(5), pp. 605−623, mai 1973.

[Ferrari 83] D. Ferrari, G. Serazzi et A. Zeigner, *Measurement and Tuning of Computer Systems*, Prentice Hall, Englewood Cliffs, NJ, 1983.

[Flatto 84] L. Flatto et S. Hahn, ''Two Parallel Queues Created by Arrivals with Two Demands'', *SIAM J. Appl. Math.*, 44(5), pp. 1041−1053, octobre 1984.

[Gehringer 82] E.F. Gerhringer et al., ''The Cm* Testbed'', *IEEE Computer*, 15(10), octobre 1982.

[Ghosal 90] D. Ghosal et al., *The Concept of Processor Working Set for Allocation Strategies in Multiprogrammed Parallel Systems*, (UMIACS TR−89−95), University of Maryland, 1990.

[Haddad 88] Y.Z. Haddad, *Performances dans les systèmes répartis : des outils pour les mesures*, Thèse de Doctarat, Université de Paris−Sud, septembre 1988.

[Harter 85] P.K. Harter et al., ''IDD: an Interactive Distributed Debugger'', *Proc. 5th Int. Conference on Distributed Computing Systems*, 1985.

[Heidelberger 82] P. Heidelberger et S. K. Trivedi, ''Queueing Network Models for Parallel Processing of Asynchronous Tasks'', *IEEE Trans on Computers*, 31, pp. 1099−1109, 1982.

[Heidelberger 83] P. Heidelberger et S. K. Trivedi, ''Analytic Queueing Models for Programs with Internal Concurrency'', *IEEE Trans on Computers*, 32, pp. 73−82, 1983.

[Jezequel 89a] J.M. Jezequel, ''Building a Global Time on Parallel Machines'', *Proc. 3rd Int. Workshop on Distributed Algorithms*, Nice, septembre 1989.

[Jezequel 89b] J.M. Jezequel, *Outils pour l'expérimentation d'algorithmes distribués sur machines parallèles*, Thèse de Doctorat, Université de Rennes I, octobre 1989.

[Joyce 87] J. Joyce et al., ''Monitoring Distributed Systems'', *ACM Trans. on Computer Systems*, 5(2), mai 1987.

[Kaashoek 91] F. Kaashoek, A. Tanenbaum, ''Group Communication in the Amoeba Distributed Operating System'', *Proc. 11th Int. Conference on Distributed Computing Systems*, pp. 222−230, Arlington, mai 1991.

[Kruatrachue 88] B. Kruatrachue et T. Lewis, ''Grain Size Determination for Parallel Processing'', *IEEE Software*, 5(1), pp. 23−32, 1988.

[Lamport 78] L. Lamport, ''Time, Clocks and the Ordering of Events in a Distributed System'', *CACM*, 21(7), pp. 558−565, juillet 1978.

[Lee 80] R. B−L. Lee, ''Empirical Results on the Speedup, Efficiency, Redundancy and the Quality of Parallel Computations'', *Proc. Int. Conference on Parallel Processing*, pp. 91−96, août 1980.

[Livny 93] M. Livny, ''Simulation of Database Systems'', *séminaire MIT*, avril 1993.

[Lint 88] B. Lint et T. Agerwala, ''Communication Issues in the Design and Analysis of Parallel Algorithms Construction and Management of Distributed Office Systems'', *IEEE Trans. on Software Engineering*, SE−7, pp. 174−188, mai 1988.

[MacDaniel 77] G. MacDaniel, ''METRIC: a Kernel Instumentation System for Distributed Environments'', *Proc. 6th ACM Symposium on Operating Systems Principles*, pp. 93−99, novembre 1977.

[Miller 84] B. P. Miller, ''Performance Characterization of Distributed Programs'', *Ph.D. Dissertation, Report No. UCB/CSD84/197, University of California, Berkeley*, August 1984.

[Nelson 85] R. Nelson et A. N. Tantawi, ''Approximate Analysis of Fork/Join Synchronization in Parallel Queues'', *IBM Research Report RC 11481*, October 1985.

[Reed 83] D. A. Reed, ''Performance Based Design and Analysis of Multimicrocomputer Networks'', *Ph.D. Thesis, Purdue University*, mai 1983.

[Reiser 80] M. Reiser et S.S. Lavenberg, ''Mean Value Analysis of Closed Multichain Queueing Networks'', *JACM*, 27, pp. 313−322, 1980.

[Robinson 79] J. T. Robinson, ''Some Analysis Techniques for Asynchronous Multiprocessor Algorithms'', *IEEE Trans. on Software Engineering*, SE−5, pp. 24−31, janvier 1979.

[Sedgewick 84] R. Sedgewick, *Algorithms*, Addison−Wesley, Reading, 1984.

[Shoch 80] J.F. Shoch et J.A. Hupp, ‘‘Measured Performance of an Ethernet Local Network’’, *CACM*, 23,pp. 711−721, 1980.

[Srikanth 87] T.K. Srikanth et S. Toueg, ‘‘Optimal Clock Synchronization’’, *JACM*, 34(3), juillet 1987.

[Stumm 90] M. Stumm, S. Zhou, ‘‘Algorithms Implementing Distributed Shared Memory’’, *IEEE Computer*, 23(5), pp. 54−64, mai 1990.

[Veran 85] M. Veran et D. Potier, ‘‘QNAP2: a Portable Environment for Queueing Systems Modelling’’, *Proc. Modelling Techniques and Tools for Performance Analysis*, édité par D. Potier, North−Holland, 1985.

Chapitre I

Évaluation de performances des systèmes répartis

I.1 Introduction 11

I.2 Modélisation des systèmes répartis 11

I.2.1 Primitives de synchronisation de type fork−join 14

I.2.2 Analyse approchée des primitives de synchronisation 15

I.2.3 Performances de programmes parallèles 16

I.2.4 Stratégies de duplication d'objets 21

I.3 Mesures de performances des systèmes répartis 31

I.3.1 Estimation du temps global dans de systèmes répartis 35

I.4 Conclusions 40

I.5 Bibliographie 42