Thèse de Doctorat de L'Université Pierre et Marie Curie

VERSION PRÉLIMINAIRE

Spécialité

SYSTÈMES INFORMATIQUES

présentée par

M. PHAM Cong-Duc

pour obtenir le titre de

DOCTEUR de l'Université Pierre et Marie CURIE

Simulations Distribuées de Réseaux ATM
Etude et Perspectives

soutenance prévue le 4 juillet 1997 devant le jury composé de

M.	**G.**	**Pujolle**	**Président**
Mme.	**M.**	**Becker**	**Rapporteur**
M.	**F.**	**Kamoun**	**Rapporteur**
M.	**O.**	**Spaniol**	**Examinateur**
M.	**R.**	**Bayou**	**Examinateur**
M.	**S.**	**Fdida**	**Examinateur**

À ma famille, à Valérie.

Remerciements

Cette thèse intitulée “Simulations Distribuées de Réseaux ATM : Etude et Perspectives”, a été soutenue à l'université Pierre et Marie Curie (Paris 6) devant le jury composé du Pr. Guy Pujolle, Pr. Monique Becker, Pr. Farouk Kamoun, Pr. Otto Spaniol, M. Rémi Bayou et du Pr. Serge Fdida.

Une partie de cette thèse s'inscrit dans le cadre d'un contrat de recherche passé entre la DRET (Direction des Recherches, Etudes et Techniques) et le laboratoire LIP6.

Je tiens à remercier tous les membres de ce jury :

- Pr. Serge Fdida, Professeur à l'université de Paris 6, qui a dirigé mes travaux de recherche, pour ces encouragements tout au long de ces années.
- Pr. Guy Pujolle, Professeur à l'université de Versailles, pour s'être intéressé à mes travaux et pour m'avoir fait l'honneur de bien vouloir présider ce jury.
- Pr. Monique Becker, Professeur à l'INT, et Pr. Farouk Kamoun, Professeur à l'université de Tunis, qui ont accepté d'être rapporteurs de ma thèse, pour l'intérêt qu'ils ont porté à ce travail.
- Mr. Rémi Bayou, Service des Recherches de la DRET, pour la confiance et le soutien qu'il nous a accordé.
- Pr. Otto Spaniol, Professeur à l'université d'Aachen, Allemagne, pour avoir bien voulu participer à ce jury.

Je remercie également tous les étudiants en thèse et les membres du Laboratoire LIP6 qui m'ont permis d'effectuer ce travail dans de très bonnes conditions et dans une ambiance particulièrement cordiale et chaleureuse.

Je remercie spécialement ma famille, à qui cette thèse est dédiée, pour m'avoir encouragé et poussé depuis le début de mes études. Cette thèse est aussi la leur.

Résumé

Les réseaux de communication sont des systèmes complexes de grande dimension qui posent un grand nombre de problèmes. Pour l'analyse de performances de ces systèmes, les outils théoriques à notre disposition aujourd'hui montrent leurs limites. La simulation nous semble plus appropriée, et plus particulièrement les techniques de simulation parallèle qui pourraient s'avérer efficaces si on les applique aux réseaux. Simuler en parallèle soulève des problèmes de synchronisation et de respect des contraintes de causalités qui rendent cette tâche ardue.

Les algorithmes de simulation parallèle et distribuée peuvent être classés en deux catégories : conservatrice et optimiste. L'approche conservatrice n'autorise l'exécution d'un événement que lorsque la certitude qu'il ne produira pas de fautes temporelles sera acquise. Pour éviter les situations d'inter-blocage, des *null-messages* qui se basent sur la notion de *prédiction* sont utilisés pour faire avancer artificiellement le temps de simulation. D'un autre coté, les approches optimistes ne font aucune tentative pour éviter les fautes temporelles mais instaurent un mécanisme de *retour-arrière* pour corriger la trajectoire de la simulation lorsqu'une telle faute survient. La sauvegarde périodique des états et des anti-messages pour corriger la propagation des erreurs est alors la contrepartie d'une plus grande liberté.

Dans cette thèse, nous nous intéressons plus spécifiquement aux modèles de réseaux ATM. Dans un premier temps, nous étudions les propriétés de ces modèles du point de vue de la simulation distribuée. Ensuite nous montrons qu'il est possible d'obtenir des gains significatifs en utilisant une méthode conservatrice lorsque les modèles de réseaux sont simples (topologie maillée, routage aléatoire uniforme). Pour des réseaux plus complexes (topologie quelconque, algorithme de routage) les gains restent non négligeables et permettent ainsi l'étude de systèmes de grande dimension en des temps raisonnables. Nous étudions ensuite l'utilisation de méthodes relâchant certaines contraintes de synchronisation dans les approches optimistes. En particulier, nous proposons l'approche statistiquement correcte qui utilise des estimations locales pour corriger la trajectoire de la simulation lorsqu'on s'intéresse à des valeurs moyennes pour les critères de performances. Un algorithme de blocage basé sur la longueur de la file des événements est également proposé pour réduire le nombre de fautes temporelles et augmenter ainsi la précision des résultats. L'utilisation conjointe de ces deux approches est prometteuse puisque les mesures de gains montrent qu'il est possible d'aller plus vite qu'une méthode conservatrice tout en préservant une grande précision.

Abstract

Communication networks are large and complex systems that raise new problems. Performance analysis of these systems by available theoretical tools show their limits. Simulation appears to be more suitable in our case and in particular parallel and distributed simulation techniques may successfully be applied to communication network models. Unfortunately distributed simulation is a difficult task and we must face synchronization problems and causality constraints.

Parallel and distributed simulation algorithms can be classified in two main categories : conservative and optimistic. The conservative approach only processes events that can definitely be considered as safe, i.e. processing an event would not result in further causality violations. To avoid deadlock situations deadlock, *null-messages* that rely on *lookahead* ability must be sent in the purpose of artificially propagating and advancing the simulation time. On the other hand, optimistic approaches do not search for safety but provisions are made to *roll back* to an earlier coherent state when causality violations occur. Periodic check-pointing and *anti-messages* to cancel bad computations are then needed as a counterpart for more freedom.

In this thesis, we are interested more specifically in ATM network models. In a first step, properties of these models are studied under the distributed simulation point of view. Then we show that significant gains can be obtained with a conservative kernel when the network models are simple (mesh topologies, random uniform routing). For more complex network models (general topologies, real routing algorithm) the gains are not negligible. Therefore, it is possible to study large systems in a reasonable amount of time. In a second step, we investigate methods to reduce some synchronization constraints in optimistic approaches. In particular, we propose the *statistically correct* approach that uses local estimations to correct the simulation trajectory when only mean value performance statistics are needed. A blocking algorithm that relies on the length of the input event queue is also proposed in order to reduce the number of time errors and increase the accuracy of local estimations. The utilization of both approaches together is promising since the time measures show that it is possible to beat a conservative approach while keeping a high accuracy level.

Table des matières

Chapitre 1

Introduction générale

1.1 LES RÉSEAUX DE COMMUNICATION

Les réseaux de télécommunications seront marqués dans les prochaines années par une demande de plus en plus diversifiée, par une compétition accrue et par des évolutions technologiques considérables qui bouleverseront les réseaux d'aujourd'hui. Parmi les techniques qui influenceront fortement une telle évolution au cours de la décennie, on retiendra surtout la transmission sur fibre optique, les techniques de codage d'images, la technique de transfert asynchrone (ATM) et le développement fabuleux de l'Internet. La technique de commutation ATM est aujourd'hui largement acceptée par la communauté aussi bien télécom qu'informatique. Il est par conséquent logique que la technique ATM participe largement à l'infrastructure des futurs réseaux à large bande, aujourd'hui en cours de normalisation à l'ITU-T et à l'ATM Forum, mais déjà présente de manière opérationnelle tant dans le domaine privé que public. L'ATM est une technique de transfert qui devra supporter les services variés des applications multimédia, coopératives et temps-réel. Une bonne adéquation entre cette technique et le besoin des applications ne sera obtenue qu'au travers d'un dimensionnement approprié des ressources du réseau. Il s'ensuit une activité intense au niveau international pour évaluer soigneusement les composantes du réseau afin d'en maîtriser le fonctionnement. En effet, un réseau étant le centre nerveux du système d'information, un mauvais dimensionnement ou une panne d'un composant peut entraîner des répercussions graves sur les applications supportées.

Ces futurs réseaux seront de grande dimension, impliqueront de multiples éléments souvent complexes et par conséquent poseront un grand nombre de problèmes. Parmi ceux-ci, on peut citer le routage, le contrôle de congestion et le comportement lors de pannes qui sont tous les trois très complexes à résoudre à grande échelle. Des critères de performances tels que le taux de perte de cellules ou la probabilité d'appels rejetés sont également très utiles pour bien dimensionner les éléments du réseau. Il est alors nécessaire de disposer de nouveaux outils d'évaluation car la complexité des systèmes étudiés augmente ainsi que leur dimension, rendant des études de phénomènes transitoires ou stationnaires délicates voire impossibles dans le contexte actuel. Deux approches complémentaires d'évaluation existent pour cela : les approches qualitatives et les approches quantitatives.

1.2 LES MÉTHODES D'ÉVALUATION

L'évaluation qualitative d'un système étudie essentiellement les propriétés de celui-ci. On s'intéressera alors plus particulièrement à mettre en évidence des propriétés de fiabilité, de robustesse, de validité etc. Les méthodes de description formelles telles que les langages formels (*Lotos, Estelle etc.*) ou les Réseaux de Petri (*RdP temporisés, stochastiques etc.*) permettent de spécifier puis de valider de manière très élégante les systèmes à l'étude.

Les méthodes d'évaluation quantitative sont celles qui nous intéressent le plus. Elles peuvent être classées en trois catégories [Heid84] : la mesure, les méthodes analytiques et la simulation (figure 1.1).

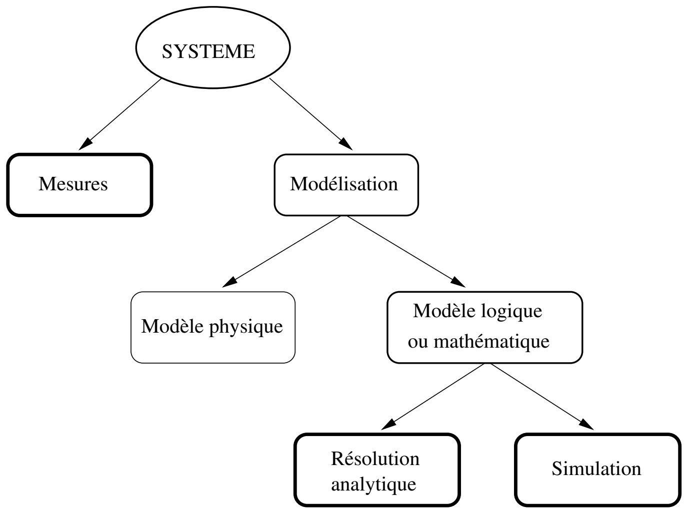


Figure 1.1 Méthodes d'évaluation de performances

Les méthodes basées sur la mesure nécessitent bien évidemment que le système à étudier soit opérationnel et qu'il soit accessible. Elles ne peuvent pas prédire les performances du système, mais constituent plutôt un moyen de surveillance et d'optimisation. Malgré la lourdeur et la délicate mise en oeuvre de toute l'instrumentation nécessaire, l'avantage de telles méthodes est d'extraire les performances d'un système réel, et non celles d'un modèle.

Maintenant si l'on se décide pour la modélisation, deux types de modèles sont à considérer : les modèles dit physiques et ceux dit mathématiques ou logiques.

Un modèle réduit d'avion en soufflerie est un exemple de modèle physique. Un grand nombre d'applications militaires reposent sur l'utilisation de tels modèles. En revanche, pour l'application qui nous concerne, il est difficilement imaginable d'utiliser de tels modèles, c'est pourquoi les modèles utilisés pour l'évaluation de performances de systèmes informatiques seront toujours des modèles du second type. Dans ce cas, les suppositions que l'on fait sur la façon dont un système donné fonctionne prennent le plus souvent la forme de relations mathématiques ou logiques. L'ensemble de ces relations constituent alors notre modèle.

Pour résoudre notre modèle nous avons le choix entre les méthodes analytiques et la simulation. La plupart des systèmes informatiques peuvent être considérés comme un ensemble de ressources et de clients concourant pour ces ressources. Des phénomènes d'attente inhérents à ces systèmes font que les méthodes de résolution analytiques utilisent naturellement le formalisme des files d'attente. Les réseaux de files d'attente à forme produit ou non donnent la possibilité de modéliser un système donné puis d'en extraire des critères de performances. Ces méthodes sont les moins onéreuses puisque la résolution se fait analytiquement. En contre partie, le champ d'application de ces méthodes est relativement réduit car celles-ci imposent souvent des hypothèses de fonctionnement très restrictives qui nécessitent de fortes simplifications pouvant rendre l'étude non réaliste.

La simulation, contrairement aux méthodes analytiques, ne présente pas de limites théoriques quant au niveau de détail du modèle. Grâce à cela, elle est devenue un outil largement utilisé dans l'évaluation de performances des systèmes informatiques. Dans le domaine des réseaux informatiques, nous assistons en ce moment à une véritable "explosion" d'outils puissants et conviviaux. Cependant cette fantastique souplesse se paye cher, et le principal facteur limitatif est le temps de simulation qui peut très vite devenir excessif lorsque le modèle est un tant soit peu réaliste. Par exemple, pour évaluer des algorithmes de contrôle d'admission, simuler au niveau appel peut être suffisant mais, dans la plupart des cas, il est nécessaire de descendre jusqu'au niveau cellule pour réellement obtenir des critères intéressants. Malheureusement, simuler à ce niveau implique l'exécution de millions d'événements, ce qui allonge dangereusement le temps de simulation. De manière générale, on peut prévoir que la simulation des futurs réseaux large bande nécessitera une grande quantité de calcul. La simulation nous semble néanmoins la méthode la plus appropriée malgré sa forte demande en puissance de calcul.

La situation est alors la suivante : d'un côté, la simulation séquentielle est dépassée par l'augmentation de la puissance de calcul requise, alors que d'un autre côté, le calcul distribué et les architectures parallèles offrent une formidable puissance très souvent sous-exploitée. Un grand effort de recherche pour utiliser cette puissance nouvellement disponible pour des tâches de simulation s'est donc déclaré depuis ces dix dernières années, et des techniques de simulation distribuée ont été proposées. Ces techniques, encore méconnues du grand public et d'un usage délicat, pourraient s'avérer efficaces si on les applique spécifiquement aux réseaux (ATM, ou autres tel que Internet). Bien que la simulation distribuée soulève des problèmes de synchronisation et de respect des contraintes de causalité, qui rendent cette tâche ardue, il est envisageable, d'un autre côté, de réduire les temps de simulation et ainsi d'augmenter la taille du système étudié.

Deux types d'environnements différents sont rencontrés lors de l'évaluation de réseaux de communication. Le premier consiste à s'intéresser aux performances d'un système possédant de nombreuses composantes différentes. Il s'agit par exemple de l'étude d'empilement de couches de protocoles dans des stations d'extrémités. Dans ce cas, la simulation parallèle semble peu performante en raison, d'une part, de la forte synchronisation des processus, d'autre part de leur faible nombre et de leur variété. Le second type est celui d'un grand réseau de commutateurs, tels que ceux rencontrés auprès des opérateurs publics ou des réseaux privés. Dans ce contexte, les commutateurs sont tous similaires et reproduits en grand nombre. Il est donc possible d'envisager une réplication du code associé à chaque commutateur dans un *processus logique* de la simulation parallèle.

1.3 LA SIMULATION PARALLÈLE ET DISTRIBUÉE

Les premières applications à bénéficier de la répartition ont été les applications militaires. Le Département de la Défense américaine (DoD) s'est très tôt intéressé à ce domaine et a largement contribué à la définition d'un protocole permettant l'échange de données entre plusieurs environnements de simulation. Ce protocole, appelé *DIS protocol*, est une composante de l'architecture HLA *High Level Architecture* qui vise à rendre homogène les environnements de simulation pour obtenir une inter-opérabilité maximale. Cependant, cette distribution de la simulation doit plutôt être vue comme une coopération de différents simulateurs que comme une réelle parallélisation de la simulation car les notions de synchronisation et de causalité ne sont pas implémentées.

Les premiers véritables algorithmes de simulation parallèle et distribuée ont été introduits indépendamment par Chandy et Misra en 1979 d'une part [Chan79] et par Bryant d'autre part [Brya77]. Ils proposent un algorithme simple qui permet une exécution cohérente des différents processus logiques. L'algorithme est dit *conservateur* car un blocage éventuel était le coût à payer pour le respect de la causalité. Des *null*-messages étaient en outre nécessaires pour éviter les configurations d'interblocage. Vers 1985, Jefferson propose une nouvelle méthode, appelée *Time Warp* [Jeff85], avec une orientation radicalement opposée. Contrairement à l'algorithme proposé par Chandy et Misra, celui-ci est sans blocage et reflète un comportement fortement *optimiste*. En effet aucun contrôle n'est fait pour éviter les *fautes temporelles*, mais lorsque celles-ci se produisent, un mécanisme de *retour-arrière* permet de défaire les erreurs commises. Malheureusement, cet algorithme qui paraît simple sur le papier est lourd à implanter et nécessite beaucoup plus de mémoire que son homologue conservateur.

L'activité de recherche de la communauté scientifique internationale sur la simulation parallèle a considérablement augmenté depuis ces dix dernières années. D'une poignée de chercheurs au début des années 80, une centaine se penchent maintenant sur ces problèmes. De nombreux travaux sont sans cesse effectués pour évaluer et augmenter les performances des algorithmes de simulation parallèle et pour en trouver d'autres.

PLAN DU DOCUMENT

- Le chapitre 2 présente les concepts généraux de la simulation. Les principes de la simulation à événements discrets tels que la gestion du temps et son architecture sont présentés. Les outils orientés langage et logiciels sont abordés, ainsi que les autres formes de simulation comme la simulation continue ou hybride.
- Le chapitre 3 introduit la notion de simulation parallèle et distribuée, sa problématique et les algorithmes qui sont proposés actuellement. Sont aussi abordés les langages et les environnements dédiés à ce type de simulation.
- Le chapitre 4 pose la problématique de la simulation parallèle de réseaux de communication ainsi que les travaux qui ont été réalisés sur le sujet. Nous concentrons ensuite notre recherche sur les réseaux ATM et nous étudions leurs propriétés sous le point de vue de la simulation.
- Le chapitre 5 met en œuvre l'approche expérimentale de la simulation distribuée de réseaux ATM en utilisant une méthode conservatrice. Il décrit également notre noyau de simulation conservateur qui fonctionne sur une machine CM-5. Nous présentons les résultats de speedup obtenus dans cette étude.
- Le chapitre 6 présente une approche dite *statistiquement correcte* qui relâche certaines contraintes de synchronisation suivant une fonction de contrôle et qui peut s'avérer efficace lorsque l'on s'intéresse à l'obtention de valeurs moyennes.
- Ce dernier chapitre présente les méthodes de réduction de l'optimisme par des stratégies de blocage et définit un algorithme à seuil basé sur la longueur de la file des événements. Cet algorithme est utilisé conjointement avec l'approche précédente pour améliorer la précision des résultats.

Chapitre 2

Aspects généraux de la simulation

Parmi toutes les méthodes d'évaluation de performances la simulation a pris une importance considérable depuis ces dernières années. On ne compte plus le nombre d'outils logiciels qui se créent chaque année, et cela dans le but d'offrir à des utilisateurs de moins en moins spécialisés les moyens d'effectuer eux-mêmes l'évaluation des performances de leurs systèmes. On peut donc affirmer que la simulation est devenue un outil "grand public", contrairement aux méthodes analytiques qui nécessitent tout de même un bagage mathématique conséquent, et qui sont donc réservés à une "élite". Quant à la mesure, outre le matériel lourd nécessaire, il faut être un spécialiste pour en tirer des résultats exploitables.

Ce chapitre aborde les points suivants :

- les concepts de base de la simulation,
- les principes de la simulation à événements discrets,
- les outils orientés langage,
- les outils orientés logiciel, et
- les autres formes de simulation.

2.1 SYSTÈME, MODÈLE ET SIMULATION

Simuler revient à imiter le comportement d'une personne, d'une machine, d'un phénomène etc., bref plus généralement d'un *système* physique. On ne simule pas pour le plaisir, on simule généralement un système pour l'étudier. Pour cela, il est nécessaire de faire des suppositions, plus ou moins précises, sur la façon dont un système donné fonctionne. L'ensemble de ces suppositions qui prennent le plus souvent la forme de relations mathématiques ou logiques constitue un *modèle*. Ce modèle servira ensuite à représenter le système, cela dans le but de mieux le comprendre et éventuellement d'étudier les impacts de telle ou telle modification.

Si les relations qui composent le modèle sont simples, alors il est possible d'utiliser un modèle mathématique qui fournira des résultats exacts sur le comportement du système—solution analytique. Par contre, si ces relations sont plus complexes, on atteint très vite les limites des modèles mathématiques, qui typiquement nécessitent très souvent des simplifications ou des hypothèses de fonctionnement très restrictives. Dans ce cas, la *simulation* est souvent le dernier recours pour l'étude du système. Cette dernière solution est de plus en plus appréciée et utilisée car il n'y a pas de contraintes théoriques quant à la complexité du système et du niveau de détail que l'on souhaite modéliser. En particulier, tandis que les méthodes mathématiques ne fournissent que des valeurs moyennes pour les critères de performance, la simulation permet en plus d'obtenir des estimations des distributions de ces derniers. De plus, celle-ci autorise l'étude du comportement d'un système non seulement à l'état stationnaire, mais aussi dans des états transitoires.

Pour effectuer une simulation, on a recours à un ordinateur pour évaluer *numériquement* le modèle du système physique que l'on considère. Le déroulement d'une simulation est constitué de trois étapes principales :

1. **La modélisation.** Cette première phase de modélisation consiste à synthétiser les connaissances que l'on a d'un système physique en un ensemble de relations mathématiques ou logiques.

2. **L'exécution.** L'exécution du modèle est ce qu'on appelle communément la phase de simulation. Elle consiste à exécuter une instance du modèle à l'aide d'un ordinateur pour recueillir des données numériques sur le comportement du modèle.

3. **L'analyse.** La phase d'analyse consiste à traiter les données recueillies pour les exploiter, et en déduire ainsi les réponses aux questions que l'on se posait avant l'étude.

Ces trois étapes ne sont pas à exécuter dans l'ordre strict dans lequel elles sont énumérées. Tout au long de la simulation, on peut aussi bien s'arrêter à une étape pour repasser à l'étape précédente, qu'itérer plusieurs fois le cycle des trois étapes. La figure 2.1 illustre ces différentes étapes et leurs interactions. La taxonomie est inspirée de celle proposée par Fishwick dans [Fish95]. La phase de modélisation peut se faire au fur et à mesure, par itération successive, lorsque l'on s'aperçoit, grâce à l'exécution et à l'analyse, que le modèle n'était pas bien adapté. Par exemple, il peut ne pas être assez proche du système auquel cas les résultats obtenus ne sont pas significatifs, soit être trop proche du système auquel cas le temps de simulation peut devenir trop long et l'analyse des résultats peut s'avérer difficile voire impossible. Il est généralement constaté que les retours en arrière sont très fréquents et que les

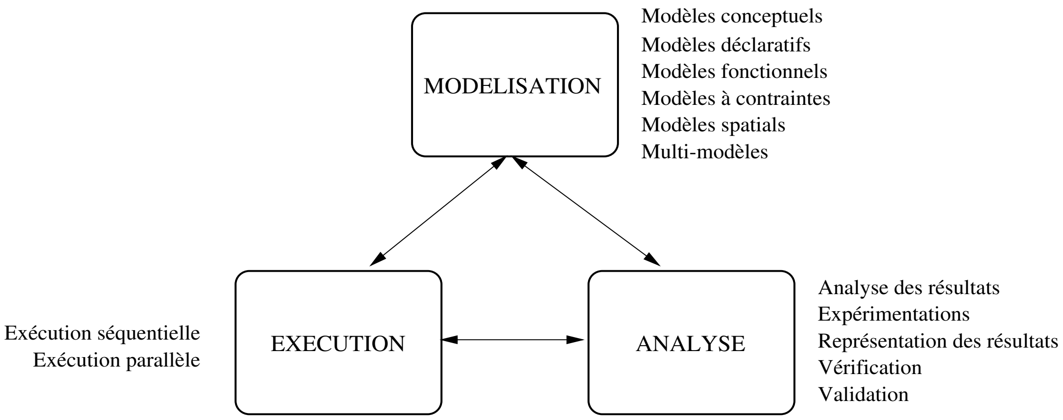


Figure 2.1 Les 3 étapes d'un processus de simulation et la taxonomie associée.

retouches au modèle sont nombreuses. Trouver un juste équilibre entre le niveau de détail modélisé et la précision des résultats obtenus est une tâche délicate.

La dernière étape qui consiste à analyser les résultats pour les exploiter n'est pas la plus facile. La *vérification* et la *validation* sont également deux aspects très importants à inclure dans cette dernière étape. La vérification est très proche de la notion de *déboggage* puisqu'il s'agit de s'assurer que le modèle fait bien ce qu'on lui demande de faire ! La validation, quant à elle, consiste à confirmer les suppositions que l'on avait faites sur le système lors de la phase de modélisation.

Contrairement aux modèles mathématiques où les résultats sont exacts—sous les hypothèses prises—ceux obtenus par la simulation sont toujours entachés d'incertitude. Cette incertitude provient des aspects aléatoires de la simulation tels que l'utilisation de générateurs de nombres aléatoires. Tandis qu'un résultat obtenu analytiquement est immuable, celui obtenu par une simulation ne représente qu'un échantillon parmi d'autres. Il est alors toujours souhaitable de préciser les *intervalles de confiance* lorque l'on fourni des résultats de simulation. L'intervalle de confiance permet de juger de la validité des résultats de simulation. Plus cet intervalle est petit et plus les résultats fournis sont susceptibles de représenter la caractéristique que l'on étudie. Le calcul de l'intervalle de confiance suppose l'existence d'un état stationnaire. A partir de cette hypothèse, il existe, pour la simulation, plusieurs méthodes de calcul parmi lesquelles nous notons les méthodes suivantes :

- ***Réplications indépendantes*** : La méthode des réplications indépendantes consiste à lancer n simulations indépendantes avec des séquences aléatoires différentes pour obtenir à chaque fois une estimation du critère à étudier. Ces valeurs permettent alors de déterminer l'intervalle de confiance. Cette méthode a cependant l'inconvénient d'être sensible aux effets du régime transitoire.
- ***Batch means*** : La méthode des batch means ne nécessite qu'une seule simulation mais la divise en n périodes. Les critères de performances dans chaque période sont alors considérés comme autant d'échantillons indépendants pour le calcul de l'intervalle de confiance. Le choix de n reste un problème ouvert.

– ***Points de regénération*** : La méthode des points de regénération repose sur l'existence dans certains systèmes de points très particuliers. Ces points particuliers, dits points de regénération, indiquent que le système a terminé un cycle et qu'il est sur le point d'en recommencer un autre en partant d'un même état initial[1]. On peut alors séparer la simulation en un nombre n de séquences pour calculer l'intervalle de confiance. Certains systèmes ne présentent malheureusement que peu ou pas du tout de points de regénération.

Dans le reste du chapitre, nous allons essentiellement nous intéresser à l'étape 2. Nous présenterons les principes de la simulation à événements discrets qui constitue de loin la méthode la plus utilisée pour l'évaluation de performances des systèmes informatiques. Ensuite quelques outils disponibles pour la simulation, à savoir les outils orientés langage et ceux orientés logiciel seront présentés. Tous ces outils sont séquentiels.

2.2 LA SIMULATION À ÉVÉNEMENTS DISCRETS

Le temps dans la nature évolue de manière continue—sans saut—et n'importe quel système physique évolue donc de manière continue. Cependant, si l'on définit un ensemble fini de changements auquels on désire s'intéresser plus particulièrement, alors on peut considérer que le système évolue de manière discrète, uniquement lorsque des changements précis se produisent. Cependant, certains systèmes restent néanmoins continus car les discrétiser complètement reviendrait à considérer des instants infinitésimaux. Par exemple, si l'on s'intéresse à la trajectoire d'un objet dans le ciel, il est clair que la discrétisation complète de ce système est impossible. On peut toujours le discrétiser en considérant uniquement certains points de la trajectoire, mais l'utilité de l'étude est bien moindre. Nous dirons donc que ce système est continu. Par contre, si on regarde maintenant une banque avec des entrées et des sorties de clients, on peut facilement discrétiser ce système en ne considérant que les changements induits par l'arrivée ou le départ de clients. Bien entendu, il se passe toujours quelque chose même si aucun de ces changements ne se produit, mais on ne s'y intéresse pas et cela ne change en rien notre étude. Nous dirons alors que ce système est discret.

La simulation à événements discrets (SED) s'applique à des modèles discrets de systèmes physiques. Les évolutions du modèle ne sont supposées se produire qu'à des instants particuliers du temps, représentées par des événements. Ainsi donc la date d'occurrence de l'événement définit l'instant de changement du système.

Nous allons dans la suite détailler la gestion discrète du temps, la distinction entre l'approche orientée événement et l'approche orientée processus puis nous donnerons l'architecture générale d'une SED.

[1]Dans les systèmes de files d'attente, un point de regénération typique se produit lorsque la taille de la file redevient nulle.

2.2.1 La gestion discrète du temps

Deux approches existent pour l'évolution du temps dans la simulation, l'approche *synchrone* ou *incrément-fixe* et l'approche *asynchrone* ou *prochain-événement*. Dans la première approche, le temps de simulation avance par incrément fixe Δt. A la fin de chaque cycle, on regarde l'ensemble des événements qui s'y sont produits et on réactualise l'état du système en conséquence. Si un ou plusieurs événements se sont produits à l'intérieur d'un cycle, on considère qu'ils se sont tous produits logiquement à la fin du cycle. La figure 2.2 montre cette approche synchrone.

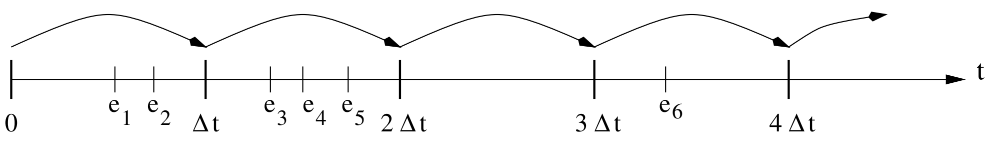


Figure 2.2 Approche *synchrone*.

Le choix de l'incrément Δt doit être fait judicieusement puisqu'il représente l'incrément minimal. L'inconvénient de cette méthode est alors de placer la contrainte du choix pour Δt sur le plus petit des incréments possibles du système et de considérer inutilement les instants où le système est inactif. Par contre, dans le cas de systèmes synchrones ou périodiques cette méthode est plus efficace.

Dans la deuxième approche, le temps de simulation va avancer par bonds variables, selon la date d'occurrence des événements. Un échéancier va contenir l'ensemble des événements estampillés, et va sélectionner le prochain événement à être exécuté. On réactualise l'état du système à la fin de chaque événement. Prenons l'exemple du système classique d'arrivées et de départs de paquets P_i dans un système de communication. Le temps de service d'un paquet P_i est noté S_i. Notons ensuite par a_i l'événement *arrivée du paquet* p_i et par d_i l'événement *départ du paquet* p_i. La figure 2.3 illustre le fonctionnement de l'approche asynchrone. L'avantage

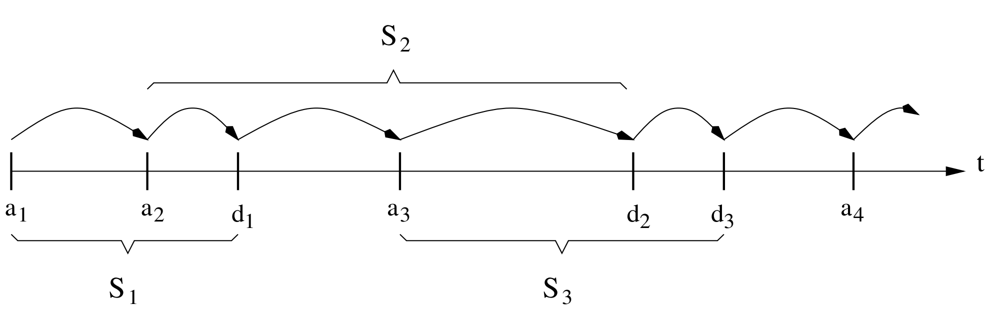


Figure 2.3 Approche *asynchrone*.

de cette méthode, par rapport à la méthode précédente est de ne considérer que les instants pertinents du système. Cette méthode est la plus utilisée par la plupart des langages et des simulateurs. Cependant, on peut assister à une explosion du nombre d'événements si les estampilles d'un événement à un autre sont trop proches. Dans ce cas, il est possible qu'une approche synchrone soit plus efficace.

2.2.2 Approche orientée événement et approche orientée processus

On peut encore distinguer, lorsque l'on parle de simulation à événements discrets, entre l'approche orientée événement de celle orientée processus. Toutes deux utilisent la notion d'événement mais diffèrent sur la façon dont un système donné est représenté et codé. Dans la première approche, on va énumérer tous les événements susceptibles de se produire dans un système, et on va associer une procédure à chacun de ces événements. Le temps n'évoluant pas à l'intérieur d'une telle procédure. Dans la deuxième approche on va associer une procédure non plus à chaque événement mais à un *processus*. Le terme processus se référant alors à une séquence ordonnée d'événements qui décrit l'évolution d'une certaine entité à travers le système. A l'intérieur de la procédure, le temps peut évoluer.

Cette distinction est relativement difficile à percevoir pour la première fois. Nous allons l'illustrer par un exemple simple. Considérons un système d'attente quelconque que nous représenterons par une file d'attente. Deux événements sont distinguables, les arrivées et les départs de clients[2]. Dans l'approche orientée événement, deux procédures vont être définies dont nous donnons l'organigramme dans la figure 2.4. Les

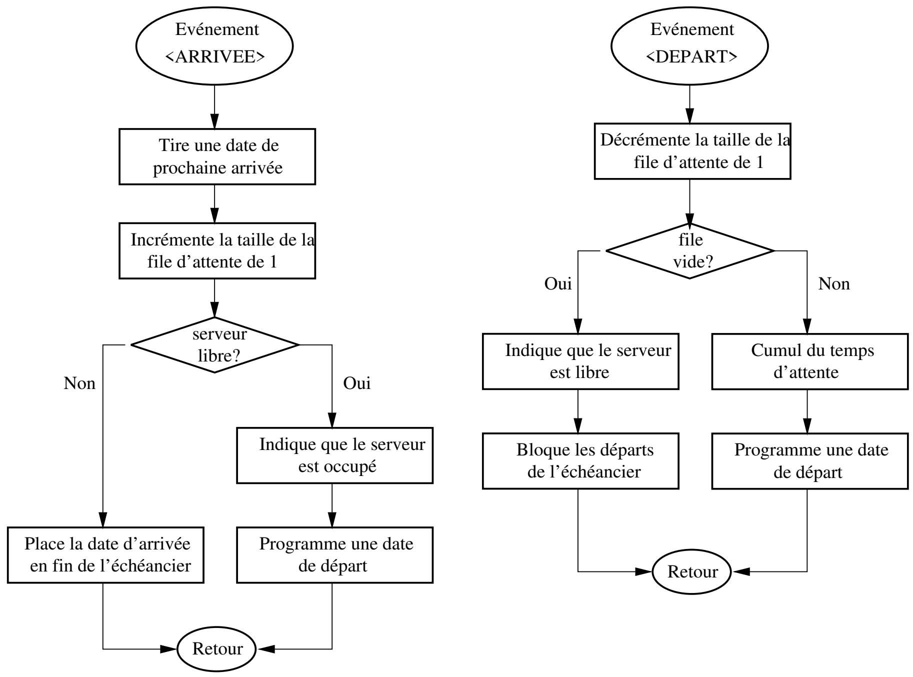


Figure 2.4 Approche orientée événement.

[2]Notons que le nombre de clients dans la file d'attente comprend celui en service.

deux procédures fonctionnent de la manière suivante :

1. **Procédure Evénement <ARRIVEE> :** Lorsqu'un client se présente, on programme la date d'arrivée du prochain client et on incrémente le nombre de clients en attente. Ensuite, si le serveur est inactif, le client est immédiatement servi et on programme une date de départ. Par contre, si le serveur est occupé, le client reste dans la file d'attente et on sauve la date d'arrivée en fin d'échéancier.
2. **Procédure Evénement <DEPART> :** Si la file d'attente devient vide après le départ on bloque les départs puisque les seuls événements possibles sont maintenant des arrivées. S'il y a par contre des clients en attente, on prend le premier et on programme une date de départ pour lui.

Dans l'approche orientée processus, une seule procédure va englober la séquence des événements associés à l'entité *CLIENT*. Cette procédure est illustrée par l'organigramme de la figure 2.5. Les transitions en pointillés indiquent que le temps de simulation peut évoluer. La procédure CLIENT fonctionne alors de la manière suivante :

- **Procédure CLIENT :** Lorsqu'un client arrive, on programme la date de prochaine arrivée. Ensuite, si le serveur est occupé, on le place en file d'attente et on attend qu'il soit servi. Pendant ce temps, le temps de simulation peut évoluer pour prendre en compte d'autres événements—autres arrivées ou un départ. Par contre, si le serveur est libre, on l'occupe, on programme une date de départ et on attend la fin du service. Pendant ce temps, le temps de simulation peut évoluer pour prendre en compte d'autres événements.

L'approche orientée processus adopte une vision plus macroscopique et est d'une certaine façon plus naturelle à concevoir. En général, on constate que le programme comporte moins de lignes si l'on utilise cette approche.

2.3 LES OUTILS DE SIMULATION

Les outils de simulation peuvent être divisés en deux catégories : les langages et les logiciels. Dans la première catégorie, beaucoup de programmation est nécessaire mais plus de flexibilité est offerte. Dans la seconde, la programmation est moins importante, voire inexistante, pour une flexibilité moindre mais par contre pour une facilité d'utilisation, une convivialité et parfois des possibilités d'analyse beaucoup plus grandes. Nous allons d'abord présenter les langages puis les logiciels.

2.3.1 Les langages de simulation

Il n'est pas extrêmement difficile d'écrire une simulation avec un langage non spécialisé tel que FORTRAN, PASCAL ou C. Il suffit en effet d'implanter un mécanisme d'horloge, des générateurs de nombres aléatoires et des structures de données pour

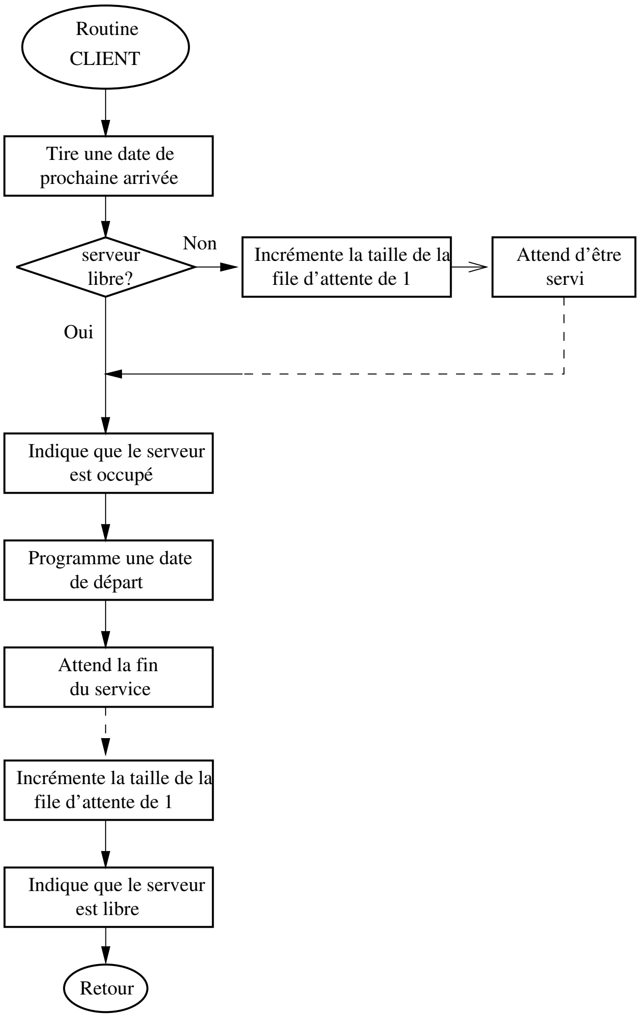


Figure 2.5 Approche orientée processus.

les événements et les listes. Les avantages à utiliser un langage général sont les suivants :

- Beaucoup de personnes connaissent déjà un langage comme FORTRAN, PASCAL ou C. Ces langages sont généralement disponibles sur la plupart des ordinateurs.
- Si le programme est bien écrit, il sera sûrement plus rapide en temps d'exécution que son homologue écrit avec un langage spécialisé.
- Les langages généraux offrent plus de flexibilité de programmation et sont généralement moins cher.

D'un autre côté, les langages dédiés à la simulation offrent en standard des fonctionnalités propres à la simulation comme les générateurs aléatoires, la gestion du temps et de la file des événements. Les avantages de tels langages sont les suivants :

- Un temps de développement plus réduit grâce à la fourniture en standard d'un grand nombre de fonctionnalités. On peut noter que la gestion de la mémoire est dans la plupart des langages faite automatiquement de manière dynamique.
- La maintenance de tels programmes est plus aisée.
- Le déboggage est plus facile car les erreurs sont moins "subtiles" et le programme est généralement moins long.

Le choix d'un langage, général ou spécialisé, se fera sans doute en fonction de la taille et de la complexité du modèle à étudier. Nous pensons que plus la complexité sera grande, et plus le choix se tournera vers un langage spécialisé. L'avantage de la vitesse d'exécution des langages généraux s'effaçant alors peu à peu, avec la disponibilité d'ordinateurs de plus en plus rapide, devant les facilités de développement offertes par les langages spécialisés. Maintenant, si on se décide pour un langage spécialisé, le choix de tel ou tel langage reste ouvert.

Nous allons dans la suite présenter brièvement SIMULA, GPSS et SIMSCRIPT II.5 qui sont les langages les plus utilisés dans le domaine de la simulation séquentielle.

SIMULA

SIMULA a été développé au Norvegian Computing Center vers 1963 pour aboutir en 1967 à SIMULA 67 [Dahl67]. SIMULA est à l'origine parti du langage ALGOL 60 pour ensuite ajouter des concepts spécialisés pour la simulation. On doit noter que SIMULA est le premier langage à introduire les concepts d'objets et de classes. C'est un langage qui possède une large communauté d'utilisateurs. Cependant, débuter avec SIMULA n'est pas très aisé car le langage n'est pas d'une grande clarté.

GPSS

GPSS (General Purpose Simulation Language) est, comme son nom l'indique, un langage de simulation très général. Il a été développé chez IBM par G. Gordon en 1961 pour se stabiliser vers 1972 sous le nom de GPSS V [IBM71][Bobi76]. IBM n'a par la suite plus supporté GPSS qui a donc été repris par d'autres personnes. J. Henriksen[3] a développé GPSS/H en 1977 alors que la société Springer Cox a développé GPSS/PC.

La simulation en GPSS est orientée processus et le langage possède un certain nombre d'instructions et de ressources de base très utiles pour la simulation. Chaque instruction peut être vue comme un *bloc* et pour construire un modèle, il suffit de connecter les différents blocs entre eux. La figure 2.6 est un petit exemple de simulation d'une unité centrale issue de [Brat87].

GPSS est facile à utiliser et convient parfaitement pour les modèles simples ou peu complexes. Cela principalement grâce à sa vision en blocs *ad hoc*. D'un autre

[3] Henriksen affirme que le "H" dans GPSS/H provient de la version du Fortran qu'il a utilisé.

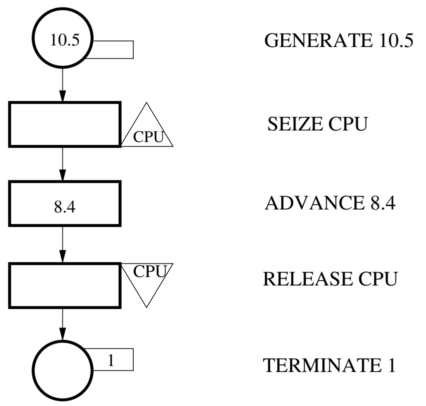


Figure 2.6 Simulation en GPSS. A gauche, l'équivalent en diagramme du programme.

côté, dès que la complexité du modèle augmente un peu, l'usage de GPSS devient difficile voire impossible.

SIMSCRIPT II.5

SIMSCRIPT II.5 est un lointain descendant de SIMSCRIPT, qui était très proche du langage FORTRAN, sans être compatible avec lui. C'est actuellement la version de SIMSCRIPT la plus répandue [Law84] distribuée par CACI Inc. Il a été développé par H. Markowitz et H. Karr avec le concours de P. Kiviat et R. Villaneuva de RAND Corporation et implémente les deux approches orientées événement et orientées processus. Le langage possède une syntaxe riche ayant malheureusement l'inconvénient d'être un peu lourde et peu concise. Malgré cela, SIMSCRIPT II.5 est un langage à instructions offrant une grande panoplie de fonctions et de structures spécialisées pour la simulation. En outre, la collecte des statistiques se fait très facilement. Un programme de simulation en SIMSCRIPT II.5 présente toujours une partie Preamble de la forme :

```
every ECOLE
 has an DIRECTEUR,
 some TEACHERS and
 owns a ENSEMBLE.CLASSES

every CLASSE
 has a PROF.PRINCIPAL,
 a NIVEAU.CLASSE and
 belongs to an ENSEMBLE.CLASSES
```

un point d'entrée Main et des routines qui peuvent être récursives. De nombreuses simulations de combats militaires et de chaînes de production ont été réalisées en SIMCRIPT II.5 avec succès. Bien que SIMSCRIPT II.5 puisse également modéliser des systèmes continus, une version plus élaborée, C-SIMSCRIPT, permet plus spécifiquement l'étude de tels systèmes.

2.3.2 Les logiciels de simulation

Les logiciels de simulation sont des simulateurs généralement dédiés à une classe particulière d'applications. Leur avantage par rapport aux langages réside dans leur facilité d'utilisation avec peu ou pas du tout de programmation. Cette convivialité est à l'origine de l'engouement croissant pour ces outils. Nous présentons ci-après quelques uns de ces outils, que nous classons selon le degré de programmation nécessaire à une mise en oeuvre minimale.

Qnap2

QNAP2 (Queuing Network Analysis Package) [QNAP84] est un outil logiciel de description et d'analyse de réseaux de files d'attente. Ce n'est pas un logiciel complet dans la mesure où la phase de modélisation doit être au préalable effectuée à l'aide de files d'attente avant d'être décrite dans un langage propre à QNAP. C'est donc plutôt un outil de description, mais surtout un outil de résolution puisqu'il comporte plusieurs modules de résolution, dont un module de simulation à événements discrets, qui fournissent des résultats numériques caractérisant les performances du modèle.

La force de QNAP réside dans la diversité de ses modules de résolution, résolution exacte ou approximative, qui peuvent tous être utilisés indifféremment pour résoudre un même modèle, à condition, bien sûr, que le modèle satisfasse les hypothèses d'application de la méthode.

Pour terminer avec QNAP, notons que c'est un produit issu du travail d'équipes de chercheurs de l'INRIA et de Bull écrit en Fortran 77. Le produit est actuellement diffusé par la société Simulog et son utilisation au sein des universités est très forte.

OPNET

OPNET (OPtimized Network Engineering Tool) [OPNET93] est un simulateur spécialisé dans la simulation de réseaux de communication. Il est commercialisé par MIL3 et constitue l'aboutissement de recherches commencées en 1984 au MIT.

Il adopte une vision hiérarchique à 3 niveaux : niveau *processus*, niveau *noeud* et niveau *réseau*. Ce dernier niveau étant le plus élevé. L'utilisateur y décrit la topologie de son réseau en plaçant les machines et les éléments d'interconnexion (bus, anneaux, etc.). La description d'une machine s'effectue au niveau noeud où l'utilisateur construit une architecture de machine en assemblant différents modules (module transport, module Ethernet, etc.) qui sont reliés entre eux par des canaux de communication. Enfin le niveau le plus bas permet à l'utilisateur de décrire un module par le biais d'un automate d'état. A chaque état peut être associé un code C décrivant les actions à effectuer.

Le langage C utilisé est enrichi de primitives propres à OPNET qui permettent de réaliser les principales actions que l'on peut effectuer sur un réseau (envoi et réception de paquets par exemple). Le logiciel est livré avec un grand nombre de modules déjà définis. Cela permet dans un premier temps de construire sans aucune programmation des modèles de réseaux tels que FDDI, Ethernet, etc.

On lance une simulation en exécutant un programme binaire résultant de la compilation des différents modules. Ce programme est centré autour d'un noyau de simulateur à événements discrets. OPNET est un outil relativement complet puisqu'il offre des outils de représentation des données et quelques fonctions d'analyse statistique. C'est un logiciel très convivial et puissant.

BONeS Designer

BONeS (Block Oriented Network Simulator) [Shan90] est également un simulateur spécialisé dans la simulation de réseaux de communication. Le principe de modélisation du système informatique est celui de la décomposition hiérarchique, la topologie du réseau étant décrite sous forme de *blocs* ou de sous-modèles. En cela, BONeS est très similaire à OPNET. Cependant, BONeS utilise de manière plus intensive la réutilisation des blocs. Contrairement à OPNET où il est courant d'écrire ses propres automates d'états, cette tâche est exceptionnelle et plus ardue dans BONeS.

De même que dans OPNET, le simulateur est un programme binaire généré à partir du code C décrivant les modules qui tourne autour d'un noyau de simulateur à événements discrets.

2.4 AUTRES FORMES DE SIMULATION

Nous nous sommes particulièrement intéressés à la simulation à événements discrets parqu'elle est très bien adaptée à l'étude des systèmes informatiques. D'autres types de simulations existent, adaptés à d'autres applications ou répondant à d'autres attentes. Nous les énumérons ci-dessous.

Simulation de type Monte-Carlo

Une simulation est dite de type *Monte-Carlo* si le temps n'a pas d'influence sur les résultats. Ce sont essentiellement des simulations *stochastiques* utilisant des nombres aléatoires —typiquement uniformes U(a,b)—pour simuler des phénomènes probabilistes. Historiquement, cette méthode prend son origine durant la seconde guerre mondiale, où elle a été appliquée pour le développement de la bombe atomique. On l'utilise également beaucoup pour l'évaluation d'expressions non-probabilistes telles que des intégrales multiples très complexes, difficilement résolvables par des méthodes d'analyse numérique.

Simulation par trace

La simulation par trace (*trace-driven simulation*) consiste à utiliser des séquences enregistrées sur un système réel pour les injecter à l'entrée d'un modèle logiciel du système que l'on étudie. On peut ensuite modifier et optimiser le modèle logiciel et évaluer l'impact de telle ou telle modification. Cette approche a été appliquée avec beaucoup de succès dans l'étude des systèmes de stockage, notamment pour les mémoires caches [Smit82]. L'avantage d'une telle méthode est que la séquence de traces appliquée en entrée reflète un scénario de fonctionnement réel et non une quelconque modélisation. Cependant, il est difficile d'obtenir des traces représentatives, et celles-ci sont malheureusement très intimement liées à une configuration de machine particulière.

Simulation à temps continu

La simulation continue s'intéresse aux systèmes où l'évolution se fait de manière continue. Dans la plupart des cas, l'évolution du système est régi par un ensemble d'équations différentielles qui détermine l'état du système aux différents instants. Des langages tels que ACSL et CSSL-IV (cf. [Prat87]) ont été spécialement développés pour supporter la simulation de systèmes continus.

Simulation hybride

Le terme de simulation hybride désigne des simulations où certaines parties sont remplacées par des modules analytiques. Par exemple, si l'on étudie un système de communication en couches, on peut s'intéresser plus particulièrement à une couche donnée en remplaçant les autres couches par des délais obtenus de manière analytique [More93]. Pour obtenir ce module analytique, on peut soit résoudre de manière analytique le modèle, soit réaliser une série de simulation permettant d'approximer la fonction de distribution des valeurs statistiques du modèle [More93][Gosw93]. Cette deuxième approche est la plus utilisée car résoudre analytiquement le modèle suppose que celui-ci soit simple.

Simulation orientée objet

SIMULA a été le premier langage de simulation à introduire la notion d'objet. Depuis, la simulation a suivi le courant de "l'orientation objet" amorcé sur les langages pour proposer la simulation orientée objet. Ce n'est pas à proprement parler un nouveau type de simulation, mais plutôt un nouveau formalisme de modélisation et de programmation. On retrouve alors les notions d'héritages (simples ou multiples), de niveau d'abstraction etc. rendues célèbres par les langages à objets.

2.5 CONCLUSION DU CHAPITRE

Nous avons présenté quelques notions essentielles dès lors que l'on souhaite se lancer dans un processus de simulation. La popularité de la simulation ne fait aucun doute. L'utilisation de simulateurs ou de langages est largement répandue, et cela par un nombre de plus en plus grand de personnes qui sont de moins en moins spécialisées[4]. Cela est une bonne chose. Cependant, le processus de modélisation et de

[4] Un peu moins vrai pour les langages toujours plus difficiles à aborder pour un néophyte.

construction du modèle reste un investissement très lourd en temps. Les problèmes sont nombreux, et même si les langages dédiés à la simulation offrent des facilités accrûes par rapport aux langages généraux, ils restent tout de même sujets aux problèmes inhérents à tout langage, à savoir la nécessité de savoir programmer, le codage tout de même long et fastidieux, et la correction des erreurs encore laborieuse. Notons également que les modifications peuvent être difficiles si le langage n'est pas assez structuré, ou si le programmeur n'a pas une grande expérience en développement. D'un autre côté, l'utilisation d'un simulateur est très attrayante, mais là également, bons nombres de pièges peuvent se présenter. En particulier, la falicité avec laquelle on peut modéliser un système avec un niveau de détail aussi grand que l'on souhaite peut très rapidement transformer le modèle en "usine à gaz" totalement inexploitable. Finalement, la maîtrise des résultats de simulation et leur interprétation demeurent essentiels.

Pour en savoir plus...

Nous avons délibérément donné un grand nombre de références tout au long de ce chapitre. Les lecteurs intéressés par un sujet précis pourront se reporter à ces références. Aujourd'hui, la littérature sur la simulation est très abondante, il serait difficile de citer tous les ouvrages tant ils sont nombreux. Les lecteurs peuvent néanmoins consulter [Bank84,Brat87,Law91] qui traitent tous les trois de manière assez complète la simulation dans son ensemble. Ils peuvent également consulter [Keij74] et [Jain92] pour avoir plus de détails concernant l'analyse des résultats. Il est difficile de conseiller la lecture d'un livre particulier pour la phase de modélisation, cependant s'il faut en conseiller un, nous pensons que [Fish95] est une bonne lecture. De manière plus générale, Fishwick propose dans [Fish94] une réflexion sur ce que doit être une simulation et ce que l'on attend d'elle, dans différents domaines d'application tels que l'intelligence artificielle ou les applications graphiques.

Chapitre 3

Les algorithmes de simulation parallèle et distribuée

Avant de lire ce chapitre...

Ce chapitre constitue l'état de l'art sur les concepts et les techniques utilisées en simulation parallèle et distribuée. Le lire entièrement peut être fastidieux. Nous donnons ci-après le plan général afin de guider le lecteur vers une lecture adaptée à ces besoins. Ce chapitre est organisé de la manière suivante :

- *Section 3.1 Introduction* : Description des différentes méthodes de parallélisation et notion de fautes temporelles (lecture conseillée).
- *Section 3.2 Notion de base et définitions* : Concepts de base de la simulation parallèle et distribuée (lecture conseillée).
- *Section 3.3 Les approches purement conservatrices* : Définition du lookahead (3.3.1), algorithme de base (3.3.2) et null-messages (3.3.3). Nous conseillons de lire ces trois sous-sections. Les lecteurs intéressés par les multiples variations au mécanisme de null-messages peuvent lire (3.3.4), ceux qui s'intéressent aux approches à fenêtre peuvent lire (3.3.6).
- *Section 3.4 Les approches purement optimistes* : Présentation de Time-Warp (3.4.1), du mécanisme du GVT (3.4.2), de la sauvegarde optimisée des états (3.4.3) et des stratégies évoluées d'annulation (3.4.4). La lecture des 4 sous-sections (3.4.1) à (3.4.4) est conseillé, excepté la dernière partie de (3.4.4) sur l'exécution conditionnelle.
- *Section 3.5 Les approches mixtes et adaptatives* : Présentation des variations au schéma optimiste (lecture de l'introduction conseillée). Les lecteurs intéressés par les protocoles à fenêtre peuvent lire (3.5.1), ceux intéressés par les approches bloquantes peuvent lire (3.5.2).

– *Section 3.6 Décomposition temporelle* : Présentation d'une autre technique de parallélisation basée sur le temps et non l'espace (lecture facultative).
– *Section 3.7 Synthèse de la taxonomie* : Classement des différents algorithmes présentés dans ce chapitre suivant des critères représentatifs (lecture conseillée, tableau récapitulatif pouvant servir de référence).
– *Section 3.8 Les outils : Langages et environnements* : Présentation des outils disponibles pour la simulation parallèle et distribuée (lecture de l'introduction conseillée). (3.8.1) présente les langages et (3.8.2) les environnements.

Nous espérons que ces indications vous seront utiles pour organiser votre lecture. Elles ne constituent en aucun cas une obligation.

3.1 INTRODUCTION

La simulation a tout d'abord été séquentielle. Cela pour deux raisons : premièrement parce que c'était la manière la plus simple de procéder, et deuxièmement parce que les ordinateurs de l'époque n'avaient pas l'incroyable puissance de calcul parallèle qu'ils ont acquise ces dernières années. Dans cette approche séquentielle, tous les processus logiques (*pl*) sont simulés sur un seul processeur et la liste des événements (LE) est ainsi centralisée. Ensuite la puissance des ordinateurs s'est considérablement accrue, en même temps que le besoin de simuler des systèmes de plus en plus grands et de plus en plus complexes. Cela a conduit tout naturellement à chercher des mécanismes qui permettraient d'une part de réduire le temps de simulation et d'autre part d'augmenter la taille du système. La parallélisation est un tel moyen et différents niveaux existent :

Niveau application : La parallélisation au niveau application est plus connue sous le terme de *réplication*. Comme il n'y a aucune synchronisation entre les différentes réplications, il n'y a aucun coût supplémentaire. Cependant cette méthode n'est pas applicable lorsque les simulations sont dépendantes les unes des autres (paramètres d'entrée dépendant des résultats de la simulation précédente) et ne permet pas de réduire le temps de la simulation lorsque le système est grand.

Niveau routine : Au niveau routine, ce sont les différentes composantes du simulateur telles que les procédures d'exécution des événements, les fonctions de collecte de statistiques ou les fonctions génératrices de nombres aléatoires qui sont distribuées. Cependant cette méthode n'est pas très performante car le nombre de routines qui peuvent être distribuées est généralement réduit.

Niveau événement - LEV centralisée : Au niveau événement, on cherche à distribuer les événements sur un certain nombre de processeurs pour les exécuter en parallèle. Lorsque la liste des événements est centralisée, la parallélisation consiste à répartir les événements de la LEV, gérée par un processeur maître, sur les processeurs esclaves. Le processeur maître doit prendre garde à préserver la cohérence de la

LEV ; en particulier il est difficile d'exécuter parallèlement des événements dont les estampilles sont très différentes.

Niveau événement - LEV distribuée : Lorsque les LEVs sont distribuées, le niveau de parallélisme est maximum dans le sens où il est maintenant possible d'exécuter de manière concurrente des événements très éloignés les uns des autres. Malheureusement, il est nécessaire d'instaurer un mécanisme de synchronisation explicite pour la préservation de la causalité. C'est néanmoins le mode de fonctionnement qui est susceptible de fournir le plus de souplesse et de gain, et c'est pourquoi il retient l'attention de la majorité des travaux sur la simulation parallèle ou distribuée.

Nous allons nous intéresser plus particulièrement à ce dernier niveau de parallélisation et dans la suite de ce chapitre, nous allons dresser un état de l'art de ce qui existe actuellement. Tout d'abord, cette distribution spatiale de la simulation soulève un nouveau problème, celui des *fautes temporelles* dont nous donnons tout de suite une définition.

> **Définition 3.1 (Faute temporelle)** *Une faute temporelle se produit lorsque le temps logique prend une valeur inférieure à celle qu'il avait à l'instant d'avant.*
>
> *Dans le cas discret où le temps change à des instants précis, si l'on note par t_i la valeur du temps à l'instant i, alors une faute temporelle survient si $t_{i+1} < t_i$.*

Cette notion de faute temporelle est très importante. Intuitivement, on se rend compte que simuler un système revient à reproduire son comportement au cours du temps. Dans notre monde réel il n'est pas possible de retourner dans le passé car le temps s'écoule inexorablement vers l'avant. L'utilisation d'une machine pour modéliser et simuler un système physique permet cependant toutes les fantaisies, et on peut à loisir modifier l'écoulement du temps puisque celui-ci est maintenant virtuel et ne dépend plus que du logiciel. Cependant, il ne sert à rien de simuler un système si son comportement ne peut être bien reproduit. Cela implique donc que notre temps logique doive suivre les mêmes lois que celles de la nature, et doit par conséquent évoluer de manière strictement croissante. Pratiquement, cela revient à s'assurer que le temps logique prenne des valeurs toujours croissantes sans jamais retourner en arrière. Les études concernant la simulation parallèle s'intéressent alors aux différents mécanismes de synchronisation entre les processus logiques pour obtenir une évolution cohérente du temps logique.

Nous allons maintenant donner quelques notions de base et définitions pour une bonne compréhension des problèmes et des mécanismes de la simulation parallèle.

3.2 NOTIONS DE BASE ET DÉFINITIONS

3.2.1 Message et événement

Les notions de message et d'estampille sont les mêmes que celles introduites par Lamport dans le cadre des horloges logiques [Lamp78]. Nous rappellons ici leur définition.

Définition 3.2 (Message) *Un message est une entité de données échangée entre deux processus logiques.*

Dans le cas de la simulation distribuée, un message envoyé par un pl A *à un* pl B *modélise une interaction entre les deux éléments physiques* A *et* B.

Définition 3.3 (Estampille d'un message) *L'estampille d'un message est une valeur réelle associée au message par la fonction temps.*

Chaque message peut se voir attribuer une signification locale ou non au sein du *pl*. Cette signification est un événement, dont nous donnons la définition suivante :

Définition 3.4 (Evénement) *Un événement est l'occurrence d'une action physique ou logique.*

Définition 3.5 (Date d'occurrence) *L'instant ou date d'occurrence d'un événement* e *est un réel associé à l'événement* e*, noté* $t(e)$*, par la fonction temps qui indiquera la date à laquelle l'événement doit se produire.*

On utilisera également le terme estampille pour désigner la date d'occurrence d'un événement.

Relation 1 (Message-Evénement) *Un message lorsqu'il arrive dans un* pl *est transformé en l'événement correspondant. La date d'occurrence de l'événement est alors l'estampille portée par le message.*

Notation 1 (D'un événement) *Nous noterons un événement* e *d'estampille* t *par* $< e, t >$*, ce qui signifie que l'événement* e *doit se produire à l'instant* t.

3.2.2 Modélisation et processus logique

De manière générale, lorsque l'on distribue les événements et les listes d'événements on adopte une modélisation qui consiste à diviser le système d'étude en un ensemble de petits systèmes, chacun d'entre eux étant simulé par un *pl* qui communique avec les autres *pls* par des échanges de messages. Ces messages sont le plus souvent estampillés pour introduire la notion de temps dans le modèle et ordonner ainsi l'exécution des événements au sein du *pl*.

La figure 3.1 illustre la modélisation d'un système physique par un ensemble de *pls* : le système physique considéré est un réseau (par exemple ATM) constitué

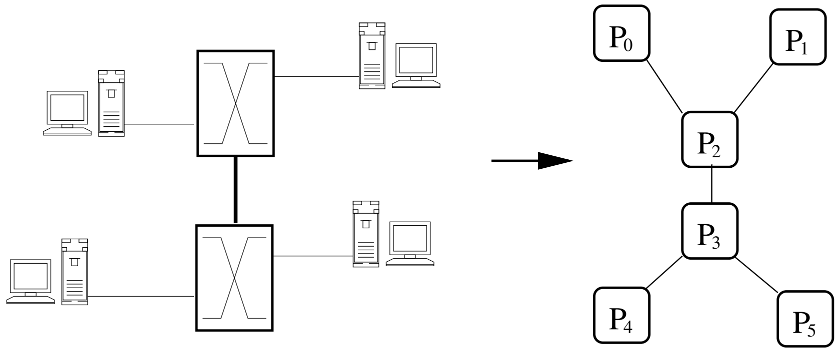


Figure 3.1 Un système physique constitué de machines et d'autocommutateurs ATM.

de machines et d'autocommutateurs. La modélisation adoptée représente chaque constituant du réseau par un processus logique.

Chaque *pl* est autonome car il possède une horloge et une file d'événements contenant tous les événements reçus. Il peut donc évoluer indépendamment des autres. Lorsque la file des événements et l'horloge logique sont uniques, on peut connaître l'état global du système entier et il n'est pas difficile d'éviter les fautes temporelles. En effet, l'unicité de la file d'événements permet un tri total qui détermine définitivement le prochain événement exécutable parmi tous les événements. Il suffira ensuite de garantir que l'horloge logique est strictement croissante pour éviter toute faute temporelle. Avec la simulation parallèle, de telles erreurs peuvent malheureusement se produire. Comme les horloges logiques et les listes d'événéments sont maintenant distribuées sur un ensemble de processeurs, la synchronisation entre les différents processus n'est plus du tout triviale. Nous allons illustrer notre propos par un exemple simple qui consiste à simuler un protocole de communication entre deux machines.

Simulation d'une communication entre deux ordinateurs N_1 et N_2. On suppose que le délai du lien est de 5 unités de temps (ut) et que les paquets à envoyer de N_1 à N_2 sont P_1, P_2 et P_3 et qu'ils se présentent à N_1 aux instants 5,12 et 25. Le protocole demande que N_2 acquitte tout paquet reçu 1 ut après la réception. Tous les événements sont listés ci-après :

Evénement	*Signification*
$< e1, 5 >$	Arrivée du paquet P_1 sur N_1.
$< e2, 10 >$	Fin du service de P_1 et envoi de P_1 à N_2.
$< e3, 12 >$	Arrivée de P_2 sur N_2.
$< e4, 15 >$	Réception de P_1 sur N_2.
$< e5, 16 >$	Envoi de $ACK(P_1)$ par N_2 à N_1.
$< e6, 17 >$	Fin du service de P_2 et envoi de P_2 à N_2.
$< e7, 21 >$	Réception de $ACK(P_1)$ sur N_1.
$< e8, 23 >$	Réception de P_2 sur N_2.
$< e9, 25 >$	Arrivée de P_3 sur N_1.
...	...

L'exécution en parallèle des événements dans cet exemple (figure 3.2) implique que N_1 n'a pas à attendre N_2 pour exécuter ses événements en attente. Cependant, en faisant cela, il prend le risque d'aller trop loin dans le futur et de générer une faute temporelle s'il exécute l'événement e_9 avant de recevoir l'acquittement—événement e_7—provenant de N_2.

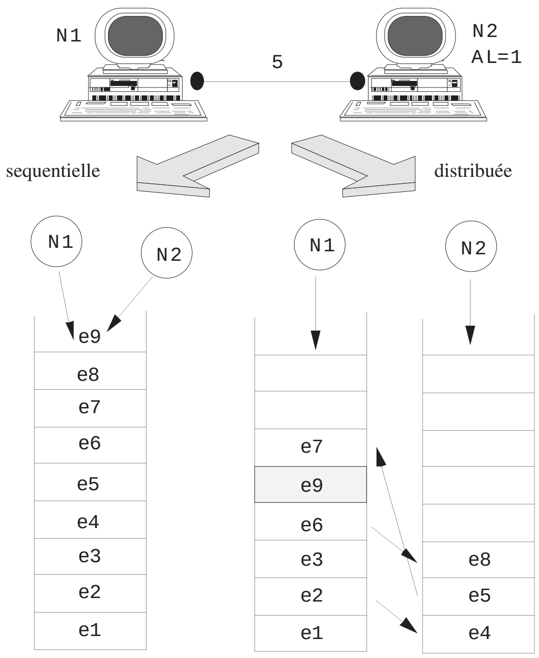


Figure 3.2 Simulation séquentielle contre simulation parallèle. Une flèche d'un événement e_x vers un événement e_y signifie que e_x a généré e_y.

3.2.3 Contraintes de causalité

Afin d'éviter les fautes temporelles, il est nécessaire d'introduire des mécanismes de synchronisation pour la gestion des horloges logiques et des files d'événements. Eviter les fautes temporelles revient en fait à respecter la causalité définie par la contrainte de précédence suivante :

> **Définition 3.6 (Contrainte de précédence)** *Soient $< e, t >$ et $< e', t' >$ deux événements tels que $t < t'$, alors l'exécution de e doit se faire avant celle de e'.*

Comme les différents *pls* évoluent de manière asynchrone, les valeurs des horloges locales sont généralement différentes ce qui rend impossible la détermination de la plus petite valeur d'horloge du système global. Cependant, dans [Fuji90] l'auteur donne une condition suffisante pour le respect de la précédence :

> **Condition 1 (Respect de la précédence)** *Si les différents* pls *ne communiquent que par échange de messages estampillés, les contraintes de*

précédence sont respectées si chaque pl *exécute les événements tels que les estampilles soient toujours croissantes.*

La difficulté de la simulation parallèle réside également dans la complexité de ces relations de causalité. Dans [Jeff91] l'auteur en énumère quatre types qui sont définis comme suit :

Relation 2 (Successeur/Prédécesseur) *Un événement* $< e, t >$ *est le prédécesseur* | *successeur de* $< e', t' >$ *si* e *et* e' *sont sur le même* pl *et* $t < t' \mid t > t'$, *et s'il n'existe pas* $< e'', t'' >$ *tel que* $t < t'' < t' \mid t > t'' > t'$.

Relation 3 (Antécédent) *Un événement* $< e, t >$ *est l'antécédent de* $< e', t' >$ *si* e *est à l'origine de* e'. *On notera* $e = a(e')$.

Relation 4 (Conduit à) *Un événement* $< e, t >$ *conduit à* $< e', t' >$, *noté* $e \to e'$, *si* e *est le prédécesseur ou l'antécédent de* e'.

Relation 5 (Influence) *Un événement* $< e, t >$ *influence* $< e', t' >$, *noté* $e \gg e'$, *s'il existe une séquence* $e_0, e_1, e_2, \ldots e_n$ *avec* $e_0 = e$ *et* $e_n = e'$ *telle que* $\forall i \in [0, n[$, *on ait* $e_i \to e_{i+1}$.

Deux approches sont proposées pour résoudre ces problèmes de synchronisation : l'approche *conservatrice* et l'approche *optimiste*.

3.3 LES APPROCHES PUREMENT CONSERVATRICES

3.3.1 La notion de *lookahead*

Le terme *lookahead* désigne la faculté de prédire ce qui se passera, ou ne se passera pas, dans un futur proche. La forme la plus courante de *lookahead* se présente dans le cas d'applications connaissant à l'avance le temps de traitement des messages reçus. L'exemple du serveur FCFS à temps de service S connu possède un horizon de prédiction égal à S. Pratiquement, cela signifie qu'entre deux messages générés par ce *pl*, soit $< e, t >$ et $< e', t' >$, on a $t' - t \geq S$.

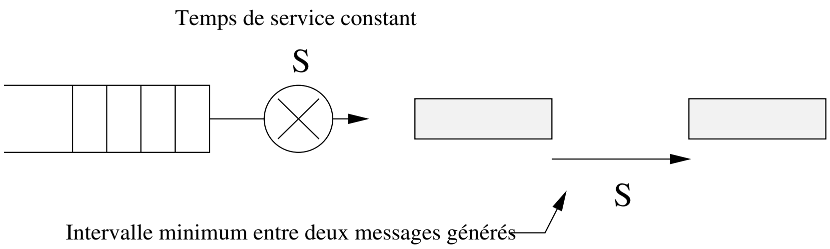


Figure 3.3 *lookahead* dans un système de file d'attente FCFS.

De manière plus générale, si un *pl* possède la connaissance de tous les événements qui se produiront jusqu'à la date T, et s'il peut prédire tous les événements qui seront générés jusqu'à la date $T + L$ alors il possède un *lookahead* de L.

Pendant la simulation, un *pl* doit souvent "regarder" dans le futur pour programmer de nouveaux événements. Dans l'exemple du serveur FCFS, à la réception d'un client, le serveur va "regarder" dans le futur pour programmer la date de départ de ce client. Cette faculté de prédiction va déterminer la rapidité avec laquelle le *pl* va prédire les nouveaux messages. Toujours pour le serveur FCFS, il est possible de connaître dès l'instant d'arrivée du client la date à laquelle celui-ci va quitter le système en fonction du nombre de clients présents dans la file. Dans ce cas, le nouvel événement pourra être programmé dès l'exécution du message d'arrivée. Par contre, si l'on instaure une priorité sur les clients, il n'est plus possible de déterminer à l'avance la date de départ car certains clients plus prioritaires peuvent arriver. Dans ce cas de figure, le *pl* doit attendre d'avoir avancé suffisamment loin dans le futur pour déterminer la date de départ.

Le plus souvent, la quantité de *lookahead* est une fonction complexe assez variable qui peut évoluer en fonction du temps de simulation et du type de message. Les performances d'une approche conservatrice sont très dépendantes du *lookahead*, il est alors nécessaire de connaître très précisémment l'application à simuler afin d'exploiter pleinement le *lookahead* pour obtenir des gains en vitesse significatifs [Fuji88].

Fujimoto fait remarquer que la valeur absolue du *lookahead* n'est en fait pas primordiale [Fuji88]. En effet, la valeur du *lookahead* doit être comparée avec la valeur de l'incrément des messages échangés. Si cet incrément est également grand, une grande valeur de *lookahead* ne signifie pas grand chose. Supposons maintenant qu'un message à la date T_{cause} produise un nouveau message à la date T_{effet} avec $T_{effet} \geq T_{cause}$, l'auteur introduit le *lookahead ratio*, noté LAR comme étant le rapport suivant :

$$LAR = \frac{T_{effet} - T_{cause}}{lookahead}$$

Lorsque LAR est grand, le "*lookahead* effectif" est petit et donc les performances sont moindres.

3.3.2 Ce que proposent Chandy-Misra-Bryant

Chandy, Misra et Bryant proposent respectivement dans [Chan79] et [Brya77] un algorithme simple permettant une exécution sans faute temporelle mais avec blocage éventuel. Cette approche dite conservatrice a été la première à être proposée pour la simulation distribuée et elle résout les problèmes de synchronisation et des fautes temporelles en se gardant dans tous les cas d'exécuter un événement $< e', t' >$ avant un événement $< e, t >$ si $t < t'$. L'exécution d'un événement $< e, t >$ ne pourra se faire que lorsqu'il sera certain qu'aucun événement $< e', t' >$ ne pourra être reçu avec $t' < t$.

Hypothèses

- Entre deux *pls* qui communiquent, il existe un lien. Une file FIFO associée à chaque lien contient les événements reçus.

– Les événements sont datés de manière strictement croissante par chaque *pl*.

– Chaque lien i possède une horloge h_i qui prend la valeur de l'estampille du premier événement de la file, si la file n'est pas vide, sinon elle contient l'estampille du dernier événement exécuté.

Algorithme

1. Chaque *pl* sélectionne de manière circulaire le lien dont l'horloge est la plus petite et exécute l'événement en tête de la file du lien sélectionné.
2. Si la file avec l'horloge la plus petite est vide, le *pl* bloque.

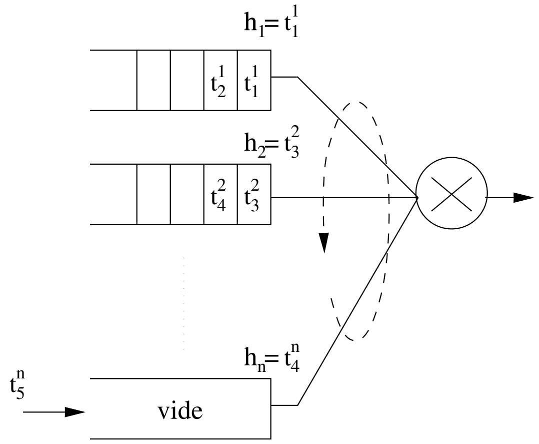


Figure 3.4 Sélection de la file i avec l'horloge h_i la plus petite.

Justification

Une fois un événement $< e, t >$ sélectionné, on est sûr qu'il n'y aura pas d'événement $< e', t' >$ avec $t' < t$. En effet comme les estampilles des messages générés par les différents *pls* sont strictement croissantes, les événements futurs auront de fait une estampille supérieure à celle de l'événement qui a été sélectionné. Lorsqu'une file est vide et qu'elle présente l'horloge la plus petite, le *pl* doit attendre car il lui est impossible de déterminer la borne inférieure du prochain événement de cette file.

Problème

L'algorithme est correct mais un blocage peut se produire à un point de fusion (fig. 3.5) ou si un cycle de files forme une chaîne de dépendance (fig. 3.6). Dans la première figure, le *pl* M ne peut rien faire si le *pl* A ne reçoit aucun événement. En effet, l'horloge h_A associée à l'entrée provenant du *pl* A est plus petite que celle associée à l'entrée provenant de B puisqu'elle ne change pas. Le *pl* M bloque donc continuellement sur cette file car rien ne lui permet d'affirmer qu'un message avec une estampille plus petite ne se présentera pas.

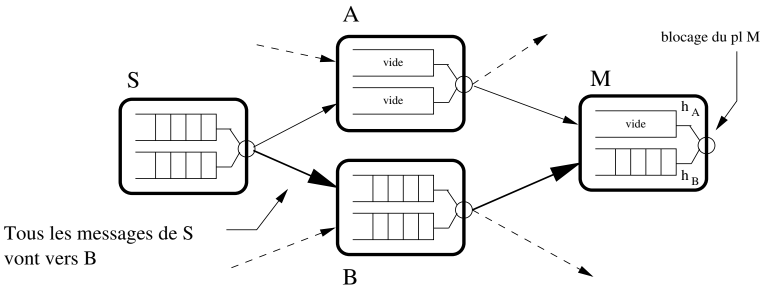


Figure 3.5 Configuration de blocage à un point de fusion.

Dans le deuxième cas, si les horloges des files vides sont inférieures aux autres files, bien que ces dernières ne soient pas vides, il est impossible d'être sûr de respecter les contraintes de précédence en exécutant un message. Si un cycle de file vide se présente, on aboutit à une configuration de blocage.

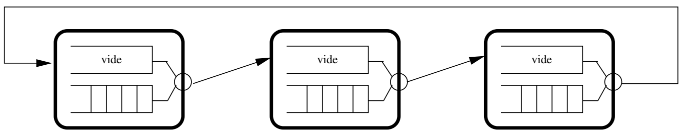


Figure 3.6 Configuration de blocage dans un cycle.

Pour résoudre ces problèmes de blocage, deux solutions radicalement opposées sont envisageables : soit éviter le blocage, soit permettre des blocages pour ensuite les détecter et débloquer les *pls*. La première solution utilise des messages fictifs pour avancer le temps de simulation, tandis que la deuxième utilise un mécanisme de détection des blocages basé sur un jeton.

3.3.3 *null* messages—schéma de base

Dans [Chan79] l'auteur propose une solution basée sur l'envoi de *null* messages pour éviter les blocages. Un *null* message n'a aucun sens réel, il permet aux *pls* de faire avancer leur horloge locale de manière artificielle. Soit un *null* message de la forme $< null, t >$ où t est une valeur d'horloge, un *pl* recevant un tel message sur un lien peut affirmer que le *pl* qui lui est connecté n'envoiera pas de messages dont l'estampille sera inférieure à t. Si on suppose que chaque *pl* envoie un *null* message après chaque événement exécuté, et que les récepteurs d'un tel message le propagent—en le mettant à jour—à tous leurs voisins, alors on évite bien les blocages. Dans l'exemple du point de fusion (fig. 3.5), bien que S n'envoie pas de messages réels à A, il peut cependant lui envoyer des *null* messages. Cela permet au *pl* A d'avancer son horloge et par conséquent celle de M puisque A propage le *null* message. Si on prend l'exemple d'une source S qui envoie à B des messages aux

time	$S \longrightarrow A$	$S \longrightarrow B$	$A \longrightarrow M$	$B \longrightarrow M$
$t = 10$	$< null, 10 >$	$< S_1, 10 >$	$< null, 10 + \delta_A >$	$< B_1, 10 + \delta_B >$
$t = 20$	$< null, 20 >$	$< S_2, 20 >$	$< null, 20 + \delta_A >$	$< B_2, 20 + \delta_B >$
$t = 30$	$< null, 30 >$	$< S_3, 30 >$	$< null, 30 + \delta_A >$	$< B_3, 30 + \delta_B >$

Tableau 3.1 Liste des *null* messages échangés.

instants $t = 10, 20$ et 30, le tableau 3.1 représente les différents messages échangés. δ_A et δ_B représentent les incréments minimaux de A et B entre deux messages générés[1]. Le *null* message $< null, 10 >$ indique à A que S ne lui enverra pas de message avec une estampille inférieure à 10. A informe donc M qu'il ne lui enverra pas de messages dont l'estampille sera inférieure à $10 + \delta_A$. Cela permet à M d'exécuter les messages provenant de B.

Le nombre de *null* messages peut cependant devenir très important par rapport au nombre de messages réels. Dans certains cas, un grand nombre de *null* messages peut être nécessaire pour faire avancer suffisamment les horloges des *pls* et permettre ainsi l'exécution des messages réels. L'exemple du cycle illustre cet aspect des choses : la figure 3.7 représente les différents *pls* ainsi que la valeur des horloges sur chaque lien[2]. Le tableau 3.2 montre les messages échangés par les différents *pls* lorsque l'horizon de prédiction de tous les *pls* est $\delta = 5$ unités de temps.

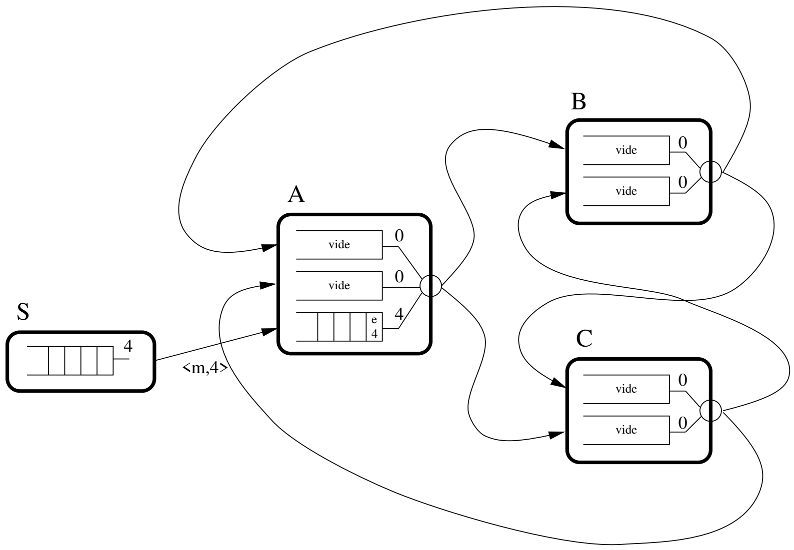


Figure 3.7 On se place à l'état initial et le *pl* S envoie au *pl* A un message $< m, 4 >$.

S, envoyant un message $< m, 4 >$ au *pl* A positionne l'horloge de la file correspondante à la valeur 4—étape 1. Cependant comme les deux autres files de A sont

[1] Nous retrouvons la notion de *lookahead* définie auparavant. Les *pls* A et B possèdent un horizon de prédiction de δ_A et δ_B respectivement.

[2] Rappelons que la valeur de l'horloge sur un lien est soit égale à l'estampille du dernier message exécuté, si la file est vide, soit égale à l'estampille du premier message de la file.

vides leur horloge est nulle. A ne peut donc qu'envoyer un *null* message $< null, \delta >$ à B et à C en se basant uniquement sur sa propre capacité de prédiction δ—étape 2 et 3. A la réception du *null* message de A, les *pls* B et C ont chacun une file dont l'horloge est à la valeur δ et une autre file dont l'horloge est nulle. B et C ne peuvent donc qu'envoyer à A le message $< null, \delta >$—étape 4 à 7. Dès la réception des deux *null* messages de B et C, A peut exécuter le message $< m, 4 >$ car la file correspondante possède maintenant l'horloge la plus petite ($4 < \delta$, $\delta = 5$).

étape	*message*	*sens*
1	$< m, 4 >$	$S \longrightarrow A$
2	$< null, \delta >$	$A \longrightarrow B$
3	$< null, \delta >$	$A \longrightarrow C$
4	$< null, \delta >$	$C \longrightarrow A$
5	$< null, \delta >$	$C \longrightarrow B$
6	$< null, \delta >$	$B \longrightarrow A$
7	$< null, \delta >$	$B \longrightarrow C$

Tableau 3.2 Messages générés pour $\delta = 5$.

Lorsque $\delta = 2$, cf. tableau 3.3, le nombre de *null* messages est deux fois plus important puisque le nombre d'étapes nécessaires à l'avancement des horloges est double du fait de la quantité de *lookahead* moitié moindre. On voit donc le rôle crucial de ce critère sur les performances de la simulation dans le cas d'une approche conservatrice comme Fujimoto le montre dans [Fuji88].

étape	*message*	*sens*
1	$< m, 4 >$	$S \longrightarrow A$
2	$< null, \delta >$	$A \longrightarrow B$
3	$< null, \delta >$	$A \longrightarrow C$
4	$< null, \delta >$	$C \longrightarrow A$
5	$< null, \delta >$	$C \longrightarrow B$
6	$< null, \delta >$	$B \longrightarrow A$
7	$< null, \delta >$	$B \longrightarrow C$
8	$< null, 2\delta >$	$A \longrightarrow B$
9	$< null, 2\delta >$	$A \longrightarrow C$
10	$< null, 2\delta >$	$C \longrightarrow A$
11	$< null, 2\delta >$	$C \longrightarrow B$
12	$< null, 2\delta >$	$B \longrightarrow A$
13	$< null, 2\delta >$	$B \longrightarrow C$

Tableau 3.3 Messages générés pour $\delta = 2$.

Un problème se pose lorsque les incréments minimaux peuvent être nuls. Dans ce cas, la solution des *null* messages n'est pas viable. Cela implique donc que certains types de problèmes, et notamment la simulation de files d'attente à temps de service nul, ne peuvent être simulés avec l'algorithme de Chandy-Misra.

3.3.4 *null* messages et améliorations

null message différé et mode à la demande

Dans [Misr86] l'auteur propose quelques solutions pour réduire le nombre de *null* messages. Si l'on décide d'adopter la stratégie qui consiste à rejeter tout *null* message $< null, t >$ suivi par un message réel $< e, t' >$ où $t' > t$, l'une des solutions utilise un *timeout* pour différer l'envoi des *null* messages. Dans ce schéma là, il peut en effet être judicieux d'attendre un temps[3] τ avant l'émission d'un *null* message dans l'espoir de recevoir un message réel dans un proche avenir $t + \varepsilon$, $\varepsilon < \tau$. Une autre solution consiste à considérer qu'un message réel inséré[4] dans une file à la position i annule tout *null* message éventuellement présent dans la file à des positions $j < i$. On peut également modifier la stratégie d'émission des *null* messages pour passer du mode automatique au mode à la demande.

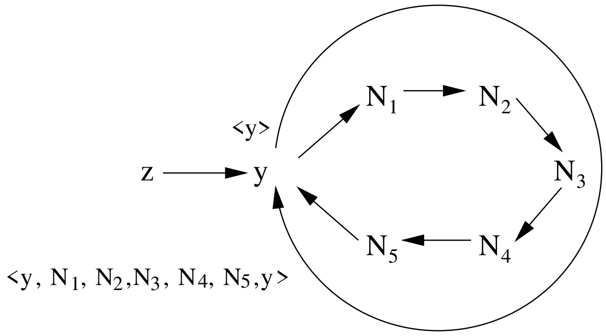


Figure 3.8 Détection du blocage par y.

Dans ce schéma de fonctionnement, un *pl* z demande explicitement à un *pl* y d'avancer l'horloge du lien (yz). y répondra alors par un *null* message si aucun message réel n'est disponible. Une phase de propagation est nécessaire lorsque l'horloge de y n'est pas supérieure à celle de z. Ce premier doit alors avancer sa propre horloge au préalable en effectuant une demande. Ainsi y ne peut envoyer un message incrémentant l'horloge de z que lorsque son l'horloge est supérieure à celle du lien (yz). Ce schéma de fonctionnement à la demande réduit le nombre de *null* messages mais augmente en revanche le délai d'obtention de celui-ci—et donc allonge le temps de simulation— puisqu'un aller-retour est nécessaire [Fuji90].

Notons également que cette stratégie permet de détecter les blocages. En effet, s'il y a blocage, le *pl* y doit propager la demande de z car son horloge est inférieure

[3] Si $\tau = 0$, on retrouve le schéma de base du *null* message ; si $\tau = \infty$, on retrouve l'algorithme sans *null* message qui peut aboutir à des cas de blocages.

[4] Misra énonce cette solution dans [Misr86] dans le cas où les files ont des capacités non nulles. L'augmentation de la taille des files influe sur les performances jusqu'à une valeur seuil pour laquelle aucune amélioration n'est ensuite constatée.

à celle du lien (yz) sinon il n'y aurait pas blocage. Si on suppose que le cycle de blocage est initialisé par y, alors ce dernier le détecte puisqu'il apparaît deux fois dans la propagation du message de demande (figure 3.8).

Ajout d'informations aux *null* messages

Bien que les améliorations décrites ci-dessus réduisent le nombre de *null* messages, ceux-ci restent néanmoins en nombre important. Dans [Cai90] les auteurs proposent d'augmenter le degré de connaissance globale du système par l'échange de messages spéciaux supplémentaires. Augmenter le niveau de connaissance revient à augmenter le degré de *lookahead* du *pl*, lui permettant ainsi de réduire le nombre d'envoi de *null* messages. Deux types de messages sont alors utilisés, les *null* messages traditionnels de la forme $< t, null >$ qui permettent d'avancer le temps de simulation, et ce que les auteurs appellent *carrier null messages* de la forme $< t, route.inf, la.inf, carrier - null >$ qui permettent l'échange d'informations supplémentaires. Pour ces derniers, outre t et $carrier - null$ qui ont les mêmes fonctionnalités que dans le cas des *null* messages, `route.inf` et `la.inf` permettent respectivement de contenir des informations de routage et de *lookahead*. Considérons le réseau de file d'attente de la figure 3.9 composé de deux sources S et S', d'un serveur, d'un puits, et de deux *merges* M et M'. Les temps de service des serveurs et des *merges* sont fixés à 1 unité de temps.

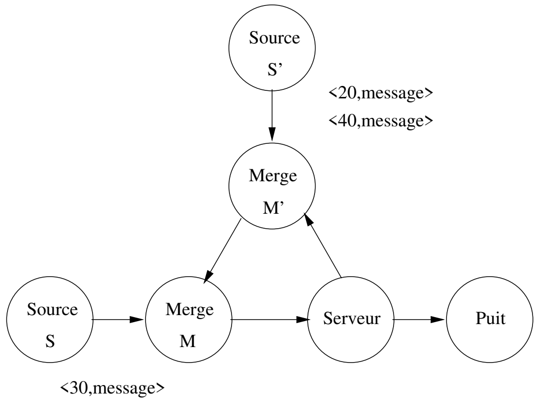
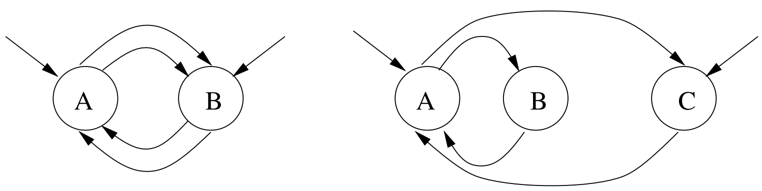


Figure 3.9 Exemple d'utilisation du carrier-null message.

Supposons que la source S envoie un premier message à M d'estampille 30 et que S' envoie deux premiers messages à M' avec les estampilles 20 et 40. Dans ce cas, un blocage a lieu puisque M attend un message de M' qui lui-même attend un message du serveur. De son côté, le serveur ne pourra envoyer des messages que s'il en reçoit de M. Ce cas de blocage est facilement résolu par l'utilisation de *null* messages le long du cycle bloquant mais en nécessite un grand nombre du fait de la faible quantité de *lookahead* (1 ut).

Dans la nouvelle approche, le champ `route.inf` permet de garder des informations de routage pour détecter l'existence d'un cycle bloquant. Lorsqu'un *pl* reçoit un $carrier-null$ message, il va rajouter son identité à la liste `route.inf` . Le champ `la.inf` permet quant à lui d'ajouter des informations de *lookahead*. La valeur de `la.inf` représente la plus petite estampille des messages réels pouvant "casser" le cycle bloquant. Dans le cas de l'exemple de la figure 3.9, nous avons `la.inf`=30. M va donc envoyer un $carrier-null$ message de la forme $< t, M, 30, carrier-null >$ vers le serveur, et lorsqu'il recevra finalement en retour un $carrier-null$ message dont le champ `route.inf` contient son identité, il pourra en conclure qu'il est l'initiateur d'un cycle bloquant. Si ensuite `la.inf` est supérieur à son horloge logique, il va émettre un *null* message de la forme $< la.inf, null >$. Dans notre cas, M va émettre $< 30, null >$ ce qui aura pour conséquence de permettre à M' d'envoyer ses messages vers M.

L'utilisation du $carrier-null$ message permet donc de réduire considérablement le nombre de messages protocolaires nécessaires dans des topologies présentant des cycles bloquants en accélérant l'acquisition d'informations globales sur le système. Dans le cas de réseaux sans cycle, cette approche se comporte comme l'approche originelle n'utilisant que des *null* messages. Cependant il apparaît que la solution proposée dans [Cai90] ne puisse pas fonctionner pour certaines configurations où le nombre d'entrées impliquées dans un cycle bloquant est supérieur à 1 (figure 3.10a), et lorsque des cycles bloquants sont imbriqués (figure 3.10b). Dans [Wood94], les auteurs généralisent l'utilisation du $carrier-null$ message en permettant son utilisation dans les cas pré-cités.



a b

Figure 3.10 Cas où le carrier-null message ne fonctionne pas.

Exploitation de la topologie du réseau

Dans [DeVr90] l'auteur propose une solution pour réduire le nombre de *null* messages utilisant la notion de prédiction pour éviter le blocage d'un *pl* sur une file vide. Rappelons que dans le schéma de base—algorithme CMB—un *pl* est obligé de se bloquer sur une file vide car il est dans l'impossibilité de déterminer de manière sûre si les autres événements sont exécutables. La solution proposée par DeVries associe à une file vide une valeur d'horloge prédite. De cette manière il n'y a plus de blocage, et donc l'utilisation de *null* messages n'est plus nécessaire dans ce cas. Le seul cas de blocage du *pl* sur une file vide se présente lorsque cette file possède l'estampille la plus petite. De Vries propose lors de la phase d'initialisation de renseigner tous les *pls* avec la valeur de l'incrément minimal des *pls* prédécesseurs. Cette solution n'est pertinente que dans le cas de réseaux de type *feedforward* et *feedback* (fig. 3.11). Dans le cas général, elle ressemble à la notion de *lookahead.*

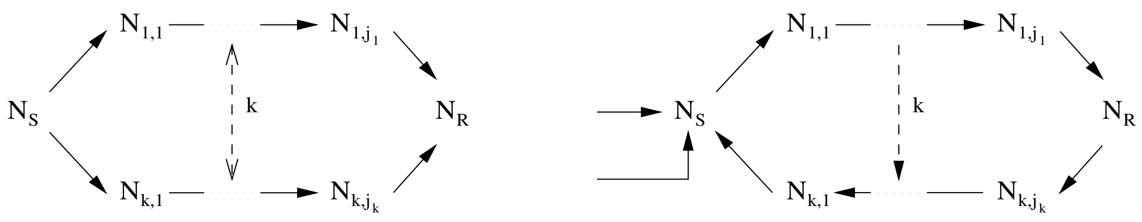


Figure 3.11 A gauche : réseau général de type *feedforward*, il existe au moins k chemins de N_R vers N_S, $k \geq 2$. A droite : réseau général de type *feedback*, il existe un chemin de N_R vers N_S et un chemin de N_R vers N_S, N_R possède exactement une sortie et N_S une ou plusieurs entrées. Dans les deux cas, les noeuds intermédiaires se contentent de faire passer les messages, ils possèdent exactement une entrée et une sortie.

A. Cas d'un réseau simple *feedforward*

Dans le cas d'un réseau simple *feedforward*, DeVries présente les algorithmes suivants pour les noeuds N_S et N_R, ce dernier étant un point de fusion. Si on note δ_i l'incrément minimal du *pl* i, avec $i = 1$ pour le *pl* source, et l_x la longueur du chemin p_x, l'auteur définit t_x comme étant l'incrément minimal le long du chemin p_x de la manière suivante :

$$t_x = \sum_{j=1}^{l_x} \delta_i$$

Considérons maintenant le réseau *feedforward* de gauche avec uniquement deux chemins définis, p_a et p_b (celui du haut et celui du bas) et notons par T_n^{end} la date de fin de traitement du message n.

Hypothèse

- Le *pl* N_S envoie avec chaque message n la valeur de T_n^{end}, un numéro de séquence S_n et une estampille indiquant la date de réception du message par le *pl* suivant.
- Le *pl* N_S reçoit lors de l'initialisation la valeur de t_a et t_b.

Algorithme pour N_S

1. $SEQ = 1$
2. N_S sélectionne de manière circulaire le lien dont l'horloge est la plus petite et exécute l'événement en tête de la file du lien sélectionné. Si la file est vide, le *pl* bloque.
3. Un message est éligible, il est exécuté et le message n résultant est envoyé avec $S_n = SEQ$. Ensuite, $SEQ = SEQ + 1$ et on passe en (2).

Algorithme pour N_R

1. Si un message n arrive avec S_n égal à celui que l'on attend alors on le place dans la file associée et on passe en (2), sinon on le retient et on reprend en (1).
2. On parcourt toutes les files et pour chaque file vide, on fixe son horloge à la valeur $T_{max}^{end} + t_x$. On choisit ensuite la file avec l'horloge la plus petite. Si la file est vide on passe en (1) sinon en passe en (3).
3. On exécute le message en tête de file et on passe en (2).

Pour N_R, l'astuce consiste à exploiter la topologie du réseau. Comme tous les messages proviennent de la même source N_S, si une file est vide on peut lui assigner une valeur d'horloge égale à $T_{max}^{end} + t_x$ où T_{max}^{end} est le maximum des T_n^{end} de tous les messages de toutes les files. En effet, le message dont $T_n^{end} = T_{max}^{end}$ est le dernier message envoyé par la source. Quelque soit le prochain message reçu, il aura toujours $T_n^{end} > T_{max}^{end}$ puisqu'il provient du même *pl* source. Ce schéma permet de s'affranchir des *null* messages car la réception d'un message sur un lien permet d'incrémenter les horloges de toutes les files vides évitant ainsi tout blocage (étape 2). L'utilisation du numéro de séquence ainsi que la manière dont on insère les messages (uniquement ceux dont le numéro de séquence est celui attendu) permet de garantir que le message avec la plus grande valeur pour T_n^{end} est effectivement le dernier message envoyé par la source et pris en compte par le récepteur—les messages hors séquence étant alors retenus, i.e. non insérés dans les files.

B. Cas d'un réseau simple *feedback*

Pour un réseau simple *feedback* (figure 3.12), DeVries note par t_a et t_b l'incrément minimal le long des chemins p_a et p_b. De la même manière que pour le réseau *feedforward*, T_n^{end} représente la date de fin de traitement d'un message n par la source N_S.

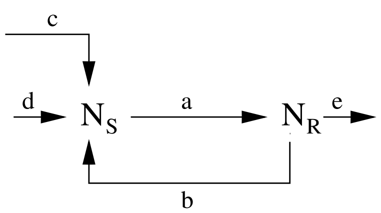


Figure 3.12 Réseau simple feedback.

Notons que dans cette configuration, un blocage se produit tout de suite lorsqu'un message se présente sur le chemin c ou d. En effet, si on se place au début de la simulation, les horloges des liens b, c et d sont nulles et celles de c ou d devenant non nulles lorsqu'un message arrive. L'horloge du lien b reste cependant toujours nulle puisque N_S n'envoie aucun message. N_S est alors bloqué. Ce cas de blocage initial, déjà évoqué dans [Misra86] et résolu avec l'émission de *null* messages, est évité par l'utilisation de la prédiction qui assigne au lien b une valeur d'horloge égale à $h_{N_S} + t_a + t_b$, où h_{N_S} est l'horloge du *pl* N_S et $t_a + t_b$ l'incrément minimal de la boucle[5].

Pour les autres cas de blocages, contrairement au cas *feedforward*, il n'est pas possible d'éviter complètement l'utilisation de *null* messages. DeVries propose pour N_S l'algorithme ci-dessous, avec l'obligation pour N_R d'envoyer un *null* message sur le chemin p_b ou p_e lorsqu'il n'envoie pas de message réel sur ces chemins en réponse à un message d'entrée. Tout comme dans le schéma de base proposé dans [Chan79], ces *null* messages servent à faire avancer les horloges associées aux liens. Nous verrons par la suite que le nombre de *null* messages est cependant plus réduit que dans le schéma de base.

Hypothèse

- Le *pl* N_S envoie avec chaque message n une estampille indiquant la date de réception du message par le *pl* suivant.

Algorithme pour N_S

1. $COUNT = 0$
2. N_S sélectionne de manière circulaire le lien dont l'horloge est la plus petite. Si la file de *feedback*—celle associée au lien b—est vide et que $COUNT = 0$, son horloge prend la valeur $h_{N_S} + t_a + t_b$, où h_{N_S} est l'horloge du *pl* N_S. Si la file ayant l'horloge la plus petite contient un message, on passe en (3) sinon on repasse en (2).
3. On exécute le message en tête de la file sélectionnée. Si le message provient des files d'entrée—chemins c ou d—alors $COUNT = COUNT + 1$. Si un *null* message provenant du chemin b a été exécuté alors $COUNT = COUNT - 1$. Si c'est un message réel qui a été exécuté, $COUNT$ reste inchangé. On passe ensuite en (2).

Le rôle du compteur $COUNT$ est de comptabiliser le nombre de messages en transit dans la boucle. L'obligation pour le *pl* N_R d'envoyer des *null* messages permet de savoir, lorsque l'on réceptionne un *null* message sur le lien b, qu'un message est parti sur le lien e et qu'il est donc sorti de la boucle ($COUNT = COUNT - 1$). Si $COUNT = 0$, cela signifie qu'aucun message n'est dans la boucle[6] et qu'il est alors nécessaire de prédire une valeur pour l'horloge du lien b.

Bien que des *null* messages soient utilisés, il y en a moins car dans le schéma de base, non seulement N_R doit en envoyer, mais également N_S pour éviter que les

[5] Si une prédiction ne suffit pas pour rendre exécutable les autres messages, plusieurs prédictions peuvent être faites. En effet, une fois que l'horloge du lien b est égale à $h_{N_S} + t_a + t_b$ et que cette valeur est encore minimale, l'horloge de N_S prend cette valeur, i.e. $h_{N_S} + t_a + t_b$, pour recommencer une autre prédiction pour le lien b. On procède ainsi jusqu'à ce que l'horloge du lien b devienne supérieure à celle des autres liens.

[6] Lorsque $COUNT$ est nul, on ne reçoit plus de messages quelqu'ils soient sur le lien b puisque les seuls messages qui peuvent être reçus par N_R proviennent de N_S (réseau simple *feedback*).

messages sur les chemins c et d ne soient bloqués lorsque la file du chemin b est vide : N_S devrait envoyer un *null* message sur le chemin a, ce qui provoque l'envoi d'un *null* message par N_R sur le chemin b. On gagne une étape avec l'utilisation de la prédiction.

L'auteur traite ensuite du cas où des réseaux *feedforward* et *feedback* sont mélangés, avec la possibilité pour un réseau *feedforward* de contenir un réseau *feedback* et vice et versa. Les algorithmes pour les *pls* sont également présentés et nous invitons le lecteur intéressé à consulter l'article pour plus de détails.

3.3.5 La détection des blocages et leur résolution

Contrairement à la stratégie visant à éviter les blocages, dans [Chan81] les auteurs proposent de laisser la simulation se bloquer, puis de détecter le blocage pour ensuite lancer une phase de résolution. Cette approche est justifiable si la probabilité de blocage est faible, auquel cas on évite ainsi l'*overhead* engendré par l'utilisation en continu de *null* messages ou de tout autre mécanisme visant à prévenir le blocage.

Chandy et Misra ont proposé deux algorithmes distribués pour la détection d'inter-blocage [Chan83]. Ces algorithmes ne sont pas spécifiquement conçus pour la simulation parallèle mais peuvent s'appliquer avec succès. Le premier s'applique aux modèles de type *ressource* et le deuxième aux modèles de type *communication*. Le premier correspond à des modèles où un certain nombre de serveurs sont en compétition pour l'acquisition d'un certain nombre de ressources. Le cas de verrou mortel se présente lorsque des serveurs attendent chacun la libération des ressources détenues par les autres. Le deuxième, plus général, est celui qui nous intéresse le plus puisqu'il considère un réseau de processus communicant de topologie quelconque. La similitude avec les problèmes de blocage en simulation parallèle est alors flagrante. Dans ce cas, les auteurs proposent un algorithme entièrement distribué avec échange de deux types de messages : les messages *query* et les messages *reply*. Les auteurs ne proposent cependant pas de moyen pour débloquer les processus.

Misra propose également l'utilisation d'un jeton (*circulating marker*) pour détecter et résoudre les blocages [Misr86]. Ce jeton, qui est en fait un message particulier, circulera d'un *pl* à l'autre suivant un chemin déterminé au préalable—par une phase de pré-traitement par exemple—afin de déterminer s'il y a blocage ou non. Chaque *pl* doit obligatoirement faire passer le jeton lorsqu'il se trouve dans un état d'attente, i.e. avec aucun message à envoyer ni à traiter. Cette technique du jeton, très courante dans les algorithmes de détection de terminaison dans les systèmes répartis, ne nécessite alors que très peu de modifications pour permettre la résolution du blocage après sa détection. La solution proposée dans [Misr86] consiste à mémoriser dans le jeton le *pl* possédant un événement d'estampille minimale. Cette estampille est également mémorisée et représente en fait la date du prochain événement exécutable en toute sécurité[7]. Le *pl* mémorisé peut alors être débloqué ; l'auteur utilise un processus central afin de diffuser aux différents *pls* la nouvelle valeur de l'horloge.

[7]En effet on peut noter que l'événement dont l'estampille est minimale parmi tous les événements du système est obligatoirement exécutable.

On peut ainsi voir cette approche comme étant constituée d'une succession de deux phases : simulation parallèle et resynchronisation.

Ces solutions ne sont cependant que très peu utilisées car elles sont assez lourdes à mettre en oeuvre—nécessité d'un grand nombre de communications et nécessité d'un serveur central. Une étude menée par Fujimoto dans [Fuji88] montre que les performances du schéma *détection de blocage/déblocage* sont plus sensibles au *lookahead* que dans l'approche évitant les blocages. En effet, l'auteur montre que lorsqu'il existe un certain nombre de *pls* possédant un faible degré de *lookahead* les performances de l'ensemble des *pls* sont très rapidement dégradées.

3.3.6 Les protocoles à fenêtre dans les approches conservatrices

L'ajout d'une fenêtre aux approches conservatrices a pour but de réduire l'espace de recherche des événements d'estampille minimale. En effet, dans l'algorithme CMB un *pl* doit obligatoirement bloquer sur une file vide. Cependant, il est fort possible que pour certaines files, il n'est pas nécessaire de les considérer car les *pls* qui y sont associés sont trop "loin" pour influencer le choix de la file la plus petite.

Supposons que tous les *pls* vont estampiller leurs messages avec un incrément minimum de 8 unités de temps. Supposons ensuite que l'on se place à l'instant 10 et que l'on se fixe une taille de fenêtre capturant les messages entre les instants [10,20]. Dans ce cas seuls les *pls* voisins sont susceptibles d'envoyer des messages dont l'estampille est à l'intérieur de la fenêtre puisque les messages provenant des autres *pls* auront obligatoirement un incrément de 16 unités de temps. Cela veut dire que les files associées aux *pls* trop "éloignés" n'ont pas besoin d'être considérées.

Cette notion de fenêtre est en fait assez proche de l'idée énoncée par DeVries lorsqu'il considérait la topologie du réseau pour estimer les valeurs d'horloges des files vides. On peut également percevoir cette approche comme le complément du *lookahead*. En effet, la manière traditionnelle d'utiliser le *lookahead* était pour un *pl* donné d'envoyer des *null* messages pour informer les autres *pls* de son état d'avancement—i.e. diffusion des informations. Maintenant, pour pouvoir écarter d'office certaines files, un *pl* donné devra obtenir/posséder plus d'informations sur ses voisins—i.e. intégration des informations.

Bounded Lag

Dans [Luba88,Luba89b] l'auteur propose un algorithme synchrone qui se démarque de manière significative de l'algorithme Chandy-Misra. Cet algorithme utilise une seule file par *pl* et fixe une borne B (*bounded lag*) entre deux événements exécutables en parallèle afin de réduire la fenêtre de recherche des événements. La notion de *délai minimum de propagation* entre deux *pls* i et j, notée $d(i,j)$, est introduite et permet de limiter la recherche des événements à uniquement quelques *pls* voisins[8].

[8]Rappelons que la recherche doit normalement s'effectuer sur l'ensemble des *pls*.

Dans ce cas, pour un *pl* i donné, tous les *pls* j tels que $d(j,i) > B$[9] ne pourront pas influencer le choix pour le *pl* i du prochain message à exécuter.

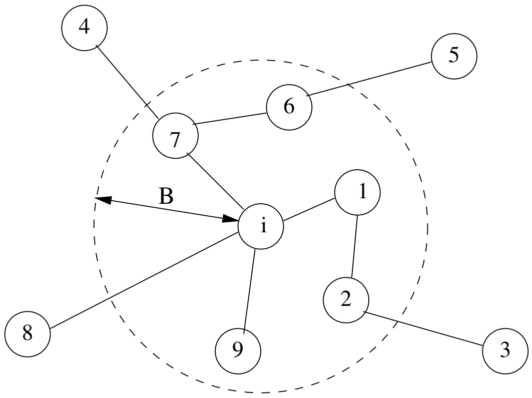


Figure 3.13 Zone de recherche limitée par la borne B.

Si on note par T_i l'estampille minimale d'un message en attente dans le *pl* i, et par $S(i,B)$ la zone de recherche centrée en i et limitée par la borne B, Lubachevsky définit le terme :

$$\alpha(i) = \min_{j \in S(i,B), j \neq i} \{d(i,j) + min\{T_j, d(i,j) + T_i\}\}$$

comme une estimation de la date minimale à laquelle le *pl* i peut recevoir un message provenant des autres *pls*. Cette relation s'explique en constatant qu'il n'existe que deux moyens pour que le *pl* i reçoive un message, à savoir :

1. Un *pl* j exécute un message d'estampille T_j et génère un message qui doit être envoyé et reçu par i en $d(j,i) + T_j$.
2. Le *pl* i exécute un message d'estampille T_i qui génère l'envoi d'un message au *pl* j qui le reçoit en $d(i,j) + T_i$. Ce dernier message génère ensuite à son tour l'envoi d'un message au *pl* i qui le reçoit donc en $d(i,j) + d(j,i) + T_i$.

L'algorithme proposé est de type synchrone car des phases de resynchronisation sont nécessaires. Cela suppose donc la définition d'une horloge globale représentant l'état d'avancement de la simulation, que l'auteur définit comme un seuil de simulation par :

$$floor = min(t) \text{ pour tout } < e,t > \text{ appartenant à } file(i), i = 1 \ldots N,$$

ainsi que la diffusion de cette valeur à l'ensemble des *pls*. Le fonctionnement de l'algorithme est sans blocage car une seule file d'entrée est considérée. Le respect des contraintes de causalités est garanti grâce à l'utilisation du *Bounded Lag B* qui limite le parallélisme en n'autorisant l'exécution en parallèle de deux événements $< e,t >$ et $< e',t' >$ uniquement si $t - t' < B$. Un *pl* i n'exécute que les événements

[9] Si le chemin n'est pas direct de i à j mais passe par k, alors $d(i,j) = d(i,k) + d(k,j)$.

dont l'estampille est inférieure à $\alpha(i)$. Cela garantit qu'aucun message avec une estampille inférieure ne sera reçu par ce *pl*. La fenêtre d'exécution associée à chaque *pl* est donc égale à $floor + B$.

Les deux principaux problèmes de l'algorithme sont d'une part que le calcul et la diffusion de $floor$ peuvent devenir très fastidieux si le nombre de noeuds augmente. D'autre part, le choix de la borne B est pour l'instant arbitraire et aucune méthode de calcul de la valeur optimale n'est proposée.

Conservative Time Window

D'une manière un peu similaire à l'algorithme *Bounded Lag*, Ayani et Rajaei proposent l'algorithme *Conservative Time Window* [Ayan92] qui associe à chaque *pl* i une fenêtre W_i telle que les événements dans chaque fenêtre W_i soient indépendants des événements dans les fenêtres W_j, $j \neq i$. Pour déterminer l'indépendance des événements les auteurs utilisent la notion de "distance" entre deux objets i et j appelée *directional distance* et notée d_{ij}, similaire donc à la notion de *délai minimum de propagation* de l'algorithme *Bounded Lag*. Un *pl* i doit donc connaître la distance qui le sépare de tous les autres *pls*. Ainsi, si $< x, t_x >$ et $< y, t_y >$ sont respectivement des événements du *pl* i et du *pl* j, alors $< x, t_x >$ est dit indépendant de $< y, t_y >$ si l'on a $t(x) + d_{ij} \geq t(y)$ (1). Cette relation s'explique par le fait que si le *pl* i, en exécutant $< x, t_x >$ génère un événement $< z, t_z >$ vers le *pl* j, on aura obligatoirement $t(z) \geq t(y)$ si (1) est vérifié. Dans ce cas d'indépendance, les événements $< x, t_x >$ et $< y, t_y >$ peuvent être exécutés parallèlement en toute sécurité.

De même que l'algorithme précédent où deux phases pouvaient être dégagées, *CTW* peut être décomposé comme une succession de trois phases séparées par des barrières de synchronisation. La première phase consiste à initialiser la borne inférieure des W_i avec l'estampille du premier événement de chaque *pl* i. La deuxième consiste alors à déterminer la borne supérieure de cette fenêtre W_i de telle manière que les fenêtres associées aux *pls* soient indépendantes. La troisième phase consiste finalement à exécuter en parallèle les événements de chaque *pl* jusqu'à concurrence de la fenêtre d'exécution.

La principale différence entre *CTW* et *Bounded Lag* réside dans le fait que la taille de la fenêtre W est dynamiquement déterminée à chaque itération et que les *pls* peuvent avoir des fenêtres locales de tailles différentes. Le défaut majeur de *CTW* est cependant que la phase de détermination de la borne supérieure de la fenêtre d'exécution requiert environ n^2 comparaisons, ce qui peut amener très rapidement à des dégradations dans les performances de l'algorithme. Les auteurs proposent pour réduire le nombre de comparaisons que chaque *pl* ne connaisse que la distance qui le sépare des *pls* immédiatement voisins.

3.3.7 Exploitation de la sémantique des messages

Dans [Ahme94], les auteurs proposent d'utiliser la sémantique des événements pour accroître le niveau de parallélisme au sein d'un *pl*. Dans une approche traditionnelle, les événements au sein d'un *pl* sont exécutés de manière séquentielle pour respecter la causalité. Il est cependant possible de relâcher cette contrainte de séquentialisation lorsqu'il peut être déterminé que deux événements de la file d'événements sont totalement indépendants. Pour cela, des attributs supplémentaires sont ajoutés aux messages afin de permettre à un *pl* de déterminer leur indépendance. Ces attributs sont cependant fortement dépendants du modèle. De tels attributs peuvent être par exemple l'adresse mémoire accédée par une instruction dans le cas de simulation d'architecture matérielle, ou bien l'index de la page de cache dans le cas de simulation d'un algorithme de cas, ou aussi le numéro de vol d'un avion si on simule un trafic aérien.

Les auteurs appliquent cette méthode conjointement avec l'algorithme CTW pour simuler une architecture de machine appelée *Enhanced Data-Driven Architecture*. Les résultats obtenus montrent qu'il est effectivement possible d'obtenir une réduction du temps de simulation en utilisant ce niveau de parallélisme supplémentaire. Le gain est cependant encore une fois fortement dépendant de l'application, et les auteurs précisent qu'il est possible, pour certaines applications, d'obtenir des temps de simulation supérieurs du fait de l'*overhead* supplémentaire introduit.

3.4 LES APPROCHES PUREMENT OPTIMISTES

L'approche optimiste découle d'une vision totalement différente de celle précédemment étudiée. Dans les approches précédentes, toutes sortes de mécanismes et de stratégies étaient mis en oeuvre pour éviter les fautes temporelles. Dans l'approche optimiste, aucune tentative n'est faite pour éviter de telles fautes, les différents *pls* exécutent les messages qui leur arrivent et avancent leur horloge en conséquence sans s'inquiéter du fait qu'ils risquent de recevoir plus tard un message avec une estampille inférieure. Cependant comme la simulation représente l'évolution d'un système physique, il est clair que l'exécution de manière anarchique des messages ne reflète pas une évolution réelle. D'autres types de mécanismes doivent être instaurés afin que le résultat de la simulation soit identique à celui que l'on obtiendrait avec une approche conservatrice ou séquentielle. L'idée à la base de l'approche optimiste consiste à laisser le système évoluer librement jusqu'à ce qu'une faute temporelle se produise en espérant que de telles fautes ne se produisent pas trop souvent. Une faute temporelle est trivialement détectée puisqu'un *pl* recevant un message dont l'estampille t est inférieure à son horloge peut affirmer qu'il est en état d'infraction. Une fois l'erreur détectée, il est nécessaire de défaire tout ce que l'on a fait à partir de la date t. Par la suite, nous appellerons *message fautif* un message arrivant dont l'estampille est inférieure à la valeur de l'horloge du *pl*. Nous allons dans ce qui suit présenter les algorithmes les plus utilisés, à savoir Time Warp et ses améliorations possibles. Ils sont tous basés sur le mécanisme du retour arrière, mais diffèrent par quelques optimisations visant à ne défaire que le minimum de choses.

3.4.1 Time Warp : Algorithme de base

Contrôle local : Local Virtual Time (LVT)

Contrairement à l'approche conservatrice, Time Warp [Jeff85] n'a pas besoin de plusieurs files d'attente en entrée et associe généralement une seule file d'attente pour tous les *pls*. Cette file contient la liste des messages à traiter, provenant de tous les *pls* entrant. L'horloge locale du *pl* (Local Virtual Time) évoluera en fonction de l'estampille des messages exécutés. Si on se place dans le cas idéal où aucune faute temporelle ne peut se produire, le schéma de fonctionnement de Time Warp est identique à celui de l'approche conservatrice avec une seule file, donc sans blocage sauf si la file est vide. Lorsque des fautes temporelles peuvent se produire, Time Warp implémente un mécanisme de retour arrière, appelé *rollback*, qui permet de défaire les actions erronées exécutées depuis l'instant de l'erreur, puis de reprendre une exécution correcte (figure 3.14). On peut voir que les événements e_4 à e_7 sont défaits, puis qu'ils sont exécutés de nouveau après avoir exécuté l'événement e_{3b}. Il faut noter que les messages exécutés e_4 à e_7 sont, après le *rollback*, marqués non exécutés.

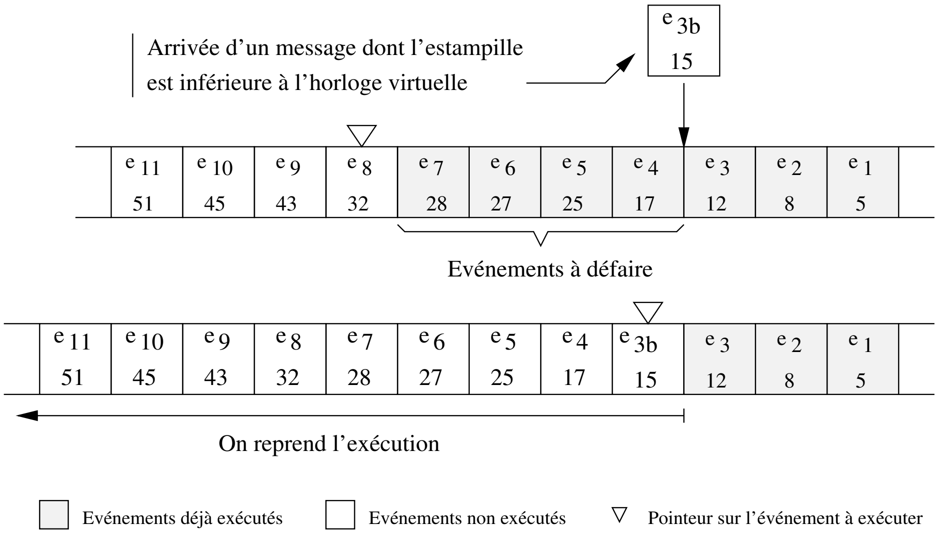


Figure 3.14 Mécanisme du *rollback*.

Afin de permettre le *rollback*, il est nécessaire de sauvegarder périodiquement l'état du *pl* dans une file. Cet état peut être représenté par un ensemble de variables, c'est la valeur de ces variables qui détermine un état particulier du *pl*. Restaurer l'état du *pl* n'est pas tout, il est nécessaire d'annuler tous les messages qui ont pu être émis vers d'autres *pls*, cela afin de leur permettre, à leur tour, de retourner en arrière. Pour cela, Jefferson introduit la notion d'anti-message. Chaque message réel envoyé est dit positif : un anti-message ne diffère d'un message réel que par son signe qui est négatif, son rôle étant d'annuler le message positif correspondant. Chaque *pl* garde alors une trace de tous les messages envoyés sous la forme d'anti-messages. Le mécanisme du retour arrière consiste alors à rétablir l'état du *pl* et à envoyer tous les anti-messages correspondant aux messages envoyés. Cette stratégie d'annulation est appelée *aggressive cancellation* car les anti-messages sont immédiatement envoyés.

Il est possible d'avoir uniquement ce que l'on appelle des *rollbacks locaux*. Un rollback est dit local si seuls les événements internes à un *pl* doivent être annulés. Dans ce cas il n'est pas nécessaire d'avoir recours à des anti-messages. Certaines approches sont spécialement conçues pour n'utiliser que, ou le plus souvent possible, des rollbacks locaux. Dans ce cas, une stratégie doit être définie pour s'assurer que les messages ne soient pas envoyés avec risque (voir Breathing Time Bucket, Breathing Time Warp, Local Time Warp, Changepoint Algorithm).

La figure 3.15 représente la structure d'un *pl* implémentant Time Warp. La file d'entrée contient les messages en attente de traitement. Une fois traités, ces messages peuvent engendrer d'autres messages dont les anti-messages sont sauvegardés. De manière périodique, l'état du *pl* est également sauvegardé dans une file. Dès qu'une erreur survient, l'état du *pl* antérieur à la date d'erreur est restauré et les anti-messages dont l'estampille est supérieure à cette date sont envoyés afin d'annuler les messages envoyés depuis cette date. Notons que les messages exécutés restent néanmoins dans la file pour une réexécution éventuelle.

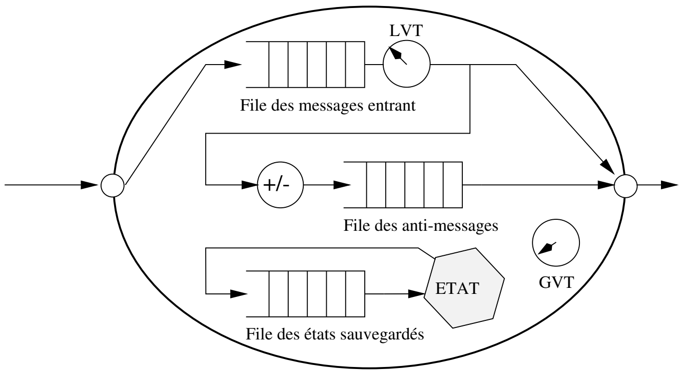


Figure 3.15 Structure d'un processus logique Time Warp

Contrôle global : Global Virtual Time

Le fonctionnement précédemment décrit représente le mécanisme de contrôle local pour un *pl*. Il est cependant nécessaire d'introduire la notion d'horloge globale pour s'assurer que la simulation du système entier avance réellement. Pour cela, dans [Jeff85] l'auteur définit le *Global Virtual Time* (GVT) par :

$$GVT = min(LVT_i, t_j)$$

avec LVT_i l'horloge locale du processus $i = 1 \dots N$, N étant le nombre de *pls*, et t_j l'estampille du message j encore en transit. Ce choix n'est pas arbitraire car si on considère le GVT comme reflétant l'état d'avancement de la simulation, il est clair qu'à un instant réel t, si le minimum de toutes les horloges et des estampilles est t_{min}, on peut affirmer que la simulation a avancé au moins jusqu'à la date t_{min}.

Le rôle du GVT est multiple. Son premier rôle concerne la gestion de la mémoire. Le mécanisme du *rollback* nécessite la sauvegarde en mémoire des informations suivantes :

- l'état du *pl* à différents instants,
- les messages déjà traités pour une réexécution éventuelle,
- les anti-messages des messages envoyés pour un *rollback*.

La mémoire étant limitée, il faut nécessairement nettoyer celle-ci des informations obsolètes. Le GVT fournit un critère pour détruire ou non une information sauvegardée. En utilisant l'équation précédente, on peut facilement voir que tout risque de *rollback* à une date antérieure au GVT est écarté puisqu'aucun message ne possède d'estampille inférieure au GVT. Dans ce cas, il est sauf de détruire toute information dont l'estampille est inférieure au GVT. Une exception s'applique cependant aux états sauvegardés car il est nécessaire de conserver au moins une sauvegarde d'un état antérieur au GVT cela afin de pouvoir le restaurer en cas de *rollback* se produisant avant la prochaine sauvegarde.

Son deuxième rôle concerne les entrées/sorties effectuées lors de la simulation. Par exemple, si des écritures dans des fichiers ou des affichages sur l'écran doivent être réalisés, il est nécessaire de les reporter jusqu'à ce qu'ils soient validés.

3.4.2 Algorithme de calcul du GVT

Le GVT, comme nous l'avons dit, est une composante essentielle de l'approche optimiste et son rôle est multiple. Sur une machine à mémoire partagée, il est plus aisé de déterminer la valeur du GVT, celle-ci pouvant même être déterminée en continu [Fuji89][Xiao95]. Maintenant, il n'est en général pas simple d'obtenir une valeur du GVT dans un environnement distribué car 2 problèmes majeurs se posent : le problème des messages en transit et celui des délais de communication non identiques entre les *pls*. Les algorithmes de calcul sont alors soit du type centralisé où c'est un *pl* donné, appelé *initiateur* qui va initier le calcul en envoyant aux autres *pls* un message `start`. Il se charge alors de recueillir l'ensemble des réponses pour calculer le GVT et le diffuser ensuite. Ce *pl* initiateur peut soit être un processus dédié ou un *pl* quelconque. Dans ce dernier cas, chaque *pl* de la simulation a la possibilité de décider, à partir de ces informations locales, d'initier une phase de calcul. L'algorithme peut aussi être du type distribué et est généralement plus complexe dans ce cas. Dans tous les cas, l'aspect performance ne doit pas être négligé car le GVT, pour remplir son rôle, doit être calculé souvent. Ce calcul doit donc interférer le moins possible avec le processus d'exécution des messages, il est même possible d'utiliser pour cela des architectures matérielles spécifiques. Le plus souvent, on ne cherchera pas à trouver une valeur exacte du GVT, ce qui nécessiterait de bloquer la simulation durant toute la phase de calcul, mais plutôt une borne inférieure notée $\widehat{GVT}$, ce qui peut être fait sans trop d'interférences.

Un grand nombre d'implémentations de Time Warp utilise l'algorithme proposé par Samadi [Sama85] qui est du type centralisé avec un processus *manager* dédié. Pour résoudre le problème des messages en transit, Samadi propose d'utiliser un mécanisme d'acquittement. Lorsqu'un *pl* p envoie un message m à un *pl* q, ce dernier doit l'acquitter en renvoyant à p un message $ack(m)$ indiquant qu'il a bien reçu m. De cette manière, si une phase de calcul du GVT venait à être initiée, le *pl* p pourrait déterminer, en comptabilisant les messages envoyés et les acquittements reçus, si oui ou non des messages sont en transit. Le grand problème de cet algorithme est qu'il introduit un trafic supplémentaire très important puisque pour chaque message réel, un message d'acquittement est nécessaire. Bellenot propose une version optimisée de l'algorithme de Samadi dans [Bell90]. Lin et Lazowska évitent les acquittements en ajoutant aux messages un champ S_n contenant un numéro de séquence [Lin90]. De cette manière, lorsqu'une phase de calcul est initiée par le *manager* chaque *pl* q envoie aux *pls* $p \in V(q)$ le dernier numéro de séquence $S_n(p)$ qu'il a reçu provenant de p. Le *pl* récepteur p peut alors déterminer l'estampille minimale des messages encore en transit.

D'Souza propose un algorithme nommé *passive response GVT* (pGVT) qui possède la particularité par rapport aux algorithmes présentés jusqu'à maintenant de donner l'initiative aux *pls* pour la mise à jour du GVT [DSou94]. Cet algorithme nécessite l'acquittement des messages car il a été conçu pour fonctionner dans un environnement non fiable. Un *GVT manager* "simplifié" existe toujours et se charge de déterminer le minimum des valeurs reportées par les *pls* et de le diffuser. Chaque *pl* met à jour son GVT local, noté LGVT, et décide lui-même d'envoyer cette valeur au *GVT manager* si : (i) un message m dont l'estampille $t(m)$ est inférieure à LGVT est reçu et (ii) lorsque le *pl* estime que le GVT, maintenu par le *GVT manager*, peut être avancé. Cette dernière décision se fait en estimant la nouvelle valeur qu'aurait le GVT grâce à une estimation moyenne du taux d'avancement du GVT, $\overline{\Delta GVT}$, diffusé par le *GVT manager*, et par une estimation moyenne K du temps, exprimée en nombre de ΔGVT, pour envoyer les informations au *GVT manager* et recevoir l'acquittement. De la sorte, si l'on a :

$$GVT + K * \overline{\Delta GVT} \geq LGVT$$

alors le *pl* envoie son LGVT au *GVT manager*. Intuitivement, on comprend que si la relation précédente est vraie, alors c'est que le GVT peut être avancé et que ne pas envoyer LGVT pénaliserait l'avancement de la simulation. L'avantage de cet algorithme tient dans le fait que l'estimation K est locale à chaque *pl*. Ainsi l'algorithme s'adapte lorsque l'environnement est très hétérogène. De plus, une conséquence immédiate de la "décentralisation" des décisions est de répartir les communications vers le *GVT manager* dans le temps.

Une approche originale est proposée dans [Toml93] où la valeur du prochain GVT est déterminée à l'avance en spécifiant un *target virtual time* (TVT) diffusé à tous les *pls* par un *pl* initiateur IP par le biais d'un événement $< TARGET, TVT >$. Soit LP l'ensemble des *pls* et $LP^- = LP - \{IP\}$, lorsque le LVT d'un *pl* $p \in LP^-$ devient supérieur ou égal à TVT, i.e. p vient d'exécuter $< TARGET, TVT >$, alors p envoie un message REPORT à IP. Celui-ci peut ensuite conclure que le nouveau GVT est maintenant TVT lorsque tous les *pls* de LP^- ont envoyé leur message REPORT. L'avantage de cette méthode est d'éviter l'utilisation d'acquittement et de présenter un *overhead* inférieur à la plupart des autres algorithmes.

Steinman propose SPEEDES GVT, implanté dans le système SPEEDES, qui n'utilise pas de mécanisme d'acquittement mais qui plutôt attend et force la réception, mécanisme de *flushing*, de tous les messages en transit avant le calcul du GVT [Stei95]. Fonctionnant avec *Breathing Time Warp*, le calcul du GVT se produit au moins à chaque fin de cycle sur initiative de chaque *pl* en utilisant une barrière de synchronisation "lâche" permettant la poursuite de l'exécution des événements. Le mécanisme du *flushing* permet en outre d'instaurer un contrôle de flux des messages dont la conséquence directe est une réduction des fautes temporelles.

Dans [Srin93] les auteurs utilisent un matériel spécifique pour le calcul du GVT. Ce matériel consiste en un réseau parallèle de réduction (RPR) et le calcul s'effectue sans interférence (ou très peu) car à chaque processeur central est associé un processeur auxiliaire entièrement dédié au calcul du GVT. Le RPR consiste en un arbre d'unité de traitement de hauteur $log_2(n)$ où n est le nombre de processeurs. La figure 3.16 montre un tel réseau pour 8 processeurs. Des acquittements sont néanmoins nécessaires au sein du RPR mais ceux-ci sont réalisés de manière matérielle et sont donc moins pénalisants. Il est alors possible d'obtenir de très bonnes performances tant en vitesse qu'en précision.

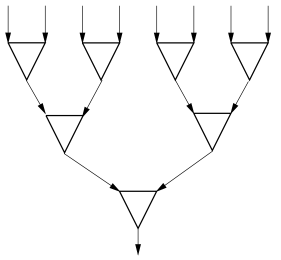


Figure 3.16 Exemple d'un réseau parallèle de réduction

3.4.3 Optimisation de la sauvegarde des états et de la mémoire

Un des problèmes de Time Warp provient du fait qu'il est nécessaire de sauvegarder l'état du *pl* de manière périodique. Il est même souvent nécessaire de sauvegarder cet état à chaque fois qu'un nouveau message est traité [Tall91]. Cependant pour optimiser la gestion de la mémoire et réduire ainsi l'espace nécessaire, certaines études visent à augmenter et optimiser l'intervalle entre deux sauvegardes d'état, *infrequent state saving* [Lin93][Rönn94a][Flei95]. Une méthode simple consistant à fixer un intervalle de sauvegarde déterminé de manière statique présente le désavantage de ne pas prendre en compte les fluctuations qui peuvent survenir pendant la simulation. Lin et al. Dans [Rönn94a], les auteurs proposent d'adapter cet intervalle pour chaque *pl* en se basant sur un compromis entre le coût d'un *rollback* et le gain obtenu en augmentant l'intervalle de sauvegarde. Ils proposent également d'adapter cet intervalle pour optimiser non plus le temps d'exécution en priorité mais plutôt l'espace mémoire. D'autres études utilisent une technique de sauvegarde incrémentale qui ne retient que les changements du vecteur d'état [Baue93][Stei93b]. Dans les deux cas, réduction du nombre de sauvegardes et réduction de la taille de la sauvegarde, le coût d'un *rollback* peut malheureusement devenir plus grand puisqu'il est nécessaire de retourner plus en arrière dans le premier cas et de reconstruire un état cohérent à partir de données non complètes dans le second. Il faut également remarquer qu'augmenter de manière non contrôlée l'intervalle de sauvegarde entraîne aussi une augmentation de la quantité de mémoire nécessaire pour conserver les messages. En effet, entre deux sauvegardes il faut conserver les événements pour pouvoir regénérer des états intermédiaires. Cependant, de manière générale, ces techniques permettent dans bien des cas de réduire considérablement la quantité de mémoire nécessaire et par conséquent le besoin d'un GVT très précis et aussi à jour que possible est moins grand.

La sauvegarde des états et des anti-messages nécessite beaucoup de mémoire. Alors que les techniques de sauvegarde optimisées permettent de réduire la quantité de mémoire, un problème se pose néanmoins lorsqu'il n'y en a plus du tout. Jefferson a été le premier a proposer un algorithme, appelé *message sendback*, qui permet à un récepteur de renvoyer un message à l'émetteur lorsqu'il n'y a plus suffisamment de mémoire pour le stocker [Jeff85]. A la réception du message retourné, l'émetteur va effectuer un *rollback* avec libération des anti-messages qui permettront alors la libération de tampons mémoire supplémentaires. Gafni propose une extension de cette approche en autorisant la libération de tampons mémoire occupés par les anti-messages et les états sauvegardés en plus de ceux occupés par les messages en entrée [Gafn88]. Ces deux approches permettent de récupérer de la mémoire au sein d'un *pl* à son initiative. Jefferson propose ensuite l'algorithme *cancelback* qui permet de récupérer de la mémoire dans n'importe quel *pl* de la simulation [Jeff90]. Facilement implémentable sur une machine à mémoire partagée, cette méthode est difficilement réalisable dans un environnement distribué. Lin propose alors *artificial rollback* qui permet d'utiliser le mécanisme existant du *rollback* à des fins de récupération mémoire [Lin91b]. Ces deux dernières propositions permettent l'utilisation de Time Warp dans un environnement très limité en mémoire ; le but étant de pouvoir rivaliser avec une simulation séquentielle en terme de quantité mémoire nécessaire. Des études analytiques et expérimentales montrent que Time Warp peut afficher de bonnes performances avec très peu de mémoire supplémentaire par rapport à une simulation séquentielle [Akyi93, Das93].

3.4.4 Améliorations de la stratégie d'annulation

Envoyer les anti-messages dès qu'une faute temporelle est détectée, comme nous venons de le voir, dénote un comportement dit d'*agressive-cancellation*. Les améliorations qui peuvent être apportées au schéma de base consistent à optimiser le mécanisme du *rollback* en ne défaisant que le minimum de chose, *lazy cancellation*, ou en essayant d'optimiser la réexécution des événements après un *rollback*, *lazy reevaluation*. Nous allons adopter la notation supplémentaire suivante pour désigner l'anti-message d'un message positif :

> **Notation 2 (Anti-message)** *Soit un événement e ayant entraîné la génération d'un message m. L'anti-message de m est noté $\overline{m}$.*

Lazy cancellation

Lorsqu'un message fautif est reçu, il est nécessaire d'annuler tous les messages générés depuis l'instant de l'erreur en envoyant les anti-messages correspondants. Dans [Gafn88] l'auteur propose une amélioration appelée *lazy cancellation*. Le principe du *lazy cancellation* repose sur la constatation qu'un message fautif ne modifie pas systématiquement la suite des messages générés. Si l'exécution du message fautif redonne les mêmes messages, les anti-messages correspondants ne sont pas envoyés. Dans ce schéma de fonctionnement, un *pl* n'envoie pas immédiatement l'anti-message $\overline{m}$ mais attend que son horloge locale ait dépassé la valeur $t(\overline{m})$ pour l'envoyer si le message positif m correspondant à l'anti-message n'a pas été recréé. Cette méthode a l'avantage de réduire le nombre d'anti-messages et d'éviter une réexécution redondante des événements. Cependant l'*overhead* qui en découle[10] peut gommer le gain obtenu. De plus, dans [Fuji90a] Fujimoto fait remarquer que l'utilisation du *lazy cancellation* peut rendre les erreurs de causalités plus pénalisantes car, en retardant l'envoi des anti-messages, la détection des fautes temporelles en est également retardée. La figure 3.17 illustre le fonctionnement d'une telle stratégie. Nous supposons que tous les événements e_i génèrent un et un seul message m_i. L'anti-message correspondant $\overline{m_i}$ est sauvegardé dans une file d'anti-messages. L'arrivée du message fautif e_{3b} devrait immédiatement provoquer l'envoi des anti-messages $\overline{m_4}\ldots\overline{m_7}$ si on utilise l'*agressive-cancellation*. Dans le cas du *lazy cancellation*, ceux-ci ne sont pas immédiatement envoyés, l'état du *pl* est juste restauré et le message fautif e_{3b} est exécuté. Celui-ci génère un message dont l'anti-message associé $\overline{m_{3b}}$ est sauvegardé dans la file des anti-messages, et les événements $e_4 \ldots e_7$ exécutés de nouveau. Si on suppose que les événements e_4, e_5 et e_7 génèrent exactement les mêmes messages qu'avant l'exécution du message e_{3b} alors les anti-messages associés sont déjà dans la file. Si seul e_6 génère un message différent m'_6, alors l'anti-message associé $\overline{m'_6}$ est sauvegardé dans la file des anti-messages. Une fois que l'horloge locale du *pl* devient supérieure à $t(\overline{m_6})$, on peut affirmer qu'aucun message positif correspondant à $\overline{m_6}$ ne sera recréé—en effet, la deuxième exécution de e_6 a généré un message différent

[10] Après le traitement d'un message fautif, il est nécessaire de vérifier à chaque génération de message que celui-ci n'ait pas déjà un anti-message correspondant, auquel cas il a déjà été produit avant le *rollback*.

de la première, l'anti-message est donc différent. Seul $\overline{m_6}$ doit donc être envoyé pour annuler l'action du message positif correspondant qui n'a pas été recréé. Le gain est donc de trois anti-messages : $\overline{m_4}, \overline{m_5}$ et $\overline{m_7}$.

Lin montre que la probabilité pour qu'un anti-message ne soit pas envoyé au cours d'un *rollback* est grande dans le cas de trafic faible et de serveurs multiples ou simple avec période de repos [Lin95]. Ces résultats sont en fait assez intuitifs dans la mesure où un trafic faible génère plus de périodes de repos et augmente par conséquent la probabilité qu'un client fautif soit inséré dans une période de repos. De ce fait, si ce client est entièrement traité avant la fin de la période de repos, les autres clients ne sont pas affectés. Dans [Radh96], les auteurs reportent qu'une accélération de 10% a pu être obtenue en simulant un réseau de files d'attente avec le système WARPED. Cependant le gain qui peut être obtenu est fortement dépendant de l'application et ces résultats sont loin d'être représentatifs. De plus, l'application en question était bien adaptée au *lazy cancellation* et un gain de 10% seulement pour ce cas laisse présager un gain bien plus modeste dans des cas plus généraux.

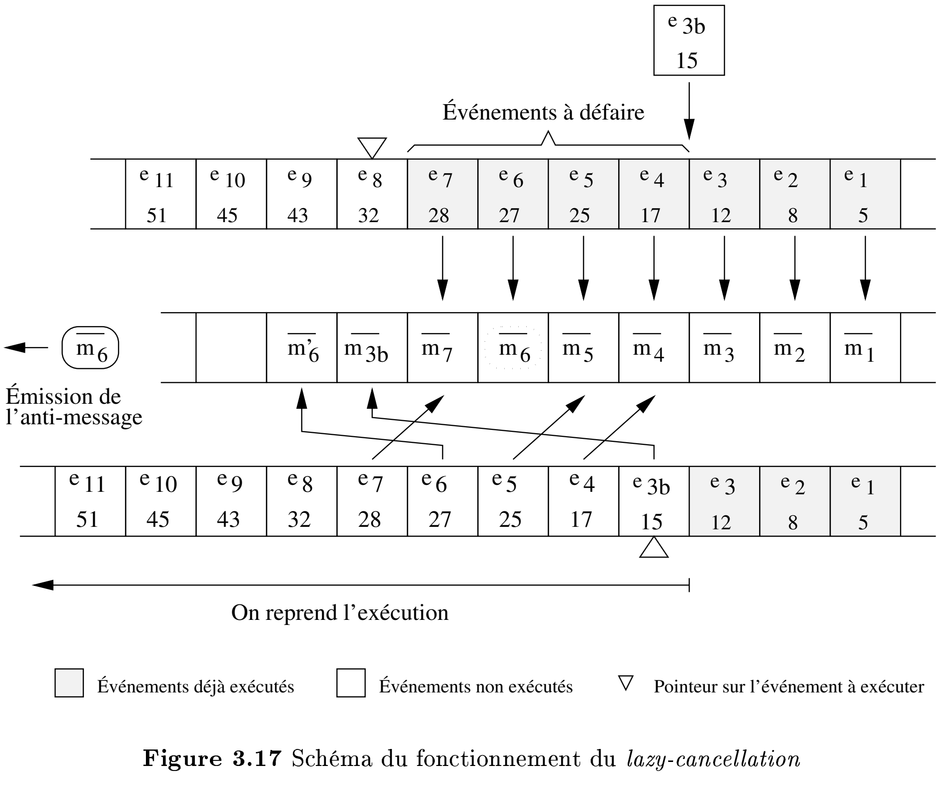


Figure 3.17 Schéma du fonctionnement du *lazy-cancellation*

Lazy re-evaluation

Toujours pour réduire le coût associé aux *rollbacks*, West propose le mécanisme du *lazy re-evaluation* [West88]. Dans cette approche, à la réception d'un message fautif, on va restaurer un état antérieur et, si l'état du *pl* ne change pas après l'exécution du message fautif et si aucun nouveau message n'est arrivé entre-temps, alors il n'est pas nécessaire d'exécuter de nouveau les événements annulés mais juste de rétablir l'état du *pl* tel qu'il était avant le *rollback*. Rétablir l'état revient donc à sauter les événements déjà exécutés. La figure 3.18 illustre le schéma de fonctionnement d'une telle méthode. Nous avons représenté par une valeur entière l'état d'un *pl*. Cet état noté S_i est sauvegardé après l'exécution d'un message e_i. Dans cet exemple, l'arrivée du message fautif e_{3b} provoque la restauration de l'état S_3. Si l'exécution de e_{3b} ne change pas l'état du *pl*, on peut sauter la réexécution des messages en restaurant directement l'état S_7. L'utilisation du *lazy re-evaluation* évite de réexécuter inutilement les messages lorsque l'on est sûr d'aboutir à un résultat déjà connu.

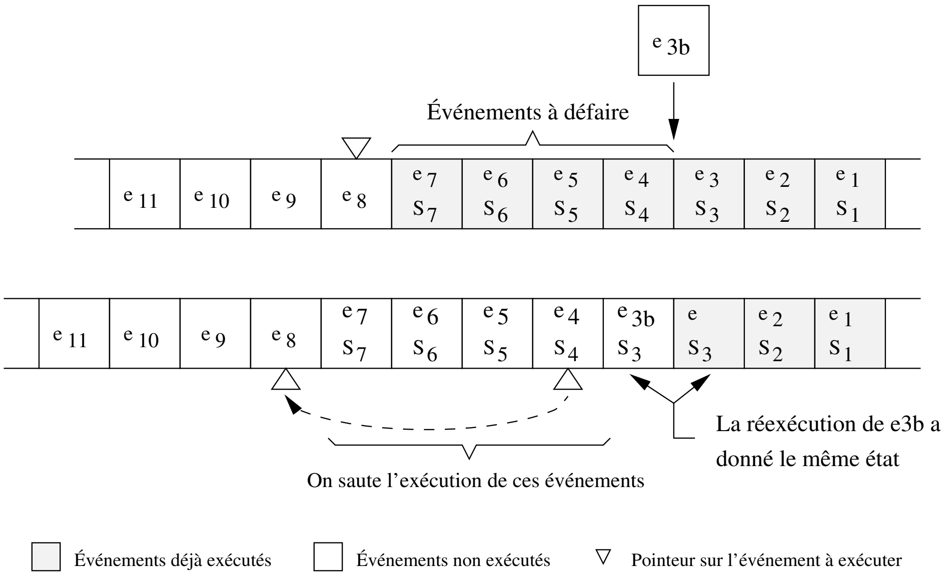


Figure 3.18 Schéma de fonctionnement du *lazy re-evaluation*

Filter—Exécution conditionnelle

Dans [Prak91] l'auteur propose un algorithme, nommé *Filter* pour limiter la propagation des erreurs de causalité dans les approches optimistes. L'algorithme utilise une liste d'assomptions, attachée à chaque message, pour déterminer si celui-ci est définitif et donc exécutable ou s'il ne sera pas plus tard annulé par un anti-message, auquel cas on ne l'exécute pas. Chaque *pl* maintient une liste d'assomptions composée d'entrées de la forme $< c, s, t >$ qui signifie que pour le lien c, aucun message fautif ne sera reçu avec un numéro de séquence supérieur à s et une estampille inférieure à t. Le numéro de séquence permet de comptabiliser les messages sur un lien particulier. Le rôle de la liste d'assomptions est de mémoriser les dépendances entre messages et garder ainsi une trace des messages fautifs présents dans le système, cela afin d'éviter l'exécution d'un message si celui-ci doit être annulé par un prochain anti-message. L'auteur présente un exemple de simulation de 4 noeuds comportant une boucle, comme illustré dans la figure 3.19. Il est alors possible de déterminer

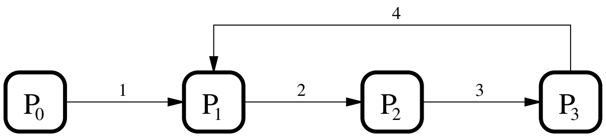


Figure 3.19 Réseau de 4 noeuds. Les chiffres indiquent le numéro du lien.

s'il est nécessaire d'exécuter ou non un message en comparant la liste d'assomptions associée au message à celle du *pl*. Si la liste du message comporte une entrée $< c_r, s_r, t_r >$ et celle du message $< c_m, s_m, t_m >$ avec $c_r = c_m$, alors le message peut être ignoré si on a :

$$s_m < s_r \text{ et } t_m > t_r \tag{3.1}$$

Dans le cas de la figure 3.19, on suppose que chaque *pl* modélise un serveur FCFS à temps de service constant égal à 5 unités de temps. Supposons ensuite que P_0 envoie trois messages d'estampille 5, 30 et 50. Les différentes étapes effectuées sont alors résumées ci-après :

Étapes

1. P_1 reçoit les messages et envoie les trois messages d'estampille 10, 35 et 55. Son horloge est alors à 55.
2. P_2 reçoit les messages et les envoie à P_3 avec les estampilles 15, 40 et 60. P_3 les reçoit et les renvoie à P_1 avec les estampilles 20, 45 et 65.
3. P_1 reçoit $< e, 20 >$ de P_3 et conclut que ce message est un message fautif puisque son horloge a la valeur 55. Il effectue un *rollback* à la date 20, envoie les anti-messages d'estampilles 35 et 55, puis exécute le message fautif ainsi que les anciens messages exécutés hors séquence. Il envoie ensuite les messages d'estampille 25, 35 et 55, les deux derniers ayant déjà été exécutés. Son horloge est alors à 55.
4. P_1 reçoit ensuite le deuxième message de P_3 d'estampille 45. Ce message est un message fautif pour P_1 car son horloge est à 55. Il doit donc effectuer un deuxième *rollback*.
5. ...

La solution proposée dans [Prak91] permet d'éviter le deuxième *rollback* (étape 4) car le deuxième message de P_3—celui d'estampille 45—n'est plus un message valide puisque P_1 a déjà envoyé un anti-message l'annulant lors de son premier *rollback* (étape 3). L'utilisation de la liste d'assomptions permet d'ignorer ce deuxième message de la façon suivante :

- Le *pl* P_1, lorsqu'il avance son horloge à une valeur T, peut affirmer qu'aucun message devrait lui arriver sur ses liens d'entrée avec une estampille inférieure à T. De même pour le numéro de séquence qui doit être supérieur à celui associé au lien. Si on suppose que le numéro du lien entre P_3 et P_1 est 4, alors P_1 peut ajouter l'entrée $< 4, 0, 35 >$ à sa liste d'assomptions et envoyer cette entrée avec le message d'estampille 35.

- Lorsque P_1 reçoit le premier message de P_3 d'estampille 20, il peut ajouter l'entrée $< 4, 1, 20 >$ dans sa liste. Lorsqu'il recevra le message d'estampille 45 de P_3, qui contient l'entrée $< 4, 0, 35 >$ mise précédemment par lui-même, il peut l'ignorer car l'équation 3.1 est vérifiée puisque l'on a $0 < 1$ et $35 > 20$.

L'idée consiste donc à faire mémoriser par P_1 les messages qu'il a émis, à l'aide des informations associées à chaque message, cela afin qu'il puisse déterminer s'il doit par la suite ignorer ou exécuter les messages qu'il reçoit et qui ont été engendrés par ceux qu'il a émit. L'exemple utilisé dans [Prak91] considère un réseau de serveurs FCFS qui représente en fait des délais. Dans un cas plus général la solution proposée fonctionne beaucoup moins bien.

3.5 LES APPROCHES MIXTES ET ADAPTATIVES

Une approche mixte dénote par exemple une approche conservatrice où l'on a ajouté une certaine dose d'optimisme pour la rendre moins pessimiste. Cela peut aussi être une approche optimiste à la base dans laquelle on a réduit l'optimisme excessif—l'approche la plus couramment utilisée. Cependant, la dose d'optimisme enlevée ou rajoutée est statiquement déterminée et ne change pas durant la simulation. Cette solution permet à moindre coût de tirer parti de chacune des approches.

Les approches adaptatives, quant à elles, peuvent également mélanger les deux approches comme dans les approches mixtes mais dans des proportions non statiques et qui peuvent évoluer au cours de la simulation. Il est aussi possible pour elles d'implémenter les deux approches, conservatrice et optimiste, pour pouvoir passer indifféremment de l'une à l'autre au cours de la simulation, cela plusieurs fois si nécessaire. Un certain critère détermine alors le changement de comportement. Cette approche résulte de la constatation que pour certains problèmes il est préférable de se bloquer plutôt que de se risquer à effectuer des opérations non valides et que pour d'autres c'est le contraire[11]. Comme il est de plus extrêmement difficile de déterminer à l'avance le type de comportement optimal pour un problème donné—comportement qui peut par ailleurs changer au cours de la simulation—l'avantage des approches adaptatives est justement leur faculté à s'adapter au problème au fur et à mesure qu'il évolue pendant la simulation. D'un autre côté, la complexité de ces protocoles est bien plus grande puisqu'ils doivent incorporer les deux types de comportements. Dans la figure 3.20a, une approche mixte est un point fixe sur la droite alors qu'une approche adaptative peut la parcourir au cours de la simulation.

[11]Généralement les approches conservatrices se comportent mieux lorsque le nombre de messages est grand alors que pour une approche optimiste c'est le contraire. Cela est facilement expliqué par le fait que plus le nombre de messages augmente et moins une approche conservatrice bloquera, tandis que le risque de *rollbacks* pour une approche optimiste augmentera.

La figure 3.20b est un exemple de courbe représentative des performances en fonction du nombre de la densité de messages. On peut ainsi voir qu'il peut exister un point de fonctionnement optimal.

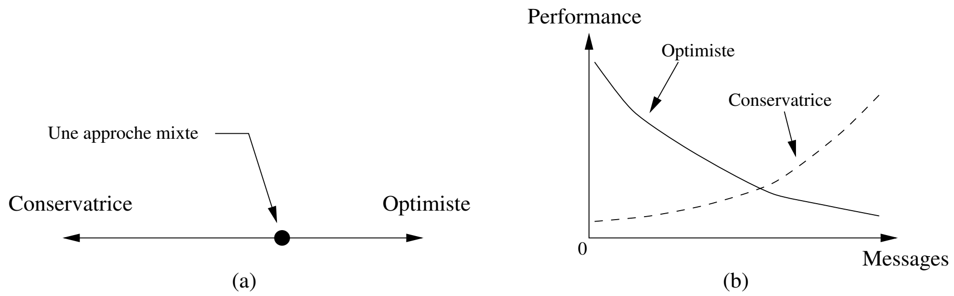


Figure 3.20 (a) approche mixte et adaptative. (b) performances en fonction de la densité de messages.

L'algorithme ADAPT proposé par Jha [Jha94] illustre parfaitement le fonctionnement d'un algorithme bien adaptatif. Cet algorithme possède deux niveaux de contrôle, un contrôle local, *Local Control Mechanism* (LCM) et un contrôle global, *Global Control Mechanism* (GCM) qui fournit une estimation de l'estampille minimale susceptible d'être reçue par un *pl*. Le LCM peut être soit du type conservateur soit du type optimiste. Pour le GCM, on peut soit utiliser le principe des *null*-messages soit utiliser un calcul de GVT. De cette manière, les différents mécanismes des protocoles conservateurs et optimistes sont complètement intégrés et interchangeables. Il est dès lors possible d'avoir une simulation où un certain nombre de *pls* sont en mode conservateur tandis que d'autres sont en mode optimiste.

Maintenant, limiter l'optimisme des protocoles peut se faire de deux manières différentes : introduire une fenêtre d'exécution qui va déterminer les événements qui seront éligibles pour l'exécution en parallèle ou alors instaurer une politique de blocage. Dans [Srin95b], Srinivasan et Reynolds définissent une classe de protocoles appelés *Asynchronous Adaptive Waiting Protocols* (AAWP) qui est suffisamment générale pour contenir la plupart des protocoles à fenêtre et à stratégie de blocage. Ce blocage peut se faire à deux endroits : soit à l'entrée, soit à la sortie. Effectuer un blocage en entrée signifie interrompre l'exécution des événements par le processus logique. Ce type de blocage est pour l'instant majoritaire et peut être défini par un processus AAWP. Récemment, Kuratti et Sanders [Kura96] ont proposé un algorithme original qui consiste à instaurer un blocage à la sortie. Le but de cette retenue est de réguler l'envoi des messages afin d'éviter de propager les erreurs et, dans une certaine mesure, d'en créer en envoyant des messages qui devront être annulés dans un futur proche.

3.5.1 Les protocoles à fenêtre dans les approches optimistes

Filtered Time Warp

Dans [Luba89] l'auteur propose une solution, nommée *Filtered Time Warp*, inspirée de l'algorithme *Bounded Lag* déjà décrit dans le cadre des approches conservatrices.

L'idée est de conjuguer Time Warp avec *Bounded Lag* afin de limiter les *rollbacks* en cascade par une limitation du parallélisme qui évite une trop grande différence d'avancement entre les *pls*. Le fonctionnement de Time Warp, à savoir l'annulation des messages lorsqu'un message fautif est détecté reste le même. Tout comme dans l'approche conservatrice, une borne statique B limite le degré de parallélisme entre les différents *pls*. La principale difficulté est alors de déterminer la valeur optimale de B.

Moving Time Window

Moving Time Window (MTW) présenté dans [Soko89] est quelque peu similaire à la solution de Lubachevsky qui restreint les événements exécutables en parallèle dans une fenêtre. Cependant cette fenêtre permet aux auteurs de définir 4 états temporels pour les messages selon la valeur de leur estampille par rapport à la fenêtre. La table 3.5.1 résume les 4 états possibles pour un message d'estampille t avec une fenêtre $[t_{inf}, t_{sup}]$ et une valeur d'horloge du *pl* de h.

Etats	*Position de t*	*Exécutable ?*
past	$t_{inf} \leq t < t_{sup}$	oui
now	$t = h$	non
current	$h < t < t_{sup}$	non
later	$t > t_{sup}$	non

Tableau 3.4 4 états possibles pour un message.

La différence réside également dans la gestion dynamique du déplacement de la fenêtre par un mécanisme de demande et par des mécanismes de recouvrement d'erreurs plus spécifiques que le *rollback*. Pour cela, des informations supplémentaires, appelées *tags* doivent cependant être véhiculées par les messages qui permettent alors de distinguer les messages modifiant l'état d'un *pl*, *Side Effecting* (SE), de ceux qui ne le modifient pas, *Non Side Effecting* (NSE). Typiquement, les messages du type NSE sont des messages de requêtes, nécessitant seulement l'envoi d'une valeur appropriée. Cette distinction autorise l'utilisation de mécanismes de récupération d'erreurs plus simples et spécialisés que le *rollback* brutal. Ceux-ci sont la révision de la file des événements et l'utilisation des données sauvegardées. Notons que cette approche permet en fait d'exécuter hors séquence des événements fautifs si ceux-ci sont du type NSE. La table 3.5.1 illustre les stratégies utilisées, lorsqu'un message fautif est reçu, en fonction du type du message et de la classe du message. Cette classe pouvant être soit ANTI, pour les anti-messages, soit NORMAL pour les messages normaux.

Remarquons que pour les états supérieurs à *past*, la révision de la file est utilisée lorsque les messages sont des anti-messages. Cela s'explique par le fait que ces anti-messages doivent annuler des messages normaux qui se trouvent dans le futur proche du *pl*. L'annulation consiste alors à supprimer tout simplement ces messages normaux comme s'ils n'avaient jamais été envoyés.

Etats	*Classe*	*Type*	*Stratégie*
past	Anti	SE	Rollback
past	Anti	NSE	Révision de la file
past	Normal	SE	Rollback
past	Normal	NSE	Utilisation des données sauvegardées
now	Anti	SE	Révision de la file
now	Anti	NSE	Révision de la file si le message n'a pas été exécuté, sinon Rollback
now	Normal	SE	Ajout à la liste
now	Normal	NSE	Ajout à la liste
current	Anti	SE	Révision de la file
current	Anti	NSE	Révision de la file
current	Normal	SE	Rollback
current	Normal	NSE	Ajout à la liste
later	Anti	SE	Révision de la file
later	Normal	SE	Ajout à la liste
later	Anti	SE	Révision de la file
later	Normal	SE	Ajout à la liste

Tableau 3.5 Stratégie utilisée en fonction du type et de la classe du message fautif reçu.

Contrairement à une approche purement synchrone, la fenêtre d'exécution est dynamiquement déplacée, bien que sa taille reste la même, pour permettre un fonctionnement continu sans phase d'attente pour les *pls* avançant plus vite. Cela semble apporter plus de souplesse par rapport à une fenêtre fixe mais nécessite un mécanisme de demande entre les *pls* qui augmente en contrepartie le nombre de messages échangés.

Breathing Time Buckets

Breathing Time Buckets (BTB) [Stei91], fût le premier algorithme utilisé dans le système SPEEDES, *Synchronous Parallel Environment for Emulation and Discrete Event Simulation*. L'algorithme est cyclique et chaque cycle peut être décomposé en deux phases : une phase de fonctionnement parallèle et une phase de libération des messages générés. Retenir les messages avant de les délivrer évite d'étendre les *rollbacks* aux autres *pls* puisque seuls les messages valides sont envoyés. Aucun anti-message n'est donc utilisé dans l'algorithme. Uniquement des *rollbacks* locaux, effectués par restauration de l'état, sont à considérer. BTB part du principe que les événements dont l'estampille est inférieure à la date du prochain nouveau message généré sont exécutables en toute sécurité, cette date est appelée *event-horizon*[12] dans

[12] L'*event-horizon* est assimilable à la borne supérieure d'une fenêtre dynamique.

[Stei91]. Les autres, ceux dont l'estampille est supérieure au *event-horizon* peuvent ne pas être valides. En effet, si on suppose que les prochains nouveaux messages générés par le *pl* A ont l'estampille t_{n_i}, alors le *pl* A possède un event-horizon à la date $t_h = t_{n_1}$. Supposons que lors de l'exécution la valeur de l'horloge dépasse t_h, et que le message d'estampille t_{n_1} est envoyé au *pl* B. Supposons ensuite que B envoie en retour un message d'estampille t_s au *pl* A, avec par exemple $t_{n_4} > t_s > t_{n_1}$. Alors ce message est un message fautif pour A qui entre-temps a exécuté hors séquence les messages d'estampille t_{n_2}, t_{n_3} et t_{n_4} (figure 3.21). La première phase de l'algorithme consiste donc à traiter tous les messages dont l'estampille est inférieure au *event-horizon*. Une fois cette limite atteinte, le *pl* doit s'arrêter pour attendre que tous les autres *pls* aient eux-mêmes atteint leur *event-horizon* local. On peut alors remarquer qu'il est impossible de recevoir des messages provenant des autres *pls* pendant cette première phase car aucun *pl* n'est autorisé à délivrer de messages[13]. Une fois tous les *pls* synchronisés, la deuxième phase commence par déterminer le *global event horizon* défini comme étant le *event-horizon* minimum parmi ceux de tous les *pls* (figure 3.22).

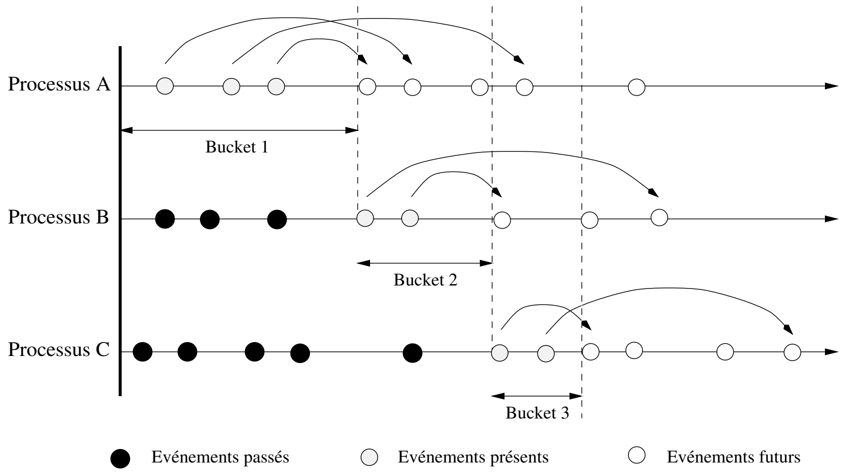


Figure 3.21 Les événements situés en dessous de l'horizon sont toujours exécutables.

Ce *global event horizon* est ensuite assimilé au *Global Simulation Time* (GST) et la deuxième phase consiste à envoyer tous les messages générés lors de la première phase dont l'estampille est inférieure ou égale au GST. Les *pls* ayant continué leur exécution au-delà du GST devront éventuellement effectuer des *rollbacks*. Ces *rollbacks* sont obligatoirement locaux et ne nécessiteront donc pas l'envoi d'anti-messages puisqu'aucun des *pls* a franchi son *event horizon* et a par conséquent envoyé aucun message aux autres *pls*—dans le cas de la figure 3.22, les *pls* A et C doivent éventuellement effectuer des *rollbacks* car ils sont tout les deux au-delà du GST. Le fonctionnement cyclique de *Breathing Time Buckets* permet d'éviter les *rollbacks* d'un *pl* à l'autre mais présente l'inconvénient d'une trop forte synchronisation car

[13]En effet, si l'*event horizon* est à la date $t_h = t_{n_1}$ et que les *pls* s'arrêtent obligatoirement avant t_h alors aucun message n'a encore été envoyé.

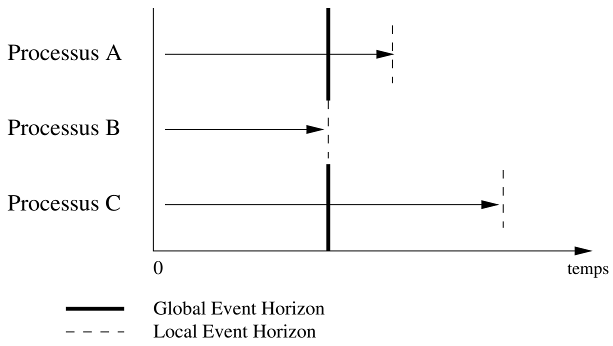


Figure 3.22 Global event-horizon.

les *pls* les plus rapides doivent attendre les *pls* les moins rapides pour continuer. Ce fonctionnement réduit considérablement le nombre d'événements exécutés par cycle.

Breathing Time Warp

Breathing Time Warp (BTW) décrit dans [Stei93a] est un mélange de *Breathing Time Buckets* et de Time Warp. Il est également implémenté dans le système SPEEDES. Notant les inconvénients des deux algorithmes, à savoir le nombre trop réduit d'événements exécutés par cycle pour BTB, et une instabilité liée aux *rollbacks* en cascade pour Time Warp, BTW essaie de concilier les deux solutions pour tirer parti des avantages de chacun. BTB en voulant éviter l'utilisation d'anti-messages et en instaurant uniquement des *rollbacks* locaux, retient les messages générés jusqu'à la certitude de leur validité. Time Warp par contre envoie les messages tout de suite en acceptant le risque de commettre une erreur. On peut résumer en disant que BTB est trop pessimiste et Time Warp trop optimiste. BTW résulte alors d'une constatation qui est la suivante :

– Plus on s'éloigne du GVT[14], et plus les messages ont des chances d'être non valides. Il est équivalent de dire que les messages les plus proches du GVT ont tendance à être corrects.

BTW peut alors être décomposé en deux phases de fonctionnement : une première phase utilisant Time Warp et une deuxième utilisant *Breathing Time Bucket*. Une borne N_1 sur le nombre d'événements délimite les deux stratégies. Time Warp est utilisé sur les N_1 premiers événements après le GVT, c'est-à-dire que les messages générés sont immédiatement envoyés avec risque de *rollback*. Cependant comme ces événements sont proches du GVT, ils ont de fortes chances d'être valides. Une fois les N_1 événements exécutés, la phase de BTB commence et les messages générés ne sont plus délivrés immédiatement, figure 3.23. On exécute les événements jusqu'à atteindre l'horizon comme défini dans BTB.

[14]Rappelons que le GVT définit le seuil atteint par la simulation. Aucun message ne possède alors d'estampille supérieure au GVT.

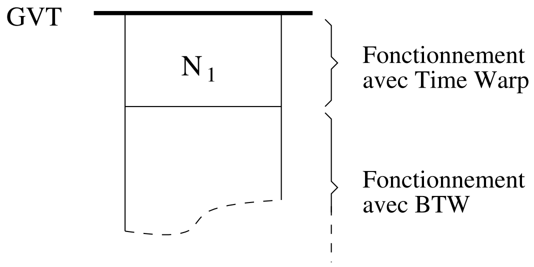


Figure 3.23 Les deux phases de fonctionnement de BTW.

L'union de BTB et de Time Warp permet de profiter des avantages de chacun, à savoir :

- Lorsque l'horizon est grand (i.e. beaucoup de *lookahead*), la plupart du travail est effectué avec BTB, ce qui réduit considérablement le nombre d'anti-messages.
- Lorsque l'horizon est réduit, la plupart du travail est effectué avec Time Warp. Cependant on réduit les risques d'instabilités dus aux *rollbacks* en ne traitant que les N_1 premiers événements après le GVT, et qui ont toutes les chances d'être valides.

Adaptive Time-Ceiling

Adaptive Time Ceiling est un protocole à fenêtre dont la borne supérieure, appelée *time-ceiling* est dynamiquement modifiée au cours de la simulation [Mats93]. Il est en ce sens bien différent de *Moving Time Window* où la fenêtre pouvait se déplacer mais où sa taille ne variait pas. La modification de la taille de la fenêtre se fait en observant les performances du simulateur et la fréquence R des fautes temporelles. La taille H de la fenêtre prend des valeurs discrètes définies à l'avance dans un tableau. Les auteurs préconisent l'utilisation d'une progression exponentielle dont un exemple est donné par la figure 3.24. La taille initiale de la fenêtre est fixée à

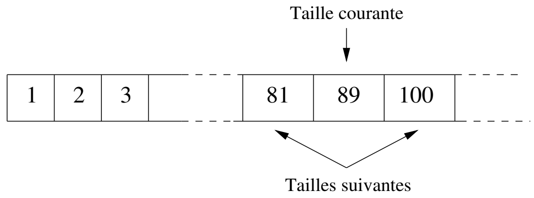


Figure 3.24 Valeurs discrètes que peut prendre la taille de la fenêtre.

la valeur minimale. L'algorithme pour déterminer la nouvelle taille de la fenêtre est alors très simple et est illustrée par l'organigramme de la figure 3.25. Dans cette figure, R est la fréquence de fautes temporelles nouvellement calculée et K une valeur représentative de la performance du simulateur en fonction de la taille de la fenêtre, calculée sur un historique de N valeurs. Les auteurs reportent cependant

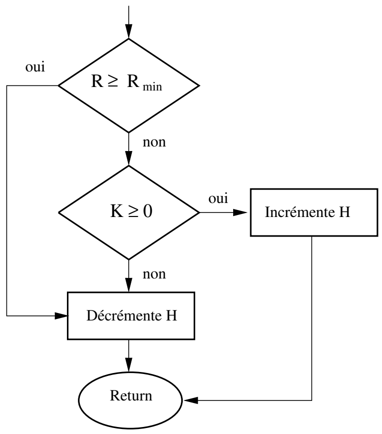


Figure 3.25 Détermination de la nouvelle taille de la fenêtre.

que quelques expériences sont nécessaires pour déterminer les valeurs optimales pour le choix du seuil R_{min} et pour celui de N.

Comme MTW, ATC tente d'éviter le phénomène d'attente inhérent à tout protocole à fenêtre lorsque les *pls* les plus rapides ont atteint le *time-ceiling*. Pour cela, les auteurs utilisent un “garde-fou”, appelé *reporting-time*, qui se situe un peu avant le *time-ceiling* et qui provoque le calcul du GVT lorsqu'il est dépassé. La borne inférieure de la fenêtre est alors déplacée jusqu'à la nouvelle valeur du GVT. Cette méthode permet d'après les auteurs de réduire le coût des diffusions par rapport à MTW car ceux-ci sont réalisées moins souvent. De plus, pendant la phase d'ajustement, les événements entre le *reporting-time* et le *time-ceiling* peuvent être exécutés.

Local Time Warp

Dans [Raja93] les auteurs proposent une décomposition du modèle en clusters qui leur permet d'utiliser une approche hiérarchique (algorithme HCTW) en instaurant à chaque niveau un contrôle différent. La décomposition en clusters s'inspire de la décomposition en groupes fonctionnels où les *pls* en relation peuvent être regroupés. Ce type de structure est très similaire à celui des réseaux informatiques où les machines sont regroupées pour former des LANs qui sont ensuite regroupés pour former des MANs et ainsi de suite. La figure 3.26 illustre une telle décomposition : les *pls* situés dans le même cluster utilisent Time Warp pour bénéficier du plus de parallélisme possible, alors qu'une approche conservatrice est utilisée pour contrôler les différents clusters. L'algorithme général est appelé *Local Time Warp* (LTW).

Chaque cluster possède deux portes, une porte d'entrée qui reçoit les messages entrant au cluster et une porte de sortie qui émet les messages sortant du cluster. A chaque porte est associée une horloge strictement croissante, IVT pour *Input Virtual Time* associée à la porte d'entrée et OVT pour *Output Virtual Time* associée à la porte de sortie. Tous les messages passant par ces portes seront estampillés de la valeur de l'horloge associée. Tous les messages inter-clusters passent obligatoirement

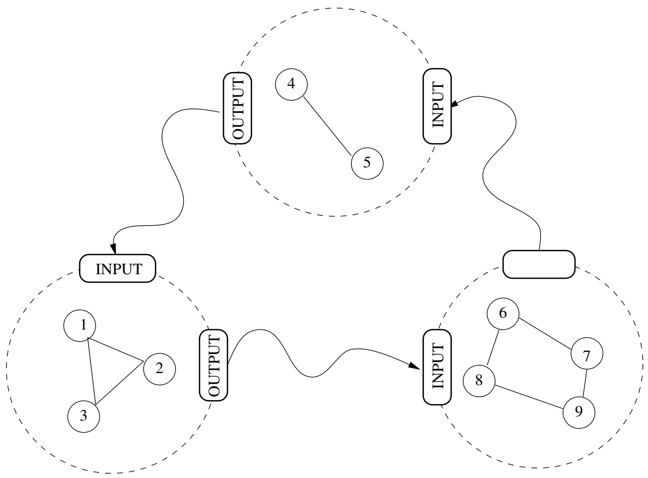


Figure 3.26 Décomposition en clusters.

par une porte. Les auteurs définissent ensuite le *Cluster Virtual Time* (CVT) de la manière suivante :

$$\forall i \in C, CVT_i = min(IVT_i, min(LVT_{i,j} \mid j = 1 \dots n_i))$$

C est l'ensemble des clusters du système et $LVT_{i,j}$, pour *Local Virtual Time*, est la valeur de l'horloge du *pl* j dans le cluster i. Cette définition du CVT est similaire à celle du GVT dans [Jeff85] à la différence qu'aucun message en transit n'est considéré, ce qui implique que les délais de transmission de messages inter-clusters soient nuls. LTW limite les *rollbacks* en cascade en évitant les *rollbacks* d'un cluster à l'autre. Pour cela, la porte de sortie retient les messages jusqu'à ce que le CVT soit devenu supérieur ou égal à l'estampille des messages à envoyer[15]. Comme un *pl* ne peut effectuer de *rollback* antérieur au CVT, du fait de la définition du CVT, aucun message d'estampille inférieure n'existe et on s'assure ainsi que les messages envoyés par la porte de sortie sont définitifs et ne risquent pas d'être annulés. Les messages délivrés aux différents *pls* du cluster par la porte d'entrée peuvent provoquer des *rollbacks* mais ceux-ci seront locaux au cluster.

3.5.2 Les approches bloquantes

Les approches bloquantes introduisent un temps d'attente fictif, et dans une certaine mesure on peut voir cela comme un peu de comportement conservateur, dans l'exécution du processus logique. La méthode de calcul du temps optimal de blocage varie du très simple au très complexe mais le but est le même, à savoir réguler le flux des événements pour réduire le nombre de fautes temporelles. Toutes les approches décrites ci-après établissent un blocage en entrée, sauf la dernière approche présentée qui, elle, retient les messages en sortie.

[15]Cette approche est similaire à BTB car le concept du *rollback* local et de la rétention des messages sont repris. Cependant le parallélisme est mieux exploité dans [Rajaei93] car le regroupement en cluster instaure un *rollback* local à chaque cluster et non à chaque *pl*.

Penalty-based throttling

Dans [Reih89] les auteurs étudient quelques solutions pour limiter l'optimisme dans Time Warp. En l'occurrence, si plusieurs *pls* sont gérés par un même noeud, alors les *pls* dans l'erreur sont pénalisés par le système qui ne leur donne pas la main. Au contraire, les bons *pls* se voient accorder plus de temps CPU. L'auteur pénalise les *pls* ayant envoyé des anti-messages en leur assignant une période de non activité. L'approche adoptée est réactive dans le sens où la décision de blocage vient après la constatation des fautes temporelles. L'auteur trouve que les performances de cette approche ne sont pas bonnes dans la mesure où il est difficile de déterminer à la fois la bonne durée de blocage et le délai de réaction de l'algorithme : un délai trop court rend l'algorithme très instable. Cette instabilité a essentiellement pour source la nature réactive de l'algorithme.

Adaptive Time Warp

Dans [Ball90] les auteurs proposent une amélioration de Time Warp qui affine et justifie la solution proposée par Reiher dans [Reih89]. En effet, dans Adaptive Time Warp un *pl* décide de suspendre temporairement le processus d'exécution des messages si un certain nombre d'erreurs vient de se produire—donc se met en inactivité. Les auteurs justifient leur solution en remarquant que les fautes temporelles ont tendance à survenir par paquets[16]. Cette attente supplémentaire est en réalité un moyen de remplir la file d'attente des événements pour avoir le plus possible d'événements éligibles. En effet, si le nombre d'événements en attente est grand, le risque de recevoir un message fautif est plus faible. Une borne BW (*Blocking Window*) détermine alors le temps d'attente du *pl*. Si BW est nul, on retrouve le schéma initial de Time Warp ; une grande valeur de BW revient à attendre un grand nombre de messages avant toute exécution. Le cas extrême où BW$\rightarrow \infty$ évite toutes les erreurs temporelles mais séquentialise la simulation puisqu'on attend que tous les événements soient arrivés et triés dans la file d'attente avant d'en exécuter un.

Les auteurs veulent minimiser le temps d'attente et le temps passé en erreur. Pour cela, la valeur de BW change dynamiquement au cours de la simulation et dépend de :

- la valeur courante de BW, notée ξ
- la proportion de temps passée en attente, notée P(B)
- la proportion de temps passée dans un état fautif[17], notée P(F)
- le taux de variation de la proportion de temps passé en attente lorsque la valeur de BW change, à savoir $\frac{d\,P(B)}{d\xi}$

[16]Ils remarquent en effet que durant la simulation, un ou plusieurs processeurs vont aller bien plus vite que certains autres en terme de temps de simulation et vont donc être sujets à des fautes temporelles répétées. Pour un processeur donné, les erreurs arrivent donc regroupées.

[17]Dès la première erreur, le *pl* est dans un état fautif, puis comme les erreurs arrivent par paquets le temps passé dans cet état fautif dépend donc de la taille du paquet d'erreurs.

– le taux de variation de la proportion de temps passé dans l'état fautif lorsque la valeur de BW change, à savoir $\frac{d\,P(F)}{d\xi}$

Ces valeurs sont périodiquement calculées et les auteurs utilisent la valeur de ξ satisfaisant l'équation (3.2) comme nouvelle valeur pour BW.

$$P(F) + \xi \frac{d\,P(F)}{d\xi} = 0 \tag{3.2}$$

Le point noir est que les auteurs ont effectué des simulations de l'algorithme plutôt que des expérimentations réelles. Les résultats pour des applications réelles ne sont toujours pas connus.

Elastic Time Algorithm

Dans [Srin95a], les auteurs ajoutent à la classe AAWP précédemment citée une sous-classe de protocoles appelés *Near Perfect State Information* (NPSI) qui se base sur l'acquisition d'informations "presque parfaites" sur le système afin de s'adapter très rapidement et d'ajuster en conséquence la valeur de leur fenêtre de blocage. Ils proposent en outre un protocole appelé *Elastic Time Algorithm* qui appartient à la classe NPSI et qui se basent sur la définition de deux fonctions $M1$ et $M2$ dont l'utilisation conjointe devrait donner un comportement adaptatif. Le rôle est de transformer les informations provenant du système en une valeur reflétant au mieux le degré d'erreur du *pl i*, notée EP_i (*error potential*). La fonction $M2$ utilise ensuite EP_i pour instaurer un contrôle sur l'exécution des événements. Les auteurs proposent pour $M1$ d'utiliser la différence entre la valeur de l'horloge locale et le GVT comme indicateur du degré d'erreur. L'obtention du GVT de manière très rapide et précise se fait en utilisant une méthode de réduction basée sur un réseau haute vitesse. Pour $M2$, les auteurs proposent de calculer une valeur de blocage δ_i de la manière suivante :

$$\delta_i = s \frac{EP_i}{MaxEP_i}$$

s est un facteur d'échelle et $MaxEP_i$ le maximum des EP_i observés. Comme pour tout protocole adaptatif, le bon choix du paramètre s est primordial et dépend malheureusement de l'application simulée. Les auteurs proposent d'utiliser deux critères de performances pour modifier la valeur de s. Ces critères sont R_c, le taux maximum d'événements définitifs[18] sur l'ensemble des *pls*, et P_{RB}, le pourcentage d'événements annulés. Le tableau 3.5.2 montre le comportement du protocole dans différentes situations. Cependant, les auteurs ne donnent aucune indication sur les valeurs représentatives de leurs paramètres. De plus, le réseau de réduction utilisé pour l'obtention rapide d'informations sur le système ne considérait que 4 processeurs. Dans le cas d'un système plus large, il est nécessaire de vérifier que la vitesse d'obtention des informations reste grande.

[18]Qui ne seront pas annulés plus tard.

R_c	P_{RB}	*Action*
petit	petit	Augmente l'optimisme rapidement, $s \uparrow$
petit	grand	Baisse l'optimisme graduellement, $s \searrow$
grand	petit	Augmente l'optimisme graduellement, $s \nearrow$
grand	grand	Baisse l'optimisme rapidement, $s \downarrow$

Tableau 3.6 Modification du facteur s.

Probabilistic Time Warp

Ferscha propose une méthode pour contrôler l'optimisme de Time Warp, appelée *Probabilistic Time Warp*, qui se base sur une étude probabiliste des estampilles des messages arrivant à un *pl* [Fers95a]. L'étude probabiliste consiste à prédire la valeur de l'estampille du prochain événement en se basant sur un historique de valeurs précédemment reçues. La probabilité de faute temporelle peut alors être déduite, et la suspension des exécutions ordonnée s'il s'avère extrêmement hasardeux de continuer. Assez complexe, cet algorithme utilise l'analyse des séries temporelles avec les méthodes ARMA (*Autoregressive Moving Average*) et ARIMA (*Autoregressive Integrated Moving Average*) pour prédire l'évolution des estampilles sur les files d'entrée. Bien que souple et générale, cette approche présente néanmoins l'inconvénient d'introduire des calculs complexes qui demandent beaucoup de place mémoire et de temps. En particulier, il est nécessaire de disposer d'un historique relativement grand pour estimer et vérifier le modèle construit.

Local Adaptive Protocol

Les algorithmes adaptatifs décrits précédemment sont essentiellement des améliorations de Time Warp. Ces améliorations visent à adapter l'optimisme dans ce protocole. Dans [Hamn94], les auteurs présentent une approche complètement adaptative entre un comportement conservateur et optimiste. Ils cherchent à trouver le point de fonctionnement optimal qui maximiserait un critère représentatif de la vitesse d'évolution de la simulation. Pour cela, un réel représentant un temps de blocage est associé à chaque entrée du *pl* et lorsque des entrées sont vides, la procédure de sélection des événements peut soit décider d'attendre sur l'entrée vide (comportement conservateur), soit de bloquer temporairement pendant une durée égale au réel associé à l'entrée vide. Dans ce dernier cas, le comportement du *pl* se situe entre un comportement purement conservateur et un comportement purement optimiste, qui se présenterait lorsque le réel associé à l'entrée est nul. Cette valeur optimale de blocage sur chaque entrée doit maximiser un critère α qui est définit comme étant la quantité d'accroissement du temps de simulation par unité de temps réel.

Changepoint Algorithm

Changepoint Algorithm est une approche originale proposée par Kuratti et Sanders [Kura96] qui instaure un blocage à la sortie. Il est un peu similaire à Probabilistic Time Warp en ce sens qu'il utilise également une analyse des estampilles des messages en entrée. Cependant, cette analyse est effectuée en détectant les sous-séquences (point de changement) dans un flot de messages afin de prédire pour chaque entrée le taux d'augmentation des valeurs d'estampilles. De cette manière, un temps de retenu des messages en sortie peut être déduit pour éviter d'envoyer un message qui sera susceptible d'être annulé dans un proche futur par un message provenant d'une autre entrée possédant un taux d'évolution des estampilles moins rapide.

L'algorithme paraît général et suffisamment adaptatif pour des processus d'arrivée irréguliers. Cependant le comportement de l'algorithme n'est pas connu pour les cas où les *pls* possèdent un grand nombre d'entrées.

3.6 DÉCOMPOSITION TEMPORELLE

Les algorithmes que nous avons présentés jusqu'à présent procèdent par décomposition spatiale du système en un ensemble de petits sous-systèmes. Contrairement à la décomposition spatiale où un processeur physique est associé à un *pl*, la technique de la décomposition temporelle (*time parallel simulation*) assigne à un processeur physique non pas un processus logique mais une tranche de temps de ce processus logique. Si on note par $P[t_1, t_2[$ la tranche d'exécution du *pl* P de la date t_1 à la date t_2, alors on peut définir D_P comme étant une décomposition temporelle de $P[0, fin[$ par :

$$D_P = P[t_1, t_2[\cup P[t_2, t_3[\cup \ldots \cup P[t_{n-1}, t_n[$$

où $t_1 = 0$ et $t_n = fin$, et $\forall i = 1 \ldots n-1, t_i < t_{i+1}$. Si l'on note $p_i = P[ti, ti+1[$ un élément de D_P, la méthode consiste à assigner à un processeur physique phy_i un élément p_i. Pour avoir une vision identique à celle d'une modélisation traditionnelle où chaque processeur physique se voit assigner un processus logique il suffit de considérer l'ensemble des phy_i comme formant un processeur physique virtuel qui simulerait alors $P[0, fin[$ (figure 3.27).

Si on note par R_i le résultat de l'exécution de p_i, il est nécessaire de fournir R_i au processeur exécutant p_{i+1}. Le gain apporté par cette technique vient du fait que si après avoir fourni R_i, p_i exécute un *rollback* et produit encore R_i, alors p_{i+1} n'a pas besoin d'effectuer de *rollback*. Cette technique est assez similaire au concept du *lazy re-evaluation* mais autorise l'exécution en parallèle de deux mêmes tranches de temps par deux processeurs différents avec des événements différents. Cette possibilité permet par exemple de lancer deux calculs en parallèle puis de choisir ensuite le bon calcul.

Le principal problème de cette approche est celui de la détermination des états initiaux S_i à fournir à chaque tranche p_i. La plupart des algorithmes démarrent avec des valeurs de S_i un peu arbitraires puis utilisent un mécanisme de correction itératif pour ajuster S_i. Les performances d'une telle approche dépendent alors fortement du nombre d'itérations nécessaires pour observer une stabilité des différents S_i.

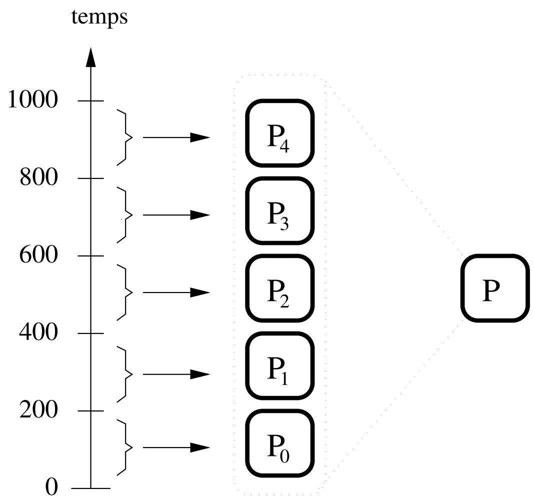


Figure 3.27 Décomposition en phase

Lin et Lazowska, [Lin91a], définissent un sous-ensemble $S_i^0 \in S_i$ appelé *état récurrent* (*regenerative substate*) défini comme un sous-ensemble dont la valeur des variables se répète indéfiniment si la simulation est infinie[19]. La détermination des différentes tranches se fait alors aux points où S_i^0 se répète. La simulation débute alors en assignant à S_i^0 une valeur de récurrence prédéfinie et aux autres variables de S_i des valeurs arbitraires. L'application simulée par les auteurs consiste en une file G/G/1 avec pour état récurrent la longueur de la file fixée à la valeur initiale 0. Les auteurs ne proposent cependant pas de méthodes générales pour déterminer ces états récurrents dans une application donnée. Wang et Abrams abordent ce problème et se proposent d'effectuer une pré-simulation pour déterminer ces états récurrents [Wang93]. Ils proposent également une méthode pour approximer les valeurs de certaines variables au lieu de leur donner des valeurs arbitraires dans le cas où il n'existe pas d'états récurrents [Wang92]. Dans [Niko94], les auteurs utilisent également une sorte de pré-simulation pour déterminer certaines valeurs initiales pour la simulation de réseaux de multiplexeurs en cascade.

De manière générale, l'application des techniques de décomposition temporelle se limite à certains systèmes relativement simples tels que les files d'attente car il reste difficile de déterminer correctement les états initiaux de chaque tranche.

3.7 SYNTHÈSE DE LA TAXONOMIE

Si deux catégories se démarquent facilement, à savoir les approches conservatrices d'une part et les approches optimistes d'autre part, il est extrêmement difficile de caractériser de manière précise les différentes variations dans une même catégorie de

[19] La notion d'état récurrent est similaire à la notion des points de regénération d'une simulation abordée dans le chapitre 2.

Protocole	ADP	AGR	JST	RSK	IDI	DDI	ADI	SYN
CMB Null			$\surd$			$\surd$		A
CMB Timeout-Null			$\surd$			$\surd$		A
CMB Demand-Driven Null			$\surd$				$\surd$	A
Carrier-null			$\surd$			$\surd$		A
CMB Deadlock Detection			$\surd$			$\surd$		LS
Bounded Lag			$\surd$		$\surd$			A
Conservative Time Window			$\surd$			$\surd$		A
Time Warp		$\surd$	$\surd$	$\surd$		$\surd$		A
Filter		$\surd$	$\surd$	$\surd$	$\surd$	$\surd$		A
Filtered Time Warp		$\surd$	$\surd$			$\surd$		A
Moving Time Window		lim	adap	$\surd$	$\surd$	$\surd$		A
Breathing Time Bucket	$\surd$		$\surd$			$\surd$	$\surd$	LS
Breathing Time Warp	$\surd$	$\surd$	$\surd$	adap		$\surd$	$\surd$	A/LS
Local Time Warp		$\surd$	$\surd$			$\surd$		A
Adaptive Time Warp	$\surd$	adap	$\surd$	$\surd$		$\surd$		A
Probabilistic Time Warp	$\surd$	adap	$\surd$	$\surd$		$\surd$	$\surd$	A
Changepoint Algorithm	$\surd$	adap	$\surd$	$\surd$		$\surd$	$\surd$	A
Elastic Time Algorithm	$\surd$	adap	$\surd$	$\surd$		$\surd$	$\surd$	A
Local Adaptive Protocol	$\surd$	adap	$\surd$	adap		$\surd$		A
Adaptive Time Ceiling	$\surd$	adap	$\surd$	$\surd$		$\surd$		A

Tableau 3.7 Caractérisation des protocoles selon la taxonomie de Reynolds.

protocoles. C'est pourquoi Reynolds [Reyn88] introduit une taxonomie qui permet de classer les protocoles de manière plus fine en utilisant les critères suivants :

- **Adaptabilité** (*Adaptability*) : L'adaptabilité (ADP) désigne la faculté pour un protocole de changer son comportement en se basant sur des observations effectuées lors de la simulation.
- **Agressivité** (*Agressiveness*) : L'agressivité (AGR) indique que l'exécution des événements peut se faire sans la connaissance totale de ce qui peut ensuite arriver. Cela revient à relâcher la contrainte de monotonicité dans l'ordre d'exécution des messages. Par contre, cela ne veut pas dire que les événements ne sont pas, finalement, exécutés dans le bon ordre grâce à un mécanisme de *rollback*.
- **Justesse** (*Accuracy*) : La justesse (JST) requiert une exécution finale des événements dans le bon ordre. Cela veut dire que le résultat d'une simulation parallèle est le même que celui d'une simulation séquentielle.
- **Risque** (*Risk*) : Le risque (RSK) indique l'envoi de messages résultant d'une politique agressive ou non-juste aux autres *pls*.
- **Intégration des informations** (*Knowledge embedding*) : L'intégration des informations (IDI) indique que le simulateur utilise des informations propres

au comportement des *pls*, cela dans un souci de performance. Cependant, cette solution rend les protocoles très dépendants du système à simuler.

- **Distribution des informations** (*Knowledge dissemination*) : La distribution des informations (DDI) indique que le bon fonctionnement de l'algorithme nécessite la transmission des informations aux autres *pls*.
- **Acquisition des informations** (*Knowledge acquisition*) : L'acquisition des informations (ADI) est similaire à la distribution des informations, mais elle est maintenant à l'initiative des *pls* récepteurs.
- **Synchronisme** (*Synchrony*) : Le synchronisme (SYN) indique le degré de synchronisation entre les *pls*. Cette synchronisation peut être *asynchrone* (A) ou *synchrone lâche* (LS), c'est-à-dire respectivement sans synchronisation et avec synchronisation périodique.

En utilisant cette taxonomie, le tableau 3.7 caractérise les différents protocoles étudiés jusqu'à présent.

3.8 LES OUTILS : LANGAGES ET ENVIRONNEMENTS

La situation est loin d'être la même pour le domaine de la simulation parallèle et celui de la simulation séquentielle. Ce dernier est maintenant un sujet bien mûr et des logiciels performants ne manquent pas. La simulation parallèle est par contre encore au stade de la recherche et les outils qui existent sont souvent issus du domaine universitaire. Tout comme en simulation séquentielle, il existe deux approches, celle orientée langage et celle orientée logiciel ou *environnement*. En fait le terme environnement décrit le mieux ces outils qui offrent des mécanismes de synchronisation de manière plus ou moins transparente à l'utilisateur. Il y a de grandes différences entre les environnements et les différents protocoles que nous avons présentés jusqu'à présent. Premièrement, un environnement de simulation distribuée essaie d'offrir une interface de programmation conviviale et cohérente, des sorties comme l'affichage ou l'impression et, le plus important, un maximum de transparence pour l'utilisateur final. Les protocoles que nous avons décrits précédemment sont souvent seulement des implémentations d'idées ou de concepts au stade expérimental. La notion de transparence est importante pour la simulation distribuée et c'est ce critère qui empêche actuellement l'utilisation générale des techniques de simulation distribuée. Par conséquent il ne faut pas s'étonner que la plupart des environnements utilisent l'approche optimiste qui nécessite, rappelons le, moins de connaissance sur le système à simuler. La deuxième différence découle de cette utilisation des méthodes optimistes qui oblige alors l'environnement de simulation distribuée à gérer ses sorties de manière plus complexe du fait de la possibilité d'annulation des actions passées. Un mécanisme comme le COMMIT des bases de données est nécessaire.

On s'aperçoit alors que la méthode conservatrice est plutôt utilisée par les langages car elle demande plus de connaissance sur le système, et plus particulièrement sur la manière dont le *lookahead* doit être extrait et exploité. Bien sûr, les langages peuvent aussi offrir une synchronisation optimiste, ou les deux comme le langage Maisie. De manière générale, ces langages choisissent plutôt d'étendre le jeu d'instruction d'un langage existant—souvent le langage C— pour offrir de nouvelles primitives adaptées à la simulation distribuée.

De manière générale, les outils que nous allons décrire dans la suite sont encore d'un usage délicat et sont loin d'être utilisables dans le cas général. Leur mise en œuvre nécessite du temps, moult expériences, et le gain en vitesse que l'on peut obtenir par rapport à une simulation séquentielle est fortement dépendant de l'application et de la manière dont le problème a été parallélisé.

3.8.1 Les langages pour écrire une simulation distribuée

Maisie

Maisie est un langage dérivé du langage MAY[20] et est basé sur le langage C. Il est développé par le Laboratoire de Calcul Parallèle de l'université UCLA. Il offre les mécanismes de synchronisation conservateurs et optimistes ainsi que les simulations séquentielles [Bagr94a, Bagr94b]. Cette intégration permet à l'utilisateur final d'écrire un modèle unique et d'utiliser les différentes méthodes de synchronisation avec très peu de modifications. C'est le seul langage à notre connaissance qui offre cette possibilité.

Un programme Maisie est un ensemble de définitions d'entités et des fonctions C. Une extension orientée objet est également proposée sous le nom de MOOSE. Une entité est du point de vue de la simulation ni plus ni moins qu'un sous-système du système à simuler. La définition d'une entité consiste alors à décrire le comportement du sous-système qu'il représente. Ces entités peuvent être créées et détruites de manière dynamique durant la simulation.

Pour améliorer les performances de Maisie, les auteurs proposent un mécanisme qu'ils appellent *optimisation sémantique* et qui permet, par exemple, d'augmenter la capacité de *lookahead* d'un *pl* ou de filtrer les *rollbacks* non nécessaires. Cette possibilité réduit un peu la transparence mais apporte à l'utilisateur expérimenté plus de contrôle.

SIMA

SIMA est un langage basé sur le langage C qui propose un moteur conservateur pour la synchronisation des processus logiques [Raja92]. Il est développé au *Royal Institut of Technology* à Stockholm. Les algorithmes utilisés sont *Three Phase Algorithm* [Ayan91] et *Conservative Time Window*[21] pour machines parallèles à mémoire partagée (Sequent Symmetry). Il est également possible d'effectuer des simulations séquentielles.

[20]MAY est un langage général de simulation qui utilise le paradigme d'envoi de messages pour modéliser et simuler les entités [Bagr87]. Il n'a, à la base, pas de mécanismes particuliers pour la simulation distribuée.

[21]Voir Chapitre 3, Section 3.3.6

La transparence a été un des soucis majeurs de SIMA qui cache la majorité des mécanismes de synchronisation au dépend d'une performance optimale. Cependant, comme dans Maisie, les auteurs offrent la possibilité pour les utilisateurs expérimentés de paramétrer plus en détail le simulateur par le biais de primitives dédiées. Le langage offre principalement un certain nombre de primitives pour la création de nœuds, l'interconnexion des nœuds et la gestion des messages.

La création d'une simulation distribuée se fait en déclarant les différents nœuds et en les connectant entre eux. Chaque nœud possède une fonction de traitement des messages et ne devient actif que sur réception d'un message. Les tests effectués sur un Sequent Symmetry à 16 processeurs montrent un speedup de 7.5 pour la simulation d'un tore 20x20 et un speedup de 12.5 pour ce même tore lorsque l'on augmente—artificiellement—le temps de service des événements.

Sim++

Sim++ [Baez90] est un langage qui ressemble à Maisie puisque les deux ont été influencés par le langage MAY. Cependant, contrairement à Maisie, Sim++ ne propose que l'utilisation d'un moteur optimiste utilisant Time Warp et dont les différents mécanismes sont totalement transparents à l'utilisateur. Comme dans Maisie, la construction d'une simulation se fait en créant des entités qui interagissent par envois de messages. Le noyau offre une interface orientée objet en C++ qui comporte des classes de base et des méthodes pour la gestion des événements. La nouveauté de Sim++ réside dans le fait qu'il est possible de regrouper un certain nombre d'entités dans des *clusters*. Les entités d'un même cluster sont situées sur le même processeur et peuvent donc accéder directement aux variables des autres entités, évitant ainsi des coûts de communication importants si elles doivent interagir très souvent. La manière dont les entités doivent être regroupées est laissée au soin de l'utilisateur qui a finalement un rôle important puisque les performances de la simulation découlent directement du partitionnement choisi.

3.8.2 Les environnements

TWOS

Sans doute un des premiers environnements de simulation distribuée, TWOS (*Time Warp Operating System*) [Reih90] a été développé au *Jet Propulsion Laboratory* (JPL) et a depuis servi à simuler diverses applications telles que les applications militaires [Wiel89], les applications de physique [Hont89], les réseaux de communication [Pres89] et les applications de biologie.

TWOS utilise Time Warp dans sa version de base et les auteurs reportent des résultats qui montrent de bons speedups dans la majorité des applications. Le noyau, développé originellement sur une Caltech Hypercube, a été porté sur un grand nombre de machines parallèles et s'avère donc portable. L'utilisation de TWOS nécessite, comme dans la plupart des cas, une décomposition du système à simuler en objets indépendants qui communiquent par envois de messages. Le noyau offre des fonctions de communication et des fonctions de synchronisation qui sont complètement transparentes à l'utilisateur. D'ailleurs, l'interface de programmation se limite essentiellement à des primitives d'envoi et de réception de messages ainsi qu'à la lecture du temps de simulation. Le mécanisme du *rollback* (copie des messages, restauration des états, envoi des anti-messages...) est pris en charge par le système lors de la détection d'une faute temporelle et il n'est pas possible de moduler le comportement du *pl* en changeant des paramètres comme les périodes de sauvegarde des états et de calcul du GVT ou la stratégie d'annulation des messages. Le calcul du GVT utilise un algorithme à deux phases décrit dans [Bell90] et s'effectue sans interrompre les processus logiques. La gestion de la mémoire prévoit également un mécanisme de retour des messages à l'envoyeur en cas de saturation du *pl* récepteur. Un mécanisme de répartition de charge permet en outre d'optimiser l'utilisation des processeurs. Les auteurs ont également proposé une stratégie de limitation de l'optimisme qui s'avère malheureusement peu performante du fait de sa nature trop réactive[22].

Dans TWOS, les messages sont de deux types : *event* et *query*. Un message de type *query* ne modifie pas l'état du *pl* et ne peut générer que d'autres messages *query* ou une réponse unique à l'émetteur. De ce fait, chaque type de message, *event* et *query*, possède sa partie de traitement distinct dans le corps du processus logique.

SPEEDES

Le système SPEEDES (*Synchronous Parallel Environnement for Emulation and Discrete Event Simulation*) a été initialement développé au JPL [Stei91] pour des applications militaires essentiellement. Dans sa première version, la synchronisation est assurée par l'algorithme *Breathing Time Bucket* qui comporte deux phases de fonctionnement : une première phase consistant en une exécution des événements sans libération de messages, puis une deuxième phase de libération des messages générés durant la première phase. *Breathing Time Warp*[23] a ensuite été rajouté pour augmenter le nombre d'événements exécutés par cycle. Des mécanismes complémentaires comme la sauvegarde incrémentale et l'utilisation du *lazy cancellation* sont également proposés. Cependant, c'est à l'utilisateur de demander explicitement la sauvegarde de l'état et de fournir une fonction de comparaison pour les états sauvegardés de manière incrémentale.

SPEEDES est un environnement écrit en C++ qui offre une interface de programmation orientée objet. Chaque objet de la simulation dérive donc d'une classe principale qui comporte un certain nombre de méthodes pour la synchronisation, le calcul du GVT (GST), etc... Récemment, les auteurs ont proposé de considérer également les événements comme des objets qui peuvent alors contenir les messages qu'ils ont générés et les modifications qu'ils ont apportées à l'état. Cette conception, assez originale, permet d'automatiser l'annulation des messages et la restauration des états lorsqu'il est nécessaire de supprimer un événement puisque que la des-

[22]Voir Chapitre 3, Section 3.5.2
[23]Voir Chapitre 3, Section 3.5.1

truction de ce dernier entraîne directement les informations qu'ils contiennent. Un mécanisme d'échange permet de récupérer les modifications qui sont contenues dans les événements lorsque ceux-ci sont entérinés.

SPEEDES offre en outre des fonctionnalités de simulation interactive pour pouvoir injecter en temps-réel des événements extérieurs et tracer la nouvelle évolution du système [Stei91]. Cette capacité d'interaction intéresse beaucoup les militaires et des simulations de champs de bataille ont été effectuées avec ce système sur une Caltech Mark III Hypercube de 32 nœuds [Stei93c]. Les gains en vitesse par rapport à une simulation séquentielle optimisée sont de l'ordre de 20.

WARPED

WARPED est un noyau de simulation optimiste développé à l'université de Cincinnati [Mart96]. Ecrit en C++ et testé sur des machines parallèles comme le Sun SparcCenter 1000 et l'Intel Paragon, il offre une vaste panoplie de mécanismes pour la sauvegarde des états (fixe, périodique), l'annulation des messages (Lazy-Cancellation, Dynamic Cancellation[24]), le calcul du GVT (pGVT, algorithme de Mattern) et l'ordonnancement (Round Robin, Lowest Event Timestamp Fisrt). Un module spécifique pour la construction de modèle à base de files d'attente est également fourni.

L'écriture d'une simulation distribuée avec WARPED consiste d'abord à définir les paramètres de fonctionnement, tels que les mécanismes que l'on désire utiliser, puis de compiler le noyau et de le lier aux programmes utilisateurs écrits en utilisant les classes et les méthodes du noyau. Produit récent, les auteurs ont expérimenté WARPED sur des modèles de réseaux de communication policiers, des modèles de machines multiprocesseurs à mémoire partagée, des modèles de réseaux de disques RAID et sur la simulation VHDL [Radh96].

PARSEVAL

PARSEVAL (PARalellizing Simulation for Performance EVALuation) [Muss94] est un environnement de simulation distribuée pour la simulation de réseaux de files d'attente. Il est développé à l'INRIA Sophia Antipolis et fonctionne sur un réseau de Transputer attaché à une station SUN. La méthode de synchronisation est du type conservateur avec deux variations par rapport au schéma de base :

- suppression des *null* messages qui sont immédiatement suivis par un message réel ou un autre *null* message,
- les *null* messages ne sont envoyés par un *pl* que lorsque celui-ci est bloqué.

S'adressant principalement à la simulation de réseaux de files d'attente, différentes méthodes d'ordonnancement (FCFS, Processor Sharing, Quantum...) sont supportées, chacune avec un calcul de *lookahead* bien spécifique. En comparant PARSEVAL à

[24] Le Dynamic Cancellation consiste à choisir pendant la simulation entre Agressive Cancellation et Lazy Cancellation

QNAP[25] [QNAP84], les auteurs constatent que les performances actuelles de PARSEVAL sont mitigées dans le sens où certaines configurations permettent effectivement un speedup de 2 à 4 alors que d'autres configurations, celles présentant des boucles, favorisent plutôt QNAP.

3.9 CONCLUSION DU CHAPITRE

La simulation distribuée semble être une solution pour la simulation de système où des méthodes séquentielles seraient énormément coûteuses en temps. Nous avons vu qu'il existe deux types d'approches initialement antagonistes : les approches conservatrices et les approches optimistes. La première approche est grandement pessimiste et limite le parallélisme contre l'assurance qu'il n'y aura pas d'erreurs de causalité. La deuxième, au contraire, favorise l'exécution en parallèle en autorisant les fautes temporelles et en instaurant un mécanisme de retour arrière. Si l'on souhaite simuler un grand système comme un réseau ATM à plusieurs centaines de noeuds, la question se pose de savoir quelle méthode doit être utilisée : conservatrice, optimiste ou une combinaison des deux ? Les méthodes conservatrices présentent des possibilités de blocages et l'overhead qui en résulte pour les éviter (*null* messages, *circulating marker...*) peut devenir énorme pour de grands systèmes. Dans [Fuji90a] l'auteur recense de nombreux problèmes inhérents aux approches conservatrices. On peut citer leur dépendance vis-à-vis du *lookahead* qui font que les performances sont grandement diminuées dès lors que les applications présentent des capacités de prédiction faible. On peut alors, dans le pire des cas, se retrouver avec une simulation séquentielle. Le parallélisme est de plus faiblement exploité du fait de l'absence d'optimisme. Dans le cas de deux événements $< e, t >$ et $< e', t' >$ où e peut influencer e' de manière directe ou indirecte, l'approche conservatrice requiert l'exécution de e avant celle de e' même s'il s'avère qu'une exécution en parallèle était possible. On peut ajouter que les mécanismes de résolution de blocage requièrent des configurations fixes interdisant toute gestion dynamique du système à simuler. Cependant, cette approche a l'avantage d'être simple et lorsque le problème s'y prête, elle affiche de très bons résultats.

Les approches optimistes sont quant à elles plus complexes à mettre en oeuvre et sont beaucoup plus délicates à optimiser car le nombre de paramètres influant sur leurs performances est plus grand. Parmi les principaux problèmes des approches optimistes, on peut citer les *rollbacks* en cascade du fait de la trop grande différence d'avancement entre les différents *pls* et l'*overhead* résultant de la sauvegarde périodique des états. Dans le pire des cas, les *pls* peuvent passer leur temps à retourner en arrière (*rollback trashing*). Les solutions pour limiter les retours arrières induisent en contre partie un overhead supplémentaire comme le synchronisme [Luba89], ou l'introduction d'informations complémentaires associées aux messages [Prak91]. Des optimisations sont possibles—*lazy-cancellation* et *lazy re-evaluation*—mais elles complexifient grandement les algorithmes. Celles visant à réduire le nombre ou la

[25] Voir chapitre 2, section 2.3.2.

taille des sauvegardes périodiques montrent de bons résultats lorsqu'elles sont correctement paramétrées. Des solutions pour limiter l'optimisme sont peut-être viables car elles permettent de réduire les risques de *rollback* en cascade. Cependant, les mécanismes basés sur l'utilisation de fenêtre d'exécution ne sont pas exempts de défauts. Parmi ceux-ci, Reiher cite dans [Reih89] la dépendance de la taille de la fenêtre par rapport au système simulé. Si théoriquement la valeur des estampilles ne doit pas être importante, seul l'ordre des événements entre-eux étant important, les méthodes basées sur la notion de fenêtre sont influencées par un changement de la valeur des estampilles, même si l'ordre global est conservé. En effet, en modifiant la valeur des estampilles, on change les performances de la simulation : la taille finie et constante à un instant donné de la fenêtre, qui détermine l'intervalle maximum entre deux événements exécutables en parallèle, fait que les événements écartés ne sont plus les mêmes (figure 3.28). Lorsque la taille de la fenêtre est dynamiquement ajustée, on constate une amélioration des performances mais le problème précédent demeure. Limiter l'optimiste par une stratégie de blocage semble être une solution

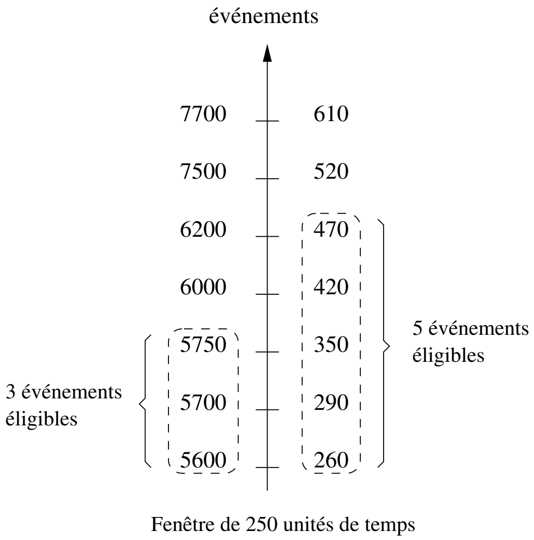


Figure 3.28 Influence de la taille de la fenêtre sur les événements écartés.

d'avenir. En effet, il est dès lors possible de varier le degré d'optimisme pour se positionner à un point donné de fonctionnement. En outre, les stratégies préventives semblent plus performantes que leurs homologues réactifs en regard des performances de l'approche proposée dans [Reih89]. On assiste d'ailleurs à une prolifération de ces approches préventives. Parallèlement aux études menées sur le logiciel, quelques pistes exploitent l'utilisation de matériels spécifiques pour la simulation distribuée. Parmi ces travaux, on peut citer ceux du groupe de Fujimoto à Georgia Tech sur un processeur dédié aux *rollbacks* [Fuji92] et le *WarpEngine Project* de l'université de Waikato. Ce projet étudie la faisabilité de coder matériellement le protocole Time Warp en entier sur une puce et en est actuellement au stade de la simulation de cette méthode. Comparées aux méthodes logicielles, les méthodes utilisant un matériel spécifique exhibent généralement des performances supérieures. Le coût supplémentaire associé à l'architecture et au matériel spécifique n'est pas négligeable mais il est bien possible qu'à l'avenir ce soit cette solution qui soit adoptée.

Le combat conservateur vs optimiste est d'autant plus rude qu'on peut trouver des applications pour lesquelles une méthode optimiste sera toujours plus performante qu'une méthode conservatrice, et vice versa. Sous des hypothèses relativement simplificatrices, Lipton montre qu'une simulation conservatrice ne peut battre une méthode optimiste que d'un facteur constant alors que ce dernier peut battre une méthode conservatrice avec un speedup linéaire [Lipt90]. Il reste cependant à savoir si ces hypothèses ne sont pas trop simples.

Pour conclure, notons que les techniques que nous avons exposées sont encore du domaine de la recherche. Elles sont d'un usage délicat et leur application est difficilement généralisable pour tous les problèmes. Ces raisons font que leur utilisation ne se soit pas encore étendue à toute la communauté "simulationniste". Pour cela, il faudrait des protocoles plus généraux et des environnements plus performants et conviviaux. La recherche va dans ce sens avec des études de plus en plus précises sur les environnements de simulation distribuée et sur la simulation distribuée interactive (DIS). Les techniques qui seront appliquées pour le DIS sont essentiellement des techniques optimistes qui permettent la création d'objets de manière dynamique et l'injection d'événements extérieurs pour changer dynamiquement le système sans avoir à reprendre la simulation depuis le début.

Dans l'étude qui nous intéresse, à savoir la simulation distribuée de réseaux ATM, le choix de la méthode la mieux adaptée est loin d'être entériné. Pour cela, il est nécessaire de regarder plus en détail les propriétés des systèmes ATM pour les caractériser du point de vue de la simulation. Ce sera l'objet des prochains chapitres.

Chapitre 4

Simulation distribuée de réseaux ATM : Etude préliminaire

L'évaluation de grands réseaux basés sur la technologie ATM est un sujet d'actualité. Lorsque nous avons commencé cette thèse, les réseaux ATM n'étaient qu'un modèle parmi d'autres pour expérimenter les mécanismes proposés en simulation distribuée. Maintenant beaucoup de travaux s'orientent nettement vers une simulation en détail de ces réseaux. Nul doute que l'on va désormais vers la voie de l´évaluation de performances de ces réseaux avec les nouveaux outils de la simulation distribuée. Le travail que nous effectuons s'inscrit pour une certaine part dans le cadre d'un contrat MASI/DRET sur les possibilités offertes par la simulation distribuée afin d'évaluer de grands réseaux ATM. ATM sera en effet utilisé par le réseau militaire Socrate, et fait déjà l'objet d'études de performances de la part du Celar et de Thomson-CSF avec l'outil Opnet.

Ce chapitre aborde les points suivants :

- Présentation de la technologie ATM, modèle de référence.
- Les travaux effectués sur la simulation distribuée de réseaux de communication ainsi que le sens que nous voulons donner à notre contribution.
- Les modèles de réseaux que nous utilisons.
- Le comportement de ces modèles du point de vue de la simulation distribuée.

4.1 LES RÉSEAUX ATM : PRÉSENTATION

Les réseaux ATM sont basés, comme leur nom l'indique, sur le mode de transfert asynchrone (*Asynchronous Transfer Mode*) qui permet de véhiculer les informations sous la forme de paquets de taille fixe appelés *cellules* [DePr93,Hand94,COMM95]. Ces cellules de 53 octets , dont 5 octets sont réservés pour l'entête, sont commutées à très haut-débit à travers le réseau par des éléments de commutation communément appelés *commutateurs ATM*, reliés entre eux par des liens en fibre optique[1]. La taille faible et fixe des cellules permet de simplifier les protocoles et les traitements effectués au sein des commutateurs. Cette technologie introduit donc de nouveaux concepts de fonctionnement que nous allons résumer ici. Cette brève présentation n'a pas pour objet d'être exhaustive, ni de débattre du futur de l'ATM. Elle nous intéresse dans la mesure où elle est de plus en plus utilisée.

4.1.1 Principe de fonctionnement

ATM fonctionne en mode connecté et il est nécessaire d'établir d'un *circuit virtuel* entre deux entités d'extrémité avant tout échange d'information. Ce circuit virtuel restera actif pendant toute la durée de la communication et sera identifié au niveau de chaque nœud par un *Virtual Chanel Identifier* (VCI) et un *Virtual Path Identifier* (VPI). Une connexion sera alors entièrement définie par une suite de couples <VCI,VPI>, qui identifient respectivement un VC et un VP, dans les tables de routage des éléments de commutation à travers le réseau. Un VP est défini comme un ensemble de VC et cette séparation VC/VP permet de réaliser un routage à deux niveaux (par exemple, les brasseurs ATM routent au niveau des VPs alors que les commutateurs agissent au niveau des VCs) et permet également au gestionnaire du réseau d'instaurer des liens permanents ou semi-permanents dans le but d'optimiser l'utilisation des ressources du réseau (figure 4.1). Le mode connecté permet également de réduire le temps de traitement des cellules dans les nœuds intermédiaires puisque la détermination du chemin de routage ne s'effectue qu'une seule fois, lors de l'établissement de la connexion.
La figure 4.2 représente l'en-tête de la cellule ATM qui comporte, par rapport aux en-têtes des protocoles actuels, un nombre très faible de champs. Le premier champ, **GFC** (*Generic Flow Control*), est codé sur 4 bits et sert à l'interface UNI pour un éventuel contrôle de flux. Ce champ disparaît de l'en-tête à l'intérieur du réseau (figure 4.2b). Suivent ensuite les identificateurs **VPI** et **VCI** codés respectivement sur 12 et 16 bits. Le champ **PT** comporte 3 bits et sert à indiquer le type de la cellule. En particulier, il sert à distinguer les cellules du trafic utilisateur de ceux du trafic de contrôle. Il est également envisagé de l'utiliser pour notifier l'existence d'une congestion dans un nœud. Le bit **CLP** qui suit sert à définir une priorité pour la cellule. Des propositions ont été faites pour utiliser ce bit comme marqueur des

[1]La nature des liens peut être différente : paire torsadée, câble coaxiale..., mais le support privilégié reste la fibre optique.

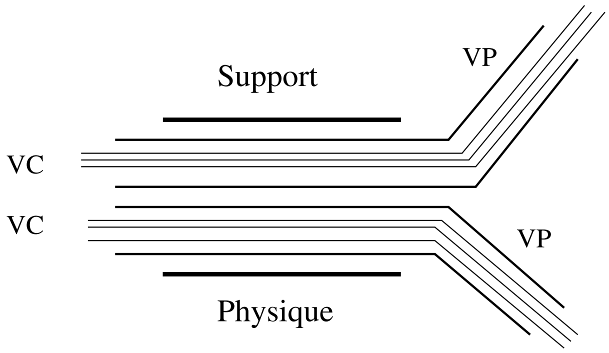


Figure 4.1 VP et VC dans ATM

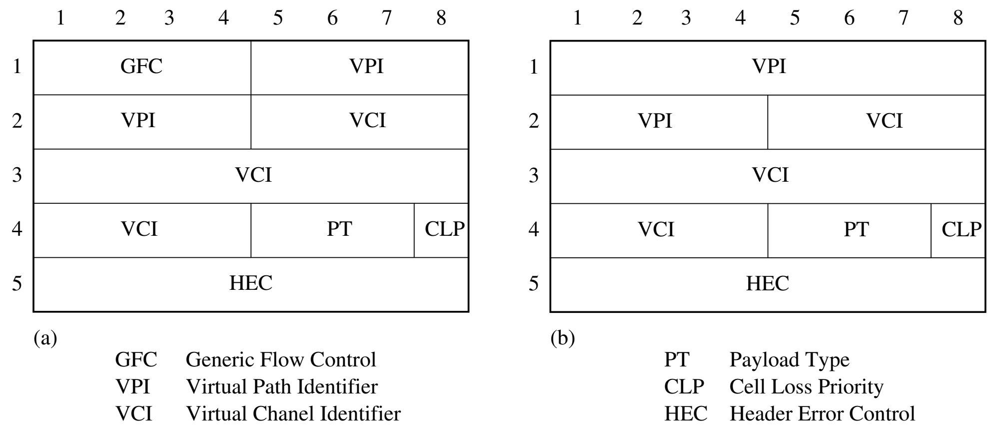


Figure 4.2 En-tête de la cellule ATM : (a) à l'interface, (b) au sein du réseau

cellules de moindre priorité. Finalement, le champ **HEC** contient un code correcteur d'erreur sur l'en-tête.

Il n'y a pas de contrôle d'erreur sur les données utilisateurs véhiculées à travers le réseau. Ce choix se justifie par le faible taux d'erreur des fibres optiques et par les contraintes plus lâches de certaines applications dont les données ne nécessitent pas un contrôle strict (images vidéo, son...). Tout cela contribue à rendre superflu un contrôle effectué à l'intérieur du réseau. Au contraire, cette tâche est plutôt relayée aux nœuds d'extrémité qui ont alors la charge de réaliser les opérations de contrôle et de reprise sur erreurs s'ils le jugent nécessaire. Le seul contrôle d'erreur réalisé au sein du réseau se fait sur l'en-tête des cellules où la moindre erreur peut rendre non exploitable la cellule entière. Ce contrôle prend la forme d'un code correcteur d'erreur, stocké sur l'octet HEC de l'en-tête, qui permet de détecter les erreurs sur les 4 premiers octets de l'en-tête. La suppression du contrôle des données utilisateurs enlève une charge non négligeable sur le traitement des cellules dans les éléments de commutation. Ce choix contribue grandement aux hauts débits offerts par ces réseaux.

Il n'y a pas non plus de contrôle de flux à l'intérieur du réseau ATM entre les éléments de commutation. Au contraire, celui-ci est relayé, comme pour le contrôle d'erreur, aux extrémités du réseau, et plus spécialement aux interfaces utilisateur/réseau (*User Network Interface*, UNI). La plupart des contrôles réalisés aux interfaces sont de nature préventive et comportent l'élaboration d'un contrat entre l'usager et le réseau, puis un contrôle d'admission, et enfin une surveillance continue du trafic utilisateur. Sont prévus dans l'en-tête de la cellule ATM trois champs permettant d'effectuer ce contrôle : le champ GFC défini uniquement à l'UNI, le champ PT et le champ CLP. Les fonctions exactes des bits GFC ne sont pas encore bien définies, par contre, des propositions concernent les champs PT et CLP pour leur permettre respectivement de notifier l'existence d'une congestion et de marquer les cellules pour rejet prioritaire. Dans ce dernier cas, les cellules marquées seront les premières à être rejetées lors d'un manque de ressources.

4.1.2 Modèle de référence

Le modèle de référence ATM comporte 3 plans (Recommendation I.321), chacun présentant un certain nombre de couches (figure 4.3). On a donc le plan *usager* qui contient les protocoles applicatifs, le plan de *commande* qui contient l'ensemble des procédures relatives aux trafics de contrôle : ouverture de connexion et autres signalisations diverses, et enfin, le plan de *gestion* qui a la charge de la gestion des différentes couches et de leur interopérabilité.

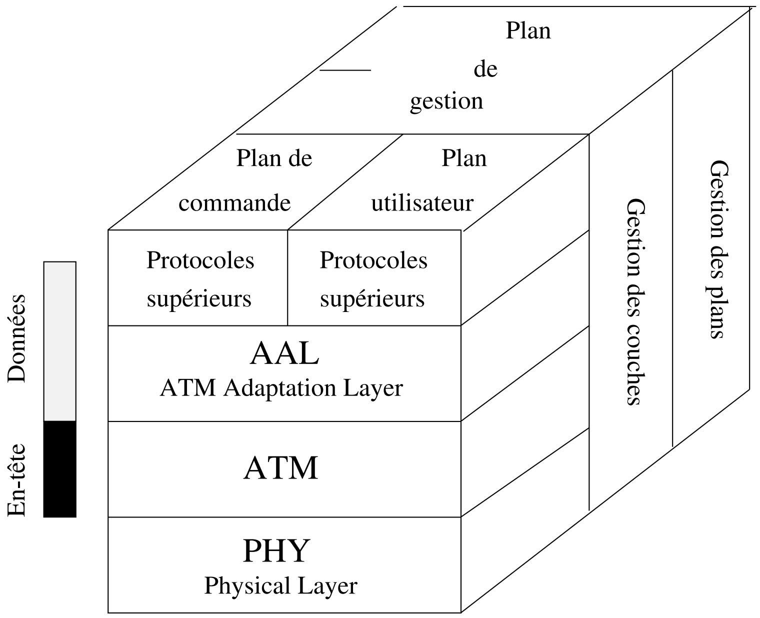


Figure 4.3 Le modèle de référence de l'ATM.

Caractéristique	*AAL 1*	*AAL 2*	*AAL 3/4*	*AAL 5*
Synchronisation	Forte	Forte	Faible	Faible
Débit	Constant	Variable	Variable	Variable
Mode de connexion	Connexion	Connexion	Sans connexion	Sans connexion

Tableau 4.1 Profils des différents AALs.

La couche *physique*, notée PHY, a la charge de l'accès au support physique qui est normalement la fibre optique. Dépendante du support, elle gère les aspects très liés au matériel comme la génération et la récupération des trames, le cadrage des cellules et la synchronisation au niveau bit. C'est également elle qui vérifie l'intégrité de l'en-tête de la cellule. La couche *ATM* assure quant à elle quatre fonctions principales :

1. Le multiplexage/démultiplexage des cellules provenant des différentes connexions à partir de plusieurs/un flot(s) de cellules. Cette fonction est réalisée grâce aux VCI et VPI.
2. La translation des VCI/VPI au niveau des nœuds ATM (routage).
3. Le traitement de l'en-tête qui consiste en l'ajout et le retrait de celui-ci lors de l'envoi et de la réception respectivement.
4. Le contrôle de flux éventuel au niveau de l'interface utilisateur/réseau grâce à certains bits de l'en-tête (bits GFC et CLP).

Cette couche constitue le principal traitement des éléments de commutation au sein du réseau. Aux extrémités du réseau, à l'interface utilisateur/réseau, on a la couche *AAL* (*ATM Adaptation Layer*) qui a la charge d'améliorer le service brut offert par la couche ATM pour l'ajuster aux exigences des applications. C'est dans cette couche que l'on trouvera les opérations de segmentation/réassemblage, les mécanismes de reprise sur erreur et la synchronisation de bout en bout comme la compensation de la gigue. Il existe maintenant[2] 4 types d'AALs : chacun proposant un certain nombre de services spécifiques pour les différents profils d'applications qui sont résumés dans le tableau 4.1.

Notons que du fait de la nature orientée connexion de ATM, le support d'un mode non connecté nécessite l'introduction d'un réseau virtuel de *serveurs en mode non connecté* en superposition sur le réseau normal en mode connecté. Pour plus de détails sur les réseaux ATM, nous invitons le lecteur intéressé à consulter [DePr93, Hand94, COMM95].

4.2 SIMULATION ET MODÈLES

Un réseau ATM est essentiellement un *système asynchrone* car les cellules sont envoyées d'un commutateur à l'autre sans aucune relation avec une horloge globale. Ainsi donc, les techniques de simulation *synchrone* ou par *incrément fixe* ne pourront pas représenter fidèlement le comportement temporel du système. Nous nous

[2] 2 types d'AALs ont fusionné pour donner l'AAL 3/4 et l'AAL 5 est apparu récemment.

intéressons donc aux approches *asynchrones* et en particulier celles dites *discrètes* où les changements apportés au système se produisent à des points discrets dans le temps et où le modèle va avancer par bonds variables selon la date d'occurrence des événements.

Dans notre travail, nous supposons l'utilisation d'une architecture parallèle ou distribuée composée de n processeurs pour exécuter la simulation. Le système à simuler est divisé en sous-systèmes disjoints, chacun d'entre eux étant alors simulé par un processus logique s'exécutant sur un processeur physique $p_i, 1 \leq i \leq n$. Les interactions qui peuvent exister dans le système réel sont représentées par l'échange de messages estampillés par une horloge locale du *pl* source. Les messages sont reçus par le *pl* destinataire et placés dans sa liste locale d'événements (LEV) en attendant d'être exécutés. D'autres méthodes de parallélisation existent mais leur champ d'application est relativement limité.

4.2.1 Les travaux effectués

La simulation des réseaux de communication est une des premières applications à avoir été distribuée [Chan79]. Tallieu et Verboven [Tall91] ont utilisé Time Warp sur un réseau de transputers pour simuler un réseau de machines TCP/IP. Chaque modèle de machine comporte donc les couches Ethernet, LLC, IP, et TCP. Les auteurs reportent cependant un nombre important de *rollbacks* et des gains en vitesse très décevants. Ces résultats négatifs ont essentiellement pour origine le degré élevé de synchronisation et d'interaction de l'application étudiée (architecture en couche, protocoles complexes). Mouftah et Sturgeon ont également utilisé Time Warp pour simuler sur 2 stations de travail un réseau d'une vingtaine de nœuds [Mouf90]. Les nombreux tests effectués permettent aux auteurs d'affirmer que le temps de simulation peut effectivement être réduit en distribuant la simulation sur les deux stations de travail. Ils notent cependant que ce gain est fortement dépendant de la taille des états et qu'une augmentation de celle-ci dégrade fortement les performances. Le coût des communications est de plus non négligeable, et le rapport temps CPU/temps de communication doit être relativement grand pour obtenir des gains intéressants. Dans [Earn92], Earnshaw et Hind utilisent une approche conservatrice pour simuler un réseau ATM sur des transputers. Le réseau est constitué d'une part de 4 commutateurs principaux formant un réseau central complètement maillé et d'autre part, de 8 multiplexeurs, possédant chacun 2 sources de trafic, reliés chacun à deux commutateurs principaux. Les auteurs reportent un taux de null-messages relativement bas lorsqu'ils simulent des taux d'utilisation supérieurs à 15%. Pour des faibles taux d'utilisation, le nombre de null-messages nécessaires à la prévention du blocage augmente assez considérablement. Les auteurs disent obtenir des speedups presque linéaires (12 transputers sont utilisés) par rapport à une version séquentielle lorsqu'ils simulent un taux d'utilisation des liens supérieur à 30%. Dans [Rönn94b], les auteurs simulent un réseau local haut débit pouvant fournir un service isochrone. Ils effectuent leur simulation sur une machine parallèle à mémoire partagée de 8 processeurs en utilisant deux protocoles : *Conservative Time Window* (CTW) pour l'approche conservatrice et Time Warp pour l'approche optimiste. L'approche conservatrice a été trouvée plus performante que l'approche optimiste. En particulier, pour un réseau de 60 nœuds, l'approche conservatrice montre un speedup de 6 alors que l'approche optimiste n'affiche qu'un speedup de 2. Les faibles performances de l'approche optimiste s'expliquent par le coût des sauvegardes d'état et la gestion des messages (allocation mémoire, copie...).

4.2.2 Notre contribution

La simulation de réseaux de communication est toujours un sujet d'actualité car cette application soulève un grand nombre de problèmes très spécifiques au domaine de la simulation distribuée. Parmi ceux-ci, on peut citer le niveau de synchronisation très souvent élevé du fait du nombre de messages échangés, la présence de boucles et le nombre élevé de canaux de communication entre les processus.

Par rapport aux travaux qui ont été effectués, nous souhaitions dans un premier temps engager une réflexion plus générale sur les mécanismes de simulation distribuée lorsqu'on les applique aux modèles de réseaux de communication. C'est l'objet de ce chapitre. Ensuite, nous voulons simuler non pas de simples transferts de cellules, comme c'est le cas dans tous les travaux qui ont été faits jusqu'à présent, mais des applications complexes et réalistes telles que le routage, la signalisation et des algorithmes de contrôle de congestion (ABR par exemple). Les expérimentations seront abordées dans le chapitre suivant.

4.2.3 Modèles de réseaux

Il est important de savoir à quel niveau de détail on souhaite modéliser le fonctionnement du réseau ATM. Dans les stations d'extrémité, les AALs possèdent un grand nombre de fonctions, dont la gestion des connexions, qui sont complexes à simuler. La couche ATM gèrent essentiellement le multiplexage/démultiplexage des cellules et l'ajout/retrait de l'en-tête. Quant à la couche PHY, c'est toute l'électronique concernant la transmission sur le support physique qui y est regroupée. Il n'est pas possible de modéliser et de simuler en détail l'ensemble de ces fonctionnalités. Les fonctions essentielles, et les plus pertinentes pour l'étude, doivent être dégagées et modélisées de manière simple et rationnelle. Nous avons décidé de ne pas nous attarder sur les AALs car l'objectif principal de notre étude est d'étudier le flux de cellules à travers le réseau. C'est pourquoi ce sont essentiellement les fonctions de la couche ATM, principale couche de traitement des éléments de commutation à l'intérieur du réseau, que nous modéliserons.

Au sein du réseau, les informations seront véhiculées sous la forme de cellules de taille fixe de 53 octets par des commutateurs ATM dont les deux principales fonctions sont la réception et l'émission de cellules. De ce fait, au niveau cellule, deux événements seront considérés dans notre modèle : un événement $< rec, t_r >$ indiquant la réception d'une cellule à l'instant t_r et un événement $< env, t_d >$ indiquant l'envoi d'une cellule à l'instant t_d. Il est ensuite possible d'ajouter l'équivalent d'un champ

Payload Type pour différencier les cellules de données des cellules protocolaires telles que les cellules `setup` et `connect` pour donner plus de réalisme sans pour cela modéliser les fonctions complexes des AALs. Pour modéliser complètement le réseau, il est nécessaire de modéliser le trafic appliqué en entrée. La modélisation de trafic est une lourde tâche, et nous n'allons pas aborder ce vaste problème. Nous proposons uniquement des sources de trafic simples qui génèrent des messages $< rec, t_r >$ où t_r sera distribuée de manière exponentielle. Nous allons dans la suite préciser les modèles de commutateurs et de réseaux utilisés dans cette étude (les lecteurs pourront également consulter [Gbur95]).

Notre modèle de commutateur

Un grand nombre d'architectures ont été proposées dans la littérature pour les commutateurs ATM. Les lecteurs peuvent consulter [DePr93, Han94] pour une complète description des architectures existantes. Pour notre part, nous adoptons un modèle de commutateur à architecture non-bloquante avec des tampons en sortie (figure 4.4). D'un point de vue strict de la simulation, les détails sur l'architecture du commutateur n'ont que très peu d'importance. En effet, il est toujours possible de considérer qu'un processus commutateur (i) reçoit des messages $< rec, t_r >$ dans sa liste d'événements, chaque message générant une suite de n événements $e_1, \ldots e_n, n \geq 1$, avec $e_n =< env, t_d >$. t_d est défini par $t_d = t_r + t_c$, où t_c représente le temps de commutation qui est considéré constant. Et (ii) qu'un ou plusieurs messages $< rec, t_a >, t_a = t_d + t_t + t_{delai\_lien}$, où t_t est le temps de transmission, seront envoyés vers d'autres processus.

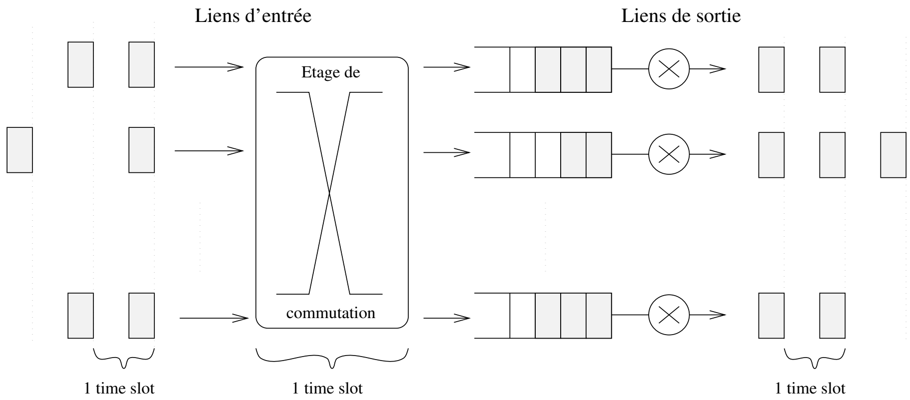


Figure 4.4 Modèle de commutateur.

Pour notre étude, nous adoptons la stratégie de commutation suivante : à chaque *time slot* une cellule arrivant sur un port d'entrée est commutée sur un port de sortie. Le choix du port de sortie est effectué selon l'algorithme de routage en vigueur. Nous utiliserons deux types de routage : le premier, simple, est aléatoire uniforme et le deuxième, plus conforme à la technique ATM, utilise la translation VCI/VPI.

Une seule cellule peut arriver sur un port d'entrée par *time slot* mais plusieurs cellules peuvent être commutées sur le même port de sortie au même instant (on a donc $t_c = 1$ time slot). Dans ce cas, elles attendent dans les tampons de sortie. Parallèlement, les cellules en tête des tampons de sortie sont transmises sur le lien de sortie à chaque *time slot* en suivant une politique *First In First Out* (on a donc $t_t = 1$ time slot). Au niveau de la simulation, l'arrivée d'une cellule sur un port d'entrée se traduit par la réception d'un événement $< rec, t_r >$. Le *pl* associé au commutateur programme ensuite pour lui-même un événement $< env, t_d >$ où t_d est l'instant de départ de la cellule du port de sortie. Un événement $< rec, t_a >$ sera ensuite envoyé au *pl* destinataire avec $t_a = t_d + t_t + delai\_lien$.

Nous verrons par la suite que si l'architecture interne des commutateurs est sans importance, il n'en est pas de même de la stratégie d'ordonnancement (i.e. FCFS, LCFS, PS...) qui peut influencer de manière appréciable les performances du simulateur.

Les topologies de réseaux

Nous proposons d'étudier deux configurations de réseaux que nous pensons être représentatives (i) de la plupart des modèles de simulation de réseaux ATM et (ii) de la plupart des problèmes soulevés par la simulation parallèle et distribuée. Dans la première configuration, le modèle consiste en un réseau complètement maillé de N_s commutateurs ATM. Une source de cellules est connectée à chaque commutateur et génère n_c cellules dont le temps d'inter-arrivée est distribué de manière exponentielle. Les liens entre deux commutateurs sont bi-directionnels, de ce fait chaque commutateur possède N_s entrées et $N_s - 1$ sorties. Lorsque les sources ont généré toutes leurs cellules, le nombre de cellulles dans le réseau reste constant et les commutateurs continuent d'envoyer les cellules jusqu'à la fin de la simulation. Le routage des cellules est aléatoire uniforme, c'est à dire qu'une cellule en entrée est commutée sur une des sortie de manière equiprobable. Le délai des liens est fixé à 2 time slot. La figure 7.2 représente le modèle de réseau pour $N_s = 5$.

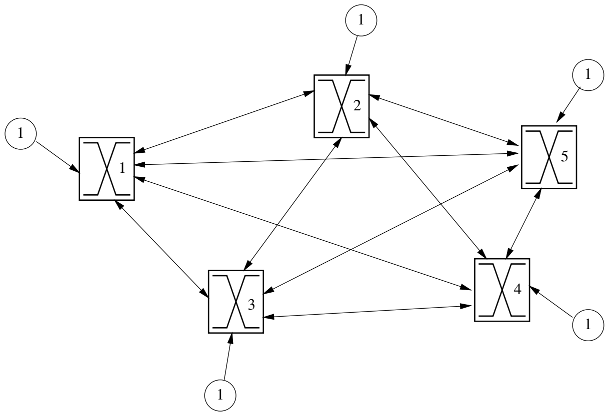


Figure 4.5 Réseau complètement maillé pour $N_s = 5$

Le choix de ce modèle n'est pas arbitraire. En effet, comme nous l'avons dit dans le chapitre 2, nous nous intéressons surtout à l'évaluation de "grands réseaux de commutateurs, tels que ceux rencontrés auprès des opérateurs publics ou des réseaux privés". Nous pensons que la grande connectivité de notre modèle est représentative des réseaux fédérateurs futurs. Cette grande connectivité a des répercussions intéressantes sur le modèle que nous simulons. Premièrement, le nombre important de messages dans le système engendre un niveau de synchronisation très élevé entre les processus, ce qui constitue un cas très "stressant" pour un protocole de simulation distribuée. Deuxièmement, et c'est favorable, le routage uniforme des messages se traduit par une répartition équitable de la charge, qui nous évite par conséquent les problèmes liés à la sous-utilisation de certains processeurs tandis que d'autres sont surchargés.

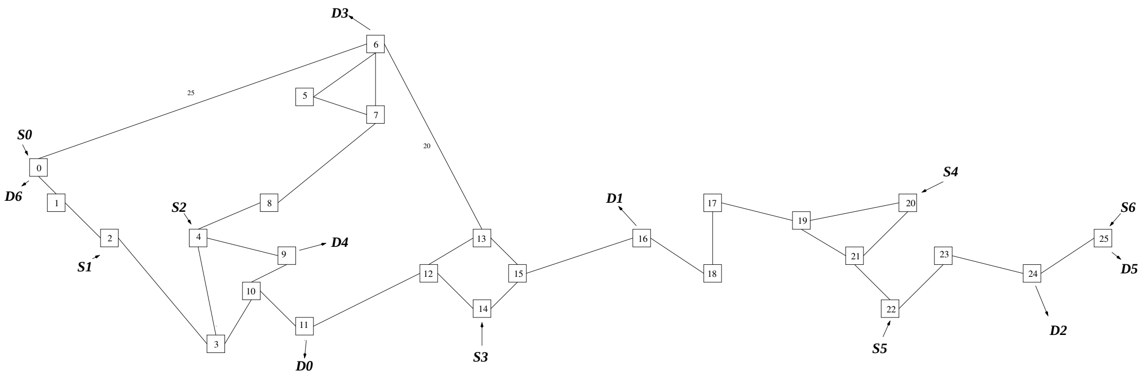


Figure 4.6 Configuration (2). Réseau de 26 commutateurs pour l'étude du routage.

Dans la seconde configuration, figure 4.6, nous proposons de simuler un algorithme de routage dans un réseau de 26 commutateurs et 7 couples (source,destination). Les mécanismes d'établissement de connexions et de translation des VCI/VPI seront également modélisés. Nous avons abordé ce problème dans le cadre d'un contrat MASI/DRET sur l'étude de la simulation distribuée de réseaux ATM, et en particulier sur l'aspect du routage avec pannes de liens dans ces réseaux. L'algorithme de routage utilisé est celui décrit dans [Jaff82] car il possède, selon les auteurs, de bonnes propriétés de reconfiguration.

La phase d'établissement se déroule de la manière suivante : chaque source de trafic qui souhaite envoyer des cellules envoie un message `setup` au commutateur auquel il est relié (ce sera souvent un multiplexeur source) pour indiquer la destination des cellules[3]. Tout au long du chemin, une nouvelle entrée est alors ajoutée pour cette nouvelle connexion dans les tables de connexion des commutateurs ATM. En particulier, cette table indique la translation VCI/VPI utilisée pour les connexions. A la réception du message `setup`, le destinataire peut envoyer en retour soit un message `connect` soit un message `reject`. Par simplicité, nous supposerons que la

[3]Nous supposons une seule et unique connexion par source de trafic.

destination renvoie toujours un message `connect` qui initie alors l'envoi des cellules par la source.

Nous allons ensuite supposer vouloir simuler sur ce réseau un algorithme ABR (Available Bit Rate) pour illustrer les problèmes qui se posent lorsque l'on désire simuler un réel algorithme de contrôle de congestion. Dans ABR, le message `setup` envoyé par la source contient une information supplémentaire : le débit crête (*Peak Cell Rate*) de la connexion. Cette information est sauvegardée dans les tables du commutateur relié à la source pour réaliser un contrôle de flux par connexion. Ce contrôle fonctionne de la manière suivante : toutes les NRM cellules envoyées pour une connexion (NRM=32 par exemple), une cellule *Resource Manager* (RM) avec un champ *Explicit Rate* (ER) est envoyé vers la destination dans le but de collecter la bande passante disponible sur le chemin. Le message RM de retour contient alors un champ ER mis à jour qui indique un nouvel *Allowed Cell Rate* (ACR) pour la connexion. L'évaluation de performance de ce problème consiste à tracer l'évolution des ACR pour vérifier l'équité et la justesse du contrôle ABR.

4.3 NIVEAU DE PARALLÉLISME ET SPEEDUP

La simulation distribuée a principalement pour but de réduire le temps de simulation par rapport à une exécution séquentielle. Il apparaît évident qu'une amélioration des performances ne pourra être obtenue que s'il y a suffisamment de parallélisme dans le système à simuler. Par conséquent, les cas pathologiques où le comportement du système est extrêmement séquentiel ne pourront pas être parallélisés convenablement—par exemple, un réseau bouclé en tandem est un cas pathologique. Heureusement, il existe dans les réseaux de communication un niveau de réplication et de distribution spatiale élevé qui fournit un haut potentiel de parallélisme. C'est pourquoi nous pensons que le problème du manque de parallélisme dans les réseaux ATM ne sera pas souvent posé et les cas pathologiques seront soient irréalistes, soient totalement inintéressants à étudier.

Cependant, le degré de parallélisme qui existe dans le système n'implique généralement pas le même degré de parallélisme dans le modèle de simulation. Ce phénomène a été précédemment décrit dans [Wagn89] pour des modèles de réseaux de files d'attente. Par exemple, si nous supposons que dans la configuration 2 le temps de commutation du commutateur 15 est plus petit que celui des autres commutateurs, alors le système réel possède un haut degré de parallélisme. D'un autre côté, le *pl* qui simulera ce commutateur devient le goulot d'étranglement dans le modèle de simulation car il reçoit des messages provenant des sources S_0, S_2 et S_3 ce qui rend sa charge 3 fois supérieure à celle du *pl* simulant le commutateur 1 par exemple. Sous l'hypothèse générale que tous les processeurs possèdent la même puissance calcul, il est clair que cette mauvaise répartition des charges limite le speedup de la simulation. Par conséquent, le parallélisme au niveau du modèle de simulation doit être séparé du parallélisme réel existant dans le système.

Obtenir un bon speedup est une tâche difficile. Ce speedup est borné par (i) le degré de parallélisme dans le modèle de simulation et (*ii*) par la surcharge introduite par les protocoles de synchronisation. Pour améliorer le premier facteur, il est possible de partitionner de manière plus optimale la répartition des *pls* sur les processeurs physiques. Par exemple, il peut être envisageable de combiner plusieurs *pls* sur un même processeur (relativement facile) ou de partager un *pl* surchargé en plusieurs tâches (plus difficile). Dans ce chapitre, nous désirons identifier les types de surcharge introduits par les protocoles de synchronisation. Dans les sections qui suivent sont étudiés les paramètres et les mécanismes qui ont un impact sur les performances du simulateur distribué.

4.4 PROPRIÉTÉS DU POINT DE VUE DE LA SIMULATION

4.4.1 Localité, densité de messages et granularité

La localité est définie comme étant la taille du voisinage. Elle peut être différente pour chaque *pl* (comme dans la configuration 2) ou identique pour tous les *pls* (configuration 1). On peut exprimer cette localité en terme de fraction de nœuds pour la rendre relative au nombre total de processus logiques. La configuration 1 présente alors une localité de $\frac{N_s}{N_s-1}$ qui est presque maximale. Le nombre moyen de messages par *pl* est par définition la *densité de messages*. Dans la configuration 1, la densité de messages est exactement le nombre de cellules générées par une source, soit n_c dans notre cas, puisque l'on se trouve dans une configuration fermée. Finalement, la *granularité* représente le temps moyen d'exécution pour un message, c'est-à-dire le temps CPU moyen nécessaire au processeur physique pour exécuter un événement.

Ces trois paramètres se révèlent avoir un grand impact sur le comportement de l'algorithme de synchronisation. Généralement, une grande localité, associée à une densité de messages faible, dégrade les performances des approches conservatrices car un grand nombre de null-messages est alors nécessaire pour correctement propager la valeur des horloges sur les liens. Maintenant, si la densité de messages augmente, alors une méthode conservatrice sera plus susceptible de battre une approche optimiste car cette dernière sera plus sujette aux *rollbacks*.

La localité et la densité de messages dépendent toutes les deux du modèle de simulation[4]. C'est dans une moindre mesure le cas de la granularité qui dépend plus de la puissance de traitement des processeurs physiques. En général, le temps de traitement d'un événement dans un modèle de réseau ATM est relativement petit. On a alors un rapport temps de traitement/temps de communication très faible, ce qui tend à pénaliser les approches optimistes car une faible granularité favorise la propagation très rapide des fautes temporelles.

Par conséquent, un fort degré de connectivité, associé à une granularité faible fait que la configuration 1 représente un cas très difficile pour une approche optimiste,

[4]On peut postuler que dans un réseau ATM le nombre de liens est grand, ce qui donne le plus souvent une grande localité. La densité de messages sera également grande car le débit de ces réseaux sera le plus souvent élevé.

même si la taille du système est petite. Le modèle de réseau de la configuration 1 est très similaire au modèle PHOLD proposé par Fujimoto pour le test des protocoles de simulation distribuée et, en se basant sur les résultats présentés dans [Fuji90b], on peut prédire qu'un système Time Warp avec l'*aggressive cancellation* serait sujet à de multiples *rollbacks* qui seront particulièrement coûteux, spécialement dans un environnement distribué (i.e. pas à mémoire partagée). Il est alors possible d'affirmer qu'une méthode conservatrice est préférable à une méthode optimiste pour simuler le réseau de la configuration 1.

4.4.2 Lookahead

La notion de *lookahead* se réfère à la faculté pour un processus logique de prédire ce qui va se passer, ou ce qui ne va pas se passer, en se basant sur une connaissance de l'application à simuler. Si un *pl* connaît tous les événements qui peuvent se produire jusqu'à l'instant t, et s'il peut également prédire tous les événements qui seront générés jusqu'à l'instant $t + L$, alors ce *pl* possède un lookahead de L. Dans [Fuji88] l'auteur introduit la notion de *lookahead ratio* (LAR) qui est définie comme étant le rapport entre la valeur de l'incrément d'horloge sur le lookahead. Ce critère s'avère plus précis que le lookahead car ce dernier perd de sa signification lorsque l'incrément d'horloge entre deux *pls* est également grand. Par conséquent, un petit LAR est synonyme d'une grande capacité à "voir" dans le futur. De nombreuses études ont été entreprises sur l'importance du lookahead et il est bien connu qu'une certaine quantité de lookahead est nécessaire aux méthodes conservatrices pour que celles-ci fonctionnent bien alors qu'une approche optimiste n'en a pas vraiment besoin. Les discussions de cette section sont donc réservées aux méthodes conservatrices.

Plusieurs délais peuvent être pris en compte dans un commutateur : le temps de commutation t_c, le temps d'attente t_w, le temps de transmission t_t et le temps de propagation t_p. t_c et t_t ne sont pas très significatifs comparés à t_w et t_p car les premiers sont généralement exprimés en centaines de μs alors que les derniers s'expriment typiquement en ms. Par conséquent, la quantité de lookahead repose principalement sur t_w et t_p. Si t_p peut être considéré comme constant, ce n'est pas le cas pour t_w qui dépend directement de la politique d'ordonnancement des commutateurs.

Les études menées sur la simulation distribuée de réseaux de files d'attente peuvent être directement utilisées ici pour caractériser le lookahead dans le commutateur. Par exemple, si les serveurs sont du type FCFS, un très bon lookahead peut être obtenu car l'événement indiquant une réception à la destination peut être envoyé dès l'exécution du message d'arrivée. En effet, le cas FCFS autorise le calcul à l'avance du temps d'attente d'une nouvelle cellule et tous les délais peuvent être ainsi regroupés en une seule et unique valeur de lookahead $L = t_w + t_c + t_t + t_p$. Maintenant, si des priorités et/ou des préemptions sont instaurées, la quantité de lookahead baisse dramatiquement. Les priorités et la préemption empêchent le calcul à l'avance du temps d'attente, et par conséquent l'événement indiquant la réception à la destination ne pourra être envoyé que lorsque le *pl* aura suffisamment avancé dans le temps, i.e. jusqu'à l'instant du début de service pour la priorité et jusqu'à l'instant de fin de service pour la préemption. Dans ce cas, le lookahead sera de $L = t_c + t_t + t_p$ pour la priorité et de $L = t_t + t_p$ pour la préemption[5]. Il est possible d'augmenter la quantité de lookahead avec des politiques d'ordonnancement autres que FCFS mais cela reste limité aux architectures à mémoire partagée et à la génération à l'avance (*pre-sampling*) des temps de service. Cette dernière méthode est malheureusement inutile dans le cas de modèles de réseaux ATM car le temps de service est généralement considéré constant.

Dans la configuration 1, toutes les conditions sont réunies pour obtenir de bonnes facultés de prédiction (FCFS sans priorité ni préemption). Exprimé en terme de LAR, le modèle présente un fort degré de lookahead puisque l'on a $LAR = 1.0$, et cette propriété devrait avantager une approche conservatrice. Pour la configuration 2, si l'algorithme ABR est étudié, la quantité de lookahead est fortement réduite puisque la nature dynamique du contrôle de flux effectué par le multiplexeur source empêche la détermination à l'avance du temps d'attente. En effet, si l'on suppose qu'il n'y a pas de préemption, le contrôle de flux résulte en une sorte de priorité instaurée sur les différentes connexions avec le niveau de priorité changeant dynamiquement. De ce fait, il n'est pas possible d'exploiter le lookahead des serveurs de type FCFS. Cependant, le lookahead représenté par les délais des liens est généralement nettement plus important que celui représenté par le temps de transmission et le temps de commutation. Cela veut dire que l'on aura souvent $L = t_p$. Si t_p est grand, i.e. les liens sont longs, la quantité de lookahead demeurera bonne même si on perd le lookahead représenté par t_w. Par contre si t_p est petit, i.e. les liens sont courts, le lookahead est fortement réduit. Dans des topologies réelles, il y aura une grande hétérogénéité dans les longueurs des liens. Il est fort probable dans ce cas que le simulateur soit pénalisé par le faible lookahead des liens courts.

4.4.3 Boucles et null-messages

Les boucles ne sont pas les bienvenues dans les approches conservatrices car elles engendrent souvent une augmentation du trafic de null-messages. Dans les réseaux de communication, ces boucles sont cependant une des composantes essentielles puisqu'elles permettent la connectivité et le transfert bi-directionnel des données. Les liens bi-directionnels entre deux nœuds de nos modèles de réseaux sont représentés dans un souci de simplicité par un lien unique comportant une flèche dans les deux sens. De manière interne, ces liens sont supportés par deux liens uni-directionnels, un dans chaque sens. Cette contrainte, nécessaire dans une approche conservatrice résulte en la création de boucles dans les deux configurations que nous étudions. Cependant, leur impact sur le simulateur n'est pas le même dans chaque cas. Dans la première configuration, les boucles sont inoffensives car toutes les entrées reçoivent suffisamment de messages réels puisque le routage des cellules s'effectue de manière uniforme. Ce n'est pas le cas dans la configuration 2 où les cellules vont

[5]En principe la préemption n'est pas supportée dans ATM.

essentiellement des sources vers les puits, en suivant les informations de routage, créant ainsi une famine en événements sur le chemin de retour. De plus, l'ajout de messages de contrôle (tels que ceux utilisés pour le routage, pour la signalisation ou pour l'algorithme ABR) crée des boucles supplémentaires d'ordre protocolaire qui peuvent être soit temporaires (cas du routage où les informations de routage ne sont envoyées que de manière périodique) soit permanentes sur toute la durée de la simulation (cas de l'algorithme ABR avec les messages RM).

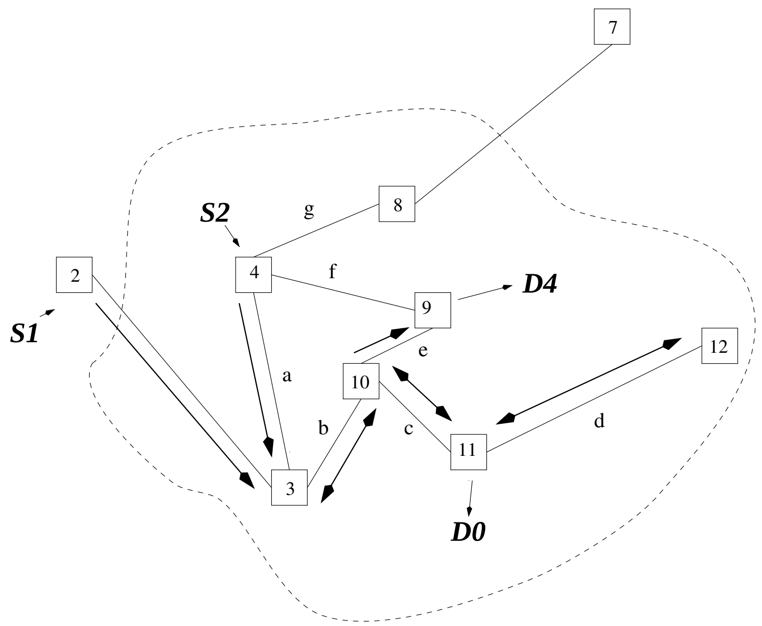


Figure 4.7 Zoom sur la configuration 2.

Dans le modèle de réseau de la figure 4.7, on peut voir que les liens b, c et d ne souffrent pas du problème de famine sur le chemin de retour car ils véhiculent des cellules dans les deux sens. Maintenant, les liens a et e ne transportent des cellules de données que dans un seul sens, c'est-à-dire des sources S_2 et S_4 aux destinations D_2 et D_4 respectivement. On peut même voir que certains liens ne transportent pas de cellule du tout : liens f et g par exemple. Dans les cas où le trafic sur le lien de retour est nul ou très faible, le simulateur va avoir besoin d'un grand nombre de null-messages pour propager l'avancement du temps de simulation sur les files d'attente associées à ces liens. Dans le cas d'un algorithme ABR, ce trafic de retour représente les messages RM qui sont retournés par le multiplexeur puits pour informer le multiplexeur source du nouveau ACR pour la connexion. Ce trafic de retour représente alors seulement $\frac{1}{32}$ du trafic en aval[6] et par conséquent, le nœud en aval doit envoyer des null-messages sur le lien de retour pour permettre l'exécution des événements au nœud le précédant. Dans le schéma classique, il est presque nécessaire pour le nœud aval d'envoyer un null-message à chaque message

[6] Rappelons qu'une cellule RM est émise toutes les 32 cellules.

reçu. Dans ce cas, le rapport message réel/null-message sur le lien de retour n'est que de $\frac{1}{32}$.

Il est alors primordial de pouvoir augmenter ce rapport pour obtenir un speedup significatif. Cela est le plus souvent réalisé en prenant en compte les particularités de l'application pour accroître la quantité de lookahead dans le simulateur. Le lookahead supplémentaire qui peut être obtenu de cette manière est payé en contrepartie par une perte de la transparence dans le noyau de simulation. En effet, prendre en compte des informations très spécifiques sur l'application implique l'ajout de morceaux de code très précis dans le noyau et sous-entend une intervention pointue de l'utilisateur. Prenons l'exemple de la configuration 2 et essayons d'accroître la quantité de lookahead. Pour cela, nous allons considérer le lien a qui connecte le commutateur 3 au commutateur source 4 dans la figure 4.7. Si nous supposons maintenant que le *pl* simulant le commutateur 3 puits[7] peut avoir la connaissance du temps d'inter-arrivée des cellules en provenance du commutateur source 4, alors il peut envoyer un null-message unique qui indiquera une approximation de l'instant auquel il devra envoyer le message RM de retour, au lieu d'en envoyer un à chaque message d'arrivée de cellule. Dans notre application, il est tout à fait possible d'obtenir cette information puisqu'un contrôle de flux est réalisé par les multiplexeurs sources avec indication du débit autorisé (ACR). Cette information peut donc être envoyée par le multiplexeur source au sein d'un message d'arrivée de cellule juste après un changement de la valeur du ACR pour la connexion. Cette information pourra ensuite être propagée tout au long du chemin.

Bien que cette solution doit être approfondie pour être réellement opérationnelle, cet exemple illustre l'utilisation d'informations spécifiques à l'application pour améliorer la prédiction au sein des *pls*. Ce n'est qu'une solution parmi d'autres qui peuvent être définies en faisant varier le niveau de complexité et le niveau de dépendance à l'application. Un compromis sera souvent nécessaire entre la complexité d'obtention et d'utilisation des informations, et le gain que l'on peut espérer.

4.4.4 Sauvegarde des états et mécanismes associés

La sauvegarde des états est un processus obligatoire dans les méthodes optimistes car le mécanisme du *rollback* requiert la récupération d'un état cohérent lorsque l'on reçoit un message fautif. Le coût de cette sauvegarde périodique peut causer une forte dégradation des performances du simulateur si la taille de l'état est grand. Par exemple, Fujimoto observe que les performances du simulateur peuvent être réduites de moitié si la taille de l'état augmente de 100 octets à 2000 octets pour des simulations de réseaux de file d'attente. Dans ce cas, il est clair qu'effectuer une sauvegarde après chaque changement d'état peut conduire à de très faibles performances. Dans [Pala93], les auteurs observent qu'en augmentant l'intervalle de sauvegarde, il est possible de réduire le temps de simulation jusqu'à obtenir une valeur optimale.

[7]Par simplicité, nous désignerons par “commutateur 3” le processus logique de la simulation qui simule l'élément commutateur 3 du système réel. Nous ferons de même pour les autres composantes.

Il est normal que la taille de l'état augmente avec la complexité du modèle. En fait, l'état du *pl* doit contenir toutes les variables qui sont susceptibles d'être modifiées lorsqu'un événement est exécuté. C'est en quelque sorte le vecteur minimum S qui caractérise le processus à n'importe quel moment. Dans la configuration 1, l'état d'un commutateur peut être défini de manière simple par $S = (o_1, .., o_j, last_1, .., last_j, n_r, n_s)$. o_j et $last_j$ sont respectivement le nombre de cellules qui attendent dans la file de sortie j et la dernière date à laquelle la sortie j a reçu une cellule. n_r et n_s sont respectivement le nombre de cellules reçues et envoyées. Les $last_j$ sont typiquement des variables introduites pour le calcul des statistiques qui doivent néanmoins être incluses dans l'état du *pl*. Cependant, même si le nombre de liens augmente, la taille de S reste relativement petite puisque l'on a seulement 2 variables par lien. Le modèle de la configuration 1 présente donc une taille d'état faible et ne pose pas de problème de sur-coût alarmant lorsque la taille du système augmente. Maintenant, pour la configuration 2, la taille de l'état des *pls* est bien plus grand puisque les informations relatives à la gestion des connexions et au routage des cellules doivent aussi y être incorporées. La taille de ces informations peut très rapidement devenir pénalisante puisque dans le cas de la configuration 2, les tables des routages à elles seules représentent plusieurs centaines d'octets. Dans ces conditions, un schéma “naïf” de sauvegarde périodique donnerait vraisemblablement de pauvres performances.

Les techniques de sauvegarde incrémentales [Stein93b] peuvent vraisemblablement être appliquées avec succès aux modèles de réseaux ATM car la sémantique des messages échangés est suffisamment faible pour ne changer qu'une petite partie de l'état du *pl*. Dans ces modèles, les variables qui sont susceptibles de changer le plus souvent sont celles qui indiquent le nombre de cellules dans les files de sortie et le nombre de cellules reçues et envoyées. Dans ce cas, la taille de la sauvegarde périodique peut être diminuée de manière assez significative. Ces techniques, de part leur complexité, ne devront être mises en œuvre que lorsque la taille de l'état pénalise lourdement le simulateur. Dans le cas de la configuration 1, il n'est sans doute pas rentable d'effectuer cette opération mais pour la configuration 2 cela est envisageable.

4.4.5 Lazy-cancellation et lazy-reevaluation

Le lazy cancellation [Gafn88] est efficace si un message fautif n'affecte pas la séquence des messages générés précédemment. Dans les modèles de réseaux de communication, la réception d'un message indique souvent l'arrivée d'un paquet d'informations qui doit être inséré dans le buffer de réception et traité. Il y a donc une forte probabilité pour que le traitement de ce paquet "en retard" ajoute un temps d'attente supplémentaire pour tous les paquets qui se trouvent derrière lui dans le buffer. Dans le modèle de simulation, cela se traduit par un décalage des estampilles des messages générés avant la réception du message fautif. Ce décalage représente le temps d'attente supplémentaire introduit par le traitement du paquet manquant. En d'autres termes, la réception d'un message fautif entraîne un retard des instants d'envoi, et par conséquent des instants de réception à la destination, des messages précédemment envoyés. Le seul moment où la séquence des messages générés ne sera pas affectée est lorsque le paquet en retard est inséré dans un buffer vide avec aucune nouvelle arrivée se produisant jusqu'à la fin de son traitement.

Tout cela nous amène à penser que le mécanisme du lazy-cancellation ne donnera pas de bonnes performances lorsqu'on l'utilise pour simuler des réseaux de communication, car non seulement les anti-messages devront finalement être envoyés, mais le fait de retarder leur envoi favorise la propagation des messages erronés. Dans les modèles ATM, cela est d'autant plus vrai que ces réseaux fonctionnent généralement à de très hauts débits, réduisant ainsi la probabilité des périodes d'inactivité. Le même raisonnement peut être tenu pour le lazy reevaluation [West88], qui se base sur le fait que l'état du *pl* ne change pas après l'exécution d'un message fautif, car la plupart des messages dans les modèles de réseaux affectent cet état (message de contrôle, message d'information de routage etc.).

4.4.6 Relaxation des contraintes de justesse

La notion de justesse demande à ce que les événements soient finalement exécutés dans l'ordre croissant des estampilles. Si ce n'est pas le cas alors le protocole est dit "non juste". Time Warp est un protocole juste car son mécanisme du *rollback* corrige les erreurs dues aux fautes temporelles. Dans le protocole MTW [Soko89], il est possible d'être non juste mais cela est seulement limité aux événements du type *Non Side-Effecting* qui n'affectent pas l'état du *pl*. Maintenant, pour être tout à fait "non juste", Theofanos a montré pour des simulations de réseaux de files d'attente que les fautes temporelles n'affectaient pas de manière significative les valeurs moyennes [Theo84]. En allant une étape plus loin, on peut penser que dans certains cas il est possible d'obtenir des résultats satisfaisants sans le respect total de l'ordre sur les estampilles établi par les approches conventionnelles. Dans ce cas, il est possible d'envisager le fonctionnement suivant : lorsqu'un message est reçu hors séquence le *pl* n'entreprend pas toujours de défaire les mauvaises actions qu'il a pu effectuer avant l'arrivée du message, mais utilise plutôt des estimations locales pour arranger les choses.

Dans les modèles de réseaux ATM, les estimations peuvent être efficaces lorsque les effets d'une réception de cellule est faible. C'est par exemple le cas dans la configuration 1 et par conséquent l'exécution de quelques événements en désordre n'affecte pas de manière significative les résultats de la simulation. Cela est d'autant plus vrai que l'on s'intéresse dans ce cas à des valeurs moyennes de performances. Deux raisons supplémentaires peuvent justifier l'utilisation des estimations locales dans ces modèles. Premièrement, les fautes temporelles qui se produisent impliquent généralement un grand nombre de *pls* du fait de la grande localité qui caractérise les réseaux ATM. Ce phénomène provoque le plus souvent de longues chaînes de *rollbacks* en cascade car un *rollback* peut en entraîner d'autres beaucoup plus coûteux. Deuxièmement, le grand nombre de messages envoyés entre les *pls* engendre implicitement une synchronisation par message très forte. Il s'ensuit une production d'un grand nombre de fautes temporelles de petite distance. Si cette synchronisation était plus faible, cela aurait produit des fautes temporelles plus occasionnelles de plus grande distance, et dans ce cas un mécanisme de *rollback* aurait été préférable, mais ce n'est pas le cas. Cependant, ces méthodes ne peuvent pas être généralisées et ne s'appliquent qu'en des cas où l'on s'intéresse à des valeurs moyennes, ce qui exclut toute étude sur les phases transitoires. Dans la configuration 2, il n'est certainement pas possible d'appliquer cette méthode car la sémantique de certains messages peut être importante et il est alors primordial de conserver un ordre strict dans l'exécution des événements.

4.4.7 Les mécanismes à fenêtre

Les algorithmes qui possèdent une fenêtre l'utilisent pour restreindre les événements qui doivent être exécutés afin de limiter la différence de valeur d'horloge entre les *pls* dans les approches optimistes. Les principaux problèmes de ces méthodes est la détermination de la taille optimale de la fenêtre d'exécution et le besoin d'une sorte de synchronisation entre les *pls* afin d'attendre ceux qui ne sont pas assez rapides. Générale à toutes ces approches, la taille de la fenêtre, si trop petite, crée une synchronisation forte entre les *pls* qui peut annihiler tout le bénéfice d'une exécution parallèle en séquentialisant en quelque sorte l'exécution des événements. Les approches à fenêtre ont été introduites pour éviter la génération de messages trop loin dans le futur et par conséquent ont essentiellement pour but de supprimer les *rollbacks* de longue distance.

Comme l'incrément temporel sera généralement faible au sein des commutateurs, du fait de la génération d'événements internes, une approche telle que Breathing Time Bucket [Stei91], où le minimum des estampilles parmi celles des messages générés détermine la borne supérieure de la fenêtre, n'est pas bien adaptée car trop peu d'événements pourront être exécutés par cycle tandis que le coût de la synchronisation deviendra grand. Maintenant, quand la taille de la fenêtre est dynamiquement déterminée par la fréquence des *rollbacks*, comme dans le protocole ATC [Mats93], le comportement d'un modèle de réseau ATM qui tend à générer un grand nombre de fautes temporelles provoquera très vraisemblablement la réduction de cette fenêtre à une taille minimale. Ce phénomène séquentialise une fois de plus la simulation. De ce fait, nous pensons que l'utilisation d'une fenêtre n'est sans doute pas la meilleure manière de simuler les modèles de réseaux ATM car la grande localité et la grande densité de messages qui les caractérisent génèrent des *rollbacks* fréquents, et de petite distance, qui ne seront de toute manière pas évités avec une taille de fenêtre raisonnable, c'est-à-dire suffisamment grande pour ne pas séquentialiser la simulation.

4.5 CONCLUSION DU CHAPITRE

Dans ce chapitre, nous avons étudié l'application des techniques de simulation distribuée pour simuler des modèles de réseaux ATM. Il n'y a pas de résultats numériques mais plutôt une réflexion générale sur le comportement de ces modèles du point de vue de la simulation. Nous trouvons en général que les modèles de réseaux ATM présentent une grande localité et un haut degré de synchronisation par envoi de messages. Combinés à une granularité la plus souvent faible, ces modèles ont tendance à produire un grand nombre de fautes temporelles de petite distance avec une approche optimiste. De plus, les optimisations telles que le lazy-cancellation et le lazy-reevaluation sont bien souvent inutiles et inefficaces avec ces modèles de réseaux. Des précautions doivent également être prises pour gérer efficacement la sauvegarde des états dans les approches optimistes car la plupart des modèles réalistes de réseaux ATM impliquent une taille d'état relativement grande. Maintenant, avec une approche conservatrice, la quantité de lookahead dans ces modèles de réseaux sera le plus souvent réduite aux délais des liens lorsque l'application induit un ordonnancement non FCFS (les algorithmes de contrôle de flux par exemple). Cependant cette quantité de *lookahead*, même réduite aux délais des liens, est bonne car ceux-ci sont nettement plus importants que les délais représentés par les temps de commutation. Le principal problème avec une méthode conservatrice est la présence d'un grand nombre de boucles dans ces modèles. Des schémas de routage réalistes et le mode connecté peuvent introduire un problème de répartition des messages entre un chemin et son homologue dans l'autre direction. Ce phénomène provoque une augmentation du trafic de null-message qui pénalise le simulateur. Ce problème peut être amoindri dans une certaine mesure mais nécessite souvent de prendre en compte de manière détaillée les particularités de l'application et de les ajouter dans le code du simulateur.

Cependant notre sentiment général penche plus vers l'utilisation de méthodes conservatrices, ou du moins à écarter les méthodes optimistes pures, pour la simulation des modèles de réseaux ATM. La forte synchronisation au niveau message et les conséquences qui en découle pour une approche optimiste nous incite à penser qu'il vaut mieux "prévenir que guérir". Dans les chapitres qui suivent, nous allons étudier l'utilisation d'une méthode conservatrice puis l'utilisation d'une méthode optimiste qui relâche certaines contraintes de synchronisation pour réduire le coût des fautes temporelles.

Chapitre 5

Simulation conservatrice d'un réseau ATM : Expérimentations

Après la partie théorique, nous abordons maintenant la partie expérimentale. Dans ce chapitre, nous allons décrire notre implémentation en C d'un noyau de simulateur conservateur pour simuler les modèles de réseaux ATM décrits dans le chapitre précédent. Le sous-système de communication inter-processus utilise l'outil PVM (*Parallel Virtual Machine*) pour réaliser les envois de messages d'un processus à un autre. Les expérimentations sont réalisées sur une machine CM-5 du *Centre National de Calcul Parallèle en Sciences de la Terre* (CNCPST) et un Cray T3E de l'*Institut du Développement et de Ressources en Informatique Scientifique* (IDRIS).

Ce chapitre aborde les points suivants :

- une description des machines parallèles utilisées et de la librairie de communication inter-processus PVM,
- une présentation générale du noyau de simulation,
- les gains obtenus pour un modèle de réseau simple (réseau entièrement maillé et routage uniforme),
- les gains obtenus pour un modèle complexe (topologie générale et routage réel),
- et les gains obtenus pour un modèle complexe de grande dimension.

5.1 LES MACHINES PARALLÈLES UTILISÉES

5.1.1 La Connection Machine CM-5

La Connection Machine CM-5 du CNCPST est une machine parallèle à mémoire distribuée (virtuellement commune, mais physiquement distribuée) acceptant les modes de programmation SIMD et MIMD. Ce modèle comporte 128 nœuds. Chaque nœud comprend un processeur SPARC cadencé à 32Mhz, quatre unités vectorielles et 32 Mo de mémoire. La puissance de crête des processeurs associés aux unités vectorielles est de 128 MFLOPS. La CM-5 comporte également des processeurs de contrôle qui chargent les codes sur les différents nœuds et effectuent les opérations séquentielles.

Les nœuds de la CM-5 sont connectés à trois réseaux d'interconnexion différents. Le premier, appelé réseau de données, assure l'échange des données entre les composants du système. La topologie de ce réseau est un "arbre gras" (*fat tree*) dont chaque nœud a 4 fils. Les feuilles de l'arbre sont les processeurs eux-mêmes, alors que les nœuds intermédiaires sont des éléments de communication. La figure 5.1 représente un tel réseau pour une partition de 16 nœuds. Par nature, le réseau est

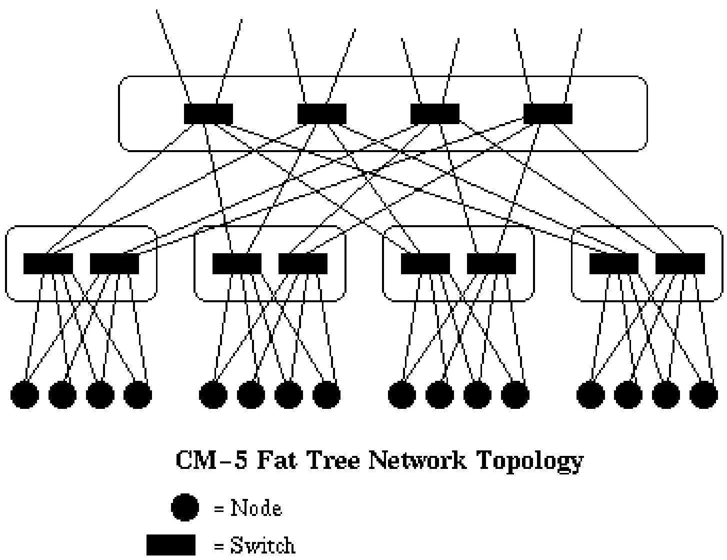


Figure 5.1 Réseau d'interconnexion des nœuds sur la CM-5. Partition de 16 nœuds.

redondant et tolérant aux pannes. Il possède une latence de 3 à 5 μs et garantit une bande passante de 5Mo/s pour chaque processeur, quelque soit l'endroit dans le réseau où les données sont envoyées. Maintenant, si les données sont envoyées dans le même groupe de 16 processeurs la bande passante est de 10Mo/s, et de 20Mo/s dans le même groupe de 4 processeurs. Le réseau de contrôle accélère les opérations coopératives telles que la diffusion et les opérations de réduction. Le troisième et dernier réseau est le réseau de diagnostic qui identifie les erreurs qui surviennent dans le système.

Les 128 nœuds de la CM-5 du CNCPST sont divisés en 3 partitions : deux de 32 et une de 64. Chaque partition dispose d'une partie du réseau qui lui est réservée et d'un processeur de contrôle appelé également processeur frontal. Les applications peuvent être lancées de manière interactive ou par *batch*. Dans le premier cas, le nombre maximum de processeurs disponible est 32. En mode *batch*, il est possible d'utiliser les 128 processeurs à condition que les partitions soient définies en conséquence.

5.1.2 Le Cray T3E

Le Cray T3E utilisé est celui de l'IDRIS. Il comporte 256 processeurs, chacun d'entre eux possédant une puissance de crête de 600 MFlops. Chaque processeur se compose d'un processeur DEC Alpha EV5 fonctionnant à 300Mhz et d'une mémoire locale de 128 Mo. La mémoire est distribuée mais globalement adressable. La topologie du réseau est celle d'un tore 3D qui constitue une grille tri-dimensionnelle connectant les nœuds dans les dimensions X, Y et Z. La figure 5.2 présente, pour une partition de 8 nœuds, la grille tri-dimensionnelle correspondante.

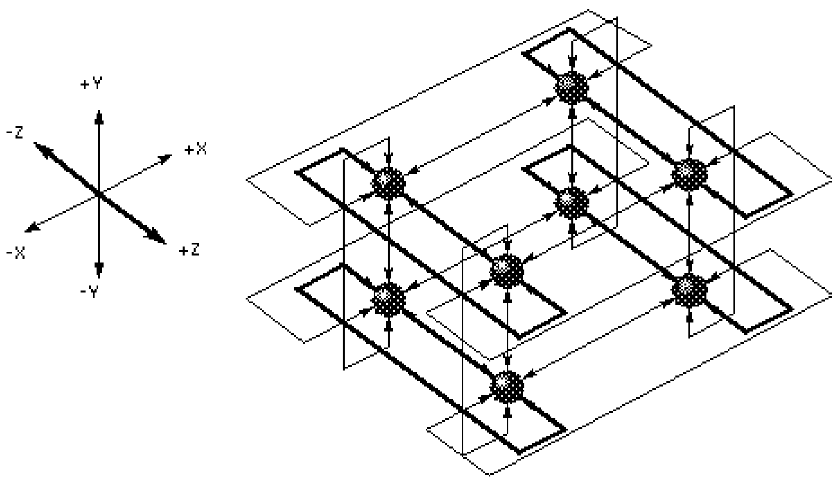


Figure 5.2 Réseau d'interconnexion des nœuds sur la T3D. Partition de 8 nœuds.

Chaque lien de communication connecte deux nœuds dans une dimension donnée. Ces liens sont constitués de deux canaux uni-directionnels par lesquels transitent les données, les informations de contrôle et les signaux d'accusés de réception. Ce réseau a une très faible latence (1 μ s) et une forte bande passante (480 Mo/s). Grâce à la technologie en tore, le réseau n'est pas bloqué en cas de liaison défaillante. De plus, des nœuds de réserve permettent de remplacer n'importe quel nœud défaillant.

Le Cray T3E ne fonctionne pas en mode "temps partagé". Des partitions dynamiques sont créées au démarrage des applications. Il n'y a pas de contraintes théoriques sur le nombre de processeurs alloués pour une application. Cependant, l'utilisation du Cray en interactif est limité à 32 processeurs. Pour utiliser plus de processeurs, il faut passer par un script et soumettre le travail en *batch*. Le parallélisme de tâches est possible grâce aux bibliothèques d'échange de messages (PVM, MPI, SHMEM) et la machine fonctionne alors en mode SPMD. Il est possible d'émuler le mode MIMD en modifiant légèrement les applications.

5.2 PVM

PVM est un outil qui a été développé au sein du projet *Heterogeneous Network Project*) avec des chercheurs d'Oak Ridge National Laboratory, du Tennessee et d'Emory [Geis93]. Ce projet a initialement pour but de permettre le calcul parallèle sur des réseaux de machines possédant des architectures et des représentations de nombres très différentes. Pour le programmeur, PVM se présente principalement sous la forme d'une librairie, pour langage C ou Fortran 77, qui offre des fonctionnalités d'échange de messages entre les différents processus. Ces processus pouvant être situés soit sur le même processeur (exécution parallèle sur un seul processeur) soit sur des processeurs différents interconnectés par un réseau quelconque de communication (machines reliées par un réseau Ethernet ou processeurs reliés par un réseau d'interconnexion rapide). Le routage des messages est transparent à l'utilisateur qui ne connaît que le *tid* des processus. Parmi l'ensemble des primitives offertes par PVM, on peut citer les primitives de gestion des messages : `pvm_send()` (envoi de messages), `pvm_recv()` (synchrone blocant) et `pvm_nrecv()` (asynchrone, non blocant). Le contenu des messages à envoyer est encodé selon un format indépendant de la machine et à chaque type de donnée (entier, réel...), correspond une fonction d'encodage qu'il faut appeler (`pvm_pkint(), pvm_pkdouble()...`). De manière similaire, la réception d'un message passe par une phase de décodage qui consiste en une suite d'appels aux primitives `pvm_upk*()`.

Outre la librairie de programmation, PVM possède deux autres composantes : un processus démon *pvmd* et une console *pvm*. Pour pouvoir lancer un programme écrit avec PVM, il est nécessaire de démarrer le processus démon *pvmd* sur chaque machine que l'on désire utiliser. Si l'on utilise la console *pvm*, celle-ci lance automatiquement les démons et permet de construire une machine virtuelle en utilisant la commande `add` suivie d'un nom de machine. Cette méthode possède l'avantage de lancer automatiquement le démon *pvmd* sur la machine distante. Il est aussi possible de préciser dans un fichier de configuration les machines que l'on désire ajouter à la machine virtuelle.

Une fois la machine virtuelle construite, les programmes peuvent utiliser la primitive `pvm_spawn()` pour lancer des tâches sur les processeurs. Trois modes d'attribution sont disponibles : le premier mode dit *transparent* place automatiquement le processus sur le site le plus approprié, le deuxième mode, dit *mode dépendant de l'architecture*, permet de spécifier sur quel type de machine le processus doit être placé, et enfin, le troisième mode dit *spécifique* indique la machine sur laquelle doit être lancé le processus.

Notre choix initial s'est porté sur PVM pour plusieurs raisons : d'une part pour sa facilité d'obtention (logiciel du domaine public) et de construction de la librairie, et d'autre part pour la simplicité de son interface de programmation qui rend sa prise de contact très rapide.

PVM sur CM-5

La version domaine public de PVM peut être utilisée sur une machine CM-5 mais elle affiche de très mauvaises performances. CM/PVM est une version adaptée et optimisée pour les machines CM-5 qui utilise la librairie de bas niveau CMAML de la machine au lieu de la librairie de haut niveau CMMD [Hopp95]. De ce fait, les performances affichées par CM/PVM sont bien supérieures à la version PVM de base.

CM/PVM est compatible avec PVM. Un programme écrit avec PVM devrait pouvoir être compilé et lié sans aucun problème avec CM/PVM. Il est également possible à des applications PVM d'interagir avec des applications CM/PVM. Cependant, quelques contraintes au niveau de la gestion des processus sur la CM-5 oblige parfois l'utilisateur qui désire avoir un seul code source à modifier légèrement la phase de lancement des processus pour satisfaire aux exigences de la CM-5.

La principale différence entre PVM et CM/PVM provient de la gestion des processus. Chaque appel à la primitive `pvm_spawn()` crée un nouveau processus, que ce soit avec PVM ou CM/PVM, et il est possible de préciser en paramètre le nombre de copies du programme à lancer. Pour PVM, exécuter plusieurs `pvm_spawn()` ou un seul avec plusieurs copies d'un même programme revient au même. Pour CM/PVM, un processus peut contenir plusieurs tâches qui s'exécuteront sur les processeurs de la partition. Il s'avère que les communications entre tâches sur la CM-5 sont bien plus rapides que celles effectuées entre processus. Par conséquent, les performances ne sont pas les mêmes lorsque l'on utilise un `pvm_spawn()` unique qui crée un processus unique composé de plusieurs tâches ou des `pvm_spawn()` multiples qui créent de multiples processus comportant chacun une tâche (figure 5.3).

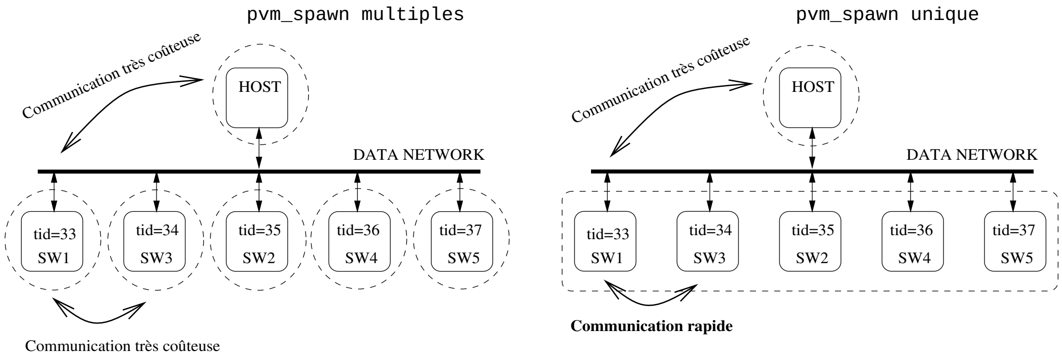


Figure 5.3 Processus et tâches sur la CM-5.

Pour obtenir des performances maximales, il est alors nécessaire sur la CM-5 de regrouper l'ensemble des processus logiques dans un seul processus CM-5, donc lancés par un seul `pvm_spawn()`. Les contraintes directes qui en découlent sont donc (i) l'impossibilité de lancer des programmes différents, les différents *pls* doivent être intégrés dans un même programme avec indication du comportement à l'initialisation (émulation du mode MIMD) (figure 5.4), et (ii) l'impossibilité pour une tâche esclave de lancer d'autres tâches esclaves si des communications intenses sont nécessaires.

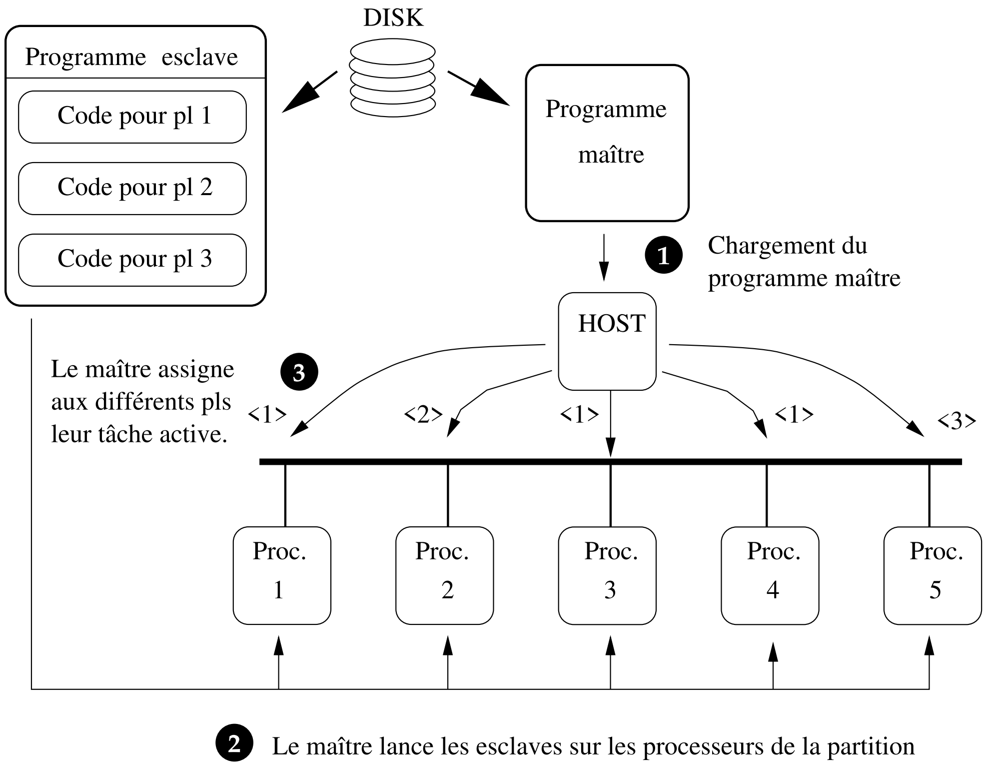


Figure 5.4 Intégration des processus et sélection du comportement actif.

La première contrainte ne nous gêne pas beaucoup car les réseaux de communication possèdent généralement des éléments identiques qui sont répliqués. Quant à la deuxième contrainte, elle se traduit pratiquement dans notre modèle par l'impossibilité pour un commutateur de lancer un processus esclave qui serait par exemple une source de trafic. Les commutateurs et les sources qui y sont connectées sont, dans notre simulateur, toujours simulés par le même processus logique.

PVM sur Cray T3E

L'utilisation de PVM sur le Cray T3E est extrêmement simple car PVM est une librairie standard sur la machine. De ce fait, il n'y a plus besoin d'avoir un véritable programme maître car les appels à `pvm_spawn` ne doivent pas être effectués au sein des programmes. La commande `mpprun` qui prend en paramètre un nombre de processeur n et un exécutable permet de lancer automatiquement l'exécutable spécifié sur une partition dynamique de n processeurs. Une fois les applications lancées, elles peuvent commencer à communiquer entre elles comme si elles avaient été lancées d'une manière plus traditionnelles.

Ce mode d'exploitation permet de fonctionner en mode SIMD et d'émuler le mode MIMD. Pour notre cas, ce n'est pas du tout gênant puisqu'il suffit simplement de pouvoir désigner un programme esclave comme un maître temporaire pour les tâches de diffusion des informations initiales.

5.3 INITIALISATION DE LA SIMULATION

Nous allons maintenant décrire brièvement le simulateur que nous avons développé pour réaliser les simulations distribuées. Seule la version pour CM-5 sera décrite car c'est la première à avoir été implémentée.

Le modèle de programmation est du type maître-esclaves. Un programme maître se charge de lancer l'ensemble des esclaves (*pl* commutateurs ATM) par un unique appel à la primitive `pvm_spawn()`. Il est important que l'appel à `pvm_spawn()` soit unique pour que l'ensemble des esclaves fasse partie du même processus afin de bénéficier de la vitesse maximale des communications sur CM-5 (figure 5.3).

Une fois lancée, chaque tâche esclave s'enregistre dans le groupe NETWORK et se voit alors attribuer un numéro d'instance dans ce groupe (étape 1, figure 5.5). La contrainte d'un appel unique à `pvm_spawn()` nous empêche de lancer les esclaves avec des paramètres en ligne différents. Les tâches esclaves se configurent donc en lisant un fichier unique de configuration et en cherchant les informations de la section correspondant à son numéro d'instance (étape 2). Ce fichier de configuration décrit la topologie du réseau et les caractéristiques de chaque composant ; un exemple est décrit ci-dessous pour $N_s = 3$ et une configuration complètement maillée. On peut modifier la topologie du réseau (commandes `IN` et `LINK`) et les différents paramètres de manière quelconque.

```
## Fichier de configuration

[[ 0                    ## Numéro de section
SELF CONST 1.0 -1 -1 20 ## Type de la source
NODE SW0                ## NODE nom
SWITCH 1 1000 3000      ## SWITCH debit nbrCell nbrTampons
LINK SW1 0 2.0          ## LINK nom sortie délai
LINK SW2 1 2.0
IN SW1                  ## IN nom
IN SW2
]]                      ## Fin de section


[[ 1                       [[ 2
SELF CONST 1.0 -1 -1 20    SELF CONST 1.0 -1 -1 20
NODE SW1                   NODE SW2
SWITCH 1 1000 3000         SWITCH 1 1000 3000
LINK SW0 0 2.0             LINK SW0 0 2.0
```

```
LINK SW2 1 2.0
IN SW0
IN SW2
]]
```

```
LINK SW1 1 2.0
IN SW0
IN SW1
]]
```

Une fois la configuration lue, les esclaves envoient vers le programme maître un couple $< tid_i, nom_i >$ (étape 3) composé de leur *tid*, obtenu en utilisant `pvm_mytid()`, et de leur nom symbolique (champ NODE du fichier de configuration, SW1 par exemple). Le maître récupère ces couples pour construire un tableau contenant tous les *tid*s et tous les noms symboliques des esclaves (étape 4). Il diffuse ensuite ce tableau à l'ensemble des esclaves qui peuvent alors récupérer les *tid*s qui leur sont nécessaires (étape 5). Désormais toutes les communications avec le maître, qui sont très coûteuses, sont inutiles et celui-ci peut s'arrêter. Précisons que pour satisfaire aux contraintes imposées par le modèle de programmation de la CM-5, les sources de trafic sont logiquement encapsulées dans les commutateurs au lieu d'être lancées par ces derniers. La figure 5.5 résume graphiquement l'ensemble des actions effectuées lors de l'initialisation. Une barrière de synchronisation à l'aide de `pvm_barrier()` est finalement placée pour s'assurer que tous les esclaves débutent leur simulation au même instant.

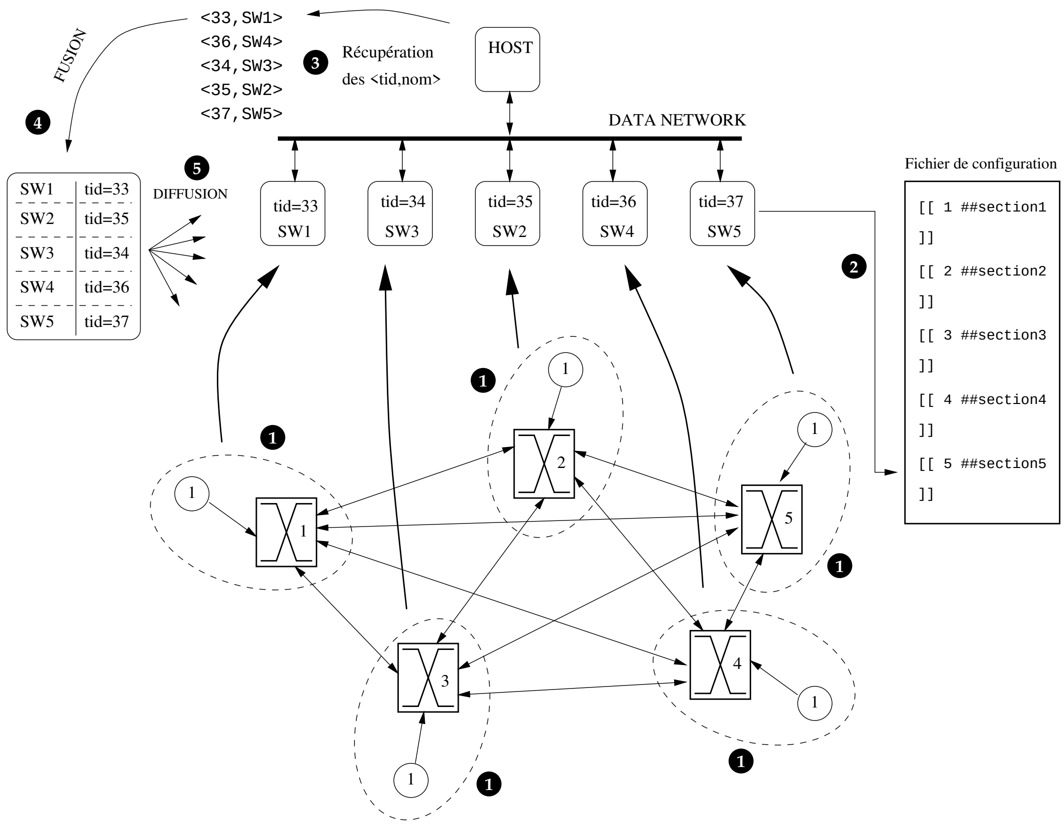


Figure 5.5 Initialisation de la simulation.

5.4 LE NOYAU DE SIMULATION

Notre moteur de simulation utilise l'approche conservatrice pour synchroniser les *pls* entre eux. Des *null-messages* sont utilisés pour éviter les situations d'inter-blocage. Chaque *pl* possède N_s files d'entrée pour recevoir les messages émanant des $N_s - 1$ autres *pls* et de leur source de trafic. La figure 5.6 illustre schématiquement la structure du moteur ; l'algorithme général est décrit ci-après.

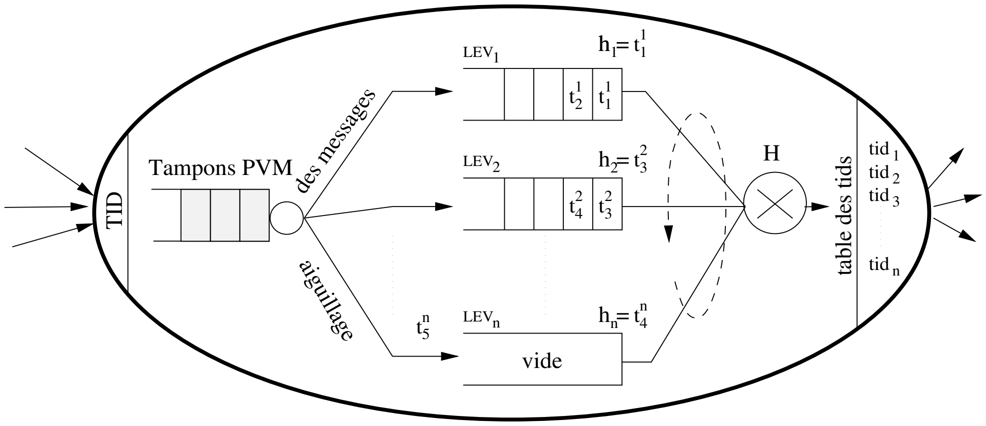


Figure 5.6 Structure du moteur de simulation.

```
REPETER
 {
   récupérer_message();
   i=1;
   REPETER
    {
      SI NON vide(i) ALORS
        SI premier_message(i)=MNULL ALORS
          SI taille(i)>1 ET estampille_premier(i)<date_dernier_null ALORS
            enlève_premier_message(i);
            i=i-1;
        SINON
          h[i]=estampille_premier(i);
          i=i+1;
    } TANTQUE i<=n;
   hmin=h[1];
   j=1;
   POUR i=2 A n FAIRE
      SI hmin>h[i] ALORS
        hmin=h[i];
        j=i;
```

```
26    SI vide(j) ALORS
27      envoyer_nullmessage();
28      attente_message(j);
29    SINON
30      executer(premier_message(j));
31
32  } JUSQUA fin_simulation();
```

La taille des messages échangés est relativement faible, inférieure à 100 octets, et ceux-ci sont envoyés en utilisant la primitive `pvm_send()`. Après chaque cycle d'exécution d'événement, la récupération des messages se fait en utilisant la primitive `pvm_nrecv()` (ligne 3) et permet à tous les messages en attente dans les tampons de PVM d'être récupérés. Ils sont au fur et à mesure insérés dans la file d'entrée correspondant au *pl* émetteur. La détermination de la bonne file se fait grâce à une table indiquant la correspondance entre le numéro de la file et le numéro d'instance du *pl* émetteur, ce dernier étant contenu dans le message. Le *pl* peut alors déterminer la file j possédant l'horloge la plus petite (ligne 21-24) en réactualisant la valeur des horloges h_i pour les files i non vides (lignes 6-17). Si cette file j est vide, des *null-messages* sont envoyés vers les autres nœuds pour leur permettre d'avancer leurs horloges (ligne 26-28). A l'aide de la primitive bloquante `pvm_recv`, on attend alors qu'un nouveau message arrive (ligne 28). L'envoi et la réception des *null-messages* ne diffèrent pas de ceux des messages réels, cependant, pour réduire leur trafic, un filtrage est effectué lors de la phase de réactualisation des horloges. Un premier filtrage consiste à ôter les *null-messages* lorsque d'autres messages (*null-messages* ou messages réels) sont présents dans la file. Le second filtrage consiste à ne retenir que les *null-messages* dont l'estampille est supérieure à celui du dernier *null-message* exécuté (ligne 10-11).

5.5 EXPÉRIMENTATION : RÉSEAU SIMPLE AVEC GRANDE CONNECTIVITÉ

Le grand nombre d'architectures supportées par PVM rend le portage d'une application très aisé d'une architecture à l'autre. En particulier, nous utilisons deux configurations pour exécuter nos simulations écrites en C. La première consiste en une SparcStation 20 monoprocesseur disposant de 64MB de mémoire centrale et la seconde en une CM-5 sur une partition de 32 processeurs. Des précautions ont en outre été prises pour s'assurer que la simulation soit bien la seule charge utilisatrice de la CM-5 lors des mesures.

Pour quantifier le gain en vitesse réalisé en distribuant la simulation sur la machine CM-5, une implantation séquentielle optimisée est obtenue en utilisant une seule file d'événements et en remplaçant le code d'envoi des messages par des insertions dans cette file. Cette version séquentielle sera exécutée sur la machine SS20. Dans le reste de cette section nous noterons par^{CM5} une simulation distribuée effectuée sur la CM-5 et seq^{SS20} la version séquentielle sur SS20. La figure 5.7 rappelle la topologie du réseau simple avec grande connectivité.

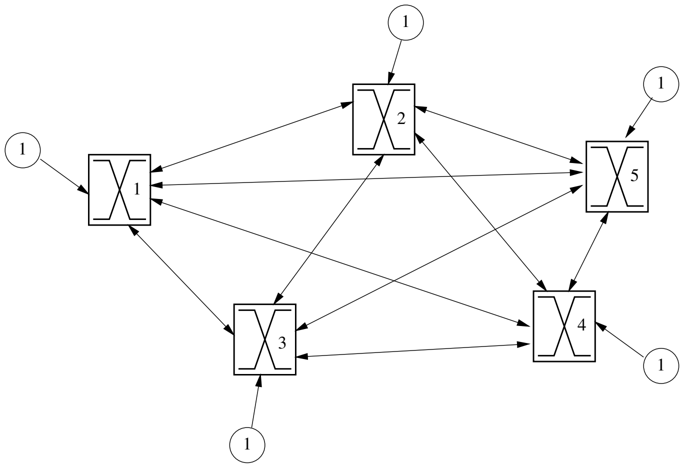


Figure 5.7 Réseau complètement maillé pour $N_s = 5$

La figure 5.8 représente le temps d'exécution, exprimé en s, de seq^{SS20} et la figure 5.9 celui de par^{CM5}. Nous avons fait varier la densité de messages et la taille du réseau. Ces temps d'exécution ont été obtenus en simulant 3.000 cellules à chaque commutateur (en effet, on constate que le temps d'exécution devient alors linéaire en fonction du nombre de cellules).

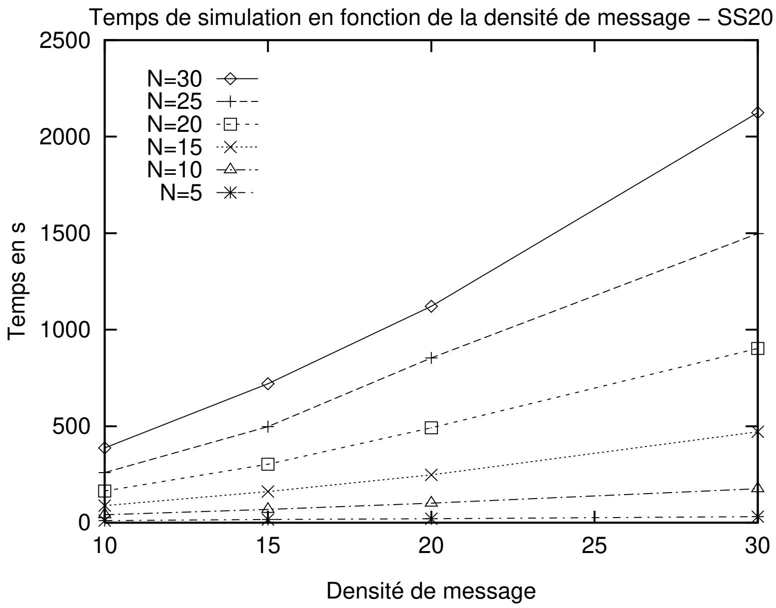


Figure 5.8 Simulation séquentielle sur SS20

Les résultats sont les suivants : au fur et à mesure que la taille du réseau et la densité de messages augmentent, le temps d'exécution de seq^{SS20} augmente très rapidement. D'un autre côté, on constate que la version parallèle demeure pratiquement

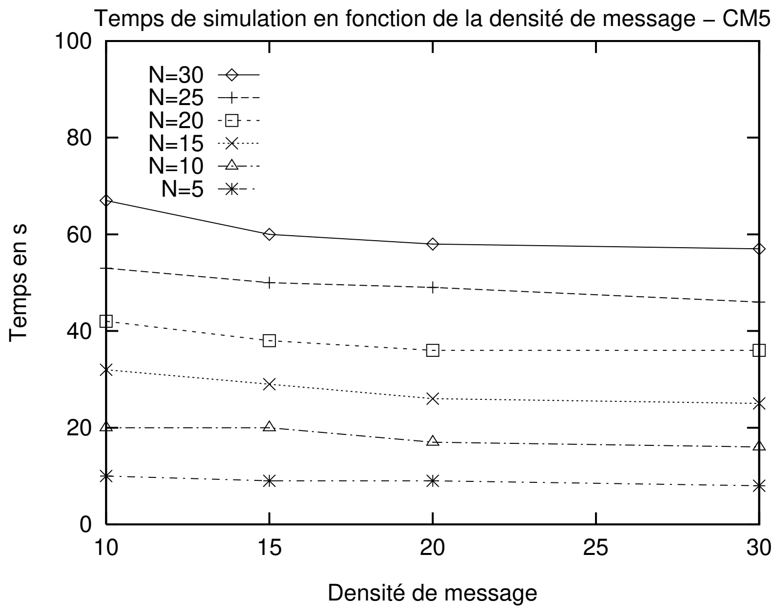


Figure 5.9 Simulation parallèle sur CM-5

insensible à l'augmentation de la densité de messages. Des tests effectués avec des densités de messages supérieures à 30 ont montré que le temps de simulation reste stable. On constate même que par^{CM5} s'avère moins performante lorsque la densité de messages est faible. Ce phénomène s'explique simplement par l'augmentation du trafic des *null-messages* nécessaires pour éviter les blocages lorsque le nombre de messages dans le système est petit. La figure 5.10 représente les gains en vitesse que l'on obtient entre seq^{SS20} et par^{CM5}.

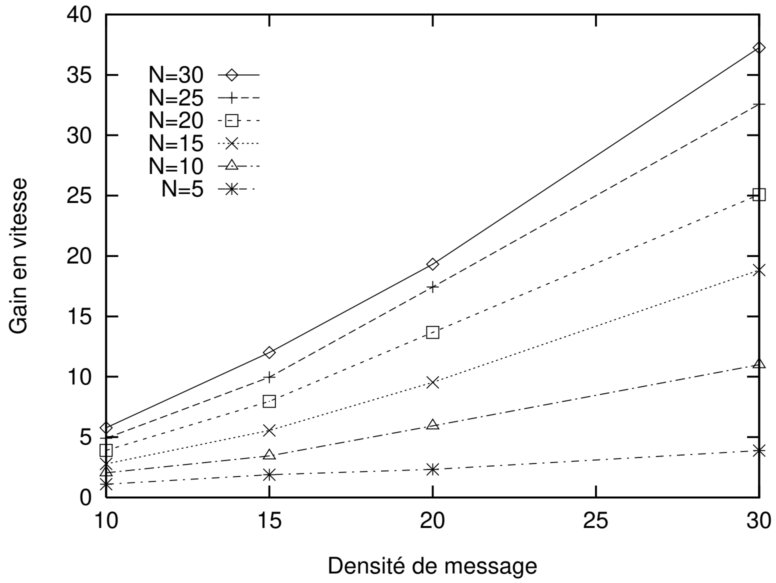


Figure 5.10 Gain en vitesse entre parallèle et séquentielle

Ces gains ne sont pas à proprement parler des speedups réels mais des gains pratiques en utilisant une machine distribuée plus puissante. Les valeurs obtenues montrent sans conteste que la distribution de la simulation sur une machine parallèle permet de réduire considérablement le temps de simulation du modèle de la configuration 1. Ces gains doivent cependant être, dans une certaine mesure, tempérés par le fait qu'il est sans doute possible d'optimiser plus facilement la version séquentielle que la version parallèle. Cependant ces premiers résultats sont très encourageants et ils démontrent l'efficacité de l'utilisation d'une approche conservatrice relativement simple pour d'une part, réduire considérablement le temps de simulation et d'autre part, augmenter la taille du système sans la contrepartie d'une dégradation majeure des performances.

5.6 EXPÉRIMENTATION : RÉSEAU COMPLEXE ET ALGORITHME DE ROUTAGE

Dans cette partie, nous allons simuler le réseau de la configuration 2 avec l'algorithme de routage. Le noyau conservateur de simulation utilisé est la seconde version du simulateur. Il est écrit en C++ et permet d'utiliser le même code source pour effectuer les simulations séquentielles et distribuées. Il a été réalisé en partie par Jobst Essmeyer, étudiant en master de l'université d'Aachen, Allemagne, durant un stage que j'ai encadré dans le laboratoire. Le simulateur fonctionne sur les machines CM-5 et Cray T3E en prenant en compte les différences relativement mineures entre les deux machines. Rappelons tout d'abord la topologie du réseau que nous souhaitons simuler.

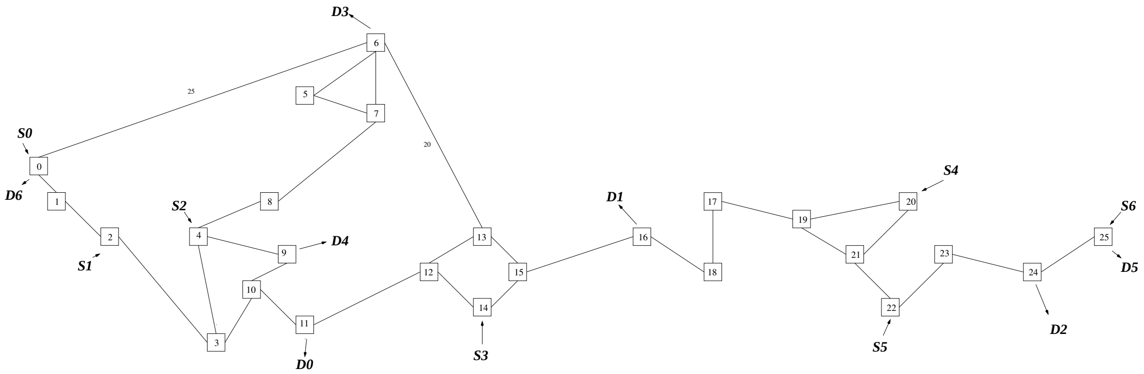


Figure 5.11 Configuration (2). Réseau de 26 commutateurs pour l'étude du routage.

Ce modèle de réseau est plus complexe que le précédent puisqu'il inclut un algorithme de routage, la gestion des établissements de connexions et la translation VPI/VCI. Des problèmes de balance de charge peuvent également se produire du fait du routage non uniforme des cellules.

5.6.1 Les algorithmes de routage

Dans l'interconnexion d'ordinateurs à travers les réseaux, il existe une fonction sans laquelle tout acheminement de l'information s'avérerait inefficace, voire impossible, c'est la fonction de routage. Cette fonction va consister, à partir d'un certain nombre de critères implicites, ou parfois explicite (donnés par le client), à calculer la "meilleur route possible" pour acheminer les paquets d'information. Sujet très vaste, nous ne donnerons ici que l'essentiel et les notions de base qui seront utiles pour notre étude. Les lecteurs qui désireront plus de détails sont invités à consulter le livre de C. Huitema [Huit95]. Nous parlerons dans les sections qui suivent du routage qui est effectué dans l'Internet puis de celui proposé dans ATM. Le routage tel qu'il est étudié dans ce travail sera ensuite présenté.

Généralités et notions de base

L'objectif des algorithmes de routage est de calculer de manière centralisée ou répartie les arcs qui composent le chemin optimal (route) entre des machines désirant communiquer entre elles. Dès lors que l'ensemble du réseau atteint une certaine taille, les méthodes centralisées ne sont plus possibles et par conséquent nous regarderons plus spécialement les approches réparties. Celles-ci peuvent être classées principalement en deux catégories selon la nature des messages échangés. Dans les approches dites "à vecteurs de distance", les nœuds s'échangent les côuts cumulés sur les différentes liaisons afin de construire leur table de routage. Dans la deuxième catégorie, dite "à état des liaisons", tous les nœuds acquièrent l'ensemble de la topologie, c'est à dire qu'ils ont tous une "carte" du réseau sous la forme d'une base de donnée, et effectuent le routage en utilisant cette base de donnée locale. Ils doivent ensuite diffuser rapidement aux autres nœuds tous les changements d'état des liaisons qu'ils auraient détectés. La tendance actuelle serait plutôt en l'utilisation de cette dernière approche car celle-ci, bien que plus complexe, permet un routage plus précis et plus fiable.

Dans l'Internet, le routage est apparu très tôt, du temps où ce réseau n'était encore que l'Arpanet. De taille relativement petite, il utilisait principalement RIP (*Routing Information Protocol*) basé sur l'approche vecteurs de distance. Il est maintenant de plus en plus délaissé au profit de l'approche à état des liaisons, représentée par OSPF (*Open Short Path First*) de l'IETF. Internet est depuis quelques années confronté a un problème de croissance du nombre de nœuds et du volume des informations échangées. De quelques milliers de réseaux et des connexions simples, Internet comprend maintenant des dizaines de milliers de réseaux qui demande de plus en plus des connexions multiples. En attendant la nouvelle version d'IP (IPv6), Internet utilise des solutions provisoires comme CIDR (Classless Internet Domaine Routing) pour gérer intelligemment l'affectation des adresses et le MBONE pour le routage multipoint.

Dans les réseaux ATM la situation n'est pas vraiment la même. Il n'existe pas encore à proprement parlé de grands réseaux ATM opérationnels et de ce fait, le routage dans ces réseaux n'est pas encore bien défini. Cependant, il est clair que les possibilités offertes par ATM donneront envie d'avoir un routage très efficace qui pourrait prendre par exemple en compte la hiérarchisation, la dynamicité, la Qualité de Service (QoS) et les connexions multipoints. La proposition de l'ATM Forum sur PNNI (Private Network to Network Interface) vise ces objectifs [ATMF96]. PNNI, basé sur l'approche à état des liaisons, devrait pouvoir supporter entre autres choses de très grands réseaux, le routage hiérarchique, la QoS, plusieurs critères de routage, le routage dynamique et les connexions anycast.

Le routage défini par Jaffe et Moss

En juillet 1995, le laboratoire MASI a passé un contrat de recherche avec la DRET. Le travail consistait à étudier l'application des techniques de simulation parallèle pour simuler des modèles de réseaux ATM. En particulier, l'intérêt de la DRET se portait sur l'étude des algorithmes de routage dans ces réseaux. Bien que le cadre d'étude de ce projet soit relativement général, les travaux que nous avons entamé devrait permettre de montrer la faisabilité d'une telle approche pour l'évaluation de réseaux complexes telle que le réseau SOCRATE (Système Opérationnel Constitué des Réseaux des Armées pour les Télécommunications) de l'armée. Ce réseau de télécommunications doit permettre de fédérer l'ensemble des réseaux des différents corps d'armées pour véhiculer des informations entres les centres Opérationnels et les unités des forces armées. Dans un tel contexte, la taille et la complexité du système limitent fortement les possibilités d'étude des approches traditionnelles (étude analytique, simulation séquentielle).

Le futur système SOCRATE de l'armée française basé sur la technologie ATM devrait comprendre à terme plus de 250 commutateurs. Le routage dans ce type de réseau sensible doit satisfaire à de nombreux critères comme la sécurité, la fiabilité et la rapidité de reconfiguration. Nous simulons l'algorithme de routage proposé par Jaffe et Moss dans [Jaff82], qui est du type vecteur de distance. Celui-ci permet selon les auteurs une reconfiguration très rapide. Les problèmes de boucles temporaires inhérentes aux approches de type vecteurs de distance sont résolus en synchronisant les envois des informations de coûts.

La figure 5.12 représente l'arbre qui est construit pour une destination D_3 donnée (cet arbre est noté $G(D_3)$). L'algorithme de routage distribué fonctionne en échangeant des informations de coût entre les commutateurs pour leur permettre de construire leurs tables de routage. A la fin de la première phase, chaque commutateur est supposé posséder une table de routage complète qui indique pour chaque destination possible le meilleur lien de sortie (ou de manière équivalente le meilleur commutateur de relais). Par conséquent, pour chaque destination possible D_i, un arbre différent $G(D_i)$ peut être construit dont la racine sera D_i et les branches représentant le meilleur lien sur lequel un nœud intermédiaire devra envoyer ses cellules pour atteindre D_i.

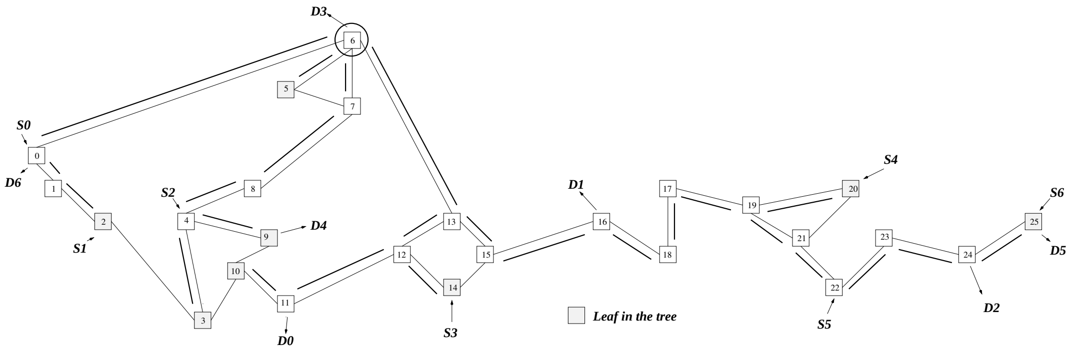


Figure 5.12 Plus court chemin pour atteindre D3.

Quand la première phase se termine, les commutateurs peuvent commencer à envoyer leurs cellules des sources vers les puits selon les informations contenues dans leur table de routage. La seconde phase d'analyse consiste alors à expérimenter des pannes de liens pour étudier les propriétés de reconfiguration de l'algorithme de routage. La figure 5.13 illustre les changements effectués dans $G(D_3)$ pour prendre en compte les deux pannes de liens.

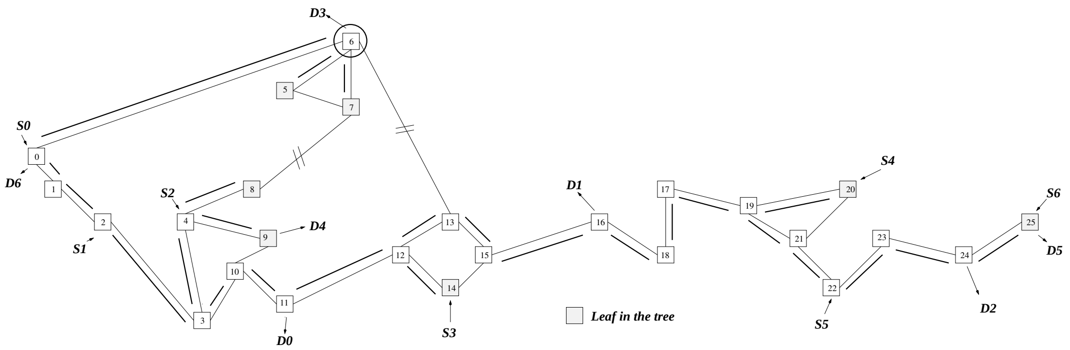


Figure 5.13 Reconfiguration du plus court chemin à D3, après pannes.

5.6.2 Résultats

Pour mesurer l'efficacité de la simulation distribuée, nous allons maintenant comparer le temps d'exécution d'une version séquentielle tournant sur 1 processeur de la machine parallèle avec celui d'une version parallèle tournant sur n processeurs (speedup en utilisant la CM-5 et le Cray T3E). Dans un premier temps, nous utilisons pour la version parallèle un simple partitionnement qui consiste à simuler chaque commutateur du réseau par un processeur de la machine parallèle (pour notre modèle, 26 nœuds sont donc utilisés). Précisons que les sources de trafic sont toujours situées dans le même processus logique que celui du commutateur auquel elles sont connectées. Les figures 5.14 et 5.15 représentent ces temps d'exécution pour une simulation de 100000 time slots (plus de 6 millions d'événements simulés et approximativement 2 millions de cellules échangées entre les commutateurs). Le

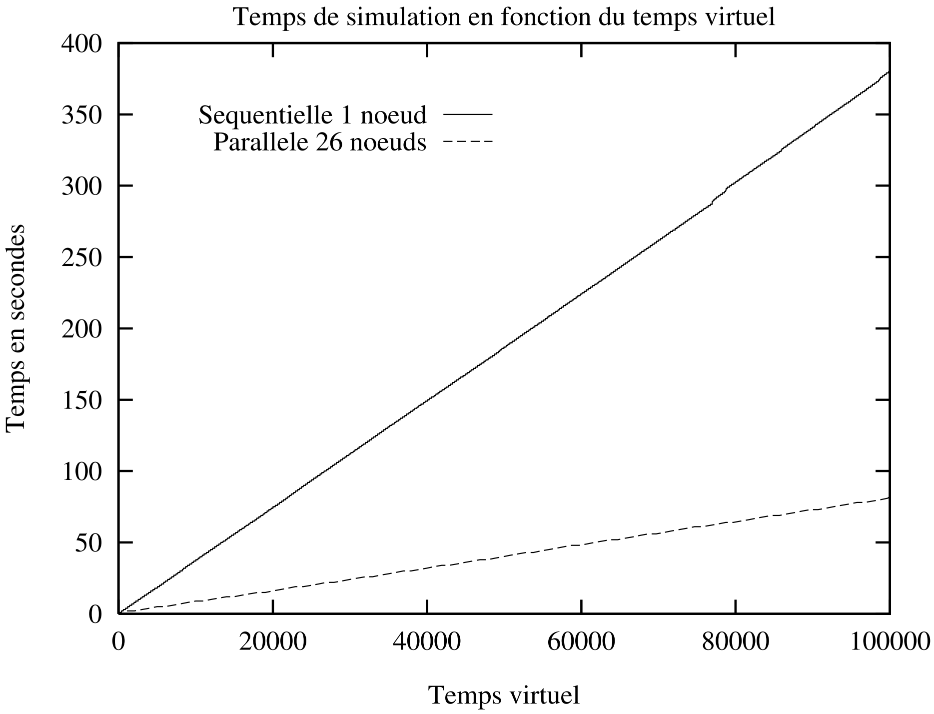


Figure 5.14 Temps d'exécution de la version séquentielle et de la version parallèle.

speedup que l'on trouve entre les deux versions est de 4.52 pour la CM-5 et de 3.3 pour le Cray T3E. Ce speedup représente une efficacité[1] de 0.26 et de 0.126, ce qui est plutôt faible. La baisse du speedup sur la machine T3E s'explique par la nette augmentation en puissance du processeur physique (DEC Alpha EV5 300Mhz et 600MFlops contre SPARC 32Mhz et 150MFlops) avec une augmentation moindre des performances du réseau de communication. C'est à dire que la version séquentielle va bien plus vite sur le Cray T3E que sur la CM-5. Par conséquent, le bénéfice de la parallélisation est moins grand.

Dans un second temps, nous assignons plus d'un commutateur sur le même processeur physique afin de réduire les coûts de communication entre les processeurs. Une simple méthode d'aggrégation est alors utilisée : nous regroupons plusieurs commutateurs adjacents et nous les simulons séquentiellement sur le même processeur. On obtient alors une simulation distribuée composée d'un certain nombre de simulations séquentielles qui interagissent en utilisant une synchronisation conservatrice. Nous pensons de cette manière pouvoir réduire le nombre de null-messages échangés

[1] L'efficacité est définie comme étant le rapport entre le speedup et le nombre de processeurs utilisés.

entre les *pls*. 3 partitionnements différents sont définis : 7, 10 et 22 processeurs. La figure 5.16 représente le partitionnement qui utilise 10 processeurs.

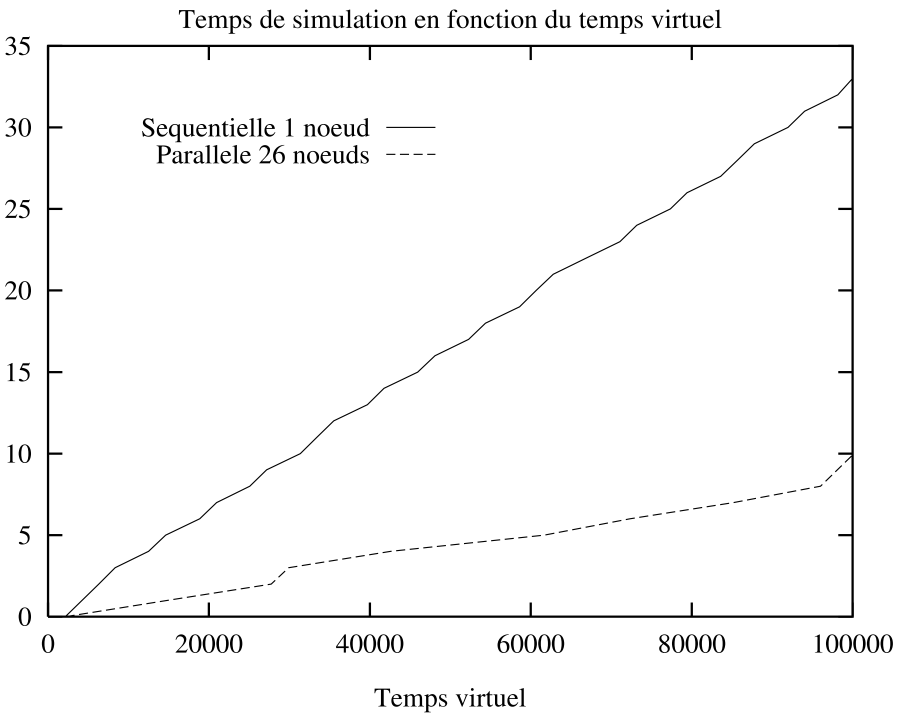


Figure 5.15 Temps d'exécution de la version séquentielle et de la version parallèle.

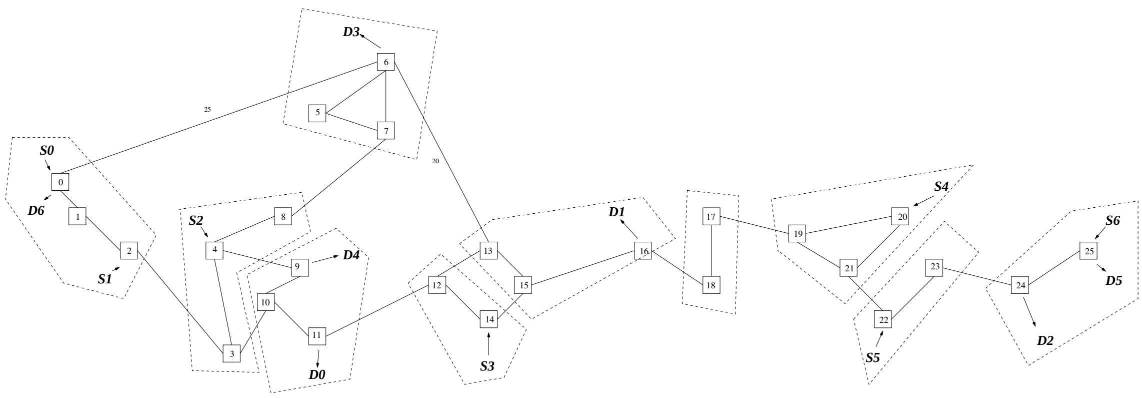


Figure 5.16 Partitionnement avec 10 processeurs.

Les résultats illustrés par les figures 5.17 et 5.18 et regroupés dans la table 5.1 montrent que malheureusement pas de meilleur speedup ne peut être obtenu avec moins de processeurs. On obtient par ailleurs le même type de courbes pour les deux machines parallèles. La raison de cet échec est relativement simple à la lumière des résultats. Prenons par exemple le *pl* qui regroupe les commutateurs 0, 1 et 2 pour illustrer notre explication. Lorsqu'on regroupe plusieurs commutateurs dans un *pl* sur un même processeur physique, le nombre de liens externes dans un *pl* reste souvent le même par rapport au cas où un *pl* ne simulerait qu'un commutateur (dans notre exemple, le nombre de liens externes est 2 dans les deux cas). Cela a deux conséquences : premièrement, le nombre de null-messages envoyés par un *pl* ne change pas de manière significative, et deuxièmement, le nombre de messages externes (messages indiquant des réceptions de cellules par exemple) reste le même car le nombre de cellules envoyées sur un lien ne dépend pas du tout du partitionnement logique.

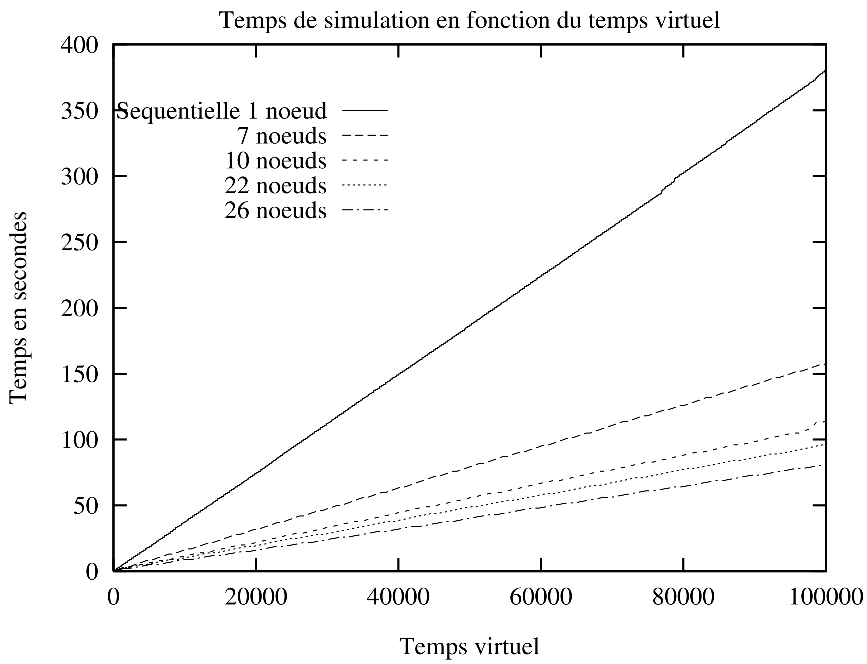


Figure 5.17 Temps d'exécution de la version parallèle en fonction du nombre de processeurs.

De ce fait, le regroupement de plusieurs commutateurs a plutôt pour conséquence d'augmenter la charge du processeur que de l'optimiser puisque celui-ci doit maintenant simuler plusieurs commutateurs au lieu d'un seul tout en gardant le même coût pour les communications externes. Ces résultats montrent que trouver un partitionnement optimal est une tâche difficile en simulation distribuée. Cela est d'autant plus vrai pour les modèles de réseaux de communication où le rapport entre le temps de traitement d'un message et son coût d'envoi (préparation des données et transmission) est très faible. Dans ce cas, regrouper plusieurs commutateurs non-adjacents dans un même *pl* ne procure pas de gain particulier puisque le niveau

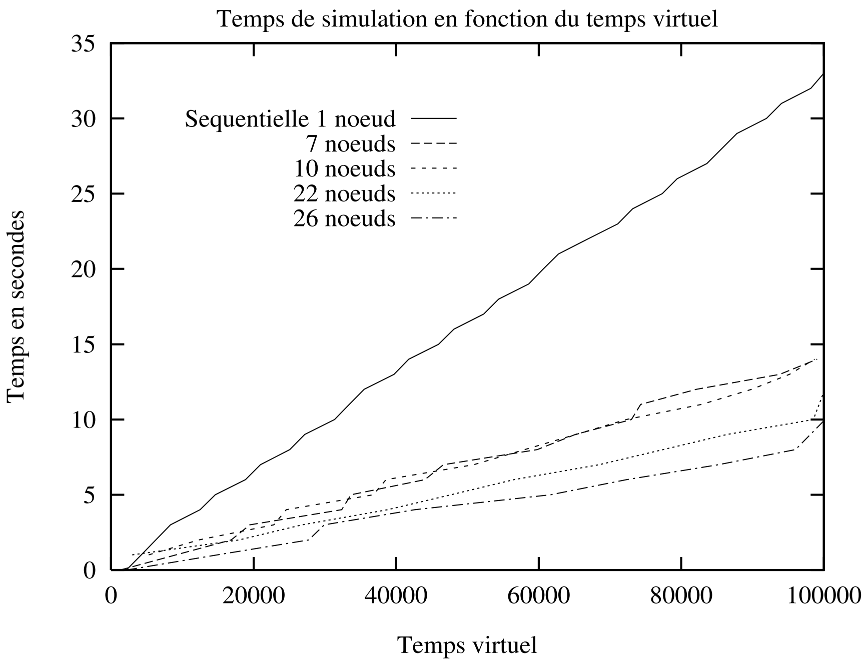


Figure 5.18 Temps d'exécution de la version parallèle en fonction du nombre de processeurs.

nproc	*temps réel*		*speedup*		*efficacité*	
	CM-5	T3E	CM-5	T3E	CM-5	T3E
1	380	33	—	—	1.0	1.00
7	158	14	2.41	2.36	0.34	0.33
10	115	14	3.30	2.36	0.33	0.23
22	97	12	3.92	2.75	0.18	0.125
26	84	10	4.52	3.3	0.26	0.126

Tableau 5.1 Temps d'exécution, speedup et efficacité en fonction de *nproc*.

de synchronisation par envoi de messages externes reste le même. De petits gains peuvent éventuellement être obtenus dans le cas de cycles tel que celui introduit par les commutateurs 5, 6 et 7 dans la figure 5.16. Dans ce cas, on peut voir que le nombre de liens externes passe de 4 (cas où le commutateur 6 est simulé séparément) à 3 (cas où les commutateurs 5, 6 et 7 sont simulés ensembles). Encore faut-il que ce regroupement n'entraîne pas une surcharge du processeur. Ce qui veut dire qu'il est difficile de regrouper de telles configurations de cycles lorsque le nombre de commutateurs impliqués est grand, même si on diminue sensiblement le nombre de liens externes. Clairement, un compromis doit être trouvé lorsque l'on regroupe les commutateurs entre la réduction des coûts de communication et l'augmentation de la charge du processeur. Il n'est pas encore certain qu'utiliser moins de processeurs que de commutateurs puisse fournir un speedup supérieur à celui fourni par un partitionnement simple 1 à 1.

Augmentation de la taille du réseau

Nous allons maintenant augmenter la taille du réseau jusqu'à 78 commutateurs afin de regarder l'évolution du speedup. Le réseau de 78 nœuds est obtenu en superposant 3 réseaux de 26 nœuds. La figure 5.19 représente succinctement un tel réseau sans préciser les numéros des commutateurs. Comme il nous est impossible pour l'instant d'utiliser plus de 32 processeurs physiques sur le Cray T3E, il est nécessaire de réaliser une aggrégation des processus logiques. Nous allons utiliser 3 partitionnements différents : 32, 16 et 8 processeurs. La figure 5.19 illustre également le partitionnement en 32 processeurs. La figure 5.20 représente les temps d'exécution que l'on

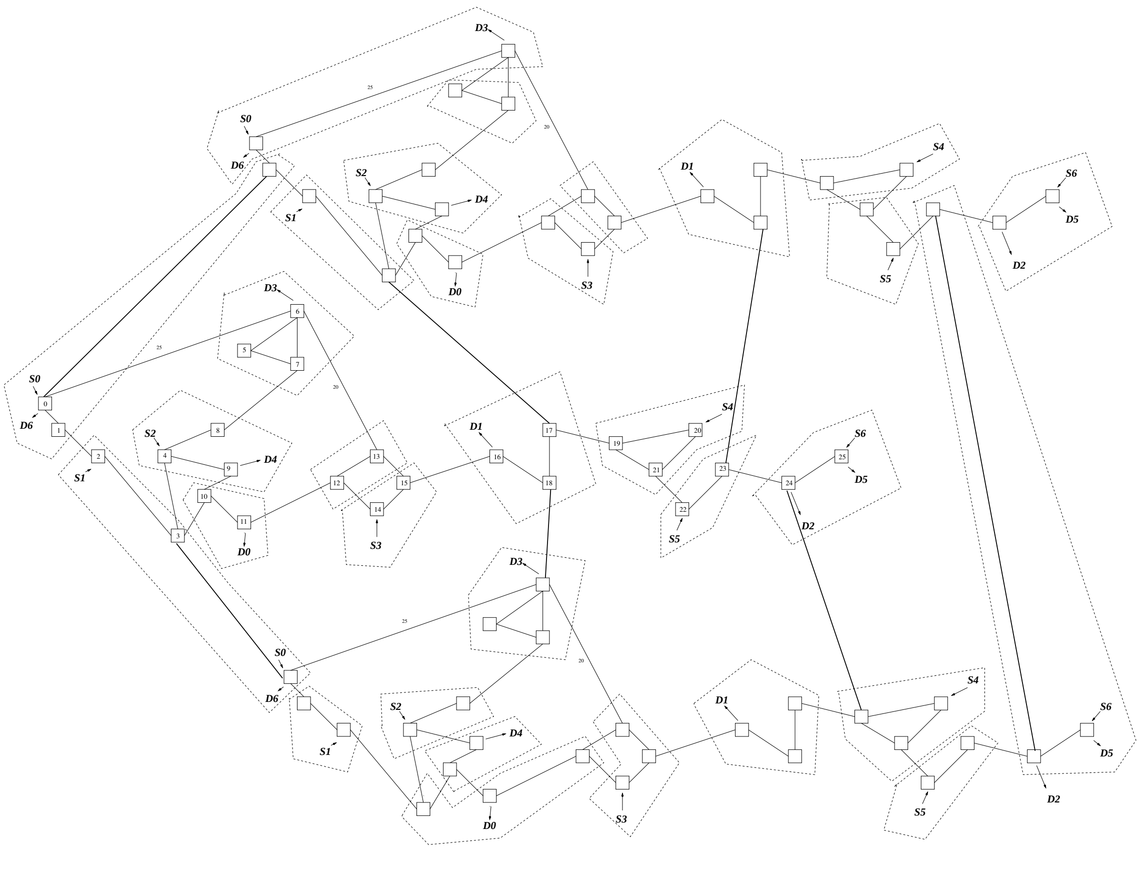


Figure 5.19 Réseau de 78 commutateurs, partitionnement en 32 processeurs.

obtient et la table 5.2 résume les valeurs de speedup que l'on obtient. Les résultats qu'on obtient sont très intéressants. Tout d'abord, on constate que :

1. le speedup maximum obtenu est maintenant de 7.68 avec 32 processeurs physiques. C'est bien plus que le 3.3 obtenu avec le modèle à 26 commutateurs.

On peut donc penser que la version distribuée de la simulation est beaucoup moins pénalisée par le facteur d'échelle.

2. la différence de speedup entre l'utilisation de 16 processeurs et de 32 processeurs est faible. Cela laisse à penser que l'aggrégation permet ici une bien meilleur utilisation des processeurs puisque les performances générales ne s'en ressentent pas. Cependant, encore une fois, le speedup décroît en fonction du nombre de processeurs physiques.

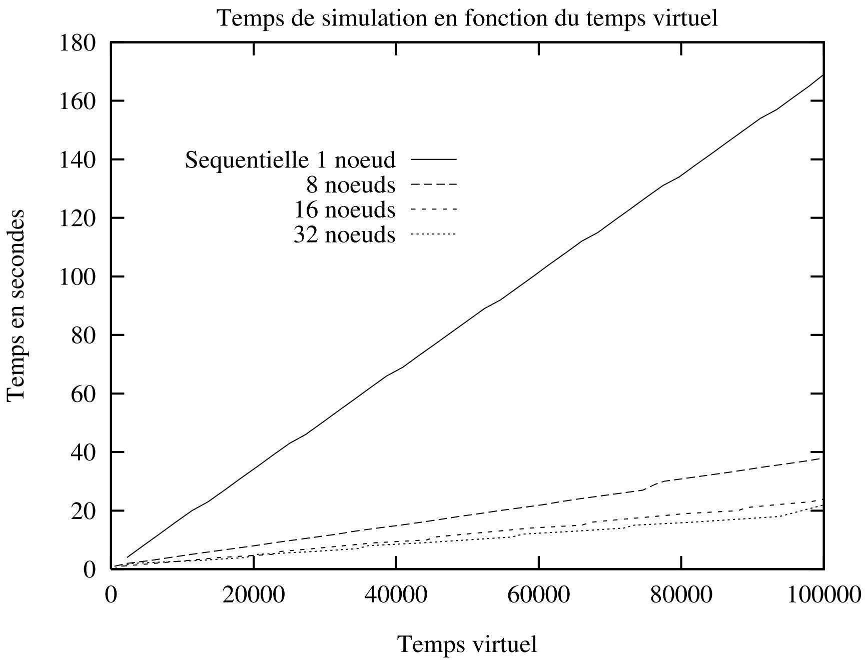


Figure 5.20 Temps d'exécution de la version parallèle en fonction du nombre de processeurs.

nproc	*temps réel*	*speedup*	*efficacité*
1	169	_	1.0
8	38	4.44	0.55
16	24	7.04	0.44
32	22	7.68	0.24

Tableau 5.2 Temps d'exécution, speedup et efficacité en fonction de *nproc*.

Ces résultats sont très encourageants car il apparaît clairement que la simulation distribuée est capable de simuler des configurations de très grandes tailles sur un nombre restreint de processeurs de manière efficaces si ces processeurs sont puissants. Le speedup qu'on obtient est alors très avantageux.

5.7 CONCLUSION DU CHAPITRE

Nous avons décrit dans ce chapitre une implémentation d'un noyau conservateur avec PVM et son application à simuler (*i*) un réseau simple avec routage uniforme et (*ii*) un réseau plus complexe comportant un algorithme de routage et des établissements de connexions. Cette première étude montre que nous pouvons déjà d'une part, accélérer la simulation et d'autre part, augmenter la taille du système sans dégradation des performances. Pour une configuration simple, les résultats de speedups sont particulièrement bon et très encourageants. Cela reste conforme aux remarques effectuées lors de la réflexion théorique précédente, à savoir que ce modèle de réseau ne comporte que des serveurs du type FCFS et qu'il présente un bon degré de lookahead. A cet égard, nous faisons remarquer que les résultats présentés concernent une implémentation qui n'exploite pas de lookahead particulièrement lié à une application donnée. Ce choix, en plus de sa simplicité, permet une meilleur transparence qui autorise l'ajout d'autres mécanismes dans les modèles sans toucher au noyau de simulation. Nous avons cependant constaté que le fait d'exploiter entièrement le lookahead dans le réseau simple complètement maillé ne donne pas de changements significatifs dans la mesure où ce modèle, de par sa structure fermée, présente suffisamment de messages réels pour des densités de messages supérieures à 10.

Pour le réseau complexe et l'étude du routage, la simulation d'un réseau à topologie réaliste avec un algorithme de routage donne un speedup de 4.52 et de 3.3, pour la CM-5 et le Cray T3E respectivement, lorsque nous assignons un commutateur à un processeur. Le speedup ne peut pas être amélioré en utilisant moins de processeurs car les coûts de communication reste sensiblement les mêmes. Lorsqu'on augmente la taille du réseau pour atteindre 78 commutateurs, on obtient un speedup de 7.68 sur le Cray T3E en utilisant 32 processeurs. Un speedup de 7.68 c'est beaucoup et ... pas assez. C'est beaucoup car cela veut dire qu'une simulation qui durerait 7 jours pourra être réalisée en un peu moins d'un jour. Maintenant, ce n'est pas assez car utiliser 32 processeurs pour un speedup de 7.68 ne représente qu'une efficacité de 0.24. Cependant, ce dernier point peut être amélioré : (*i*) nous pensons que plus les modèles seront grands et meilleur sera le speedup car la version distribuée est plus extensible et dépend seulement du nombre de processeurs disponibles[2], et (*ii*) ce speedup doit être relié au fait que nous n'avons pas tenté d'optimiser la topologie ni le degré de lookahead en utilisant des informations spécifiques aux applications. Enfin, le problème de l'aggrégation des objets de simulation dans les processus logiques reste un problème ouvert qui mérite d'être exploré.

[2] Les architectures multi-processeurs deviennent de plus en plus courantes et nul doute que l'on disposera bientôt de machines possédant un grand nombre de processeurs bon marché.

CHAPITRE 6

L'approche statistiquement correcte

6.1 INTRODUCTION

Nous avons vu que les techniques de simulation distribuée pouvaient être grossièrement partagées en deux catégories : conservatrice et optimiste. Ces méthodes offrent des mécanismes de synchronisation pour permettre une exécution correcte et ordonnée des événements. Cependant, ce souci de préserver les contraintes de causalités occasionne une lourdeur dans ces mécanismes de synchronisation. Rappelons à ce sujet que les méthodes conservatrices introduisent un blocage pour garantir qu'aucun événement ne pourra être reçu dans le passé (faute temporelle). Les méthodes optimistes autorisent quant à elles l'exécution immédiate d'un événement mais utilisent un mécanisme de retour-arrière (*rollback*) pour corriger les fautes temporelles lorsque celles-ci se produisent.

Dans la terminologie définie par Reynolds [Reyn88], un fonctionnement *correct* nécessite une exécution finale des événements dans l'ordre croissant des estampilles. Si cela n'est pas le cas, le protocole est dit *non-correct*. De nombreuses améliorations ont été proposées mais très peu d'entre elles remettent en cause ce principe de correction. Et encore, celles qui le font, ne le réalisent que dans des cas bien particuliers [Soko88,Dick90]. Les approches optimistes comme Time Warp sont correctes car le mécanisme du *rollback* offre à l'utilisateur final la vision d'une exécution ordonnée des événements.

Notre contribution pour accélérer les simulations distribuées consiste à relâcher ce paramètre de correction pour définir un protocole non-correct permettant néanmoins de fournir des résultats exploitables. Cette démarche est motivée par le constat qu'une synchronisation forte au niveau de l'ordre des messages n'est sans doute pas totalement nécessaire pour simuler certaines applications. L'idée principale consiste donc à relâcher certaines contraintes de synchronisation pour pouvoir s'affranchir, dans une certaine mesure, de la lourdeur des mécanismes de synchronisation. Dans [Theo84], l'auteur montre que dans le cas d'une application comme la simulation distribuée de réseaux de files d'attente, un certain nombre de fautes temporelles n'affecte pas de manière importante les critères moyens de performance. C'est pourquoi, en allant une étape plus loin, nous pensons que dans bien des cas des résultats tout à fait exploitables peuvent être obtenus sans le respect strict d'un ordonnancement total sur les estampilles des messages tel qu'il est établit dans les approches conventionnelles.

Ce chapitre expose donc les concepts généraux d'une approche visant à relâcher certaines contraintes de synchronisation pour accélérer la simulation. Il est organisé de la manière suivante : la section 2 présente les principes de base de l'algorithme, la section 3 détaille la procédure de calcul des estimations locales, la section 4 et 5 présentent respectivement les configurations d'étude et les résultats obtenus. La conclusion du chapitre est donnée dans la section 6.

6.2 PRINCIPES DE BASE

Pour définir l'approche statistiquement correcte, nous avons besoin de quelques définitions de base.

6.2.1 Définitions

Définition 6.1 (Entièrement-correcte) *Une simulation est dite entièrement-correcte si les événements sont exécutés par un processus logique dans l'ordre croissant de leur estampille sans fautes temporelles.*

Définition 6.2 (Correcte) *Une simulation est dite correcte si les fautes temporelles qui ont pu se produire ont toutes été corrigées par un mécanisme semblable au mécanisme du* rollback.

Définition 6.3 (Statistiquement-correcte) *Une simulation est dite statistiquement-correcte si sur l'ensemble des fautes temporelles qui ont pu se produire, certaines d'entre elles n'ont pas été corrigées par un mécanisme semblable au mécanisme du* rollback.

Définition 6.4 (Séquence sans-*rollback* et k-*rollback*) *Soit φ une séquence de n événements $< e_i, t_i >$ où t_i est l'estampille de l'événement e_i. La séquence φ est dite sans-*rollback *si $\forall i,$ on a $t_{i+1} \geq t_i$. D'un autre coté, une séquence k-*rollback *est une séquence où il existe k t_i tel que l'on ait $t_i < t_{i-1}$. Par extension, une séquence avec-*rollback *est une séquence k-*rollback *où $k \geq 1$.*

Une approche conservatrice produit typiquement des simulations entièrement-correctes alors qu'une méthode optimiste comme Time Warp produit généralement des simulations correctes.

6.2.2 Relaxation des contraintes de synchronisation

En développant notre nouvelle approche, nous voulions relâcher les contraintes de synchronisation de manière paramétrable pour pouvoir produire aussi bien des simulations correctes que des simulations statistiquement-correctes. Pour obtenir ce résultat, notre approche statistiquement correcte doit établir un contrôle qui offrirait à l'utilisateur la possibilité d'obtenir des résultats couvrant un spectre de fonctionnement allant de correct à statistiquement correct. Notons alors Φ_c et Φ_s les contrôles pour obtenir respectivement une simulation correcte et une simulation statistiquement-correcte. Pour l'instant, aucune autre hypothèse n'est formulée pour ces contrôles, nous essayons dans un premier temps de formaliser notre approche.

Nous pouvons maintenant introduire les notations TW_φ et $ST_\varphi(\Phi)$ pour désigner un *pl* effectuant une simulation qui utilise respectivement le protocole Time Warp et notre approche statistiquement correcte sous le contrôle Φ, ceci avec la séquence φ. Une simulation effectuée avec une séquence φ signifie que le *pl* exécute successivement les événements i de φ, $i = 1..n$. Pour simplifier la notation, nous écrieront $a = b$ pour indiquer que le résultat de la simulation a est identique au résultat de la simulation b. Finalement, nous pouvons écrire les relations suivantes :

1. si φ est une séquence sans-rollback alors TW_φ et $ST_\varphi(\Phi)$ sont tous deux des simulations entièrement correctes quelque soit le contrôle Φ. Nous aurons $TW_\varphi = ST_\varphi(\Phi)$.
2. si φ est une séquence avec-rollback alors TW_φ et $ST_\varphi(\Phi_c)$ sont tous deux des simulations correctes et nous aurons $TW_\varphi = ST_\varphi(\Phi_c)$. D'un autre coté, $ST_\varphi(\Phi_s)$ est une simulation statistiquement-correcte et dans la plupart des cas, nous aurons $TW_\varphi \neq ST_\varphi(\Phi_s)$.

Pour une méthode optimiste, le mécanisme du *rollback* est le seul moyen d'obtenir des simulations correctes. Par conséquent, le rôle du contrôle est de décider si oui ou non le mécanisme du *rollback* doit être utilisé lorsqu'une faute temporelle survient. Il devient évident que Φ_c est le contrôle traditionnel de Time Warp qui demande un *rollback* à chaque faute temporelle alors que Φ_s peut être basé sur un certain nombre d'heuristiques qui ne déclencherait le *rollback* qu'en des cas particuliers. Nous allons maintenant définir plus précisément un tel contrôle, et comment il doit être réalisé.

Comme une séquence sans-rollback ne génère pas de fautes temporelles, une simulation utilisant n'importe quel mécanisme de synchronisation, existant ou à venir, avec une telle séquence sera obligatoirement une simulation entièrement-correcte. La relation 1 indique qu'avec une séquence sans-rollback, l'approche statistiquement correcte produit des résultats qui sont identiques à ceux produit par une simulation Time Warp traditionnelle, cela indépendemment du contrôle. Maintenant si φ est une séquence k-rollback, un algorithme Time Warp produit des simulations correctes grâce à son mécanisme du *rollback*. D'un autre coté, nous voulons que notre approche produise des simulations statistiquement-correctes selon le contrôle Φ. Cela signifie que lorsqu'une faute temporelle se produit, le mécanisme lourd du *rollback* n'est pas systématiquement utilisé pour corriger les erreurs. On voit alors clairement la nature du contrôle qui consiste à filtrer le déclenchement du *rollback* pour éviter son utilisation systématique. Bien sûr, pour obtenir néanmoins des résultats cohérents et exploitables, des alternatives au *rollback* doivent être trouvées. Nous proposons alors d'utiliser des estimations locales et un schéma différent d'évolution du temps virtuel.

6.2.3 Estimations locales

La vie d'un processus logique consiste à recevoir et à exécuter les messages qui arrivent. La plupart du temps, l'exécution d'un message modifie l'état du processus et peut éventuellement générer un certain nombre de messages qui doivent être envoyés à d'autres processus. Lorsqu'un message m_{past} d'estampille t_{past} arrive au temps t_{now}, une approche optimiste conventionnelle comme Time Warp doit effectuer un retour-arrière par le biais du mécanisme du *rollback*. Cela est réalisé en restaurant l'état du processus avant l'instant t_{past}, à retourner jusqu'à l'instant t_{past} et à envoyer les anti-messages pour annuler les mauvais messages envoyés. A ce moment seulement les événements après t_{past} sont réexécutés. Tout cela est nécessaire car l'exécution du message fautif m_{past} peut changer l'évolution du processus, modifiant ainsi tous les états et les messages produits depuis t_{past}. Ce mécanisme élégant peut cependant diverger lorsque le nombre de fautes temporelles devient grand, créant ainsi un effondrement des performances du fait des retours en arrière incessants (*rollback-trashing*).

Il n'y a pas d'autres alternatives que le *rollback* si on veut corriger exactement les erreurs propagées par les mauvais messages. Maintenant, si on accepte d'introduire quelques erreurs en n'annulant pas tous les mauvais messages envoyés, il est possible d'utiliser un mécanisme de récupération autre que le *rollback*. En particulier, on peut utiliser des estimations locales pour corriger localement l'état du processus afin de prendre en compte le message fautif m_{past}. C'est l'idée clé de notre approche statistiquement correcte. Ces estimations doivent permettre de fournir des messages corrects de sortie, résultant de l'exécution de m_{past}, et de réévaluer les états du processus de l'instant t_{past} à l'instant t_{now}.

Nous proposons des estimations locales basées sur un historique d'informations sauvegardées. Ces informations peuvent être inclues dans l'état du processus et périodiquement sauvegardées. Lorsqu'un message fautif m_{past} arrive, l'horloge virtuelle du processus ne change pas et les informations sauvegardées avant t_{past} sont utilisées pour exécuter m_{past} et générer des messages de sortie. Une réévaluation des états antérieurs à t_{past} est effectuée pour prendre en compte les modifications occasionnées par m_{past} à l'état du processus.

La manière dont ces estimations sont calculées est cependant très dépendante de l'application, ce qui ne permet pas leur usage général. Cependant, les estimations sont susceptibles de donner de bons résultats lorsque la sémantique associée aux messages échangés est faible. Pour des applications comme celles envisagées dans ce document, c'est effectivement le cas puisque beaucoup de messages échangés ne représentent que des réceptions de cellules. Dans ce cas, l'exécution d'un certain nombre d'événements en "désordre" ne devrait pas affecter de manière significative les résultats de la simulation, cela est d'autant plus vrai lorsqu'on s'intéresse à des valeurs moyennes de performance. Il faut également préciser que l'utilisation des estimations est motivée par le fait que le comportement d'un modèle de réseau est susceptible de produire un grand nombre de fautes temporelles de petite distance. Si cela n'est pas le cas, par exemple si on a peu de fautes mais des distances plus grandes, une approche traditionnelle pourrait sûrement donner de bons résultats. La petite distance des fautes plaide aussi en la faveur des estimations puisqu'on peut ainsi espérer que les erreurs commises seront faibles.

6.2.4 Contrôles proposés

Le principal souci de l'approche statistiquement correcte est de réduire le coût du mécanisme de synchronisation tout en fournissant des résultats utiles et exploitables. L'utilisation des estimations locales possède l'avantage d'éviter le lourd mécanisme du *rollback* mais peut en contre partie se tromper et trop altérer les résultats. Une solution est alors d'établir un contrôle qui déciderait quand le processus doit faire un *rollback* et quand il peut utiliser les estimations locales.

Pour quantifier le degré de précision de notre approche, nous pouvons remarquer que lorsque les simulations sont déterministes, il est possible d'obtenir des résultats identiques d'une exécution à l'autre si la séquence des événements appliquée en entrée est la même pour chaque exécution. Dans ce cas, nous pouvons alors définir un critère simple de précision en comparant les résultats obtenus par notre approche avec ceux fournis par une simulation entièrement-correcte ou correcte obtenus respectivement avec une méthode conservatrice et optimiste prise comme référence. Si les deux résultats sont identiques, nous dirons que le maximum de précision a été atteint. De la même manière, nous pouvons aussi définir un critère de coût en prenant le contrôle traditionnel de Time Warp comme valeur maximale du coût.

Par conséquent, au point maximum de précision et de coût, e.g. $precision = cout = 1$, nous avons le contrôle Φ_c de Time Warp qui produit des simulations correctes. Pour obtenir des résultats du type "best effort", e.g. $precision \in [0,1]$, mais à un coût minimum, e.g. $cout = 0$, nous avons :

- *Le contrôle nul* noté Φ_s^{null}. Avec le contrôle nul un processus utilise toujours les estimations locales et n'utilise pas du tout le mécanisme du *rollback*.

Entre ces deux points extrêmes de fonctionnement, le contrôle doit pouvoir déterminer, dans une certaine mesure, quand l'utilisation des estimations locales est susceptible de fournir de mauvais résultats. Soit d la distance d'une faute temporelle causée par l'arrivée d'un message $< e_i, t_i >$ définie comme $d = LVT - t_i$, LVT étant la valeur de l'horloge virtuelle du processus récepteur. Quelques observations suggèrent d'utiliser cette distance comme critère de déclenchement du mécanisme du *rollback* car plus cette distance est grande et :

1. plus les estimations peuvent être fausses car trop de changements ont pu se produire,

2. plus les messages erronés envoyés vers d'autres *pls* sont nombreux,
3. et plus le coût des réévaluation d'état est susceptible de dépasser le coût d'un *rollback*.

Cela nous amène à définir le contrôle suivant :

– *Le contrôle par distance*, noté $\Phi_s^{d_0}$. Avec un contrôle par distance, un processus peut utiliser le mécanisme du *rollback* si la distance d'une faute est plus grande qu'un seuil d_0.

Notons que l'on a $\Phi_c = \Phi_s^{d_0}$ quand $d_0 = 0$. La figure 6.1 représente le continuum de précision couvert par l'approche statistiquement correcte.

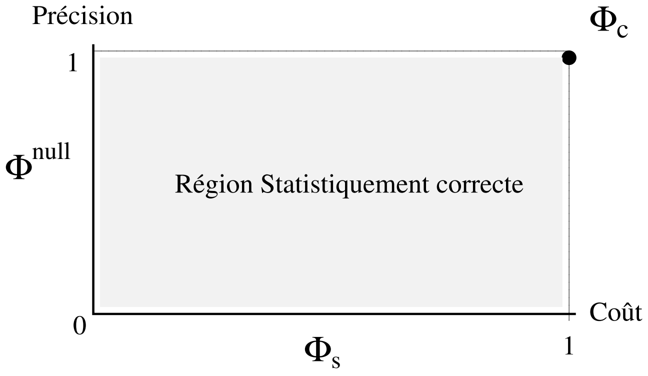


Figure 6.1 Continuum de précision selon le coût du contrôle.

Avec une séquence quelconque φ on peut obtenir un maximum de précision, mais aussi à un coût maximum, en utilisant $ST_\varphi(\Phi_c)$. Cela est équivalent à une simulation réalisée avec TimeWarp. D'un autre coté, $ST_\varphi(\Phi_s)$ offre la possibilité de réduire le temps de la simulation tout en gardant un niveau de précision que l'on espère acceptable. Les simulations se trouvant sur l'axe des y utilisent toujours le contrôle nul. Si on obtient tout de même un maximum de précision, alors la simulation est entièrement correcte, ce qui revient à dire que la simulation a été réalisée avec une séquence sans-rollback. Les simulations sur l'axe des x doivent être rejetées car elles utilisent un contrôle Φ_s qui introduit des coûts sans produire de résultats corrects. Nous pensons que ce cas ne devrait pas se produire pratiquement. Maintenant, la région grisée représente les simulations utilisant des combinaisons de contrôles Φ_s et de séquences φ. Il est évident que les cas les plus favorables se situent dans le coin supérieur gauche de la région grisée car ceux-ci offrent un maximum de précision pour un coût minimum.

6.2.5 Évolution du temps virtuel

Dans l'algorithme du *rollback*, une faute temporelle oblige le processus à retourner dans le passé (en changeant la valeur de son horloge virtuelle) jusqu'à l'instant de l'erreur pour réexécuter dans l'ordre les événements. Dans le cas où l'on utilise maintenant des estimations locales, il n'est plus nécessaire de retourner dans le passé. L'exécution d'un message fautif peut être réalisée dans ce que nous appelons un *environnement d'estimation.*

La première faute temporelle bloque l'horloge du pl au temps t_{now} jusqu'à ce qu'un message possédant une estampille t supérieure à t_{now} soit reçu. Comme le temps virtuel est gelé, les prochains messages d'estampille inférieure à t_{now} peuvent également produire des fautes temporelles. Nous appellerons ce dernier type d'erreur des erreurs *secondaires* contrairement aux erreurs *primaires.*

La figure 6.2 représente les évolutions possibles du temps virtuel pour (a) une approche conservatrice, (b) une approche optimiste et (c) l'approche statistiquement correcte. Dans ce dernier cas, un point blanc indique l'exécution d'un message passé, relativement à t_{now}, dans un environnement d'estimation. Avec le contrôle $\Phi_s^{d_0}$ la distance d d'une faute, primaire ou secondaire ne peut pas dépasser le seuil d_0 alors qu'elle n'est pas limitée avec un contrôle Φ_s^{null}.

6.3 CALCUL DES ESTIMATIONS LOCALES

Le calcul des estimations locales est dépendant de l'application et des critères de performances qu'on désire étudier. Le rôle de ces estimations est double. D'une part elles servent à produire des messages de sortie corrects et d'autre part, elles servent à la réévaluation de l'état du processus. Pour la première phase, l'idée conductrice part du constat suivant : les messages fautifs qui arrivent signalent tous l'arrivée d'une cellule dans le passé. De ce fait, les estimations devront donc déduire l'instant de départ de cette cellule si elle avait été reçue en séquence. Cela peut se faire simplement car on suppose un service FCFS. Quant à la phase de réévaluation, elle consiste à corriger les différentes variables qui constituent l'état du pl, y compris celles définies pour le calcul des critères de performances. Dans notre cas, le critère auquel nous nous intéressons est la longueur moyenne des files de sortie. Pour obtenir ce critère, nous utilisons la méthode qui consiste à calculer la somme $\Sigma S(n)$ où $S(n)$ représente le temps passé avec n cellules dans la file. Ainsi donc, à chaque arrivée et à chaque départ, un terme supplémentaire est ajouté à la somme totale. La réévaluation de l'état du processus implique alors la réévaluation de cette somme Σ. Dans les paragraphes qui suivent, nous allons détailler les actions effectuées par ces deux phases.

Algorithme

Soit L_s la liste des états sauvegardés et $S_i = (o_1^i, \dots o_k^i, last_1^i, \dots last_k^i, n_r^i, n_s^i)$ le i^{me} état sauvegardé. Si on suppose que le commutateur possède k sorties, $o_1^i, \dots o_k^i$ indiquent le nombre de cellules en attente dans la sortie k. $last_1^i, \dots last_k^i$ contiennent l'instant à laquelle une cellule a été la dernière fois commutée vers la sortie k. Finalement, n_r^i et n_s^i représentent respectivement le nombre de cellules reçues et envoyées. L'estampille de S_i est notée T_i et les index i augmentent au fur et à mesure que les T_i augmentent. Après l'arrivée d'un message fautif $< rec, t >$, $t < LVT$,

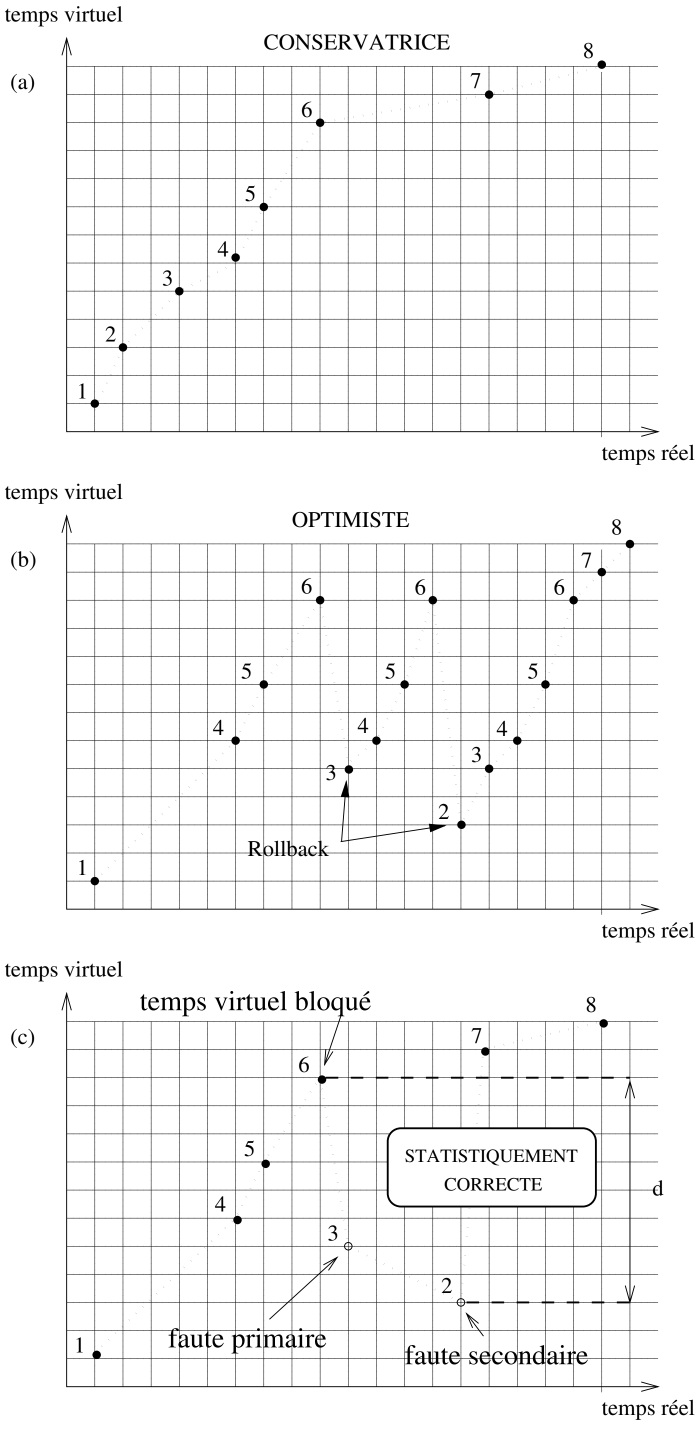


Figure 6.2 Evolution du temps virtuel pour (a) une approche conservatrice, (b) une approche optimiste et (c) l'approche statistiquement correcte.

indiquant la réception d'une cellule dans le passé, nous utilisons l'algorithme suivant pour obtenir les estimations locales effectuées par le commutateur.

1. Récupérer le premier état avant l'instant t de la liste L_s. L'état récupéré est S_i avec $T_i \leq t < T_{i+1}$.
2. Déterminer la sortie j et l'instant de départ t_d de cette nouvelle cellule en calculant son temps d'attente dans la file. Programmer ensuite pour soi-même un événement $< env, t_d >$. (Phase 1)
3. Insérer le nouvel état dans L_s avant l'état récupéré S_i (donc à l'index $i+1$; tous les index des états ont été incrémentés).
4. (a) Réévaluer tous les états S_k avec $k > i+1$. (b) Réévaluer la somme $\Sigma S(n)$. (Phase 2)

Comme les informations d'état sont sauvegardées après chaque exécution d'un événement $< rec, t >$, le premier message fautif qui arrive récupère un état qui est exactement celui qu'il aurait trouvé s'il était arrivé en séquence. Pour les messages fautifs suivants, l'état récupéré peut être un état réévalué. Nous détaillons les étapes 2 à 4 ci-après.

Détails

Nous rappelons que le modèle de commutateur suppose 1 time slot pour le temps de commutation t_c et également 1 time slot pour le temps de transmission t_t. Par souci de simplicité, nous avons utilisé ces valeurs dans les équations qui suivent. Une version littérale pourra être trouvée à la fin de ce chapitre. Maintenant, soit t_w le temps d'attente de la cellule avant le début de son traitement à la sortie j. L'instant de départ t_d de la cellule de la sortie j est calculé de la manière suivante (le temps est exprimé en time slot) :

■ Etape 2
$$\begin{cases} \text{if } t = last^i_j \text{ then } t_w = max\{o^i_j - 1, 1\} & (1) \\ \text{else } t_w = max\{o^i_j - 1 - (t - T_i), 0\}. & (2) \\ t_d = t + t_w + 1 + 1. & (3) \end{cases}$$

Remarquons que si $t = last^i_j$ nous devons avoir $t = T_i$ et $o^i_j \geq 1$. D'un autre coté, si $t \neq last^i_j$ il est possible d'avoir $t = T_i$ mais cela n'est pas obligatoire. Comme nous avons pris l'hypothèse que plusieurs cellules peuvent être commutées de plusieurs entrées sur une sortie pendant un time slot, la relation $t = T_i$ exprime le fait qu'il y a plus d'une cellule commutées vers une sortie dans le time slot. La relation $t = last^i_j$ est plus précise puisqu'elle indique qu'au moins une cellule a déjà été commutée vers la sortie j dans le même time slot. Donc si $t = last^i_j$ et $o^i_j > 1$ (eq. 1) la cellule doit attendre après toutes les autres cellules qui se trouvent devant elle dans la sortie j à l'instant t, moins celle qui aura déjà été envoyée lorsque la cellule arrivera au port de sortie (figure 6.3a). Cela est nécessaire car la transmission de la cellule en tête d'une sortie requiert également 1 time slot. Maintenant, si $o^i_j = 1$ alors la cellule doit

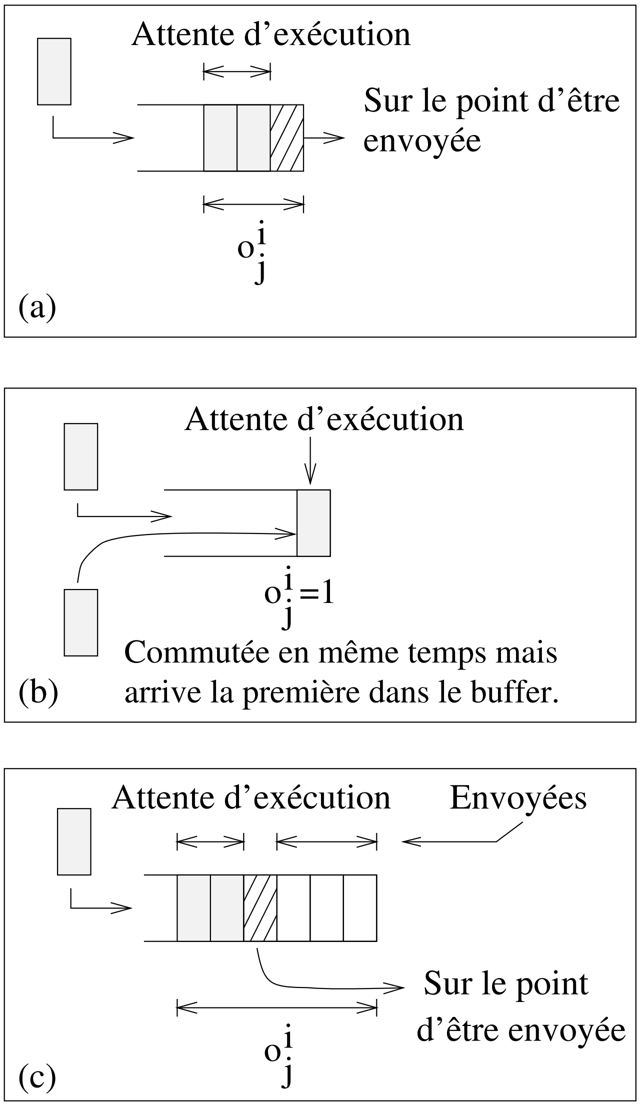


Figure 6.3 Estimation du temps d'attente d'une cellule.

seulement attendre après celle qui se trouve devant elle dans la sortie j car toutes deux ont été commutées dans le même time slot (figure 6.3b).
Si $t \neq last_j^i$ (eq. 2) alors la cellule doit attendre après toutes les autres cellules qui se trouvent devant elle dans la sortie j à l'instant T_i, moins celles qui ont déjà été envoyées durant l'intervalle $t - T_i$, et moins celle qui aura été envoyée lorsque la cellule arrivera sur le port de sortie (figure 6.3c). Bien sûr, si le port de sortie est vide, t_w est nul. Finalement, l'instant de départ t_d est facilement obtenu en additionnant à t le temps de commutation (un time slot), le temps de transmission (un time slot) et le temps d'attente t_w (eq. 3).

L'état du processus après l'exécution du message fautif $< rec, t >$ est noté S_{i+1} car il est sauvegardé à l'index $i+1$ et nous avons :

$$\blacksquare \text{ Etape 3} \begin{cases} o_j^{i+1} = max\{o_j^i - (t - T_i), 0\} + 1. \\ last_j^{i+1} = t. \\ n_r^{i+1} = n_r^i + 1. \\ n_s^{i+1} = n_s^i + min\{o_j^i, t - T_i\}. \end{cases}$$

Si $o_j^i < t - T_i$ alors toutes les cellules qui attendent dans la file de sortie j ont été envoyées durant l'intervalle $t - T_i$. Dans ce cas, nous avons $o_j^{i+1} = 1$, qui indique que la nouvelle cellule attend toute seule dans la sortie j. D'un autre coté, si $o_j^i > t - T_i$ alors il reste quelques cellules et la nouvelle doit attendre avec les autres $o_j^i - (t - T_i)$ cellules restantes. De la même manière, le nombre de cellules envoyées est incrémenté du nombre de cellules qui ont été envoyées entre t et T_i.

La réévaluation de tous les états S_k avec $k > i+1$ est réalisée de la manière suivante :

$$\blacksquare \text{ Etape 4a} \begin{cases} \text{if } T_k - t < o_j^{i+1} \text{ then } o_j^k = o_j^k + 1 \text{ else } o_j^k \text{ reste inchangé.} \\ last_j^k \text{ reste inchangé.} \\ n_r^k = n_r^k + 1. \\ \text{if } t_d < T_k \text{ then } n_s^k = n_s^k + 1 \text{ else } n_s^k \text{ reste inchangé.} \end{cases}$$

L'inégalité $T_k - t < o_j^{i+1}$ signifie que la nouvelle cellule n'a pas encore été envoyée à l'instant T_k, donc que o_j^k doit être incrémenté. D'un autre coté, si $T_k - t \geq o_j^{i+1}$ alors la nouvelle cellule a été envoyée et o_j^k reste inchangé. De la même manière, si $t_d \leq T_k$ alors la nouvelle cellule a été envoyée avant, ou à l'instant T_k, donc n_s^k doit être incrémenté. Notons que si $t_d \leq T_k$ alors nous avons $T_k - t \geq o_j^{i+1}$ car les deux conditions sont équivalentes.

La réévaluation de la somme pondérée $\Sigma S(n)$ est nécessaire pour prendre en compte dans les statistiques permettant d'obtenir la longueur moyenne des files d'attente l'arrivée d'une cellule en retard. On doit pour cela distinguer 2 cas de figure : soit la cellule est insérée alors que le buffer est vide, soit elle est insérée alors que le buffer contenait encore des cellules en attente. Dans le premier cas, il suffit de rajouter à $\Sigma S(n)$ le temps de transmission d'une cellule, c'est à dire dans notre modèle $t_t = 1$ time slot. En effet, si le buffer est vide, le terme qu'il est nécessaire d'ajouter est la durée pendant laquelle le buffer contient 1 cellule, soit 1 time slot. Maintenant dans le deuxième cas, lorsque la cellule doit être insérée dans un flot de cellules en attente, on doit reporter le temps d'attente supplémentaire des autres cellules. En fait, cela est équivalent à remarquer que lorsque la cellule est insérée dans un buffer contenant n cellules, chacune de ces cellules va attendre le temps de transmission d'une cellule supplémentaire. Au total, le buffer aura "vu"pendant le temps nécessaire à transmettre $n+1$ cellules 1 cellule supplémentaire. Pratiquement, cela se fait simplement en constatant qu'il suffit de rajouter le temps de transmission nécessaire à transmettre le nombre de cellules qui se trouvaient dans la file plus celui de la cellule supplémentaire, soit en fait $(n+1)t_t$.

■ Etape 4b $\begin{cases} \text{if } t - T_i > o_j^i \text{ then } \Sigma S(n) = \Sigma S(n) + 1 \\ \text{else} \\ \Sigma S(n) = \Sigma S(n) + \{o_j^i - (t - Ti)\} + 1 \end{cases}$

La figure 6.4 illustre un exemple où on a représenté le nombre de cellules dans le buffer en fonction du temps de simulation exprimé en time slot.

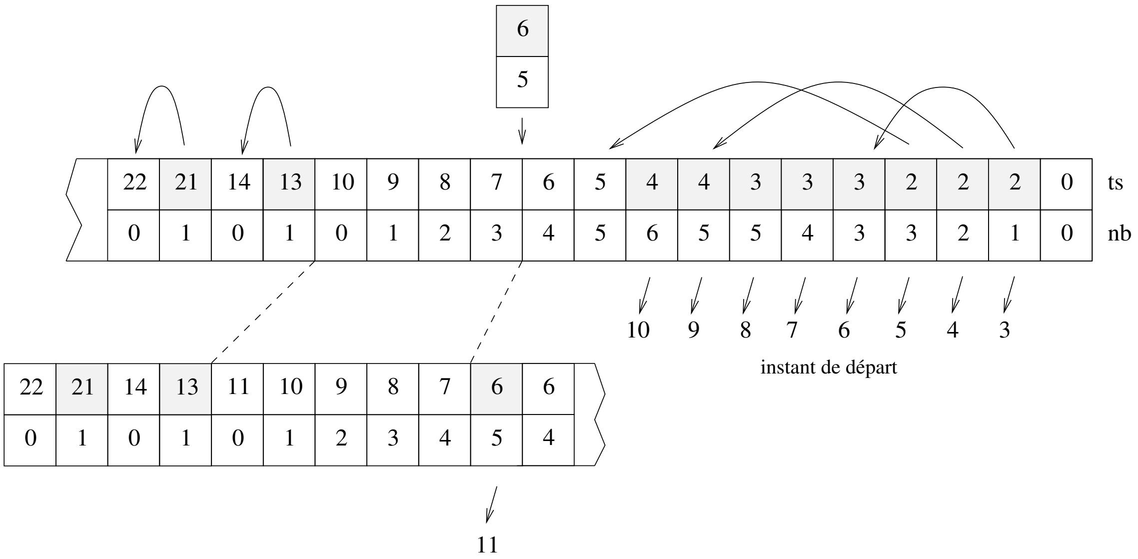


Figure 6.4 Réévaluation de la somme pondérée $\Sigma S(n)$.

Les cases grisées indique une réception de cellule. Il est possible d'avoir plusieurs réception pour un même instant de simulation. Il est également possible d'avoir un départ en même temps qu'une arrivée, auquel cas, le nombre de cellules dans le buffer reste inchangé par rapport à celui de l'instant précédent. On voit dans cette figure que l'insertion de la cellule supplémentaire décale les instants de départ des autres cellules et que cette attente représente au total 5 temps de transmission. Un calcul rapide permet de le vérifier simplement.

Avant la cellule fautive

$$\begin{aligned} \Sigma S(n) \quad = \quad & (2-0).0 + (2-2).1 + (2-2).2 + (3-2).3 + (3-3).3 \\ & +(3-3).4 + (4-3).5 + (4-4).5 + (5-4).6 + (6-5).5 \\ & + \underbrace{(7-6).4 + (8-7).3 + (9-8).2 + (10-9).1 + (13-10).0}_{changent} \\ & +(13-10).0 + (14-13).1 + (21-14).0 + (22-21).1 \end{aligned}$$

Après la cellule fautive

$$\begin{aligned}\Sigma S(n) &= \ldots + (7-6).5 + (8-7).4 + (9-8).3 + (10-9).2 \\ &\quad +(11-10).1 + (13-11).0 + \ldots\end{aligned}$$

D'où une différence de 5 time slots.

6.4 CONFIGURATIONS DE L'ÉTUDE

Pour réaliser nos tests, nous utilisons 2 configurations de réseaux. La première, configuration (a), consiste en un seul commutateur avec 3 sources de trafic (figure 6.5). Les sources de trafic et le commutateur sont semblables à ceux déjà décrits dans le chapitre précédent avec routage aléatoire uniforme. Nous ne rappellerons donc pas leur principe de fonctionnement.

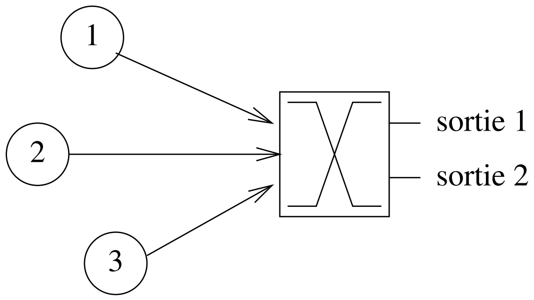


Figure 6.5 Configuration a.

Dans la configuration (b), un réseau de 5 commutateurs est utilisé. Ce réseau est fort semblable à celui utilisé au chapitre précédent, hormis le fait que celui-ci n'est pas complètement maillé (figure 6.6)—liens non bi-directionnels. Nous avons choisi de réduire le niveau de synchronisation pour simplifier dans un premier temps le problème. De ce fait, chaque commutateur possède seulement 3 entrées et 2 sorties (contre N_s entrées et $N_s - 1$ sorties au chapitre précédent). N_s est le nombre de commutateurs). Chaque commutateur reçoit les messages d'une source de trafic qui génère des messages de réception de cellules jusqu'à une date $t_{endsource}$ inférieure à la durée de la simulation. Lorsque les sources n'émettent plus, le nombre de cellules dans le réseau reste constant. Nous avons fixé $t_{endsource} = 40$ pour toutes les simulations. Le temps de propagation entre deux commutateurs est resté le même, à savoir 2 *time slots*.

Trois cas différents sont considérés en utilisant les deux configurations décrites. Ils sont présentés ci-dessous :

1. Cas 1 : nous utilisons la configuration (a) avec des sources de trafic exponentielles. Les trois sources génèrent en continue des arrivées de cellules.

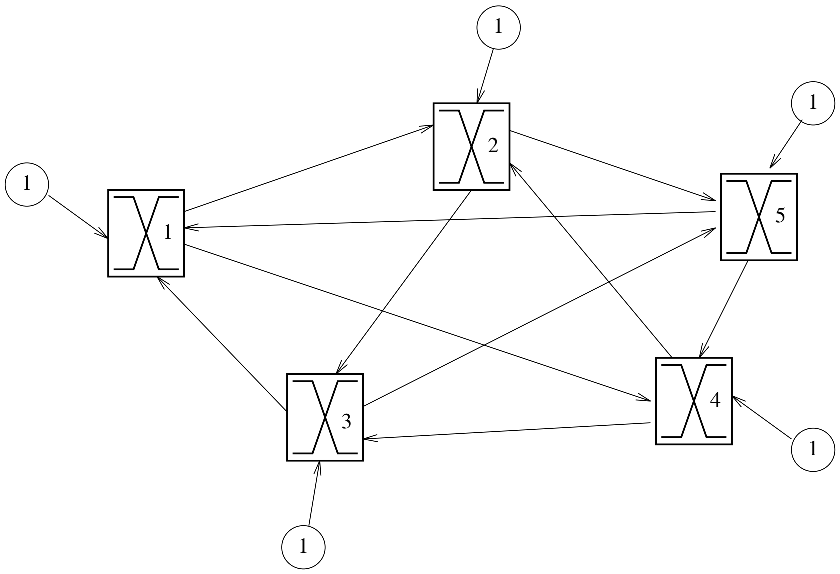


Figure 6.6 Configuration b.

2. Cas 2 : nous utilisons la configuration (b) où toutes les sources de trafic sont exponentielles.
3. Cas 3 : nous utilisons la configuration (a) avec trois sources du type ON/OFF. Les durées des périodes de silence et de burst sont toutes les deux distribuées de manière exponentielle. Pendant un burst, la source génère une cellule à chaque time slot.

La sauvegarde des informations pour le calcul des estimations locales est effectuée après chaque exécution d'un événement $< rec, t >$. L'exécution d'un événement auto-généré, comme $< env, t >$, ne provoque pas une sauvegarde de l'état. Ce choix permet de réduire la quantité des informations sauvegardées.

6.5 RÉSULTATS ET DISCUSSION

Les simulations sont effectuées sur un réseau de stations de travail et nous avons réalisé l'implantation de notre approche en C avec la librairie PVM pour les primitives d'envoi de messages inter-processus comme précédemment. Afin de pouvoir quantifier facilement le niveau de précision de l'approche statistiquement correcte, nous utilisons des sources de trafic déterministes. C'est-à-dire que d'une simulation à l'autre, les sources génèrent la même séquence d'événements $< rec, t >$. Nous obtenons cependant des différences dans le nombre de fautes temporelles grâce au caractère aléatoire des délais de transmission.

Le temps de simulation est fixé à 4000 *time slots*. Le critère de performance auquel nous nous intéressons est la longueur moyenne des files de sortie des commutateurs exprimée en nombre de cellules. Ce critère répond au caractère "moyen" qui est nécessaire pour pouvoir utiliser les estimations locales de manière efficace. Les résultats montrent en outre le nombre de fautes primaires et secondaires qui ont été détectées. Pour quantifier le niveau de précision par rapport à la méthode conservatrice, nous utilisons ce que nous appelons un pourcentage d'erreur. Ce pourcentage d'erreur, noté $err_{\%}$, est obtenu en considérant la différence entre la valeur fournie par la méthode conservatrice, valeur de référence, et celle fournie par la méthode statistiquement correcte. Cette différence est ensuite transposée en pourcentage par rapport à une valeur d'erreur de référence fixée à 100% qui se présente lorsque la valeur fournie par la méthode statistiquement correcte est nulle. Par exemple, soient x_c et x_s les valeurs fournies respectivement par la méthode conservatrice et notre méthode, le pourcentage d'erreur se calcule de la manière suivante :

$$err^{\%} = 100\frac{x_c - x_s}{x_c}$$

Les valeurs fournies par l'approche conservatrice sont présentées avec un intervalle de confiance à 95% calculé sur 10 échantillons indépendants. De ce fait, les valeurs fournies par la méthode statistiquement correcte qui rentrent dans l'intervalle de confiance sont tout à fait acceptables. Le calcul du pourcentage d'erreur comme nous l'avons réalisé possède la particularité d'être dépendant de la valeur de référence. En effet, si la valeur de référence est grande, la même différence sera traduite par un pourcentage d'erreur moins élevé. Cela reflète d'une certaine manière le fait qu'il est moins pénalisant d'obtenir une différence de d par rapport à une grande valeur que par rapport à une petite valeur (si on trouve 306 au lieu de 308, c'est moins grave que si on trouve 2 au lieu de 4) même lorsque les intervalles de confiance sont les mêmes.

Nous n'avons implanté que le contrôle nul qui fourni des résultats du type "best-effort". En ce qui concerne le contrôle basé sur la distance, nous avons été confrontés à l'ampleur de la tâche qui consiste à écrire un noyau de simulation Time Warp.

6.5.1 Résultats pour les cas 1 et 2

Les tables 6.1 et 6.2 montrent la longueur moyenne d'une sortie pour le cas 1 obtenue respectivement avec une approche conservatrice et notre approche statistiquement correcte avec le contrôle nul.

On peut remarquer que le nombre de fautes primaires est faible. Cela s'explique facilement par le fait que le système étudié est petit et que le transfert des cellules est uniquement unidirectionnel. Tout cela réduit considérablement le degré d'interaction entre les différentes composantes du système et contribue à abaisser le nombre de fautes temporelles. Par conséquent, il est normal que l'on obtienne un bon degré de précision. On peut généraliser ce résultat à des configurations de commutateurs en cascade (figure 6.7) puisque ce modèle est facilement réplicable.

Cas 1	SW1
Buffer 1	0.812 ±0.019
Buffer 2	0.846 ±0.020

Tableau 6.1 Simulation conservatrice du cas 1.

Cas 1 1	SW1
Fautes primaires	5
Fautes secondaires	22
Buffer 1	0.809 → 0.37%
Buffer 2	0.841 → 0.59%

Tableau 6.2 Simulation statistiquement correcte du cas 1.

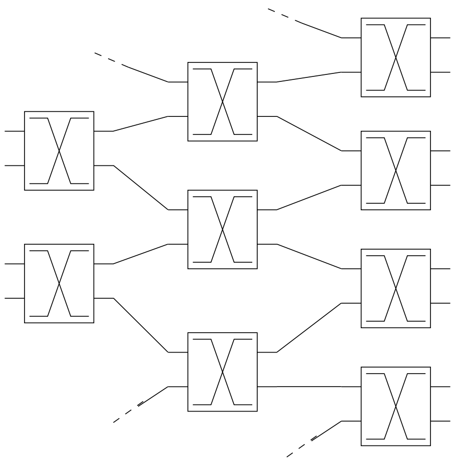


Figure 6.7 Commutateurs en cascade.

Pour le cas 2, les choses se compliquent car les interactions entre les composantes du réseau sont maintenant plus complexes. Dans ce scénario, un commutateur a peut recevoir d'un commutateur b une cellule qu'il lui avait originellement envoyé. Si entre-temps le commutateur a a exécuté plusieurs événements provenant d'autres commutateurs, le risque de provoquer une faute temporelle est beaucoup plus grand. La table 6.3 et la table 6.4 montrent les résultats obtenus. En comparant avec les résultats du cas 1, nous trouvons effectivement que le nombre de fautes primaires est maintenant plus élevé. Cependant, et cela est encourageant, les résultats qu'on obtient affichent un degré de précision très satisfaisant.

Pour les deux cas, la longueur moyenne des buffers fournie par notre approche statistiquement correcte est toujours plus petite que celle fournie par une approche traditionnelle. Ce phénomène est en partie prévisible car les erreurs introduites en n'utilisant pas le *rollback* se traduit par une sous-estimation du temps d'attente des

Cas 2	SW1	SW2	SW3	SW4	SW5
Buffer 1	2.584	2.588	2.581	2.572	2.578
Buffer 2	2.578	2.574	2.561	2.558	2.558

Tableau 6.3 Simulation conservatrice du cas 2.

Cas 2	SW1	SW2	SW3	SW4	SW5
Fautes primaires	93	109	113	98	112
Fautes secondaires	135	157	151	137	148
Buffer 1	2.425	2.374	2.421	2.385	2.463
Buffer 2	2.383	2.436	2.401	2.434	2.395
$err^{\%}$	6.15%	8.26%	6.19%	7.27%	4.45%
$err^{\%}$	7.56%	5.36%	6.24%	4.84%	6.37%

Tableau 6.4 Simulation statistiquement correcte du cas 2.

cellules précédemment envoyées. A la lumière des résultats obtenus, cette erreur est plutôt faible pour les cas que nous avons étudiés jusqu'à présent.

6.5.2 Résultats pour le cas 3

Le cas 3 utilise des sources de trafic plus sporadiques du type ON/OFF. Ce cas est introduit pour étudier l'influence du "motif" du trafic sur la précision des estimations locales. Nous avons enregistré et tracé l'évolution du nombre de cellules dans la sortie 1 de commutateur Sw1. La figure 6.8 représente cette évolution pour une simulation conservatrice et la figure 6.9 pour une simulation statistiquement-correcte—toujours avec le contrôle nul.

On peut remarquer un burst manquant dans la figure 6.9. Comme l'enregistrement des arrivées de cellule se fait en-ligne, c'est-à-dire lors de l'arrivée effective de la cellule, ce burst manquant ne peut s'expliquer que par un déséquencement des messages introduisant des fautes temporelles. Ce cas se produit lorsque le processus a simulé de manière optimiste trop loin dans le futur et, à l'arrivée du burst hors séquence, le processus va utiliser les estimations locales et la réévaluation des états pour tenter de reconstruire une évolution cohérente sans enregistrer l'arrivée du burst.

Les tables 6.5 et 6.6 montrent la longueur des buffers pour le cas 3 obtenue respectivement avec une simulation conservatrice et une simulation statistiquement correcte avec un contrôle nul. En comparant avec les résultats précédemment obtenus pour le cas 1, on constate que le nombre de fautes secondaires est bien plus grand. Cela s'explique par le fait que la sporadicité des sources engendre des fautes temporelles par paquets. Par conséquent, les erreurs introduites en négligeant le temps d'attente supplémentaire des cellules précédemment envoyées sont plus grandes et modifient de manière appréciable la précision des résultats. Il faut cependant noter que seules les cellules envoyées pendant la durée du busrt manquant sont concer-

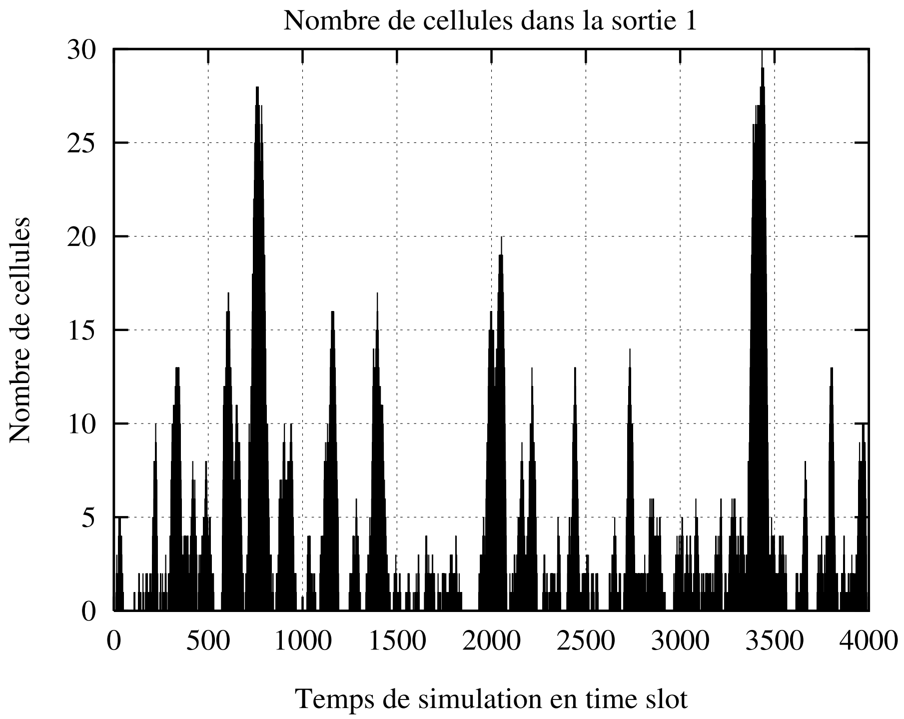


Figure 6.8 Evolution du nombre de cellules—Simulation conservatrice.

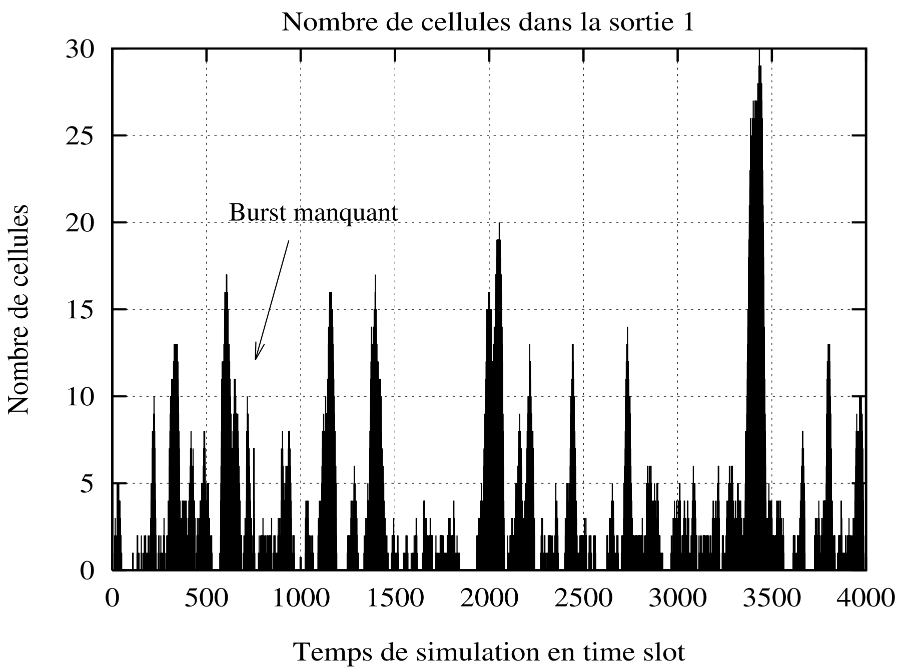


Figure 6.9 Evolution du nombre de cellules–Simulation statistiquement correcte

Cas 3	SW1
Buffer 1	3.975 ±0.021
Buffer 2	3.938 ±0.020

Tableau 6.5 Simulation conservatrice du cas 3.

Cas 3	SW1
Fautes primaires	2
Fautes secondaires	123
Buffer 1	3.682 $\rightarrow$ 7.37%
Buffer 2	3.881 $\rightarrow$ 1.44%

Tableau 6.6 Simulation statistiquement correcte du cas 3.

nées par cette sous-estimation du temps d'attente. Malheureusement, si le burst est suffisamment long, un grand nombre de cellules ont pu être traitées pendant cette période (figure 6.10) et cela peut entraîner une chute de la précision.

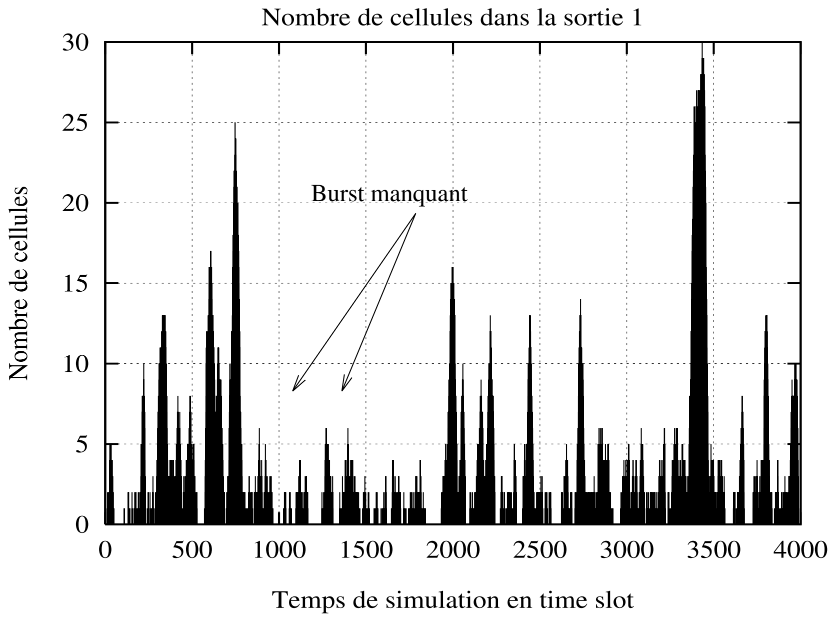


Figure 6.10 Evolution du nombre de cellules–Simulation statistiquement correcte

Dans la table 6.7 on peut voir que le nombre de fautes secondaires est nettement plus élevé que dans les autres cas. Ce dernier cas montre les limites des estimations locales puisque les résultats obtenus avec notre approche diffèrent de manière appréciable de ceux obtenus avec une simulation traditionnelle. En fait, ce n'est pas vraiment le nombre de fautes temporelles qui affecte la précision des résultats, mais le nombre de cellules qui ont été envoyées pendant le burst manquant.

Cas 3	SW1
Fautes primaires	9
Fautes secondaires	317
Buffer 1	3.400 $\rightarrow$ 14.46%
Buffer 2	3.521 $\rightarrow$ 10.58%

Tableau 6.7 Autre simulation statistiquement correcte du cas 3.

6.6 CONCLUSION DU CHAPITRE

Dans ce chapitre, nous avons proposé une nouvelle approche appelée *statistiquement correcte* pour la simulation distribuée d'application où une précision stricte n'est pas toujours nécessaire. Cette approche à pour but de réduire le coût de la simulation tout en gardant un niveau de précision appréciable.

La manière dont sont obtenus les estimations est très dépendante de l'application et cette méthode donne de meilleurs résultats lorsque les messages générés et les états des processus ne changent pas beaucoup après l'exécution du message manquant. Bien sûr, nous sommes conscients que cette démarche n'est sans doute pas généralisable et qu'elle ne trouve son utilité que dans certains cas d'utilisation. Sans remplacer les méthodes conventionnelles, la méthode que nous proposons s'instaure dans le contexte d'une prospection préliminaire où le niveau de précision des résultats n'est pas encore capitale. Au contraire, la rapidité d'obtention des résultats, pour se faire une "idée", est plus attractive.

Les tests que nous avons effectués montrent que les estimations locales donnent de bons résultats, mais nous avons remarqué qu'un grand nombre de fautes temporelles peut les rendre inefficaces. A cet effet, deux aspect importants restent à étudier : le gain en vitesse de notre approche par rapport à une simulation conservatrice et le contrôle de la précision des résultats fournis par notre approche. Ce sera l'objet du chapitre suivant.

6.7 ANNEXE

Version littérale des estimations locales

$$\blacksquare\ \text{Etape 2}\ \begin{cases} \text{if } t = last_j^i \text{ then } t_w = max\{(o_j^i - 1)t_t, t_t\} \\ \text{ else } t_w = max\{o_j^i - 1 - \frac{(t-T_i)}{t_t}, 0\}. \\ t_d = t + t_w + t_c + t_t. \end{cases}$$

$$\blacksquare\ \text{Etape 3}\ \begin{cases} o_j^{i+1} = max\{o_j^i - \frac{(t-T_i)}{t_t}, 0\} + 1. \\ last_j^{i+1} = t. \\ n_r^{i+1} = n_r^i + 1. \\ n_s^{i+1} = n_s^i + min\{o_j^i, \frac{t-T_i}{t_t}\}. \end{cases}$$

$$\blacksquare\ \text{Etape 4a}\ \begin{cases} \text{if } \frac{T_k - t}{t_t} < o_j^{i+1} \text{ then } o_j^k = o_j^k + 1 \text{ else } o_j^k \text{ reste inchangé.} \\ last_j^k \text{ reste inchangé.} \\ n_r^k = n_r^k + 1. \\ \text{if } t_d < T_k \text{ then } n_s^k = n_s^k + 1 \text{ else } n_s^k \text{ reste inchangé.} \end{cases}$$

$$\blacksquare\ \text{Etape 4b}\ \begin{cases} \text{if } \frac{t-T_i}{t_t} > o_j^i \text{ then } \Sigma S(n) = \Sigma S(n) + t_t \\ \text{else} \\ \Sigma S(n) = \Sigma S(n) + \{o_j^i - \frac{(t-Ti)}{t_t} + 1\}t_t \end{cases}$$

Chapitre 7

Limitation de l'optimisme

7.1 INTRODUCTION

Nous avons vu dans le chapitre précédent que les estimations locales donnent de bons résultats mais qu'elles peuvent être inefficaces lorsque le nombre de fautes temporelles augmente de façon importante. Le maintien de cette méthode implique 2 directions possibles : (i) réduire le nombre de fautes temporelles et (ii) améliorer la précision des estimations locales. Nous allons regarder dans un premier temps la première solution.

Dans une approche purement optimiste, la manière habituelle d'arriver à réduire le nombre de fautes est de freiner l'optimisme du protocole. Plusieurs solutions ont été proposées et celles-ci peuvent être classées en deux catégories : celles basées sur l'utilisation d'une fenêtre et celles basées sur l'utilisation d'une stratégie de blocage. Dans ce chapitre, nous étudions une stratégie de blocage basée sur la longueur de la file d'événements et son utilisation en conjonction avec les estimations locales. Le fonctionnement d'un pl est alors le suivant : le pl doit suspendre l'exécution des événements jusqu'à ce que la longueur de sa file d'attente des événements devienne supérieure à un certain seuil L_{min}. Cette stratégie de blocage nous permet de réduire le nombre de fautes temporelles afin de permettre aux estimations locales de mieux traiter les erreurs restantes. Nous appliquons cette approche à la simulation d'un réseau ATM et nous cherchons dans un premier temps à trouver empiriquement un point de fonctionnement optimal entre la longueur initiale de blocage, le gain en vitesse par rapport à une méthode conservatrice et la précision des résultats. Dans un deuxième temps, nous essayons d'ajuster automatiquement la longueur initiale de blocage en spécifiant le degré de précision souhaité par le biais du taux de fautes temporelles.

Ce chapitre est organisé comme suit : la section 2 retrace brièvement les travaux effectués dans le domaine[1], la section 3 présente les motivations pour utiliser une stratégie de blocage, la section 4 présente l'algorithme de blocage et la section 5 présente les configurations d'étude ainsi que les résultats obtenus. La conclusion du chapitre est donnée dans la section 6.

7.2 TRAVAUX EFFECTUÉS

Les approches qui sont basées sur une fenêtre d'exécution pour réduire l'optimisme ont été utilisées dans des protocoles tels que *Filtered Time Warp* (FTW) [Luba89], *Moving Time Window* (MTW) [Soko89], *Breathing Time Bucket* (BTB) [Stei91] et *Adaptive Time-Ceiling* (ATC) [Mats93]. Ces protocoles limitent l'optimisme en confinant les événements à considérer dans une fenêtre temporelle. La taille de la fenêtre peut être statique comme dans FTW et MTW—même si ce dernier déplace dynamiquement la fenêtre—ou dynamiquement ajustée. Par exemple, BTB divise le temps de simulation en *buckets* de taille variable. Un bucket est une sorte de fenêtre dynamique dont la borne supérieure, appelé *event-horizon*, est définie comme étant l'estampille minimale parmi toutes les estampilles des événements générés dans le bucket. Dans le protocole ATC, un seuil temporel, qui peut être assimilé à la borne supérieure d'une fenêtre, est dynamiquement ajusté en surveillant les performances du simulateur et le taux de fautes temporelles.

D'un autre coté, la plupart des protocoles basés sur une stratégie de blocage ajoutent un comportement conservateur en suspendant l'exécution des *pls* pendant un certain temps. On assiste en ce moment à un formidable engouement pour ce type de protocoles qui utilisent le blocage pour se positionner entre les deux points extrêmes représentés par les approches traditionnelles conservatrices et optimistes. Srinivasan et Reynolds [Srin95b] ont défini une classe de protocoles appelée *Asynchronous Adaptive Waiting Protocols* (AAWP) qui englobe une grande partie des protocoles existants, et en particulier ceux décrits ci-après.

Une des plus ancienne approche utilise une stratégie de blocage qui repose sur la notion de pénalité (*penalty-based*) et a été proposée dans [Reih89]. Cet algorithme suspend l'exécution d'un *pl* pendant une durée fixe à chaque fois que celui-ci envoi un anti-message. Comme l'envoi d'anti-messages apparaît comme un mauvais comportement signifiant que des calculs erronés ont été effectués, retarder l'exécution des prochains événements est un moyen d'éviter d'autres erreurs. Ball et Hyot proposent quant à eux le protocole *Adaptive Time Warp* [Ball90] qui suspend l'exécution du *pl* si un nombre anormalement élevé de fautes temporelles se sont produit dans un passé récent. Une estimation du temps optimal de blocage pour chaque *pl* est calculée en minimisant une fonction indiquant le taux d'inactivité. Dans [Fers95], un modèle de coût pour parcourir un continuum entre une exécution optimiste et une exécution conservatrice est construit en échantillonnant les estampilles des messages qui arrivent. Les auteurs calculent à partir de ce modèle le temps optimal de blocage de la

[1] Les lecteurs ayant lu dans l'état de l'art le chapitre concernant les approches mixtes et adaptatives peuvent lire très rapidement cette section.

CPU pour chaque événement, avant son exécution. Hamnes et Tripathi proposent le protocole *Local Adaptive Protocol* qui couvre le continuum conservateur–optimiste en associant à chaque entrée un temps de blocage (*real-time blocking window—RTWB*) [Hamn94]. Quand certaines entrées sont vides, le RTWB contrôle la procédure de sélection des événements pour choisir entre attendre, se bloquer ou continuer. La valeur optimale du RTWB est périodiquement calculée et doit maximiser le taux d'avancement du temps de simulation. Dans [Srin95a], les auteurs introduisent une nouvelle sous-classe, appelée *Near Perfect State Information* (NPSI), à la classe de protocoles AAWP. Les protocoles de cette sous-classe ont la caractéristique de reposer sur l'acquisition d'informations globales à un coût très faible pour ajuster leur fenêtre de blocage à une valeur optimale. Ils proposent en outre un protocole NPSI appelé *Elastic Time Algorithm* (ETA), qui considère la différence entre le LVT et le GVT, obtenu par une méthode NPSI, pour réduire son optimisme.

7.3 MOTIVATIONS POUR LE BLOCAGE

Même si une stratégie de blocage interrompt l'exécution des événements, on peut noter que dans beaucoup de cas, la probabilité de provoquer une faute temporelle décroît lorsque le nombre d'événements dans la LEV augmente. Cela peut être expliqué par le fait que le *pl* a plus de "choix" lorsqu'il sélectionne un événement pour l'exécuter. Ensuite, exécuter un événement dès qu'il est disponible peut être très intéressant, si cet événement est effectivement celui que l'on attend (l'événement est sur le chemin critique de la simulation), mais peut s'avérer très hasardeux sinon. En effet, cette "agressivité" peut conduire à un nombre excessif de fautes temporelles qui dégradent fortement les performances du simulateur. Finalement, le blocage permet de réduire la probabilité de fautes temporelles en permettant à plus d'événements d'être placés dans la LEV. Bien sûr cela est seulement avantageux si le temps passé en blocage est inférieur au temps passé à corriger les fautes temporelles. Dans [Srin95b], les auteurs ont fait remarquer qu'une mauvaise décision de blocage peut causer un grand nombre de fautes temporelles supplémentaires. De ce fait, une stratégie de blocage doit être manipulée avec beaucoup de soin.

Contrairement à l'approche basée sur les pénalités dont la stratégie de blocage peut être classée comme *réactive*, notre approche, ainsi que toutes les autres précédemment citées, peut être qualifiée de *préventive*[2]. Dans une stratégie réactive, le *pl* ne suspend son exécution que sur occurrence d'une faute temporelle. Ce schéma de fonctionnement peut être efficace si de telles fautes sont relativement rares et arrivent par paquets. Malheureusement, cette hypothèse est fausse dans le cas de systèmes comportant un grand nombre de sous-systèmes qui interagissent de manière intensive. Dans ce cas, un grand nombre de messages sont échangés d'un *pl* à l'autre ce qui tend à rendre les fautes temporelles très fréquentes et arrivant, non pas par paquet, mais de manière uniforme tout au long de la simulation. De manière générale, les stratégies réactives sont d'un paramétrage délicat car la condition de

[2]Le terme *pro-actif* peut également être utilisé lorsqu'un contrôle bouclé est réalisé.

réaction et le temps de réaction doivent être soigneusement déterminés, sinon l'algorithme peut devenir très instable et exhiber de très mauvaises performances. Au contraire, une approche préventive semble plus prometteuse pour réduire le nombre de fautes temporelles, même au prix du coût supplémentaire pour surveiller de manière continue l'avancement de la simulation.

Peu de travaux ont cependant été effectués sur l'utilisation d'un seuil basé sur la longueur de file plutôt que sur le temps. De manière paradoxale, il est pourtant plus naturel de fixer une longueur de seuil plutôt qu'un temporisateur car on peut difficilement prédire les instants d'arrivée des messages. D'une part, les approches basées sur le temps autorisent un réglage très précis puisque le temporisateur peut prendre des valeurs réelles mais d'un autre coté, avec une longueur de seuil, il est possible de réduire le coût du blocage car la condition peut être satisfaite immédiatement. Par exemple, considérons le cas où le temps de blocage optimal est T_{opt} et la longueur optimale correspondante L_{opt}. Avec un temporisateur, il est nécessaire d'attendre effectivement T_{opt} unités de temps pour être sûr de réduire la probabilité de fautes temporelles, même s'il s'avère ensuite que cela était inutile. Maintenant, s'il n'était pas nécessaire d'attendre, une approche basée sur la longueur de la file d'attente peut "sauter" la phase d'attente. Il peut donc y avoir des cas où les premiers doivent bloquer et pas les seconds, sans risque supplémentaire.

7.4 APPROCHE BASÉE SUR LA LONGUEUR DE FILE

La stratégie de blocage que nous proposons est basée sur la longueur de la file des événements. L'exécution du *pl* est suspendue jusqu'à ce que la longueur de sa file devienne supérieure à un seuil L_{min}.

Contrairement aux approches basées sur un temps de blocage, il n'est pas possible de fixer la longueur de blocage L_{min} à une valeur initiale arbitraire L_{init} sans faire face à des problèmes d'inter-blocage. En effet, la population de messages dans le système est généralement finie et les messages sont le plus souvent liés entre eux. Par conséquent, si aucun message n'est exécuté, aucun message ne sera délivré. Donc, si L_{min} est plus grand que le nombre de messages qui peut arriver sur la file d'entrée, on obtient une configuration de blocage car tous les *pls* attendent après d'autres *pls*. Pour résoudre ce problème, il est nécessaire de relâcher la condition de seuil après un certain temps. Pour cela, nous proposons de décrémenter la valeur de L_{min} toutes les D_t unités de temps (ut). Ainsi, la condition de seuil est relâchée petit à petit. Un comportement fortement agressif est obtenu lorsque $L_{min} = 1$ car le *pl* est prêt à exécuter le premier message qui arrive. La figure 7.1 représente l'organigramme de l'algorithme.

7.4.1 Valeurs pour L_{init} et D_t

Il y a deux alternatives pour le choix de L_{init} et D_t : soit fixer L_{init} grand et D_t petit, soit le contraire. Dans la première solution, si les messages arrivent rapidement, alors la condition de seuil sera très rapidement satisfaite. Maintenant si les messages

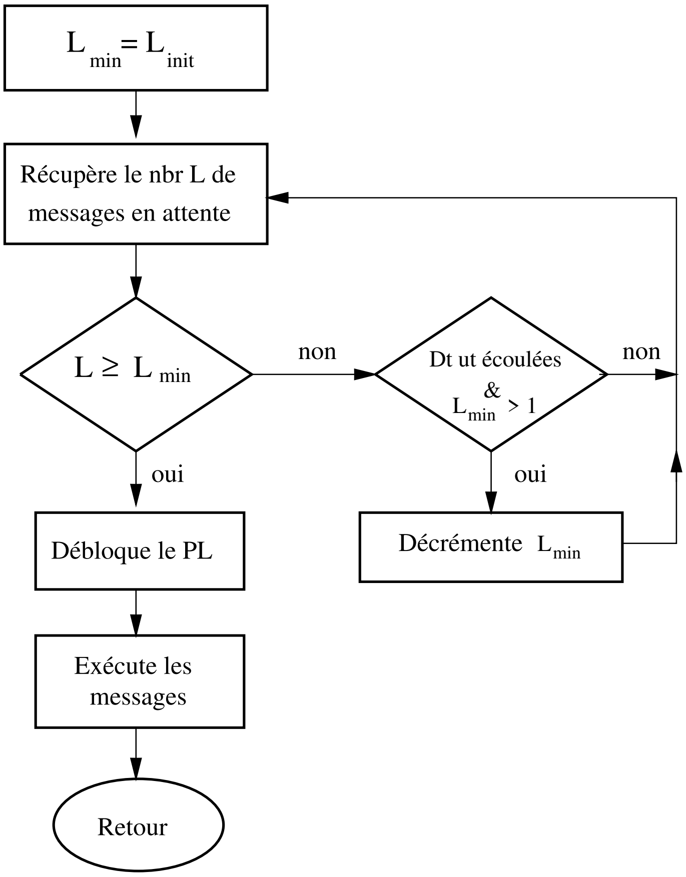


Figure 7.1 Organigramme de l'algorithme de blocage.

arrivent lentement, alors la valeur du seuil L_{min} diminuera néanmoins rapidement pour éviter les attentes inutiles. Dans la seconde solution, si les messages arrivent rapidement, alors la condition de seuil sera très rapidement satisfaite comme précédemment, mais si ce n'est pas le cas le *pl* peut perdre beaucoup de temps à attendre jusqu'à ce que la condition de seuil soit suffisamment relâchée. Nous préférons la première solution qui est à notre sens plus adaptative. L'espoir sous-jacent d'une telle stratégie basée sur la longueur de file est que les messages arrivent rapidement au *pl* afin d'éviter dans la plupart des cas des attentes inutiles.

Maintenant, la longueur initiale L_{init} ne doit pas être choisie trop grande sinon la condition de seuil sera trop difficile à satisfaire. Il peut sembler intéressant de fixer ce seuil assez haut pour diminuer le taux de fautes temporelles au maximum. Le cas extrême où L_{init} est si grand que le *pl* bloqué a dans sa file d'événements au moins un message de chaque *pl* communicant avec lui est théoriquement équivalent à une méthode conservatrice si les inter-blocage pouvaient être évités. Ce n'est cependant pas le but de cette approche car un certain risque est nécessaire pour obtenir un gain intéressant par rapport à une méthode conservatrice.

7.5 CONFIGURATION ET RÉSULTATS EXPÉRIMENTAUX

Nous sommes partis de l'implantation optimiste déjà évoquée dans le chapitre précédent et nous avons ajouté l'algorithme de blocage que nous proposons. Les simulations sont effectuées sur deux configurations de machines : une SparcStation 20 avec 64MB de mémoire vive (configuration 1) et une CM-5 à 128 processeurs (configuration 2) du *Centre National de Calcul Parallèle en Sciences de la Terre (CNCPST)* comme dans le chapitre 4. Des précautions ont en outre été prises pour s'assurer que la simulation est bien la seule charge utilisatrice de la CM-5 lors des mesures.

Le réseau ATM que nous étudions est identique à celui utilisé au chapitre 4 (dans ce chapitre, nous utilisions une méthode conservatrice). Par rapport à celui utilisé au chapitre précédent, celui-ci est complètement maillé et par conséquent présente un niveau d'interaction beaucoup plus élevé. Nous rappelons ci-dessous le réseau de l'étude pour $N_s = 5$ commutateurs.

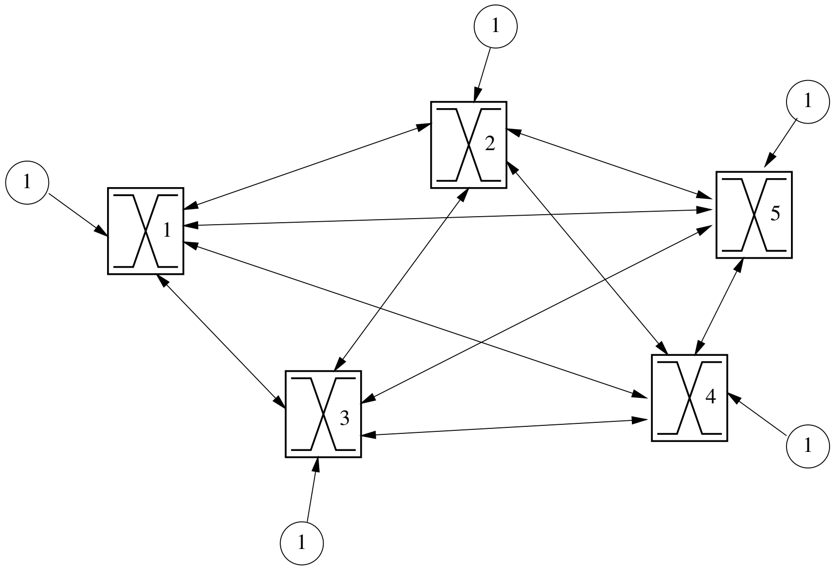


Figure 7.2 Réseau complètement maillé pour $N_s = 5$.

Les premiers tests sont effectués avec $N_s = 5$. Comme précédemment, nous nous sommes intéressés à la longueur moyenne des buffers de sortie (MBL) exprimée en nombre de cellules. Seuls les résultats pour la sortie 1 du commutateur 1 sont présentés car les autres résultats sont similaires. Avec la configuration 1, le nombre de cellules dans le réseau est fixé à 100 ($n_c = 20$) et nous avons expérimenté pour 10 valeurs de L_{init} : 1, 2, 4, 5, 6, 8, 10, 11, 13 and 15. D_t est fixé à 15ms pour toutes les simulations. Les simulations s'arrêtent lorsque tous les commutateurs ont simulé 20000 réceptions de cellules. Avec la configuration 2, un paramétrage automatique de L_{init} est réalisé sous une charge de trafic différente ($n_c = 10$). Enfin, pour quantifier la précision des résultats obtenus avec notre approche, nous les comparons à ceux obtenus avec une méthode conservatrice.

Les critères de performances que nous utilisons sont d'une part le gain en vitesse par rapport à une méthode conservatrice et d'autre part le *Time Error Ratio* (TER) défini de la manière suivante :

$$TER = \frac{\text{Nombre de fautes temporelles}}{\text{Nombre d'événements exécutés}}$$

Le numérateur représente le nombre de fautes temporelles, causées par des messages fautifs, qui ont été corrigées par les estimations locales. Par conséquent, le TER représente la proportion d'événements qui devraient déclencher le mécanisme du *rollback* dans une simulation optimiste traditionnelle.

7.5.1 Valeur empirique optimale pour L_{init}

La figure 7.3 montre le gain en fonction de la valeur de L_{init} obtenu sur la SS20. L'axe des x indique les différentes valeurs que l'on utilise. Le gain obtenu est représenté sur l'axe des y.

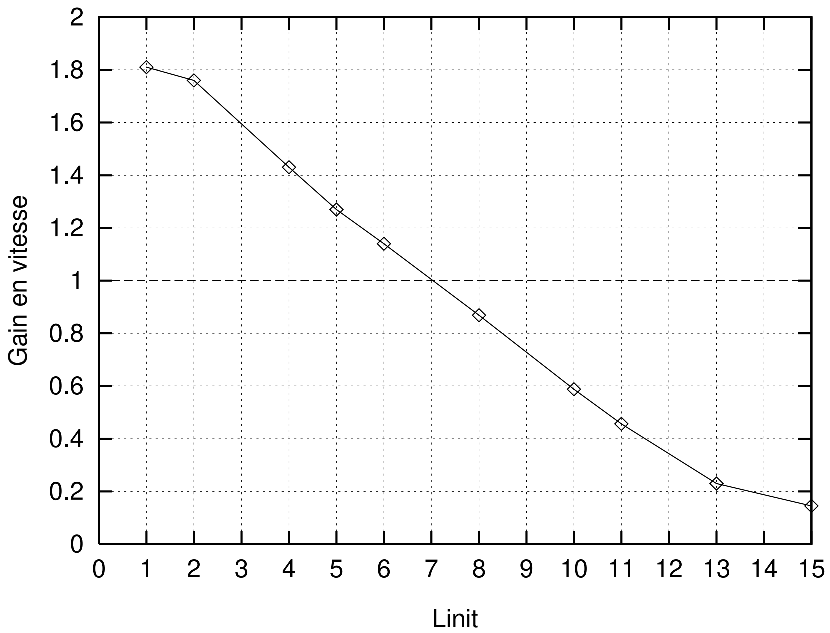


Figure 7.3 Gain en fonction de L_{init}

Il est intéressant de remarquer que la courbe interpolée est presque linéaire en fonction de L_{init}. Quand la valeur initiale du seuil L_{init} est trop élevée, on observe une chute du gain car le *pl* perd trop de temps à attendre inutilement. Les valeurs intéressantes pour L_{init} doivent bien sûr donner des gains supérieurs à 1.

Le TER est présenté dans la figure 7.4 avec les différentes valeurs pour L_{init} sur l'axe des x. Cette courbe convexe présente un minimum pour $L_{init} = 10$. Au fur et à mesure que L_{init} augmente de 1 à 10, la stratégie de blocage réussi à réduire le TER en autorisant la réception de plus de messages dans la file d'événements. Cependant, pour des valeurs supérieures à 10, on observe une remontée du TER. Ce phénomène étrange que nous n'avions pas prévu peut être expliqué de la manière suivante. Une grande valeur de L_{init} rend la condition de seuil très difficile à satisfaire et par la même occasion suspend tous les *pls* qui vont attendre pour des messages. Malheureusement, comme les messages dans notre modèle de réseau sont très étroitement liés entre eux, les *pls* se bloquent mutuellement et attendent une relâche de la condition de seuil. Lorsque L_{min} devient petit, certains *pls* sont brutalement tirés de leur phase d'attente et recommencent à produire des messages. La réception de ces messages par les autres *pls* entraîne leur exécution immédiate car le seuil L_{min} de ces *pls* est aussi devenu très bas. De ce fait, les messages sont en fin de compte exécutés de manière très agressive. Cet effet de bord causé par une condition de seuil trop élevée est alors similaire au problème déjà évoqué dans [Srin95b] où un temps de blocage trop grand dans les approches basées sur un temporisateur introduit des fautes supplémentaires.

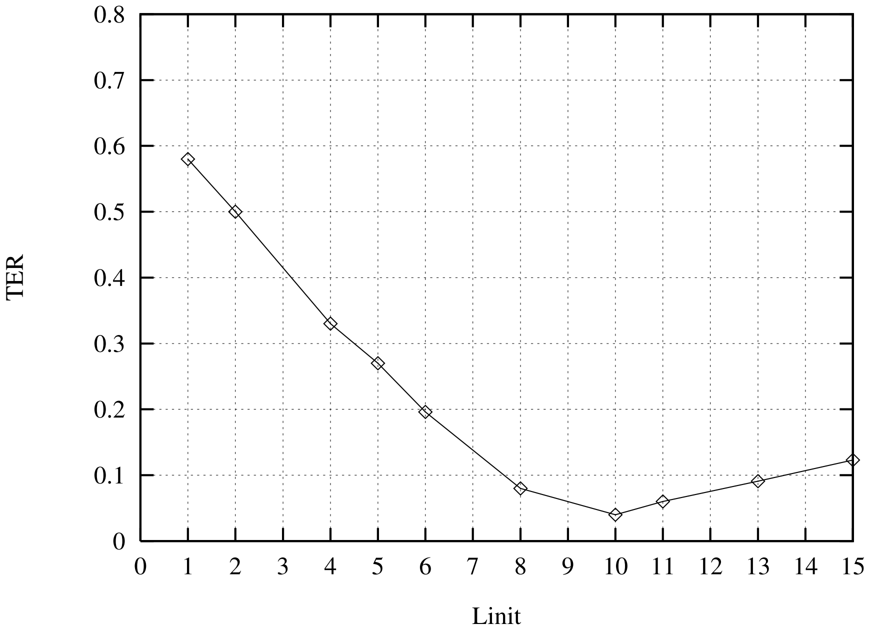
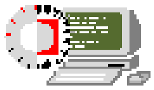


Figure 7.4 TER en fonction de L_{init}

La table 7.1 compare la longueur moyenne de la sortie du commutateur obtenue avec une méthode conservatrice (CMB pour Chandy-Misra-Bryant) à celle obtenue avec notre méthode utilisant les estimations locales et la stratégie de blocage. Les résultats sont donnés en fonction de L_{init} et la valeur du TER correspondant est également indiquée.



L_{init}	MBL	$err^{\%}$	TER
15	3.48	5.45%	0.123
13	3.40	3.03%	0.091
11	3.45	4.54%	0.060
CMB	3.30	0%	0
10	3.20	3.03%	0.044
8	3.19	3.33%	0.080
6	3.15	4.54%	0.196
5	2.93	11.21%	0.270
4	2.85	13.64%	0.330
2	2.65	19.70%	0.500
1	2.52	23.64%	0.580

Tableau 7.1 Pourcentage d'erreur et TER en fonction de L_{init}.

Pour $L_{init} = 1, 2$, les résultats qu'on obtient avec notre méthode sont bien au dessous de ceux obtenus avec une méthode conservatrice avec un pourcentage d'erreur de plus de 20%. Au fur et à mesure que L_{init} augmente, la précision s'améliore du fait de la réduction du nombre de fautes temporelles (cf. Figure 7.4). Pour $L_{init} = 4, 5$ le pourcentage d'erreur est inférieur à 15%. Pour $L_{init} \geq 6$ celui-ci devient inférieur à 5%.

Finalement, à la lumière des résultats obtenus, le meilleur compromis entre la précision et le gain est réalisé pour $L_{init} = 4, 5$. Cependant, ces valeurs dépendent de paramètres tels que la vitesse de la machine sur laquelle sont réalisées les simulations ou des délais introduits par le système de communication. Si la simulation est effectuée sur une machine différente, les valeurs optimales pour L_{init} risquent de changer. Par exemple, sur une véritable architecture multi-processeurs, ces valeurs seront très certainement plus petites du fait de l'augmentation de la puissance de calcul.

7.5.2 Paramétrage automatique pour L_{init}

Changer la valeur de D_t n'est pas vraiment intéressant car cela ne ferait que décaler la valeur optimale de L_{init}. Le choix de L_{init}, qui rappelons le est la valeur initiale que l'on donne au seuil L_{min}, est par contre essentiel car il affecte directement la précision de la simulation. De manière idéale, il devrait être possible d'indiquer le degré de précision souhaité, par exemple un pourcentage d'erreur inférieur à 15%, et de laisser le noyau de simulation déterminer la valeur correspondante pour L_{init}.

L'idée est de spécifier un niveau d'erreur en utilisant le TER comme un critère de mesure. Lorsque que l'on regarde la table 7.1, on peut théoriquement associer une valeur de TER de référence, notée TER_{ref}, à un niveau d'erreur même si l'environnement de la simulation change. En effet, la précision de l'algorithme ne devrait dépendre que du nombre de fautes temporelles. Dès lors, trouver la valeur de L_{init} correspondant au niveau d'erreur désiré peut être fait dynamiquement en calculant périodiquement un TER, noté TER_{comp}, sur les derniers événements exécutés. La valeur initiale de L_{min} qui produit la valeur théorique du degré d'erreur devrait également produire une valeur de TER_{comp} inférieure ou égale à TER_{ref}.

Pratiquement, on peut trouver la valeur optimale pour L_{init} en commençant par une valeur arbitraire, 10 par exemple, et en calculant une valeur TER_{comp} tous les N événements exécutés. Le TER_{comp} obtenu peut alors être comparé à la valeur spécifiée, TER_{ref}, afin d'ajuster (incrémenter ou décrémenter) L_{init} jusqu'à trouver un TER_{comp} inférieur ou égal à TER_{ref}. L'algorithme proposé est décrit ci-dessous en pseudo-C.

```
TER_tmp=INFINITY ;
found=FALSE ;
stop_if_lesser=stop_if_greater=FALSE ;
L_init = 10 ;

do {
Simul(N) ;
if (not found)
{
      TER_comp=ComputeTER() ;
      if (TER_comp < TER_ref) {
         if (stop_if_lesser) found=TRUE ;
         else { L_init = L_init - 1 ;
                stop_if_greater=TRUE ; } }
      else {
            if (stop_if_greater) {
               L_init = L_init + 1 ;
               found=TRUE ; }
            else {
                  if (TER_comp > TER_tmp) {
                     L_init = L_init - 1 ;
                     TER_tmp = 0 ;
                     stop_if_lesser=FALSE ; }
                  else {
                        TER_tmp = TER_comp ;
                        L_init = L_init + 1 ;
                        stop_if_lesser=TRUE ; }
               }
         }
   } } while(not end_of_simul) ;
```

A la première itération, on peut choisir N assez grand pour pouvoir atteindre

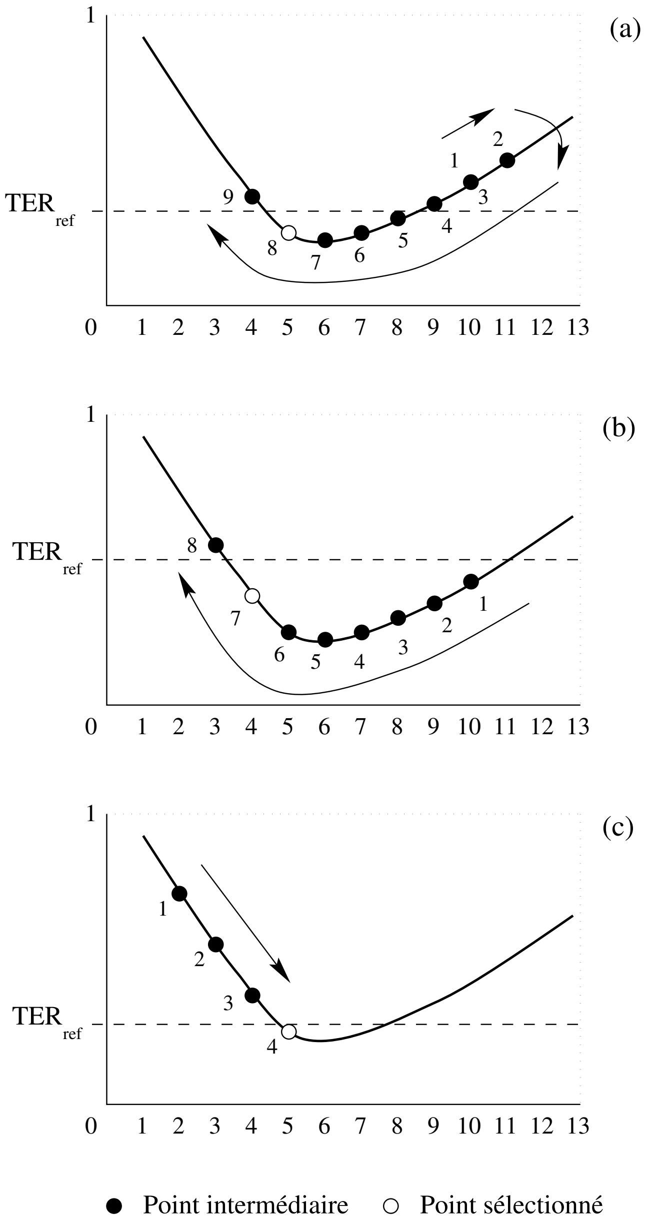


Figure 7.5 Point initial sur la mauvaise branche.

un état stationnaire. Pour les itérations suivantes, N peut être choisi moins grand. En fait, ce qui complique l'algorithme, c'est la possibilité de spécifier un TER_{ref} sur la mauvaise branche de la courbe (graphique a et b). En effet, si le premier point donne un $TER_{comp} > TER_{ref}$, alors incrémenter L_{init} ne diminue pas le TER mais l'augmente au contraire (graphique a). Il est donc nécessaire de détecter cette augmentation du TER afin de réajuster L_{init} dans le bon sens. Le cas décrit dans le graphique b montre un point initial sur la mauvaise branche mais avec un $TER_{comp} < TER_{ref}$. Dans ce cas, il est légitime de diminuer L_{init} pour augmenter le TER jusqu'à obtenir le point qui donne un TER immédiatement inférieur au TER_{ref} avec une valeur de L_{init} minimale. Le graphique c montre un cas favorable où le point initial se situe sur la bonne branche de la courbe. Dans ce cas, augmenter L_{init} permet effectivement de réduire le TER. De manière générale, le nombre d'itérations n'est pas grand car l'éventail des valeurs possibles est relativement limité. Dans la table 7.2 un TER_{ref} est spécifié à l'avance et un TER_{comp} est obtenu en lançant la simulation sur une CM-5. La valeur correspondante de L_{init} est retournée. Le pourcentage d'erreurs par rapport à une simulation conservatrice est également indiqué.

TER_{ref}	TER_{comp}	MBL	$err^{\%}$	L_{init}
0.044	0.044	1.27	1.60%	6
CMB	0	1.25	0%	—
0.080	0.061	1.22	2.40%	5
0.196	0.121	1.20	4.0%	4
0.270	0.268	1.08	13.60%	3
0.330	0.270	1.07	14.40%	3
0.500	0.498	0.93	25.60%	2
0.580	0.496	0.93	25.60%	2
1.0	0.740	0.77	38.40%	1

Tableau 7.2 L_{init} en fonction du TER_{ref}

La figure 7.6 présente le gain obtenu sur une CM-5. Cette courbe n'est plus linéaire et présente une caractéristique intéressante dans la mesure où l'on peut noter un pic se situant à $L_{init} = 4$ alors que pour des petites valeurs de L_{init} on s'aperçoit que le gain est moindre. Ce phénomène peut être expliqué de la manière suivante : lorsque L_{init} est petit, le nombre de fautes temporelles est grand, ce qui veut dire qu'un grand nombre d'estimations sont faites. Bien que ces estimations ne doivent pas prendre beaucoup de temps (car la distance des fautes est relativement petite), un grand nombre de fautes peut cependant ralentir de manière significative la simulation. Lorsque L_{init} est grand, on perd beaucoup de temps à la fois en blocage et en estimations car le TER est élevé (cf figure 7.7). La courbe de gain précédente obtenue sur une SS20 était presque linéaire car, sur cette machine, nous avons des coûts de communication inter-processus, et surtout de gestion de contextes, toujours plus grands que le coût des calculs des estimations locales. De ce fait, ce dernier coût était masqué par les autres. Avec $L_{init} = 4$, le gain que l'on obtient par rapport à la méthode conservatrice est de 1.8, ce qui correspond à un $TER_{comp} = 0.121$ et un pourcentage d'erreurs de 3.72%. C'est certainement le meilleur compromis entre la précision et le gain en vitesse.

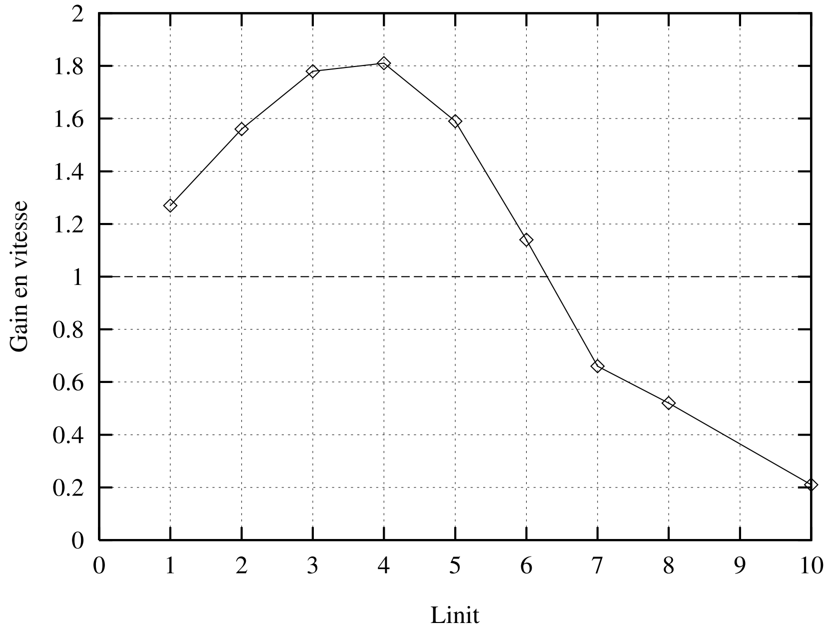


Figure 7.6 Gain en fonction de L_{init}.

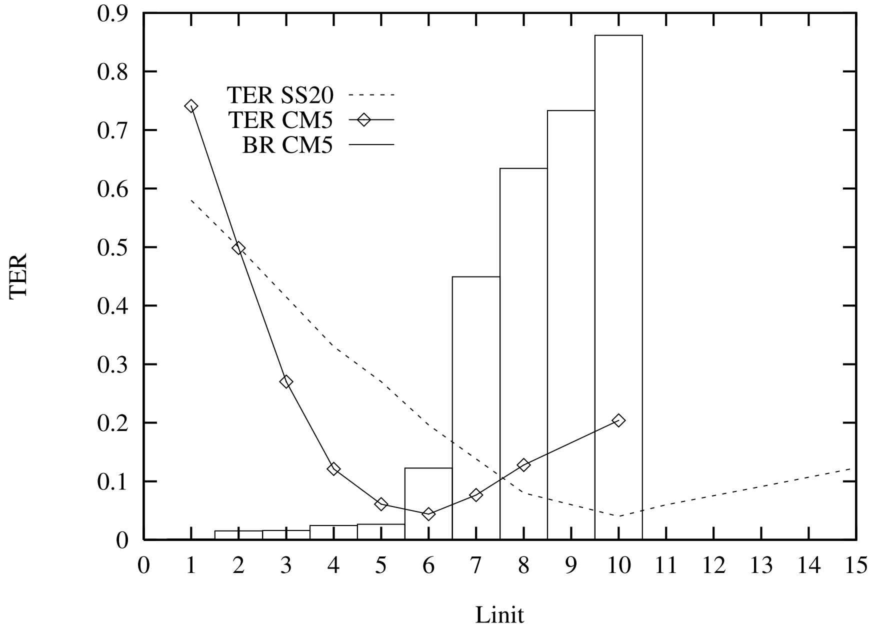


Figure 7.7 TER et BR en fonction de L_{init}.

La figure 7.7 compare le TER obtenu sur une SS20 (précédemment tracée) avec celui obtenu sur une CM-5 en fonction de L_{init}. La proportion de blocage (*Blocking Ratio, BR*) sur la CM-5, définie comme étant le rapport entre le temps total passé en attente et le temps total de la simulation, est également représentée. On observe une diminution beaucoup plus rapide du TER sur la CM-5 et un décalage de la valeur de L_{init} qui minimise la courbe du TER (la valeur minimale du TER reste cependant la même pour les deux environnements). Ces deux phénomènes s'expliquent par le faible coût des communications et par l'augmentation de la puissance de traitement qui contribuent tous deux au passage plus rapide des messages entre les processus logiques. Comme précédemment, le TER recommence à croître lorsque L_{init} est trop élevé. En ce qui concerne la proportion de blocage (BR), pour $L_{init} < 6$ le BR est inférieur à 3% ce qui donne un coût de blocage très faible. Par exemple, pour $L_{init} = 4$ (valeur optimale) nous avons un BR de seulement 2.45% pour une réduction du taux de fautes temporelles de 83% par rapport au cas purement optimiste ($L_{init} = 1$).

7.5.3 Augmenter la taille et la densité de messages

Nous avons ensuite fait varier la taille du réseau (N_s) pour étudier le comportement de l'algorithme. La figure 7.8 représente le TER en fonction de L_{init} pour une densité de messages $n_c = 10$ et 3 tailles de réseau : 10, 20 et 30 commutateurs.

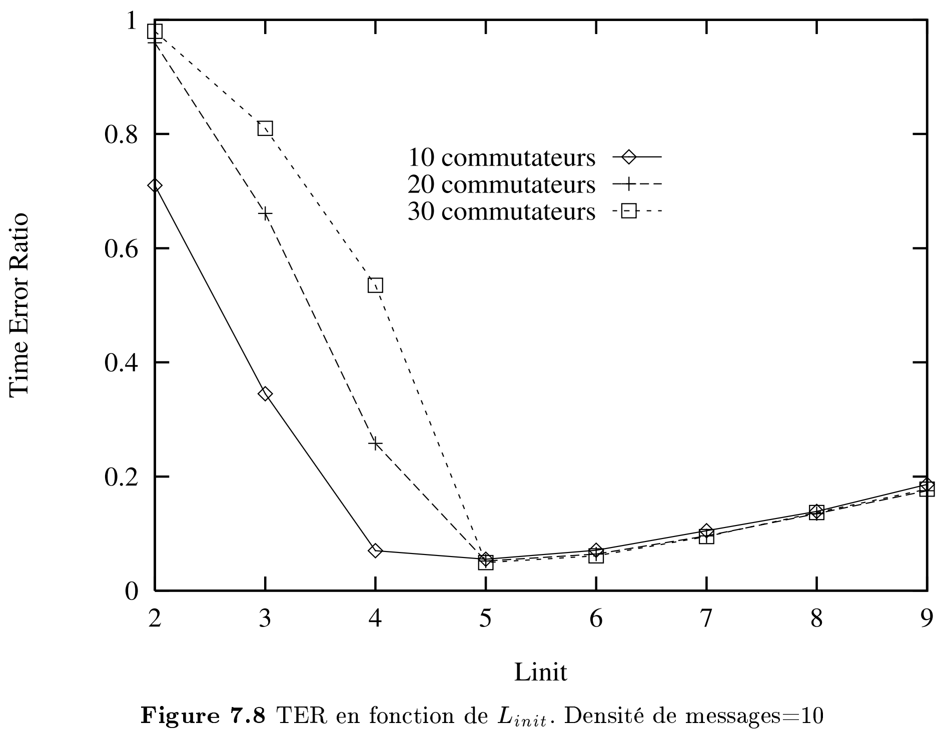


Figure 7.8 TER en fonction de L_{init}. Densité de messages=10

Les premiers résultats sont les suivant : premièrement, lorsque N_s augmente, la valeur optimale de L_{init} n'augmente pas de manière linéaire avec N_s même si le nombre d'entrées pour un *pl* augmente. Deuxièmement, il apparaît qu'accroître la taille du réseau permet à l'algorithme de réduire beaucoup plus rapidement le taux de fautes temporelles même pour des petites valeurs de L_{init}. L'avantage que l'on peut en tirer est une proportion de blocage très faible pour une réduction importante du taux de fautes. Cependant, l'inconvénient notable est qu'il est alors difficile de paramétrer l'algorithme de manière précise car le seuil initial L_{init} ne peut prendre que des valeurs entières. La figure 7.9 représente le gain par rapport à une méthode conservatrice.

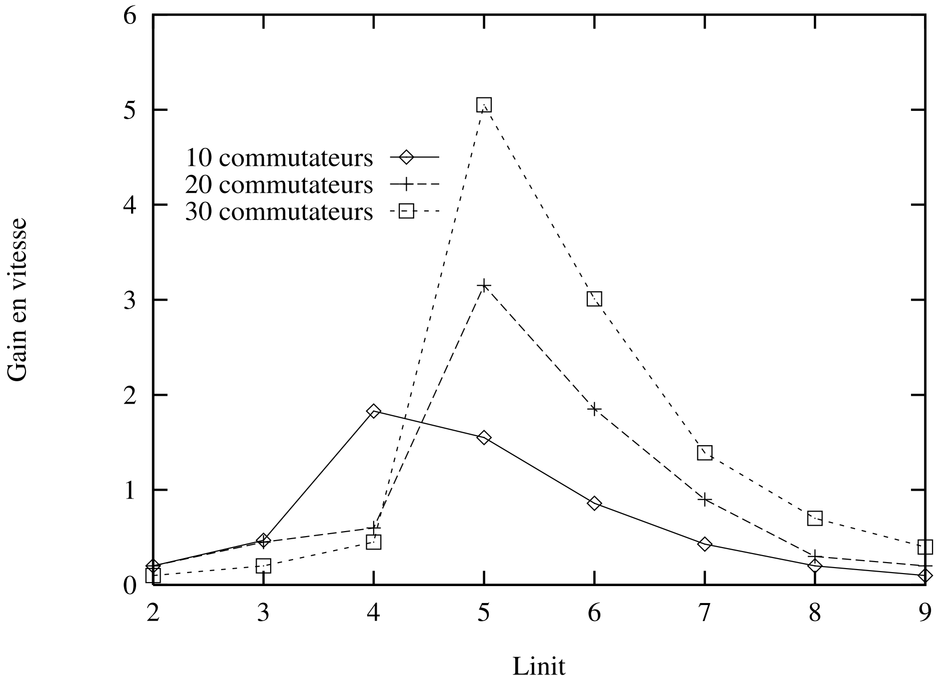


Figure 7.9 Gain en fonction de L_{init}. Densité de messages=10

On peut constater que ce gain est supérieur à celui précédemment obtenu pour 5 commutateurs et que le bénéfice de cette approche apparaît maintenant plus clairement. Si on augmente la densité de messages en prenant $n_c = 20$ on constate deux phénomènes relativement prévisibles qui sont d'une part un léger décalage de la valeur de L_{init} qui minimise le TER et d'autre part une courbe de TER plus régulière (figure 7.10) qui rend l'algorithme un peu moins instable. En ce qui concerne le gain, une densité de messages plus grande aurait tendance à améliorer les performances d'une méthode conservatrice grâce à la réduction du trafic de null-message, mais les résultats montrent des gains équivalents à l'expérience précédente (figure 7.11) et maximisés pour les valeurs de L_{init} qui minimisent le TER.

En ce qui concerne la précisions des résultats fournit par cette approche, on a pu constater que la meilleur précision est bien sûr fournie par les valeurs de L_{init} qui minimisent le TER. Dans toutes ces expériences, le pourcentage d'erreur avec ces valeurs de L_{init} n'excède pas 5%.

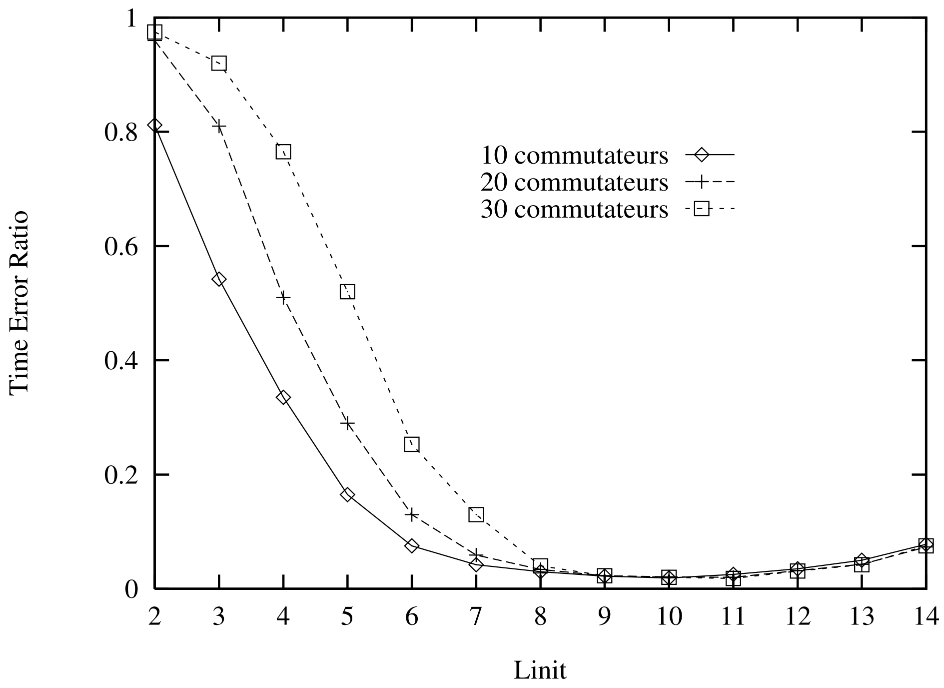


Figure 7.10 TER en fonction de L_{init}. Densité de messages=20

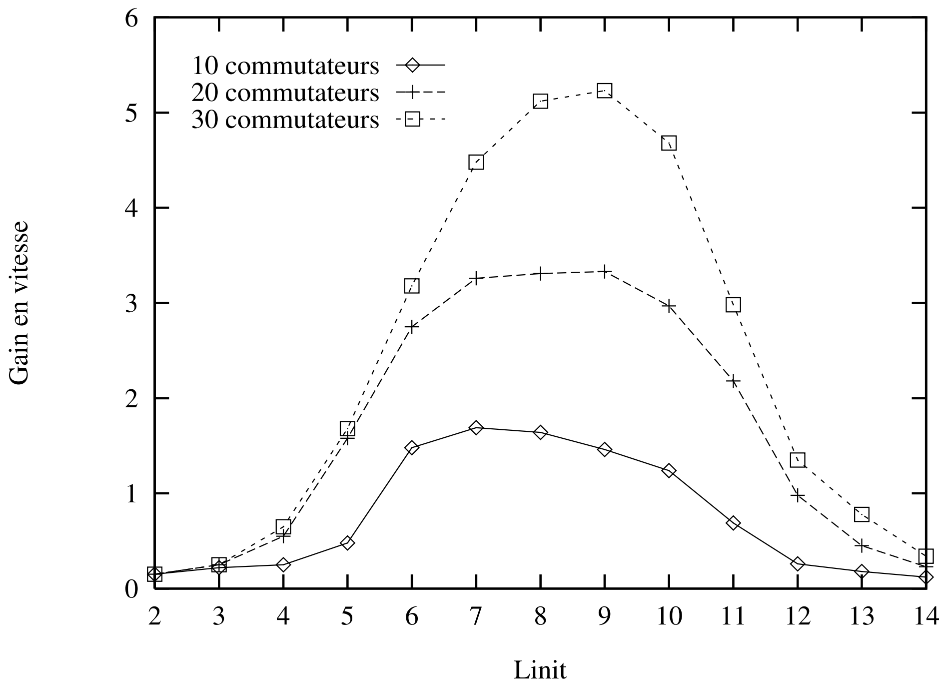


Figure 7.11 Gain en fonction de L_{init}. Densité de messages=20

7.6 CONCLUSION DU CHAPITRE

Dans ce chapitre, nous avons proposé une méthode simple de blocage qui se base sur la longueur de la file d'événements. Cette stratégie permet de réduire le nombre de fautes temporelles afin d'augmenter la précision des estimations locales. En simulant un réseau ATM complètement maillé de 5 commutateurs, nous obtenons un pourcentage d'erreurs inférieur à 5% et un gain par rapport à une méthode conservatrice de 1.8. Lorsque l'on augmente la taille du réseau, le gain augmente jusqu'à 5.2 pour 30 commutateurs. Le bénéfice de notre approche apparaît alors plus clairement puisque nous avons pu constater que la précision des résultats demeurait très bonne (erreur inférieure à 5%) pour ces valeurs de gains.

Nous avons trouvé que le seuil initial L_{init} ne doit pas être choisi trop grand car cela diminue considérablement le gain pour très peu d'améliorations sur la précision des résultats. De plus, de grandes valeurs pour L_{init} introduisent des effets de bord qui tendent à accroître le nombre de fautes temporelles en introduisant de mauvaises décisions de blocage. Une fois correctement paramétrée, cette simple stratégie de blocage réussie à réduire le nombre de fautes temporelles très efficacement. Les résultats expérimentaux montrent que l'algorithme est plus efficace lorsque les messages arrivent rapidement. Dans ce cas, la valeur optimale de L_{init} est relativement petite et de très légères modifications sur L_{init} peut entraîner une baisse, ou une hausse, notable du TER. Ce phénomène est très avantageux dans la mesure où il est dès lors possible de réduire très rapidement le TER avec une proportion de blocage très faible. Même si nous avons utilisé cet algorithme de blocage conjointement avec les estimations locales, il est suffisamment général pour être utilisé avec d'autres protocoles optimistes.

Nous pensons que l'adaptation de l'optimisme par blocage est une méthode prometteuse pour contrôler le flux d'événements dans des modèles de simulation qui présentent un haut degré de communication entre les *pls*. Ces méthodes ont de plus l'avantage de fonctionner lorsque la distance des fautes temporelles est faible, cas où une méthode basée sur une fenêtre échouerait.

Chapitre 8

Conclusion

L'objectif de cette thèse est d'étudier l'application des techniques de simulation parallèle et distribuée pour l'évaluation de performances des modèles de réseaux de communication. Nous pensons que cette évaluation sera un des problèmes majeurs dans les années qui viennent car les futurs réseaux seront complexes et de grande dimension. L'étude préalable est une étape indispensable à un déploiement réussi et il est souhaitable de pouvoir étudier le comportement du système entier pour répondre à des problèmes de dimensionnement et de conception.

La taille et la complexité des modèles que nous prévoyons dans un futur proche excluent l'utilisation des méthodes analytiques et des simulations séquentielles pour étudier le système dans son intégralité. Nous nous sommes donc tourné vers les possibilités offertes par la simulation parallèle et distribuée. Ce travail peut être décomposé en 3 étapes. La première a consisté en une étude générale pour appréhender le comportement des modèles que nous proposons sous le point de vue de la simulation parallèle et distribuée. En particulier, cette étape nous a permis de mieux renseigner nos modèles en terme de degré de parallélisme, de localité et de granularité, de *lookahead*, et de prévoir leurs comportements face à des mécanismes comme l'utilisation de fenêtres, les null-messages pour l'approche conservatrice, la sauvegarde des états, le *lazy-cancellation* et le *lazy re-evaluation* pour l'approche optimiste. Dans une seconde étape, nous nous sommes efforcé de construire un outil flexible et opérationnel pour simuler de manière conservatrice nos modèles de réseaux. Ce travail nous a permis d'acquérir une expérience et une compétence sur la simulation distribuée de modèles de réseaux ATM. Cette étape a également démontré que des gains significatifs peuvent être obtenus sur des topologies générales et des modèles complexes. Elle ouvre donc la voie vers l'exploitation réelle de ces techniques encore mal connues pour appréhender des systèmes trop grands pour pouvoir étudiés sinon.

La troisième étape s'est davantage situé dans le cadre des algorithmes et des mécanismes de simulation à proprement parlé. L'idée conductrice a été ici d'extraire les particularités de ces modèles pour optimiser les algorithmes de simulation. Dans cette optique, nous avons proposé une approche originale basée sur des estimations locales pour relâcher certaines contraintes de synchronisation. Cette démarche nous semble réalisable dans le cadre bien précis de la prospection sur des valeurs moyennes de performances dans les modèles de réseaux de communication. L'approche statistiquement correcte proposée permet de réaliser un compromis sur la précision des résultats avec l'avantage de réduire le temps de simulation. La notion de contrôle doit permettre de paramétrer le coût des synchronisations pour réaliser ce compromis entre vitesse et précision. Dans la continuité de cette proposition, nous avons également conçu une méthode de blocage basée sur la longueur de la file des événements en entrée. Cet algorithme préventif permet de réduire efficacement le nombre de fautes temporelles avec une proportion de blocage très faible pour les valeurs optimales de seuil. L'utilisation conjointe de ces deux techniques permet l'obtention de résultats précis avec des gains non négligeables par rapport à des méthodes conservatrices.

Une contribution importante de ce travail est d'avoir déterminé des directions concernant l'utilisation d'une méthode conservatrice plutôt qu'une méthode optimiste. Cela se confirme maintenant au vu des travaux réalisés par ailleurs. L'outil que nous avons développé nous permet de bien nous situer sur le plan international et nous pensons que dans un futur proche *simulation distribuée* et *réseaux ATM* seront deux mots-clés très souvent associés.

Les travaux que nous avons effectués peuvent être étendus dans différentes directions. En particulier, il nous semble très important d'utiliser les outils proposés ailleurs afin de parfaire notre expérience. Nous pensons au langage MAISIE proposé par l'équipe de UCLA qui permet l'utilisation de divers mécanismes de synchronisation. Nous pensons également au noyau WARPED proposé par l'université de Cincinnati pour valider nos prévisions concernant les méthodes optimistes. Par ailleurs, nos travaux sur les stratégies de blocage préventives méritent d'être poursuivis car ils sont suffisamment généraux pour être appliqués à d'autres environnements. L'outil que nous avons développé mérite également d'être étendu. En particulier, nous proposons d'ajouter des modèles supplémentaires afin de disposer d'un ensemble relativement complet pour la simulation distribuée de réseaux ATM. C'est une étape indispensable pour diffuser au mieux ces techniques distribuées encore mal perçues par l'ensemble des "simulationnistes". Enfin, nous pensons que dans cette perspective, il est nécessaire de proposer des mécanismes permettant, à partir d'un simulateur séquentiel existant, d'intégrer le parallélisme avec le moins de modifications possibles [8].

Publications

1. C. D. Pham and S. Fdida, "Relaxation of Synchronization Constraints in Parallel Simulation of ATM Networks", *Proceedings of the Anglo-French Workshop on Formal Methods, Modelling and Simulation for System Engineering, Dassault Electronique, St-Quentin en Yvelines, France, February 13-14 1995.*
2. C. D. Pham and S. Fdida, "The Statistically Correct Approach to Distributed Simulation of ATM Network", *Proceedings of the 6th IFIP Conference on Performance of Computer Networks, Istanbul, Turkey, October 23-26, 1995.*
3. C. D. Pham and S. Fdida, "Length-Based Blocking and Local Estimation in Distributed Simulation : A Case Study", *Proceeding of the 29th Annual Simulation Symposium Louisiana, New Orleans, April 8-11 1996.*
4. C. D. Pham, "Simulations Distribuées de Réseaux ATM", *Calculateurs Parallèles, Vol. 8(2), Juin 1996.*
5. C. D. Pham and S. Fdida, "Evaluation de Performance des Réseaux ATM : Etude et Perspective en Utilisant la Simulation Distribuée", *Actes de la Conférence Francophone sur l'Ingénierie des Protocoles, Rabat, Maroc, 14-17 octobre 1996.*
6. C. D. Pham, J. Essmeyer et S. Fdida,"Simulation of a Routing Algorithm using Distributed Simulation Techniques", *A paraître dans Proceedings of the Euro-Par'97 conference, August 26-29 1997.*
7. C. D. Pham et S. Fdida, "Perspectives in Performance Evaluation of Large ATM Networks", *A paraître dans Proceedings of the 5th IFIP Workshop on Performance Modelling and Evaluation of ATM Networks, July 21-23 1997.*
8. C. D. Pham, J. Essmeyer et S. Fdida, "A Conservative Simulator for Distributed Simulation of ATM Networks", *Soumis à Winter Simulation Conference, WSC'97.*
9. C. D. Pham, "Vitual Input Channels in Conservative Approach : Reducing the Difference between Distributed and Sequential Simulators", *Soumis à Winter Simulation Conference, WSC'97.*

Bibliographie

[Ahme94] H. Ahmed, L. Barriga and R. Ayani, "Parallel Discrete Event Simulation Using Space-Time Events", *Proceedings of the 2nd Euromicro Workshop on Parallel and Distributed Processing, Spain*

[Akyi93] I. F. Akyildiz, L. Chen, R. Das, R. M. Fujimoto and R. F. Serfozo, "The Effect of Memory Capaciy on Time Warp Performance", *Journal of Parallel and Distributed Computing, Vol. 18(4), August 1993, pages 411-422.*

[Ayan91] R. Ayani, "Parallel Discrete-Event Simulation on Shared Memory Multiprocessors", *International Journal in Computer Simulation, June 1991, pages 81-97.*

[Ayan92] R. Ayani and H. Rajaei, "Parallel Simulation Using Conservative Time Windows", *Proceedings of the 1992 Winter Simulation Conference, pp709-717.*

[Baez90] D. Baezner, G. Lomow and B. W. Unger, "Sim++ : The Transition to Distributed Simulation", *Proceeding of the 1990 SCS Multiconference on Distributed Simulation, San Diego, California, pages 211-218.*

[Bagr87] R. L. Bagrodia, K. M. Chandy and J. Misra, "A Message-based Approach to Discrete-Event Simulation", *IEEE Transactions on Software Engineering, pages 654-665.*

[Bagr94a] R. L. Bagrodia and W. T. Liao, "Maisie : A Language for the Design of Efficient Discrete-Event Simulations", *IEEE Transactions on Software Engineering, April 1994, pp225-238.*

[Bagr94b] R. L. Bagrodia, "Language Support for Parallel Discrete-Event Simulations", *Proceedings of the 1994 Winter Simulation Conference, Walt Disney World Swan Hotel, Lake Buena Vista, Florida, December 11-14, 1994, pp1324-1331.*

[Ball90] D. Ball and S. Hoyt, "The Adaptive Time-Warp Concurrency Control Algorithm", *Proceeding of the 1990 SCS Multiconference on Distributed Simulation, San Diego, California, pp174-177.*

[Bank84] J. Banks and J. S. Carson, "Discrete-Event System Simulation", *Prentice-Hall, 1984.*

[Baue93] H. Bauer and C. Sporrer, "Reducing Rollback Overhead in Time-Warp Based Distributed Simulation with Optimized Incremental State Saving", *Proceeding of the 26th Annual Simulation Symposium, March 1993, pp12-20.*

[Bell90] S. Bellenot, "Global Virtual Time Algorithms", *Proceeding of the SCS Multiconference on Distributed Simulation, San Diego, California, January 17-19, 1990, pages 122-127.*

[Bobi76] P. A. Bobillier, B. C. Kahan and A. R. Probst, "Simulation with GPSS and GPSS V", *, 1976.* Bull and INRIA, "QNAP2 Reference Manual", 84.

[Brat87] P. Bratley, B. L. Fox and L. E. Schrage, "A Guide to Simulation, Second Edition", *Springer-Verlag, 1987.* Bull and INRIA, "QNAP2 Reference Manual", 84.

[Brya77] R. E. Bryant, "Simulation of packet communications architecture computer systems", *Massachussets Institute of Technology, MIT-LTC-TR-188, 1977.*

[Cai90] W. Cai and S. J. Turner, "An Algorithm for Distributed Discrete-Event Simulation - The Carrier Null Message Approach", *Proceeding of the SCS Multiconference on Distributed Simulation, San Diego, California, pp3-8.*

[Chan79] K. M. Chandy and J. Misra, "Distributed Simulation : A Case Study in Design and Verification of Distributed Programs", *IEEE Transactions on Software Engineering, 1979, pp440-452.*

[Chan81] K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", *Communication of the ACM, Vol. 24(11), April 1981, pp198-206.*

[Chan83] K. M. Chandy, J. Misra and L. M. Haas, "Distributed Deadlock Detection", *ACM Transactions on Computer Systems, May 1983, pp144-156.*

[COMM95] "Issues and Challenges in ATM Networks", *C. o. t. ACM, Vol. 38(2), February 1995.*

[Dahl67] O. J. Dahl and K. Nygaard, "Simula : A Language for Programming and Description of Discrete Event Systems. Introduction and user's manual, 5th edition.", *, 1967.*

[Das93] S. R. Das and R. M. Fujimoto, "A Performance Study of the Cancelback Protocol for Time Warp", *Proceedings of 7th Workshop on Parallel and Distributed Simulation, San Diego, California, pp135-142.*

[DePr93] M. DePrycker, "Asynchronous Transfer Mode : Solution for Broadband ISDN", *Second edition, Ellis Horwood, 1993.*

[DeVr90] R. C. DeVries, "Reducing Null Messages in Misra's Distributed Discrete Event Simulation Method", *IEEE Transactions on Software Engineering, Vol. 16(1), January 1990, pp82-91.*

[Dick90] P. M. Dickens and P. F. Reynolds, "SRADS With Local Rollback", *Proceeding of the 1990 SCS Multiconference on Distributed Simulation, San Diego, California, January 17-19, 1990, pp161-164.*

[D'So94] L. M. D'Souza, X. Fan and P. A. Wilsey, "pGVT : An Algorithme for Accurate GVT Estimation", *Proceedings of 8th Workshop on Parallel and Distributed Simulation (PADS'94), Edinburg, Scotland, U. K., July 6-8, 1994, pages 102-109.*

[Earn92] R. W. Earnshaw and A. Hind, "A Parallel Simulator for Performance Modeling of Broadband Telecommunication Networks", *Proceedings of the 1992 Winter Simulation Conference, Arlington, Virginia, pp1365-1373.*

[Fers95] A. Ferscha, "Probabilistic Adaptive Direct Optimism Control in Time Warp", *Proceedings of 9th Workshop on Parallel and Distributed Simulation, Lake Placid, New York, June 14-16, 1995.*

[Fish94] P. A. Fishwick, "Computer Simulation : Growth through Extension", *Computer & Information Sciences Department, 1994.*

[Fish95] P. Fishwick, "Simulation Model Design & Execution : Building Digital Worlds", *Prentice-Hall, 1995.*

[Flei95] J. Fleischmann and P. A. Wilsey, "Comparative Analysis of Periodic State Saving Techniques in Time Warp Simulators", *Proceedings of 9th Workshop on Parallel and Distributed Simulation, Lake Placid, New York, June 14-16, 1995, pages 50-58.*

[Fuji88] R. M. Fujimoto, "Lookahead in Parallel Discrete Event Simulation", *Proceedings of the 1988 International Conference on Parallel Processing, pp34-41.*

[Fuji89] R. M. Fujimoto, "The Virtual Time Machine", *Proceedinds of the 1989 International Symposium on Parallel Algorithms and Architectures, pp199-208.*

[Fuji90a] R. M. Fujimoto, "Parallel Discrete Event Simulation", *Communication of the ACM, Vol. 33(10), October 1990, pp31-53.*

[Fuji90b] R. M. Fujimoto, "Performance of Time Warp under synthetic workloads", *Proceeding of the 1990 SCS Multiconference on Distributed Simulation, San Diego, California, pp23-28.*

[Gafni88] A. Gafni, "Rollback Mechanisms for Optimistic Distributed Simulation Systems", *Proceeding of the 1988 SCS Multiconference on Distributed Simulation, July 88, pages 61-67.*

[Gbur95] P. Gburzynski, T. Ono-Tesfaye and S. Ramaswamy, "Modeling ATM Networks in a Parallel Simulation Environment : A Case Study", *Proceedings of the SCSC'95 Conference, Ottawa, Ontario, Canada, July 24-26, 1995.*

[Geis93] A. Geist and al., "PVM 3 User's Guide and Reference Manual", *May 1993.*

[Gosw93] K. K. Goswami and R. K. Iyer, "Use of Hybrid and Hierarchical Simulation to Reduce Computation Costs", *Proceedings of the MASCOT'93 Conference, La Jolla, California, pp197-202.*

[Hamn94] D. O. Hamnes and A. Tripathi, "Investigations in Adaptive Distributed Simulation", *Proceedings of 8th Workshop on Parallel and Distributed Simulation (PADS'94), Edinburg, Scotland, U. K., pp20-23.*

[Hand94] R. Händel, M. N. Nuber and S. Schröder, "ATM Networks : Concepts, Protocols, Applications", *Addison-Wesley, 1994.*

[Heid84] P. Heidelberger and S. S. Lavenberg, "Computer Performance Evaluation Methodology", *IEEE Transactions on Computers, Vol. 33(12), December 1984, pages 1195-1220.*

[Hopp95] H.-C. Hoppe, P. Ossadnik and W. Stuttgen, "CM-PVM : An Efficient Implementation of PVM 3.3 for the CM-5", *Proceedings of the EuroPVM'95, Lyon, France, pages 53-58.*

[Hont89] P. Hontalas and B. Beckman, "Performance of the Colliding Puck Simulation on the Time Warp Operating System (Part 2 : A Detailed Analysis)", *Proceedings of the 1989 Summer Computer Simulation Conference, San Diego, California, pages 91-95.*

[Huit95] C. Huitema, "Le Routage dans l'Internet", *Eyrolles, 1995.*

[IBM71] IBM, "General Purpose Simulation System V Users's Manual, 2nd edition.", *IBM New-York, 1971.*

[Jaff82] J. M. Jaffe and F. H. Moss, "A Responsive Distributed Routing Algorithm for Computer Networks", *IEEE Transactions on Communications, Vol. 30(7), July 1982, pages 1758-1762.*

[Jain92] R. Jain, "The Art of Computer Systems Performance Analysis", *John Wiley, 1992.*

[Jeff85] D. R. Jefferson, "Virtual Time", *ACM Transaction on Programming Languages and Systems, Vol. 7(3), July 1985, pp405-425.*

[Jeff90] D. R. Jefferson, "Virtual Time II : Storage Management in Distributed Simulation", *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing, August 1990.*

[Jeff91] D. Jefferson and P. Reiher, "Supercritical Speedup", *The 24th Annual Simulation Symposium, New Orleans, Louisiana, pp159-168.*

[Jha94] V. Jha and R. L. Bagrodia, "A Unified Framework for Conservative and Optimistic Distributed Simulation", *Proceedings of 8th Workshop on Parallel and Distributed Simulation (PADS'94), Edinburg, Scotland, U. K., pp12-19.*

[Klei74] J. P. C. Kleijnen, "Statistical Techniques in Simulation, Part I & II", *D. B. Owen, 1974.*

[Kura96] A. Kuratti and W. H. Sanders, "A Changepoint Algorithm for Parallel Simulation", *Submitted to TOMACS.*

[Lamp78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM, July 1978, pp558-565.*

[Law84] A. M. Law and C. S. Larmey, "Introduction to Simulation Using SIMSCRIPT II.5", *CACI Products Company, 1984.*

[Law91] A. M. Law and W. D. Kelton, "Simulation Modelling & Analysis, Second Edition", *MacGraw Hill, 1991.*

[Lin90] Y. B. Lin and D. Lazowska, "Determining the Global Virtual Time in a Distributed Simulation", *Proceedings of the 1990 International Conference on Parallel Processing, ppIII :201-209.*

[Lin91] Y. B. Lin and E. D. Lazowska, "A Time-Division Algorithm for Parallel Simulation", *ACM Transactions on Modeling and Computer Simulation, Vol. 1(1), January 1991, pages 73-83.*

[Lin91] Y. B. Lin and B. R. Preiss, "Optimal Memory Management fr Time Warp Parallel Simulation", *ACM Transactions on Modeling and Computer Simulation, Vol. 1(4), October 1991, pages 283-307.*

[Lin93] Y. Lin, B. R. Preiss, W. M. Loucks and E. D. Lazowska, "Selecting the Checkpoint Interval In Time Warp Simulation", *Proceedings of 7th Workshop on Parallel and Distributed Simulation, San Diego, California, May 16-19, 1993, pp3-10.*

[Lin95] Y. Lin, "Estimating the Likelihood of Success of Lazy Cancellation in Time Warp Simulations", *To appear in the International Journal in Computer Simulation*

[Lipt90] R. J. Lipton and D. W. Mizell, "Time Warp versus Chandy-Misra : A Worst-Case Comparaison", *Proceeding of the 1990 SCS Multiconference on Distributed Simulation, San Diego, California, pp137-143.*

[Luba88] B. D. Lubachevsky, "The Bounded Lag Distributed Discrete Event Simulation", *Proceeding of the 1988 SCS Western Multiconference on Distributed Simulation, pp183-191.*

[Luba89a] B. Lubachevsky, A. Shwartz and A. Weiss, "Rollback Sometimes Works ... If Filtered", *Proceedings of the 1989 Winter Simulation Conference, Washington DC, pp630-639.*

[Luba89b] B. D. Lubachevsky, "Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks", *Communication of the ACM, Vol. 32(1), January 1989, pp111-123.*

[Mart96] D. E. Martin, T. J. McBrayer and P. A. Wilsey, "WARPED : A Time Warp Simulation Kernel for Analysis and Application Development", *Proceedings of the 29th Hawaii International Conference on System Sciences, Maui, Hawaii, January, 1996.*

[Mats93] Y. Matsumoto and K. Taki, "Adaptive Time-Ceiling for Efficient Parallel Discrete Event Simulation", *Proceedings of the 1993 Object-Oriented Simulation Conference, pp101-106.*

[Misr86] J. Misra, "Distributed Discrete Event Simulation", *Computing Surveys, Vol. 18(1), March 1986, pp39-65.*

[More93] Y. Moret, "Création et Validation d'un Modèle Hybride", *Laboratoire MASI, Rapport de stage, 1993.*

[Mouf90] T. Mouftah and R. T. Sturgeon, "Distributed Discrete Event Simulation for Communications Networks", *IEEE Journal on Selected Areas in Communication, Vol. 8(9), December 1990, pp1723-1734.*

[Muss94] P. Mussi and H. Rakotoarisoa, "PARSEVAL : Workbench for Queueing Networks Parallel Simulation", *Inria Sopia Antipolis, Rapport No. 2234, 1994.*

[Niko94] I. Nikolaidis, R. Fujimoto and C. A. Cooper, "Time-Parallel Simulation of Cascaded Statistical Multiplexers", *Proceedings of the ACM Sigmetrics 1994, Nashville, Tennessee, pp231-240.*

[OPNET93] MIL3 Inc., "OPNET : OPtimized Network Engineering Tool, Version 2.4a", *MIL3 Inc., 1993.*

[Pala93] A. C. Palaniswamy, S. Aji and P. A. Wilsey, "Performance Measures for Several Optimizations to a Distributed Digital System Simulator", *Proceeding of the 26th Annual Simulation Symposium, pp21-29.*

[Prak91] A. Prakash and R. Subramanian, "Filter : An Algorithm for Reducing Cascaded Rollbacks in Optimistic Distributed Simulations", *The 24th Annual Simulation Symposium, New Orleans, Louisiana, pp123-132.*

[Pres89] M. Presley, M. Ebling, F. Wieland and D. Jefferson, "Benchmarking the Time Warp Operating System with a Computer Network Simulation", *Proceeding of the 1988 SCS Multiconference on Distributed Simulation, San Diego, California, pages 8-13.*

[QNAP84] Bull and INRIA, "QNAP2 Reference Manual", *Bull et Inria, 1984.*

[Radh96] R. Radhakrisnhan, T. J. McBrayer, K. Subramani, M. Chetlur, V. Balakrisnan and P. A. Wilsey, "A Comparative Analysis of Various Time Warp Algorithms Implemented in the WARPED Simulation Kernel", *Proceedings of the 29th Annual Simulation Symposium, Fairmont Hotel, New Orleans, Louisiana, USA, April 8-11, 1996, pages 107-116.*

[Raja92] H. Rajaei, "SIMA : An Environment for Parallel Discrete-Event Simulation", *Proceedings of the 25th Annual Simulation Symposium, pages 147-155.*

[Raja93] H. Rajaei, R. Ayani and L. E. Thorelli, "The Local Time Warp Approach to Parallel Simulation", *Proceedings of 7th Workshop on Parallel and Distributed Simulation, San Diego, California, pp119-126.*

[Reih89] P. L. Reiher and D. Jefferson, "Limitation of Optimism in the Time Warp Operating System", *Proceedings of the 1989 Winter Simulation Conference, Washington DC, pp765-769.*

[Reih90] P. L. Reiher, "Parallel Simulation Using the Time Warp Operating System", *Proceedings of the 1990 Winter Simulation Conference, pages 38-45.*

[Reyn88] P. F. Reynolds, "A Spectrum of Options for Parallel Simulation", *Proceedings of the 1988 Winter Simulation Conference, December 1988, pp325-332.*

[Rönn94a] R. Rönngren and R. Ayani, "Adaptive Checkpointing in Time Warp", *Proceedings of 8th Workshop on Parallel and Distributed Simulation (PADS'94), Edinburg, Scotland, U. K., July 6-8, 1994, pages 110-117.*

[Rönn94b] R. Rönngren, H. Rajaei, A. Popescu, M. Liljenstam, Y. Ismailov and R. Ayani, "Parallel Simulation of a High Speed Lan", *Proceedings of 8th Workshop on Parallel and Distributed Simulation (PADS'94), Edinburg, Scotland, U. K., pp132-138.*

[Sama85] B. Samadi, "Distributed Simulation : Algorithms and Performance Analysis", *PhD Thesis, Computer Science Department, UCLA, Los Angeles, 1985.*

[Shan90] K. S. Shanmugan and W. Larue, "Modeling and Simulation of Communication Networks using the Block Oriented Network Simulator", *Proceedings of the System Design and Networking Conference, Santa Clara, California*

[Smit82] A. J. Smith, "Cache Memories", *Computing Surveys, Vol. 14, 1982, pages 473-530.*

[Soko89] L. M. Sokol, B. K. Stucky and V. S. Hwang, "MTW : A Control Mechanism for Parallel Discrete Simulation", *Proceedings of the International Conference on Parallel Processing, 1989, pp250-254.*

[Srin93] S. Srinivasan and J. P. F. Reynolds, "Non-Interfering GVT Computation via Asynchronous Global Reductions", *Proceedings of the 1993 Winter Simulation Conference, Los Angeles, California, December 12-15, 1994, pages 740-749.*

[Srin95a] S. Srinivasan and P. F. Reynolds, "NPSI Adaptive Synchronization Algorithms for PDES", *Proceedings of the 1995 Winter Simulation Conference.*

[Srin95b] S. Srinivasan and P. F. Reynolds, "Adaptive Algorithms vs. Time Warp : An Analytical Comparaison", *Proceedings of the 1995 Winter Simulation Conference.*

[Stei91] J. S. Steinman, "Interactive SPEEDES", *The 24th Annual Simulation Symposium, New Orleans, Louisiana, April 1-5,1991, pp149-158.*

[Stei93a] J. S. Steinman, "Breathing Time Warp", *Proceedings of 7th Workshop on Parallel and Distributed Simulation, San Diego, California, pp109-118.*

[Stei93b] J. S. Steinman, "Incremental State Saving in SPEEDES Using C++", *Proceedings of the 1993 Winter Simulation Conference.*

[Stei93c] J. S. Steinman, "Synchronization of Parallel and Distributed Interactive Military Simulations Using SPEEDES", *Proceedings of the 1993 Summer Computer Simulation Conference, Lafayette Hotel, Boston, Massachusetts, USA, July 19-21, 1993, pages 701-710.*

[Stei95] J. S. Steinman, C. A. Lee, L. F. Wilson and D. M. Nicol, "Global Virtual Time and Distributed Synchronization", *Proceedings of 9th Workshop on Parallel and Distributed Simulation, Lake Placid, New York, June 14-16, 1995, pages 139-148.*

[Tall91] F. Tallieu and F. Verboven, "Using Time Warp for Computer Network Simulations on Transputers", *The 24th Annual Simulation Symposium, New Orleans, Louisiana, April 1-5, 1991, pp112-117.*

[Theo94] M. Theofanos, "Distributed Simulation of Queuing Networks", *Master's Thesis, The University of Virginia, January 1984.*

[Toml93] A. I. Tomlison and V. K. Garg, "An Algorithm for Minimally Latent Global Virtual Time", *Proceedings of 7th Workshop on Parallel and Distributed Simulation, San Diego, California, May 16-19, 1993, pages 35-42.*

[Wagn89] D. B. Wagner and E. D. Lazowska, "Parallel Simulation of Queuing Networks : Limitations and Potentials", *Proceedings of the ACM Sigmetrics 1989, pages 146-155.*

[Wang92] J. J. Wang and M. Abrams, "Approximate Time-Parallel Simulation of Queuing Systems With Losses", *Proceedings of the 1992 Winter Simulation Conference, pages 700-708.*

[Wang93] J. J. Wang and M. Abrams, "Determining Initial States for Time-Parallel Simulations", *Proceedings of 7th Workshop on Parallel and Distributed Simulation, San Diego, California, May 16-19 1993, pages 19-26.*

[West88] D. West, "Optimizing Time Warp : Lazy Rollback and Lazy Reevaluation", *PhD Thesis, MS Thesis, 1988.*

[Wiel89] F. Wieland, L. Hawley, A. Feinberg, M. D. Loreto, L. Blume, J. Ruffles, P. Reiher, B. Beckman, P. Hontalas and S. Bellenot, "The Performance of a Distributed Combat Simulation with the Time Warp Operating System", *Concurrency : Practice and Experience, pages 35-50.*

[Wood94] K. R. Wood and S. J. Turner, "A Generalized Carrier-Null Method for Conservative Parallel Simulation", *Proceedings of 8th Workshop on Parallel and Distributed Simulation (PADS'94), Edinburg, Scotland, U. K., pp50-57.*

[Xiao95] Z. Xiao, F. Gomes, B. Unger and J. Cleary, "A Fast Asynchronous GVT Algorithm for Shared Memory Multiprocessor Architectures", *Proceedings of 9th Workshop on Parallel and Distributed Simulation, Lake Placid, New York, June 16-16, 1995.*