Protocole d'appel de multiprocédure à distance dans le système Gothic : définition et mise en oeuvre.

Christine Morin

Numéro d'ordre : 498

THESE

présentée devant

l'Université de Rennes 1

Institut de Formation Supérieure en Informatique et Communication

pour obtenir

Le Titre de Docteur de l'Université de Rennes I

Mention INFORMATIQUE

par

Christine MORIN

Titre de la thèse

Protocole d'appel de multiprocédure à distance dans le système Gothic : définition et mise en œuvre

Soutenue le 18 décembre 1990 devant la commission d'examen :

MM. :	Jean-Pierre	BANÂTRE	**Président**	
	Michel	DIAZ	**Rapporteur**	
	Claude	JARD	**Rapporteur**	
	Roland	BALTER		**Examinateurs**
	Michel	BANÂTRE		
	Jean	CAMILLERAPP		
	Liba	SVOBODOVA		
	Jean-Pierre	VERJUS		

Numéro d'ordre : 498

THESE

présentée devant

l'Université de Rennes 1

Institut de Formation Supérieure en Informatique et Communication

pour obtenir

Le Titre de Docteur de l'Université de Rennes I

Mention INFORMATIQUE

par

Christine MORIN

Titre de la thèse

Protocole d'appel de multiprocédure à distance dans le système Gothic : définition et mise en œuvre

Soutenue le 18 décembre 1990 devant la commission d'examen :

MM. :	Jean-Pierre	BANÂTRE	**Président**	
	Michel	DIAZ	**Rapporteur**	
	Claude	JARD	**Rapporteur**	
	Roland	BALTER		**Examinateurs**
	Michel	BANÂTRE		
	Jean	CAMILLERAPP		
	Liba	SVOBODOVA		
	Jean-Pierre	VERJUS		

Année universitaire **1989-1990**

UNIVERSITE de RENNES I

U.E.R. SCIENCES et PHILOSOPHIE

DOYENS HONORAIRES

M. LE MOAL H.
M. MARTIN Y.
M. BOCLE J.

PROFESSEURS HONORAIRES

M. FREYMANN R.
M. ROHMER Y.
M. SALMON-LEGAGNEUR F.
M. VALET P.
M. PHILIPPOT A.
Mlle CHARPENTIER M.
M. VACHER M.
M. VILLERET S.
M. ORTIGUES E.

M. LE MOAL H.
Mlle DURAND S.
M. LE BOT J.
M. MARTIN Y.
M. RAZET P.
M. MAILLET P.
M. ALLEGRET P.
Mlle GOAS M.
Mlle GOAS G.
M. CLAUSTRES G.
M. FOLLIOT R.
M. MEVEL J.

MAITRES de CONFERENCES HONORAIRES

Mlle HAMON M.R.

MATHEMATIQUES

Professeurs

M. BERTHELOT P.
M. CALOZ G.
M. CAMUS J.
M. CERVEAU O.
M. CONZE J.P.
Mme COSTE-ROY M.F.
M. CROUZEIX M.
M. DESHAYES J.
M. FERRAND D.
M. GIORGIUTTI I.
M. GUERINDON J.
M. HENNION H.
M. HOUDEBINE J.
M. MAHE L.
M. MEMIN J.
M. METIVIER G.
M. MIGNOT A.
M. MORET-BAILLY L.
M. PETRITIS D.
M. RAUGI A.
M. TOUGERON J. Cl.
M. TOURNEMINE G.
M. WOLF J. (ENSAT L.)
M. LERNER N.

Habilités

M. MERRIEN J.
M. PERRIN G.
Mme TOUGERON M.

Docteurs d'Etat

M. BOSSARD Y.
M. GRAVEREAUX J.B.

INFORMATIQUE

Professeurs

Mme ANDRE F.
M. KOTT L.
M. LENFANT J.
M. LERMAN I.
M. MARIE R.
M. RAOULT J. C.
M. RAYNAL M.
M. SEGUIN J. (ENSSAT L.)
M. BOSC P. (ENSSAT L.)
Mme CORDIER M.O.
M. JALBY W.
M. HELARY J.M.
M. LORETTE G.
M. DIVAY M. (IUT L.)

Habilités

M. BOUATOUCH K.

Docteurs d'Etat

M. TALLUR B.
Mme MORIN A.M.

PHYSIQUE

Professeurs

M. ARQUES P.Y.
M. BARON A. (IUT R.)
M. BENIERE F.
M. BERTEL L.
M. BRUN P.
M. CAILLEAU H.
M. COLIN Y. (IUT R.)
M. CORAZZA M. (ENSSAT L.)
M. DANIEL J.P.
M. DECAMPS E.A.
M. DUBOST G.
M. DURAND A.
M. FOUCHE G. (IUT R.)
M. FUCHS J.J.
M. GROSVALD (IUT R)

M. GUIDINI J.
M. HAEUSLER Cl.
M. JOUBERT P. (IUT L.)
M. LE FLOCH A.
M. LE MEN J.F. (IUT L.)
M. LEROUX EX
M. LE TRAON A.
M. LEVASSEUR M. (IUT R.)
M. MALHERBE J.C. (ENSSAT L.)
M. MEINNEL J.
M. NUSIMOVICI M.
M. RIAUX E. (IUT R.)
M. STEPHAN G. (ENSSAT L.)
M. TERRET Cl.
M. THOMAS G.
M. VEZZOSI G.
M. BONNAUD O.
M. BERTHAULT M.
M. BIDEAU D.
M. JEZEQUEL G.
M. LE DOUCEN R.

Habilités

M. COATRIEUX J.L.
M. CHARBONNEAU G.
M. COLLOREC R. (IUT R.)
M. ECOLIVET C.
M. GIRARD A.
M. LANGOUET L.
M. MESSAGER J.C.
M. PILET J.C.

Docteurs d'Etat

M. ANDRIAMIRADO (IUT L.)
M. BALCOU Y.
M. BERNARD D.
M. BESNIER G.
M. BOULIOU A.
M. CHAGNEAU (IUT R.)
Mme COUSIN C.
M. BOULIOU A.
M. CLEC'H G. (IUT L.)
M. DAUDE A.
M. DEFRANCE A.
M. FORTIN B. (IUT R.)
M. DEFRANCE A.
M. GOMET J.C.
M. GOULPEAU L.
M. HAGENE B.
Mlle HAGENE M.
M. HOUDEAU J.P.
M. JEZEQUEL G.

M. LARVOR M.
M. LE BLOA A. (IUT R.)
M. LE CLEAC'H (IUT L.)

M. LE COMTE A.
M. LE DOUCEN R.
M. LENORMAND J. M.

M. POEY P.
M. PRIOL M.
M. QUEFFELEC J. L.
M. RABACHE P. (IUT L.)
M. RAOULT F. (IUT R.)
M. REBOURS B. (IUT L.)
M. RIYET Y. (IUT L.)
M. SEIGNAC A.
M. TANGUY P.
M. TACHE D.
M. THOUROUDE D.
M.T'KINT de ROODENBEKE A. (IUT R.)
Mme T'KINT de RRODENBEKE M. (IUT R.)
M. TROADEC JP (IUT R.)

Docteurs d'Université

CHIMIE

Professeurs

M. BARIOU B. (IUT R.)
M. BOTREL A. (ENSCR)
M. BRAULT A. (IUT R.)
M. CARRIE R.
M. DABARD R.
M. DIXNEUF P.
M. FOUCAUD A.
M. GRANDJEAN D.
M. GUERILLOT Cl.
M. HAMELIN J.
M. LAURENT Y.
M. LE CORRE M.
M. LE GUYADER M.
M. LISSILOUR R.
M. LUCAS J.
M. MARTIN G. (ENSCR)
M. MAUNAYE M. (ENSCR)
M. PATIN H. (ENSCR)
M. ROBERT A.
M. SARRAZIN J.C.
M. SOYER N. (IUT R.)
M. TALLEC A.
M. VERDIER P. (IUT R.)

M. CAREL Cl.
M. DANION D.
M. DARCHEN A. (ENSCR)
M. GUERIN P. (ENSCR)
M. MOINET Cl.
M. POULAIN M.
M. LE GUYADER J.
M. SAILLARD J.Y.
M. DORANGE G. (ENSCR)

M. CARO B. (IUT L.)
M. PRIGENT Y. (IUT R.)

Habilités

M. CAILLET P.

M. JOUCLA M.
M. LAPLANCHE A.(ENSCR)

M. PRIGENT Y. (IUT R.)

M. TONNARD F. (IUT R.)

Docteurs d'Etat

M. AUFFREDIC J.P.
Mme BARS née BEAULIEU
Mme DANION née BOUGOT
Mme LE ROUZIC née BELLEVRE (ENSCR)
Mme TEXIER née BOULLET
M. BROCHU R.

Mme POMMERET née CHASLE (IUT R.)
M. CORRE F.

M. FAYAT Ch.
M. GADREAU Cl.
M. GAUDE J.
Mme LOUER née GAUDIN
M. GUILLEVIC J.
M. HAZARD R.
M. HERCOUET A.
Mme PAPILON née JEGOU (IUT R.)
M. LEBORGNE G.
M. LE COQ A.
M. LE FLOCH Y(ENSCR)
Mme UTJES née LE GALL
Mme RIVET née LE GUELLEC
Mlle LE PLOUZENNEC M
M. MARTELLI J.
M. MEYER A.

M. MORVAN J (ENSCR)
M. PERSON H.
Mme de COURVILLE née PICHEVIN A.
M. PICOUAYS B.
M. PLUSQUELLEC D. (ENSCR)
M. POCHAT F.
M. RAOULT E.
M. RAPHALEN D (ENSCR)
M. RAULET Cl.
Mme CARLIER née ROLLAND (IUT R.)

Mme TEXIER Fr.
M. VENIEN F.(ENSCR)

Mlle LARPENT C. (ENSCR)
M. LE CLOIREC P. (ENSCR)

BIOCHIMIE - BIOLOGIE CELLULAIRE et MOLECULAIRE

Professeurs	Habilités	Docteurs d'Etat
M. BLANCO C.	M. JEGOU B.	
M. BOISSEAU C.		
M. DUVAL J.		M. BOUTRY Jean-Luc (ENSCR)
M. GOURANTON J.		M. COILLOT J.P.
M. JEGO P.		M. GOURRET J.P.
M. JOLY J.M.		
M. KERCRET H.		M. HAMON Cl.
M. LE PENNEC J.P.		
M. PHILIPPE M.		M. LE GUELLEC R.
M. VALOTAIRE Y.		Me LE GUELLEC C.
M. WROBLEWSKI H.		

PHYSIOLOGIE - BIOLOGIE des ORGANISMES et des POPULATIONS

Professeurs	Habilités	Docteurs d'Etat
M. BARBIER R.		M. BERNARD J.
M. CALLEC J.J.		M. BERNARD Th.
M. CITHAREL J.		M. BERTRU G.
M. DAGUZAN J.		M. BRIENS M.
M. GAUTIER J.Y.		M. CANARD A.
M. HUON A.		M. CHAUVIN G.
M. LARHER F.		M. CLEMENT B.
Mme LEMOINE C.		M. COILLOT J.P.
M. NENON J.P.		M. DENIS Ch.
M. TOUFFET J.L.		Mlle FORGEARD F.
M. TREHEN		M. GLOAGUEN J.Cl.
M. WEBB D.		M. GOURRET J.P.
		M. GUILLET J. Cl.
		M. GUYOMARC'H J. Ch.
		Mme HUBERT née GUERGADY
		M. LE GARFF B.
		M. MICHEL R.
		MMe RICHARD M.A.
		Mlle ROZE F.
		M. SAVOURE B.
		Mme GUYOMARC'H née COUSIN C.

GEOLOGIE

Professeurs	Habilités	Docteurs d'Etat
M. BRUN J.P.		
M. CHAUVEL J.J.	M. MARTIN H.	M. BLAIS S.
M. CHOUKROUNE P.		Mme ESTEOULE née CHOUX
M. HAMEURT J.		M. COGNE J. P.
M. JAHN B.M.		
M. LARDEUX H.		M. HENRY J.L.
M. WILLAIME Ch.		Mme MORZADEC née KERFOURN
M. AUVRAY B.		M. LAGARDE J. L.
M. FOURCADE S.		M. LE CORRE Cl.
M. GILET Ph.		M. LEFORT J.P.
		M. MORZADEC P.
		Mme OLLIVIER née PIERRE

PHILOSOPHIE

Professeurs	Habilités	Docteurs d'Etat
M. CLAIR A.		
M. FOLSCHEID D.		
M. NEF F.		
M. VETO M.		

PERSONNEL C.N.R.S.

Directeurs de Recherche	Habilités	Docteurs d'Etat
MATHEMATIQUES		
INFORMATIQUE		
M. QUINTON P.	M. DARONDEAU P.	M. BASSEVILLE M.
CHIMIE		
M. CHEVREL R.	M. LAPINTE CL.	M. DEMERSEMAN B.
M. DENIS J.M.	M. SIMONNEAUX G.	Mme BAUDY née FLOC'H
M. COEURET (ENSCR)		M. FONTENEAU G.
M. GREE R. (ENSCR)		M. GUYADER J.
M. LOUER D.		M. HAMON J.R.
M. SERGENT M.		M. HERCOUET A.
M. SIMONET J.		M. LE BOZEC H.
		M. MARCHAND R.
		M. MARTIGNY P.
		M. MATECKI M.
		M. MOREL G.
		M. NOEL H.
		M. PADIOU J.
		M. PENA O.
		M. PERRIN A.
		Mme PERRIN C.
		M. POTEL M.
		Mme RAULT-BERTHELOT
		M. VAULTIER M.
PHYSIQUE		
		M. DANG TRAN Q.
		M. SANQUER M.
		M. TOUDIC B.
GEOLOGIE		
M. CAPDEVILA R.		M. BERNARD-GRIFFITHS J
M. COBBOLD P.		M. PARIS F1.
		M. PEUCAT J.J.
		M. ROBARDET M.
ANTHROPOLOGIE		
M. BRIARD J.		M. MONNIER J.C.

BIOLOGIE CELLULAIRE et GENETIQUE

Mlle GARNIER D.

M. OSBORNE B.

M. CHARBONNEAU M.
M. THIEULAND M. L.
M. THOMAS Daniel

BIOLOGIE des ORGANISMES

Mme GAUTIER A.
M. GAUTIER J. P.
M. VANCASSEL R.

Mme BAILIOT-DELEPORTE S.
Mme CLOAREC A.
Mme DELORTE S.
M. DELETTRE Y.
Mlle RIVAULT C.
Mle EYBERT M.

M. VIDAL J.M.

Docteurs d'Université

CHIMIE

M. BONDON A. (ENSCR)
Mme GUERCHAIS V. (ENSCR)

à mes parents,
à Béatrice,
à Denis.

Remerciements

J'exprime ma sincère reconnaissance à Jean-Pierre Banâtre, nouveau directeur de l'Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA) et professeur à l'Institut National des Sciences Appliquées (INSA) de Rennes pour m'avoir accueillie dans l'équipe Langages et Systèmes Parallèles (LSP) à l'IRISA. Son soutien attentif, sa disponibilité, la motivation qu'il a sû me communiquer ont contribué à la réussite de cette étude.

Mes remerciements sont aussi particulièrement adressés à Michel Banâtre, nouveau responsable de l'équipe LSP, directeur de recherche à l'INRIA, pour l'intérêt qu'il a porté à ce travail. Ses critiques et remarques pertinentes furent d'un appoint essentiel.

J'exprime ma profonde gratitude à Michel Diaz, directeur de recherche au Laboratoire d'Automatique et d'Analyse des Systèmes (LAAS) à Toulouse, et à Claude Jard, chargé de recherche au CNRS pour avoir accepté d'être les rapporteurs de cette thèse.

Je tiens à remercier Liba Svobodova, ingénieur au centre de recherche IBM de Zürich pour l'honneur qu'elle me fait en participant à ce jury.

Roland Balter, directeur du centre de recherche Bull à Grenoble, Jean Camillerapp, professeur à l'INSA de Rennes et Jean-Pierre Verjus, professeur à l'Université de Grenoble ont accepté de faire partie de ce jury. Je les en remercie vivement.

Mes remerciements vont aussi aux membres de l'équipe LSP. Je tiens à remercier plus particulièrement :

- Bruno Rochat pour son esprit d'équipe et l'aide généreuse qu'il m'a apportée,
- Philippe Joubert et Philippe Lecler pour leur grande disponibilité à mon égard,
- Gilles Muller pour m'avoir initiée à la pratique des systèmes d'exploitation ainsi que pour sa participation à la démonstration,
- Béatrice Michel pour ses encouragements.

Je remercie vivement Yves Prunault, Christiane Zimmermann et Maurice Saulnier de l'atelier micro de l'IRISA qui se dévouent à la maintenance de nos machines.

Enfin, je tiens très fort à remercier mon mari, Denis Martin, qui par son soutien sans faille et ses encouragements multiples a véritablement contribué à la bonne marche de cette étude. J'adresse toute ma reconnaissance à ma sœur, Béatrice, pour avoir veillé à mes moments de détente, à mes parents pour leur soutien chaleureux et à mes beaux-parents pour leur compréhension.

Chapitre I

Introduction

Un système informatique distribué est constitué d'un ensemble de postes de travail et de périphériques reliés entre eux par un système de communication. L'histoire des systèmes distribués est étroitement liée à l'évolution des outils de communication.

Des systèmes géographiquement répartis existent depuis la fin des années cinquante mais ces systèmes étaient entièrement dédiés à une seule application (par exemple réservation de places d'avion dans le système SABRE) qui assurait entièrement la gestion des communications. Les premiers projets de réseaux à grande distance ont été développés dans le courant des années soixante. Ils visaient à fournir un système de communication autonome. Le réseau Arpanet a été la base du développement des premiers protocoles notamment pour le transfert de fichiers.

L'apparition vers le milieu des années soixante-dix du réseau Ethernet[1] [Metc76], réseau local à haut débit utilisant un réseau à diffusion, marque une étape importante : premiers postes de travail individuels dotés de facilités graphiques, organisation client-serveur pour l'accès aux ressources partagées.

Parallèlement se sont développés des réseaux utilisant une structure en anneau. Les premiers systèmes répartis utilisant les réseaux locaux, qui apparaissent à la fin des années soixante-dix, sont réalisés en interconnectant plusieurs systèmes homogènes (notamment des systèmes Unix [2]). La plupart de ces systèmes étendent simplement le système de fichiers pour offrir un accès transparent aux fichiers locaux ou distants ; quelques uns permettent la création et l'exécution de processus distants.

Ce n'est qu'au début des années quatre-vingt qu'apparaissent les systèmes distribués intégrés conçus au départ comme répartis. Ces systèmes donnent alors l'impression aux usagers qu'ils disposent d'un système d'exploitation aussi souple que les systèmes à temps partagé mais avec de nouvelles possibilités.

[1] Ethernet est une marque déposée de XEROX

[2] Unix est une marque déposée des laboratoires AT & T.

De nombreuses applications distribuées ont été développées pour s'exécuter sur les systèmes distribués. Pour le programmeur d'applications distribuées se pose néanmoins un problème nouveau : celui de la communication.

La gestion des communications est une charge pour le programmeur du fait des imperfections de la transmission des messages (perte par exemple) et des défaillances possibles des sites du réseau de communication. Le développement de protocoles de communication fiable permet alors de décharger le programmeur de la lourde tâche du traitement des erreurs de transmission et des défaillances susceptibles de survenir pendant une communication.

L'un de ces protocoles est l'appel de procédure à distance qui est l'extension naturelle de l'appel de procédure dans le cadre des applications distribuées. La conception et la réalisation de protocoles d'appel de procédure à distance a mis la conception d'applications distribuées, jusqu'alors réservée à un petit nombre de spécialistes, à la portée d'un très grand nombre de programmeurs.

Cependant l'expérience dans la conception d'applications distribuées montre que le type de communication client/serveur offert par l'appel de procédure à distance est mal adapté pour les applications constituées de plusieurs processus coopérants. Dans ce type d'applications, la communication a souvent lieu entre un processus et un groupe de plusieurs processus. De nombreux travaux ont été menés ces dernières années dans le domaine de la conception de protocoles permettant une communication de type 1 vers N. Deux approches se sont dégagées. La conception de protocoles de diffusion fiable constitue la première approche. La seconde consiste à étendre le mécanisme d'appel de procédure à distance pour permettre l'exécution d'une procédure sur N sites en parallèle.

Le langage de programmation de GOTHIC [3], appelé POLYGOTH, introduit le concept de multiprocédure pour la construction d'applications distribuées. La multiprocédure est une généralisation de la procédure permettant l'exécution en parallèle de plusieurs calculs. Nos travaux portent sur la conception de protocoles de communication fiable pour la mise en œuvre d'un protocole d'appel de multiprocédure à distance.

Le projet GOTHIC [Bana88b], développé dans l'équipe Langages et Systèmes Parallèles de l'IRISA (Institut de Recherche en Informatique et Systèmes Aléatoires) à Rennes, vise à construire un système distribué intégré tolérant les fautes sur une architecture matérielle constituée d'un réseau local de type Ethernet reliant un ensemble de machines multiprocesseurs. L'originalité de cette architecture réside dans le fait qu'à chaque processeur est associée une mémoire stable rapide. La mémoire stable rapide est non volatile et permet la mise à jour atomique des informations qui y sont conservées. Les opérations atomiques de la mémoire stable sont mises en œuvre par matériel. La tolérance aux fautes dans GOTHIC est réalisée en utilisant les propriétés de la mémoire stable.

[3] Le projet GOTHIC est un projet INRIA/BULL

L'originalité de notre travail repose sur le fait que les protocoles proposés exploitent sans restriction aucune le concept de mémoire stable. La possibilité offerte par la mémoire stable de mettre à jour un groupe d'objets de façon atomique permet de garantir l'intégrité de toutes les données vitales de notre système de communication. La sauvegarde d'une grande quantité d'informations en mémoire stable est envisageable du fait de ses très bonnes performances en terme de rapidité d'accès (équivalentes à celle d'une mémoire vive).

Ainsi, nos protocoles à base de mémoire stable conjuguent-ils fiabilité et efficacité.

La suite de ce document est divisée en trois parties.

Dans une première partie, nous présentons les travaux existants en matière de communication fiable dans les systèmes distribués. Cette étude est elle-même divisée en trois chapitres. Le premier chapitre porte sur les protocoles développés pour mettre en œuvre l'appel de procédure à distance. Dans le deuxième chapitre, nous décrivons quelques protocoles de diffusion fiable représentatifs des travaux actuels dans ce domaine. En particulier, nous consacrons une large part de ce chapitre aux protocoles développés dans le cadre du système Isis [Birm87]. Enfin, dans le troisième chapitre, nous étudions quelques protocoles qui étendent l'appel de procédure à distance en permettant l'exécution en parallèle sur plusieurs sites de la procédure appelée.

La deuxième partie est consacrée au protocole d'appel de multiprocédure à distance. Nous présentons tout d'abord le concept de multiprocédure et le schéma de contrôle associé. Nous décrivons ensuite le protocole RMPC que nous proposons pour la mise en œuvre de l'appel de multiprocédure à distance.

Notre protocole d'appel de multiprocédure à distance suppose l'existence d'un protocole de communication fiable par messages et d'un protocole de diffusion atomique avec ordonnancement total des messages. Dans la troisième partie, nous décrivons les deux protocoles mis en œuvre dans le cadre du système GOTHIC. Nous présentons tout d'abord le système GOTHIC dans lequel s'intègre la mise en œuvre de nos protocoles de communication fiable. En particulier, nous donnons une description précise de la mémoire stable qui est directement impliquée dans nos travaux. Nous présentons ensuite le protocole de communication fiable par messages (RMC) et le protocole de diffusion fiable (RBC) sur lesquels s'appuie le protocole d'exécution des multiprocédures à distance (RMPC). Outre la présentation des protocoles, nous donnons quelques éléments de preuve qui montrent le bon fonctionnement de ceux-ci en présence de pannes. Cette partie se termine par un chapitre consacré à la mise en œuvre et à une analyse détaillée des performances des protocoles RMC et RBC dans le système GOTHIC.

Dans la conclusion générale du document, nous rappelons les points forts de nos travaux. L'un des points intéressants de notre étude est que les protocoles RMC et RBC peuvent être utilisés dans un cadre différent de celui de l'appel de multiprocédure à distance. Nous décrivons brièvement dans cette conclusion un protocole utilisant le protocole RMC qui met

en œuvre la propriété d'atomicité du rendez-vous CSP [Hoar78]. Nous indiquons également les améliorations qui pourraient être apportées au système de communication fiable de GOTHIC. Pour terminer, nous présentons quelques perspectives de développement du projet GOTHIC.

Première Partie

Protocoles de communication fiable dans les systèmes distribués

La conception du protocole d'appel de multiprocédure à distance nous a amené à étudier les travaux effectués dans le domaine des protocoles de communication fiable dans les systèmes distribués, auxquels est consacrée cette première partie.

La multiprocédure généralise le concept de procédure en permettant l'exécution de plusieurs composants en parallèle. Les protocoles d'appel de procédure à distance gèrent les problèmes posés par la mise en œuvre de l'appel de procédure dans les systèmes distribués où l'appelant et l'appelé peuvent s'exécuter sur des sites distincts. Dans un appel de multiprocédure à distance, l'appelant et les composants appelés peuvent s'exécuter sur plusieurs sites distincts. L'appel de procédure à distance n'est donc en fait qu'un cas particulier d'appel de multiprocédure à distance dans lequel la multiprocédure appelée est constituée d'un seul composant. La mise en œuvre de l'appel de multiprocédure à distance pose donc des problèmes similaires à ceux rencontrés pour la mise en œuvre de l'appel de procédure à distance.

Dans le premier chapitre, nous présentons plusieurs protocoles d'appel de procédure à distance qui se distinguent par la façon dont sont traitées les pannes. Nous décrivons plus particulièrement les solutions apportées au célèbre problème des orphelins.

Les problèmes résolus dans les protocoles d'appel de procédure à distance ne constituent qu'un sous-ensemble des problèmes soulevés par la mise en œuvre d'un protocole d'appel de multiprocédure à distance. En effet, dans un appel de procédure à distance le transfert de contrôle prend place entre deux processus seulement. En revanche, un appel de multiprocédure à distance implique le transfert de contrôle du ou des composants appelants vers le ou les composants appelés. Nous nous sommes donc intéressés aux protocoles de diffusion qui permettent d'envoyer un message vers N destinataires de façon fiable.

Le deuxième chapitre est consacré aux protocoles de diffusion fiable. De nombreux protocoles de diffusion fiable ont été proposés dans la littérature. Ils diffèrent essentiellement par le modèle de système considéré et par l'ordre dans lequel les messages diffusés sont délivrés à leurs destinataires. Nous ne présentons dans le chapitre III que quelques protocoles qui nous semblent représentatifs des travaux dans le domaine.

L'appel de multiprocédure à distance n'est pas la seule extension proposée pour l'appel de procédure à distance. Nous concluons cette partie par un chapitre sur les mécanismes d'appel de procédure multiple qui étendent également l'appel de procédure à distance en permettant l'exécution d'une procédure en parallèle sur plusieurs sites.

Chapitre II

Appel de procédure à distance

II.1 Introduction

Nous présentons dans ce chapitre l'appel de procédure à distance. La notion d'appel de procédure à distance émerge en 1970 [Robe70] mais jusqu'au début des années 80 la communication par messages prévaut sur l'appel de procédure à distance. Bien que certains aspects de l'appel de procédure à distance soit traités dans divers travaux réalisés entre 1970 et 1980 [Feld71, Whit77, Brin78, Ball80], Nelson est le premier à aborder l'appel de procédure à distance dans toute sa généralité [Nels81]. Depuis lors, l'appel de procédure à distance est très largement étudié et plusieurs mécanismes sont mis en œuvre [Shri82, Birr84, Lisk84, Corp81, Bers87].

L'appel de procédure à distance est une mise en œuvre de l'appel de procédure dans un environnement distribué, qui respecte le schéma de contrôle de l'appel de procédure classique. Lors d'un appel de procédure local, la procédure appelante et la procédure appelée s'exécutent sur le même processeur (fig. II.1) tandis que dans le cas d'un appel de procédure à distance la procédure appelante et la procédure appelée s'exécutent sur deux processeurs distincts.

Les sites sur lesquels s'exécutent les procédures appelante et appelée sont dénommés respectivement site *client* et site *serveur*. Un site peut naturellement jouer simultanément le rôle de client et de serveur pour des appels de procédure à distance distincts.

De façon schématique, un appel de procédure à distance se déroule de la manière suivante (fig. II.2). Le site client construit un message de type *APPEL* contenant l'identification de la procédure appelée à distance et les paramètres d'entrée, puis le transmet au site serveur. Le processus appelant est suspendu pendant l'exécution de la procédure. Le site serveur, à la réception du message *APPEL* crée un processus pour exécuter la procédure. A la fin de l'exécution de cette dernière, un message de type *RETOUR* contenant les résultats de l'exécution est transmis au site client. A la réception du message *RETOUR* sur le site client,

l'appelant est débloqué.

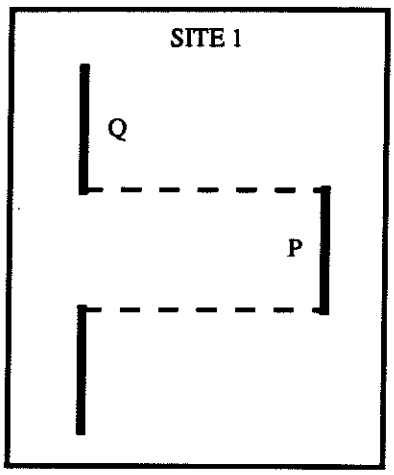


Figure II.1 Appel de procédure local

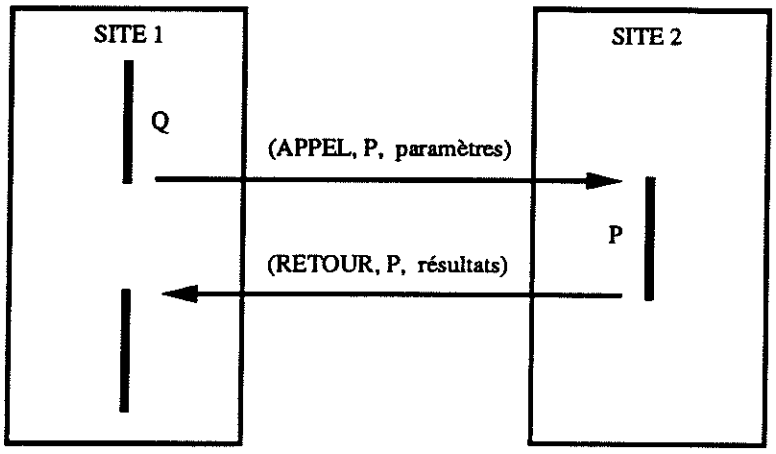


Figure II.2 Appel de procédure à distance

L'intérêt d'utiliser la communication par appel de procédure à distance est double. D'une part il permet de structurer les applications distribuées en procédures comme c'est le cas pour les applications séquentielles. D'autre part le mécanisme d'appel de procédure à distance permet de masquer au programmeur les détails de la communication inter-site et de le décharger en grande partie du traitement des défaillances.

La conception d'un mécanisme d'appel de procédure à distance implique l'étude de plusieurs points appartenant au domaine langage ou au domaine système.

D'un point de vue langage, un mécanisme de vérification de la conformité du type des paramètres effectifs avec le type des paramètres formels doit être développé pour les procédures qui sont appelées à distance.

Un autre aspect langage de l'appel de multiprocédure à distance est relatif à la définition d'une politique de passage des paramètres. En général, le passage des paramètres se fait par valeur dans les appels de procédure à distance. L'exécution d'une procédure appelée à distance s'effectue dans un espace d'adressage distinct de celui de la procédure appelante alors que procédures appelante et appelée s'exécutent dans le même espace d'adressage dans le cas d'un appel local. Par conséquent, le passage d'une adresse comme paramètre dans un appel de procédure à distance est dénué de sens.

Le concepteur d'un mécanisme d'appel de procédure à distance peut être également amené à faire face à des problèmes d'hétérogénéité. Les procédures appelante et appelée peuvent ne pas être écrites dans le même langage. En outre, les machines et les systèmes sur lesquelles les procédures s'exécutent peuvent être différents. Ces divers aspects de l'appel de procédure à distance sortent du cadre de cette étude bien qu'ils constituent une part importante dans la conception d'un mécanisme d'appel de procédure à distance.

D'un point de vue système, il est nécessaire de concevoir un éditeur de lien capable de réaliser la liaison entre la procédure appelante et la procédure appelée distante.

Un autre problème système est le contrôle de la concurrence. Un appel de processus est bloquant : l'exécution de l'appelant est suspendue jusqu'au *RETOUR* du résultat de la procédure appelée. Or une procédure appelée à distance peut à son tour appeler une procédure distante. Le mécanisme de contrôle de la concurrence doit permettre au site serveur d'exécuter d'autres requêtes pendant l'attente du résultat de la procédure imbriquée.

Enfin la transparence des appels de procédure à distance est un point très important. Le mécanisme d'appel de procédure à distance doit respecter le schéma d'exécution des procédures locales. L'occurrence de pannes ne va pas sans poser de problèmes. Dans le cas de l'appel de procédure local, l'appelant et l'appelé s'exécutent sur le même site. La panne de ce site provoque la panne de l'appelant et de l'appelé. En revanche, lors d'un appel de procédure à distance, les sites sur lesquels s'exécutent l'appelant et l'appelé sont distincts et peuvent tomber en panne indépendamment l'un de l'autre. Nous étudions maintenant plus précisément le schéma d'exécution des appels locaux et distants.

Considérons tout d'abord un appel local. Deux cas sont à envisager pour le déroulement d'un appel de procédure local : la panne de machine et l'absence de panne. En l'absence de panne, la procédure appelée est exécutée exactement une fois et ses résultats sont transmis à l'appelant à l'issue de l'exécution. Si la machine tombe en panne au cours du transfert de contrôle entre la procédure appelante et la procédure appelée ou pendant l'exécution de la procédure appelée, l'appel ne se termine pas. La procédure appelée a pu être exécutée zéro ou une fois ou seulement partiellement. Si le programme est relancé après la panne soit par

l'utilisateur soit automatiquement à partir d'un point de reprise (ensemble des informations permettant à un processus de reprendre son exécution après une panne), l'appel de procédure est réexécuté et c'est le résultat de la dernière exécution (influencé par les effets de bords éventuels de l'exécution interrompue par la panne) qui est rendu à l'appelant.

Dans le cas d'un appel de procédure à distance, le problème des pannes est plus complexe du fait de la distribution. Un appel de procédure à distance se déroule sur deux machines indépendantes. La panne de l'une des machines n'empêche pas l'exécution de se poursuivre sur l'autre. Ainsi, l'exécution du processus serveur continue en dépit de la panne du site client. Même en l'absence de panne des sites client ou serveur, une et une seule exécution d'un appel de procédure à distance n'est pas garantie puisque le réseau de communication n'est pas fiable. En outre, un site ne peut pas toujours déterminer de façon certaine le type des pannes qui surviennent. Ainsi comment un site S_i peut-il différencier la panne d'un site distant S_j et la défaillance du lien de communication entre S_i et S_j ? Dans les deux cas, toute communication entre S_i et S_j est impossible. Enfin, la durée d'une panne pouvant être longue, il apparaît qu'il est impossible pour le système de masquer complètement les défaillances matérielles aux applications. En effet pratiquement, il n'est pas envisageable d'attendre indéfiniment la terminaison d'un appel de procédure à distance perturbé par une panne de longue durée.

Comme nous venons de le voir, l'occurence de pannes donne naissance à de nombreuses configurations exceptionnelles pour lesquelles il est nécessaire de définir l'issue d'un appel de procédure à distance. Il est clair que les pannes ne peuvent pas être totalement masquées aux utilisateurs mais il est essentiel qu'un mécanisme d'appel de procédure à distance garantisse aux programmeurs un comportement déterministe pour l'exécution des procédures à distance. Ces garanties définissent ce qui est couramment appelé par abus de langage la sémantique d'appel de procédure à distance.

Le comportement d'un appel de procédure à distance en présence de pannes est en fait l'un des problèmes les plus épineux dans la conception d'un mécanisme d'appel de procédure à distance et fait l'objet de la présente étude.

La suite de ce chapitre est organisée comme suit. Dans le paragraphe II.2 nous présentons le modèle du système et des pannes considérés. La terminologie employée dans la suite de ce chapitre est précisée dans le paragraphe II.3. Nous étudions dans le paragraphe II.4 plusieurs protocoles qui ont pour objectif de rendre aussi transparente que possible l'exécution de l'appel de procédure à distance dans le contexte d'un système distribué non fiable. Le traitement des orphelins créés par les pannes constitue le problème le plus délicat à résoudre dans la majorité de ces protocoles. Le paragraphe II.5 est entièrement consacré à l'examen des méthodes qui peuvent être employées pour éliminer les orphelins. Dans le paragraphe II.6, nous décrivons une mise en œuvre originale de l'appel de procédure à distance fondée sur la redondance de l'exécution de la procédure. Nous terminons par une conclusion dans laquelle nous comparons les différents mécanismes d'appel de procédure à distance et

dégageons les enseignements de cette étude.

II.2 Modèle de système et hypothèses sur les défaillances

II.2.1 Modèle de système

En premier lieu, nous décrivons le modèle du système que nous considérons dans la suite de ce chapitre.

Un système distribué est constitué d'un ensemble de sites ou nœuds reliés par un réseau de communication. Chaque site comporte un processeur associé à une mémoire vive et facultativement à une mémoire stable. Une mémoire stable [Lamp81a] est un support non volatile qui résiste avec une très forte probabilité aux défaillances matérielles du processeur auquel elle est associée. L'écriture de données en mémoire stable est une opération atomique. Une opération atomique est *indivisible* et *recouvrable* [Lisk83].

- Une opération est indivisible si ses états intermédiaires ne sont pas observables par des calculs concurrents.
- Une opération recouvrable n'a que deux issues possibles. Soit elle se termine normalement et les données modifiées passent de leur état initial à leur état final. Soit elle ne produit aucun effet c'est-à-dire que les données modifiées restent dans leur état initial. C'est le principe du tout ou rien. En cas de panne pendant le déroulement d'une opération recouvrable, il est possible soit de terminer l'opération soit de restaurer l'état initial de tous les objets modifiés.

Sur chaque site s'exécutent un ou plusieurs processus. La notion de processus fournit un modèle pour représenter l'activité résultante de l'exécution d'un programme sur une machine [Krak85].

II.2.2 Modèle de défaillances

Le comportement d'un système distribué est affecté par les pannes matérielles dont il est victime. Les sites peuvent tomber en panne. Nous supposons qu'une panne entraîne l'arrêt immédiat et complet d'un site [Schn83]. Le contenu de la mémoire volatile est perdu. Les données conservées en mémoire stable ne sont pas altérées.

Un site peut donc se trouver dans l'un des trois états suivants :

- opérationnel

  C'est l'état normal d'un site pendant lequel les applications peuvent s'exécuter.

- en panne

  Un site est en panne suite à une défaillance. Aucun calcul ne progresse sur un site en panne.

- recouvrement

  Un site est en phase de recouvrement lorsqu'il reprend son exécution suite à une panne. Le recouvrement consiste à restaurer un état sain du système. Les applications ne peuvent être relancées que lorsque la phase de recouvrement est terminée.

Le réseau de communication peut être défaillant. Les messages qui transitent sur le réseau peuvent être perdus ou délivrés avec retard au site destinataire. Si aucune information ne peut être transmise entre deux sites, on dit que le réseau de communication est partitionné. Un partitionnement de réseau peut durer quelques secondes, plusieurs minutes voire plus.

II.3 Terminologie

Nous introduisons dans ce paragraphe quelques termes et notations utiles à la lecture de la suite de ce chapitre.

Considérons un appel de la procédure P à distance entre le site A et le site serveur B. On appelle processus client ou encore *père* le processus qui effectue l'appel à P sur le site A. L'exécution de P sur le site B est effectuée par un processus serveur ou encore processus *fils*. Un appel de procédure à distance Q imbriqué dans P est dit appel *fils*. P est l'appel *père*. Un appel P qui n'a pas d'appel père est dit *racine*. Les sites pères d'un site N sont tous les sites qui ont un appel de procédure à distance en cours dont le processus serveur s'exécute sur N. Les relations *ancêtre* et *descendant* sont construites par application transitive des relations *père* et *fils* en suivant l'arbre résultant de l'imbrication des appels.

II.4 Protocoles pour l'exécution de l'appel de procédure à distance

II.4.1 Introduction

Les divers mécanismes d'appel de procédure à distance se distinguent par les garanties qu'ils offrent aux programmeurs pour l'exécution d'une procédure distante, autrement dit

par leur sémantique d'appel. La sémantique d'appel caractérise combien de fois la procédure appelée à distance est exécutée suite à un appel en l'absence ou en présence de pannes.

La palette des sémantiques d'appel s'étend de la sémantique *peut-être une fois* à la sémantique *au plus une fois* en passant par des sémantiques intermédiaires telles que la sémantique *au moins une fois* et ses variantes. Nous passons en revue dans ce paragraphe différents protocoles mettant en œuvre les diverses sémantiques possibles pour un mécanisme d'appel de procédure à distance.

II.4.2 Exécution incertaine de la procédure distante

La sémantique d'appel la plus faible proposée dans la classification de Spector [Spec82] est la sémantique *peut-être une fois* (*may-be semantics*). C'est la plus simple à mettre en œuvre. Le message *APPEL* est envoyé une seule fois au site serveur. En l'absence de panne la procédure est exécutée exactement une fois. En cas de panne, quel que soit le type de la panne, la procédure est exécutée zéro, une fois ou partiellement sans que le client puisse connaître le nombre d'exécutions.

Cette sémantique d'appel est mise en œuvre par le mécanisme d'appel de procédure à distance « Concurrent CLU RPC » du système CDCS [Baco87].

Ce mécanisme est très peu coûteux mais il est peu utilisable en pratique car le client ne peut savoir si l'exécution de la procédure distante a eu lieu ou pas. Le comportement garanti par ce mécanisme pour un appel de procédure à distance ne correspond absolument pas à celui d'un appel de procédure local.

II.4.3 Exécution au moins une fois de la procédure distante

Certains mécanismes d'appel de procédure à distance garantissent que la procédure appelée à distance est exécutée au moins une fois. Ils respectent la sémantique *au moins une fois* (*at least once semantics*).

Le mécanisme « Concurrent CLU RPC » [Baco87] propose cette sémantique en option.

En l'absence de panne (des sites ou du réseau), la procédure appelée est exécutée exactement une fois. En présence de pannes, la procédure appelée est exécutée une ou plusieurs fois. En cas de panne du site client ou du site serveur, la réception par le client du message de type *RETOUR* lui garantit que la procédure a été exécutée au moins une fois. En l'absence du message *RETOUR*, le client ignore combien de fois la procédure a été exécutée : zéro, une ou plusieurs fois avec la possibilité d'exécutions partielles de la procédure en cas de panne du site serveur.

Un protocole très simple permet de garantir au moins une exécution de la procédure.

Si le site client ne reçoit pas de message *RETOUR* au bout d'un délai fixé, il retransmet le message *APPEL* au site serveur.

La réception d'un message *RETOUR* indique au client que la procédure a été exécutée une ou plusieurs fois, le site serveur ayant pu recevoir plusieurs messages *APPEL* pour la procédure du fait des retransmissions.

Si après un nombre fixé de retransmissions le site client n'a toujours pas reçu le message *RETOUR*, l'appel est abandonné et une exception est signalée à la procédure appelante. Dans ce dernier cas, aucune hypothèse ne peut être faite quant au nombre d'exécutions de la procédure (qui a pu ne pas être exécutée du tout).

Ce type de mécanisme peut convenir à l'exécution d'une classe particulière de procédures : les procédures idempotentes. Une procédure est idempotente si quel que soit le nombre d'exécutions, elle délivre le même résultat et produit les mêmes effets. Cependant, l'application doit prévoir un traitement d'exception dans le cas où aucun message *RETOUR* ne parvient au site client pendant le délai fixé.

Une autre solution est de ne pas fixer de délai d'attente du message *RETOUR* mais dans ce cas le processus client reste évidemment bloqué jusqu'à réception du message *RETOUR*.

II.4.4 Exécution au moins une fois de la procédure distante avec retour du résultat de la dernière exécution

Dans le mécanisme d'appel de procédure à distance précédent, qui garantit au moins une exécution de la procédure, le résultat retourné au client n'est pas forcément celui calculé lors de la dernière exécution de la procédure appelée.

Lampson propose dans [Lamp81a] un protocole pour garantir que le résultat délivré à l'appelant est celui de la dernière exécution de la procédure. La sémantique d'appel assurée par ce protocole est appelée *last of many* dans la littérature. Nous donnons les grandes lignes de ce protocole et en montrons les limites.

Comme dans l'algorithme précédent, le site client retransmet le message *APPEL* en attendant la réception d'un message *RETOUR*.

Pour garantir que le résultat délivré à la procédure appelante est celui de la dernière exécution de la procédure appelée, le site client doit éliminer tous les messages *RETOUR* correspondant aux exécutions précédentes. A la réception d'un message *APPEL*, le site serveur n'exécute la procédure que si celle-ci n'est pas déjà en cours d'exécution. Pour ce faire, chaque message du type *APPEL* contient deux identificateurs uniques (*uid*) : *NA* et *NS*. *NA* identifie l'appel auquel se rapporte le message *APPEL*. *NS* identifie chaque message *APPEL*. Autrement dit, *NS* permet de numéroter en séquence tous les messages *APPEL* émis pour un même appel de procédure à distance identifié par *NA*.

Pour chaque appel de procédure, le site client conserve le dernier message *APPEL* envoyé au site serveur. Pour chacun des sites clients avec lesquels il communique, un site serveur conserve l'uid NS_d du dernier message *APPEL* reçu. De plus le site serveur conserve chaque message *APPEL* qu'il prend en compte (i.e. qui donne lieu à l'exécution de la procédure appelée) jusqu'à ce qu'il envoie le message *RETOUR* correspondant.

Lors de la réception d'un message m de type *APPEL* contenant les uid NA_m et NS_m, le site serveur compare NS_m et NS_d. Si NS_m est inférieur ou égal à NS_d, le message m est ignoré. Dans le cas contraire NS_d prend la valeur de NS_m ; m ne donne lieu à une exécution que s'il n'existe aucun message contenant l'uid NA_m parmi les messages *APPEL* conservés par le site serveur. Si un tel message existe, son champ *NS* prend la valeur NS_m mais la procédure n'est pas exécutée. A la fin de l'exécution de la procédure appelée, le site serveur envoie un message *RETOUR* au site client. Ses champs *NA* et *NS* prennent la valeur de ceux conservés dans le message *APPEL* correspondant. Ce dernier est détruit et le message *RETOUR* n'est pas mémorisé.

Le site client ne prend en compte un message *RETOUR* lui parvenant que si son champ *NS* est identique à celui du message *APPEL* conservé pour l'appel de procédure concerné par le message *RETOUR*.

Cet algorithme garantit que, en l'absence de panne de sites, la procédure est exécutée au moins une fois et que le résultat délivré à la procédure appelante est celui calculé lors de la dernière exécution de la procédure. Il offre les mêmes garanties en cas de panne des sites client ou serveur mais seulement pour les appels de procédure non imbriqués. Par contre, il est mis en défaut pour les appels de procédure imbriqués si une des machines de la chaîne des appels tombe en panne comme l'illustre l'exemple suivant (voir figure II.3).

L'appel à distance effectué sur le site *A* entraîne l'envoi du message (*APPEL*, *A1*, 1) au site *B*. La procédure appelée effectue un appel de procédure à distance vers le site *C* entraînant l'envoi du message (*APPEL*, *B1*, 1) à *C*.

Le site *B* tombe alors en panne, puis reprend son exécution. Comme le site *A* n'a pas reçu de message de *RETOUR* pour l'appel *A1*, il réémet un message d'appel de procédure à distance (*APPEL*, *A1*, 2). A la réception de ce message, le site *B* lance l'exécution de la procédure puisque suite à la panne il a perdu toutes les informations qu'il avait conservées. L'exécution de la procédure sur *B* conduit vers un appel à distance vers *C* (message (*APPEL*, *B2*, 1)). A la réception du résultat de cet appel (message (*RETOUR*, *B2*, 1)), *B* envoie le message (*RETOUR*, *A1*, 2) au site *A*. Lorsque *B* reçoit ensuite le message (*RETOUR*, *B1*, 1), il l'écarte.

Dans cet exemple, ce n'est pas le résultat de la dernière exécution qui est reçu par le site *A*. De plus, deux exécutions de la même procédure ont eu lieu concurremment sur le site *C* alors qu'une seule exécution aurait dû avoir lieu. Cet exemple met en évidence le célèbre problème des orphelins que nous étudions dans la suite.

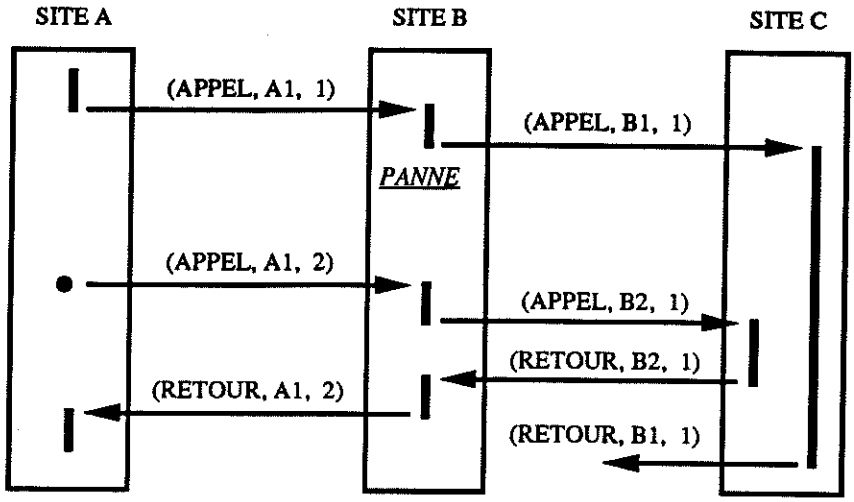


Figure II.3 Exemple d'appels imbriqués

L'algorithme de Lampson doit donc être complété par un algorithme empêchant les orphelins de s'exécuter pour que la sémantique *last of many* soit garantie en présence de pannes de sites pour les appels de procédure imbriqués. Nelson parle dans ce cas de sémantique *last one* qui assure un comportement similaire pour les appels de procédure locaux et distants en cas de pannes de sites.

L'utilité de ce protocole garantissant au moins une exécution de la procédure appelée avec *RETOUR* des résultats de la dernière exécution est limitée aux applications idempotentes puisque, même en l'absence de panne de sites, une seule exécution de la procédure appelée n'est pas garantie. En effet le site serveur écarte, au moment de l'envoi du message *RETOUR*, les informations relatives à l'appel qu'il avait conservées. Par conséquent si ce message *RETOUR* est perdu au cours de sa transmission, le site client réémet un nouveau message *APPEL* qui n'est pas identifié comme un message retransmis par le site serveur et qui est donc de nouveau pris en compte.

II.4.5 Exécution au moins une fois de la procédure distante avec garantie d'une seule exécution en l'absence de panne de sites

Le protocole décrit dans le paragraphe précédent peut être légèrement modifié pour garantir une seule exécution de la procédure appelée en l'absence de panne de sites. Nous venons de voir que le protocole de Lampson n'offre pas cette garantie à cause de l'envoi non fiable du message *RETOUR*. Pour rendre fiable l'envoi du message *RETOUR*, il suffit que

le site serveur le conserve jusqu'à ce qu'il soit certain qu'il a été reçu par le site client.

Ainsi, si le site serveur reçoit un message *APPEL* concernant une procédure qu'il a déjà exécuté, il réémet au site client le message *RETOUR* qu'il a conservé sans exécuter à nouveau la procédure.

Un message *RETOUR* peut être éliminé par le site serveur à la réception d'un nouveau message du type *APPEL* en provenance du même processus client (acquittement implicite).

Ce protocole met en œuvre la sémantique *exactement une fois* (*exactly once semantics*) mais seulement en l'absence de panne de sites.

Elle est préconisé par Nelson car elle permet de garantir un même comportement en l'absence de panne pour les appels de procédure locaux et distants.

II.4.6 Exécution au plus une fois de la procédure distante

Liskov [Lisk87] dans le cadre du système Argus et Svobodova [Svob84a] pour les calculs distribués résilients proposent des protocoles pour l'exécution de procédures à distance qui garantissent au plus une exécution de la procédure appelée en présence de pannes matérielles des sites ou du réseau de communication. Le choix de la sémantique *au plus une fois* (*at most once semantics*) est motivé par la construction de certaines applications distribuées très fiables telles qu'un système de réservation de places d'avion ou un système bancaire.

Argus [Lisk83] met en œuvre un mécanisme d'actions atomiques. Les actions sont des opérations atomiques qui permettent de mettre à jour des données stables. Une action se termine soit par une validation (*commit*), auquel cas les données sont dans leur état final, soit par une annulation (*abort*) qui laisse les données dans leur état initial.

Les actions concurrentes sont sérialisées par un mécanisme de verrouillage. La copie en mémoire stable des données mises à jour par une action n'est pas modifiée avant que l'action ne soit validée. Une version de ces données est conservée en mémoire vive et un protocole de validation à deux phases standard [Gray78] est utilisé pour mettre à jour la copie en mémoire stable.

Une action peut être composée de sous-actions, encore appelées actions imbriquées ; cette structure étant invisible à l'extérieur de l'action. Les sous-actions peuvent être validées ou annulées indépendamment les unes des autres. L'annulation d'une sous-action ne provoque pas l'annulation de l'action mère. La validation d'une sous-action est conditionnelle jusqu'à la terminaison de l'action mère. Si cette dernière est annulée, toutes ses sous-actions le sont également.

L'appel de procédure à distance (appel de *handler* dans le système Argus) est mis en œuvre par une sous-action. La défaillance du système de communication entraîne l'annulation de la sous-action. La sous-action peut être également annulée à la demande de la procédure

appelante.

Un appel de procédure à distance est donc exécuté au plus une fois dans le système Argus : une fois quand la sous-action correspondante est validée, zéro fois quand elle est annulée. L'appel de procédure à distance est dit atomique : soit la procédure est exécutée exactement une fois soit elle n'a aucun effet de bord. En outre l'appelant connaît le nombre d'exécutions (zéro ou une) ayant eu lieu.

Dans le protocole décrit dans [Svob84a], l'appel de procédure à distance est mis en œuvre par une action atomique comme dans Argus. De plus, un point de reprise est sauvegardé lors de chaque appel de procédure à distance. Les processus clients peuvent ainsi traiter les réponses leur parvenant après la panne de leur site et concernant les appels entrepris avant la panne. Ce protocole met en œuvre la sémantique *au plus une fois* ou plus précisément *only once type 2* selon la classification établie par Spector [Spec82].

La sémantique *au plus une fois* est coûteuse à mettre en œuvre. Elle s'avère cependant indispensable pour les applications très robustes tel qu'un système de transactions bancaires. Un appel de procédure à distance avec la sémantique *au plus une fois* est plus proche d'une transaction que d'un appel de procédure local puisque nous avons vu qu'en cas de pannes de sites, une procédure appelée localement pouvait être exécutée plusieurs fois.

La sémantique au plus une fois rend aisé le traitement des pannes pour les applications puisqu'il est garanti qu'une procédure ne produit pas d'effets de bord en cas de panne.

Cependant il y a des procédures pour lesquelles la sémantique *au plus une fois* n'est pas parfaitement bien adaptée : les procédures qui réalisent une opération irréversible. Des exemples de telles procédures sont l'impression d'un listing, la distribution de billets de banque dans un guichet automatique, le lancement d'un missile ... L'effet de ces procédures ne peut pas être annulé. Dans [Lisk83], une technique permettant de limiter considérablement mais pas complètement les actions dont les effets ne peuvent être annulés est proposée. Une procédure réalisant une opération irréversible est structurée en deux actions séquentielles ; le rôle de la première est de vérifier que l'opération irréversible peut bien avoir lieu, la seconde réalise l'opération irréversible mais n'est exécutée que si la première action a été validée.

II.4.7 Discussion

Face à la diversité des mécanismes d'appel de procédure à distance, nous pouvons nous demander quel est le mécanisme idéal.

Dans sa thèse, Nelson avance que les appels de procédure à distance doivent être avant tout transparents pour le programmeur. Il se positionne en faveur d'un mécanisme qui offre les mêmes garanties (ni plus, ni moins) pour les appels de procédure à distance que pour les appels locaux : en l'absence de panne de sites une procédure appelée à distance est

appelée exactement une fois ; en présence de pannes de site, aucun orphelin ne subsiste et c'est le résultat de la dernière exécution de la procédure appelée à distance qui est rendu à l'appelant.

Un tel mécanisme a l'avantage de pouvoir être mis en œuvre efficacement. Cependant la transparence ne peut être totale puisque le programmeur doit prévoir le traitement de nouvelles exceptions telles que la panne du site serveur, l'expiration du délai d'attente du résultat, etc. ...

Une autre approche est adoptée dans le système Argus et dans la mise en œuvre des calculs distribués résilients [Svob84a]. Le mécanisme d'appel de procédure à distance garantit au plus une exécution de la procédure appelée, même en présence de pannes de sites (exécution atomique des procédures à distance).

L'avantage d'un tel mécanisme est la simplicité du traitement des exceptions puisque soit la procédure est exécutée exactement une fois soit aucune exécution n'a lieu. L'inconvénient majeur est le coût puisqu'un système de gestion de transactions est intégré au mécanisme d'appel de procédure à distance.

Selon le type d'application, il sera préféré l'une ou l'autre des deux approches. Pour l'exécution de procédures idempotentes, nous pensons qu'un mécanisme tel que celui proposé par Nelson est bien adapté car dans ce cas les performances peuvent être privilégiées par rapport à la fiabilité. En revanche, pour des applications telles qu'un système de transactions bancaires ou un système de réservation de places d'avion, nous pensons qu'un mécanisme d'appel de procédure à distance atomique est indispensable puisque la fiabilité doit être privilégiée.

Shrivastava opte pour la séparation dans deux couches logicielles distinctes du mécanisme d'appel de procédure à distance et des techniques assurant l'exécution atomique des procédures appelées à distance [Shri82, Shri81]. Un mécanisme d'appel de procédure à distance très simple (garantissant une seule exécution pour les appels de procédure à distance non perturbés par les pannes des sites client ou serveur) et par conséquent très performant peut être mis en œuvre. Les techniques qui offrent un degré de fiabilité plus élevé (atomicité d'exécution) sont mises en œuvre au dessus du mécanisme d'appel de procédure à distance. Cette approche est adoptée par les concepteurs du système d'appel de procédure à distance « Concurrent CLU » [Baco89]. L'intérêt de cette approche est que seules les applications qui requièrent un degré de fiabilité élevé en subissent le coût.

A notre avis, cette approche est valable dans un système général dans lequel coexistent des applications robustes et des applications qui ne nécessitent pas un niveau élevé de fiabilité. Elle privilégie les performances de ces dernières au détriment des performances des applications robustes. Il nous semble en effet que la mise en œuvre *efficace* d'applications distribuées robustes nécessite l'intégration des mécanismes de fiabilité et de communication.

II.5 Le traitement des orphelins

II.5.1 Introduction

Dans ce paragraphe, nous concentrons notre attention sur le traitement des exécutions orphelines. Afin d'éviter l'écueil de l'anthropomorphisme, nous introduisons le néologisme *exéline* pour désigner un orphelin ou une exécution orpheline.

La création d'exélines est due aux pannes qui surviennent dans le système distribué. Rappelons les circonstances qui conduisent à la création d'exélines. Les retransmissions du message *APPEL* peuvent provoquer plusieurs exécutions de la procédure appelée si aucune précaution particulière n'est prise sur le site serveur. Toutes les exécutions de la procédure sauf une sont des exélines. De même, le déclenchement du délai armé pour prévenir une attente infinie du message *RETOUR* par le client peut engendrer des exélines. En effet, le fait que le message *RETOUR* ne parvienne pas au site client n'empêche pas pour autant l'exécution de la procédure appelée sur le site serveur. Cette exécution devient une exéline à partir du moment où le client cesse d'attendre le message *RETOUR*. Enfin, en cas de panne du site client les exécutions de procédure à distance qu'il a initialisées avant de tomber en panne deviennent des exélines.

Les exélines sont indésirables pour plusieurs motifs. En premier lieu, elles peuvent interférer avec les exécutions valides en cours. En outre, les exélines acquièrent des ressources du système et par conséquent augmentent les conflits d'accès aux ressources. Elles consomment du temps d'unité centrale et contribuent ainsi à une baisse des performances.

En univers centralisé, les exélines disparaissent au moment de la panne du site. Aucun mécanisme particulier n'est employé pour les faire disparaître. Il n'en va pas de même dans un système distribué. Des algorithmes spéciaux doivent être mis en œuvre pour traiter le problème des exélines. De nombreuses études y sont consacrées [Nels81, Lamp81b, Shri82, Shri83]. Il existe essentiellement trois approches pour ce problème : la prévention, l'élimination et la récupération des exélines. Chacune de ces approches fait l'objet d'un paragraphe dans la suite.

II.5.2 Prévention des exélines

Nous avons vu que la création d'exélines peut être engendrée par la réémission du message *APPEL*. Il est possible de prévenir l'exécution d'exélines dans ce cas particulier. La méthode consiste à associer un numéro unique (uid) à chaque appel de procédure à distance et à l'insérer dans le message *APPEL*. Le serveur conserve l'uid des messages *APPEL* qu'il prend en compte et écarte tous les messages *APPEL* dont l'uid est identique à l'un des uid conservés. Ainsi, les messages réémis ne donnent pas lieu à de nouvelles exécutions et la

création d'exélines (de ce type) est évitée. Ce mécanisme fonctionne bien en l'absence de panne mais doit être perfectionné de façon à traiter correctement les cas de pannes de sites.

Pour tolérer la panne du serveur, il faut ranger en mémoire stable les identificateurs uniques (uid) des appels ayant donné lieu à une exécution. Le résultat de l'exécution d'une procédure appelée à distance doit être également conservé en mémoire stable sur le site serveur de façon à ce que celui-ci puisse le retransmettre au client sans exécuter de nouveau la procédure en cas de retransmission du message *APPEL*.

II.5.3 Elimination des exélines

Alors que des techniques appropriées permettent de ne pas engendrer d'exélines dues aux réémissions du message *APPEL*, la panne du site client entraîne inévitablement la création d'exélines. Plusieurs méthodes permettent de les éliminer.

Nous décrivons en détail quatre méthodes d'élimination des exélines et terminons par un rapide survol d'autres algorithmes applicables.

Première méthode : élimination des exélines à l'initiative du site client

L'élimination des exélines est réalisée au moment de la reprise après panne du site client. L'algorithme d'élimination est lancé pendant la phase de recouvrement avant que le système ne soit rendu disponible pour l'exécution des applications. Seuls les appels de procédure à distance faisant partie de l'algorithme d'élimination des exélines sont exécutés par un mécanisme spécial pendant cette phase.

Le principe de l'algorithme est le suivant. L'algorithme comporte deux phases successives : l'élimination suivie de la notification. Soit N le site en phase de recouvrement ; pendant la phase d'élimination, N doit contacter tous les sites sur lesquels il a un appel de procédure à distance en cours afin de leur demander d'éliminer ces appels. Les sites contactés demandent à leur tour aux sites serveurs qui exécutent un appel de procédure à distance fils d'un appel issu de N d'éliminer ces appels. Ainsi, récursivement, tous les appels de procédure à distance descendant d'un appel effectué par le site N sont éliminés.

La phase de notification consiste à avertir tous les appels pères des appels s'exécutant sur N de la terminaison anormale (élimination) de ces appels.

L'algorithme nécessite la conservation en mémoire stable, sur chaque site S_i, de deux ensembles de sites : l'ensemble des sites pères des appels servis par le site S_i, appelé *sites_appelants* et l'ensemble des sites fils des appels de procédure à distance effectués par des processus s'exécutant sur le site S_i, appelé *sites_appelés*.

En outre, chaque site conserve en mémoire vive la liste des processus actifs à un instant

donné. A chaque processus p sont associées trois informations :

- *p.site_père* représente l'identité du site père de l'appel exécuté par p,
- *p.processus_père* représente l'identité du processus père de l'appel exécuté par p,
- *p.site_fils* représente l'identité du site vers lequel le processus p a, le cas échéant, effectué un appel de procédure à distance.

Dans l'exemple de la figure II.4, p_3.*site_père* vaut A, p_3.*processus_père* vaut p_1 et p_3.*site_fils* vaut C.

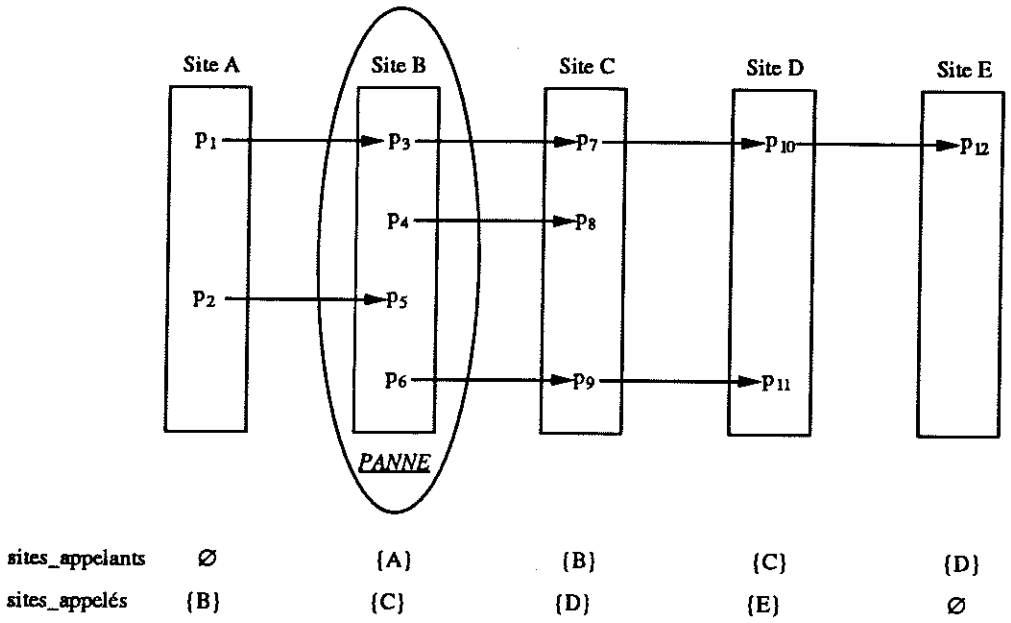


Figure II.4 Un exemple

Le site N appelle à distance la procédure *élimination_totale* sur tous les sites S_j contenus dans l'ensemble *sites_appelés*. L'exécution de la procédure *élimination_totale* sur un site S_j a pour premier effet de détruire tous les processus p tels que *p.site_père* est égal à N. La procédure *élimination_appel* est ensuite appelée à distance sur tous les sites *p.sites_fils* de tout processus p éliminé par la procédure *élimination_totale* avec comme paramètre le couple (S_j,p).

L'exécution de la procédure *élimination_appel* sur le site S_k a pour effet de détruire les processus q pour lesquels le couple *(q.site_père, q.processus_père)* est égal au couple passé en paramètre et d'appeler récursivement à distance la procédure *élimination_appel* sur les

sites *q.site_fils* avec en paramètre le couple (S_k, q). Cet algorithme permet d'éliminer tous les appels de procédure à distance issus de N ainsi que tous les appels imbriqués de ces appels.

La figure II.5 montre le déroulement de l'algorithme d'élimination suite à la panne du site B sur l'exemple introduit dans la figure II.4.

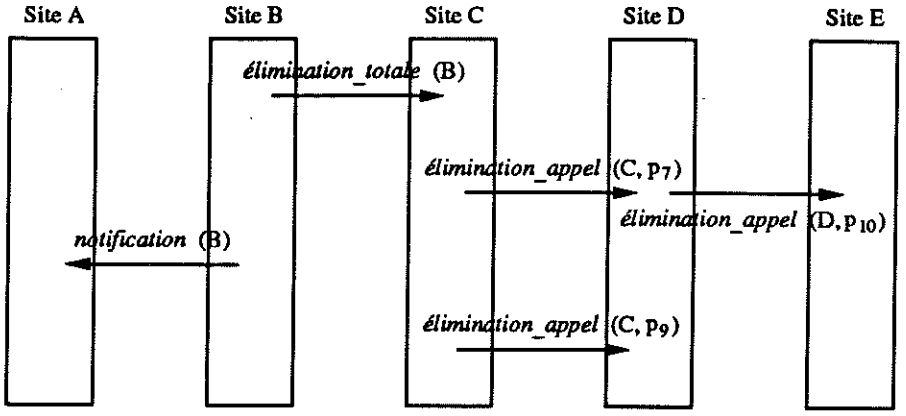


Figure II.5 Algorithme d'élimination

La phase de notification, exécutée une fois la phase d'élimination terminée, consiste à prévenir les processus pères des appels de N éliminés de leur terminaison anormale. Pour ce faire, le site N appelle à distance sur tous les sites présents dans *sites_appelants* la procédure notification avec en paramètre l'identité du site N.

La procédure notification a pour effet de signaler une exception à tous les processus p tel que *p.site_fils* est égal au paramètre de la procédure notification.

L'algorithme d'élimination est récursif. Or, les relations *site_père* et *site_fils* peuvent introduire des cycles. Pour assurer la terminaison de l'algorithme, l'exécution à distance des procédures *élimination_totale* ou *élimination_appel* n'a lieu sur un site en phase de recouvrement que si ce site n'a pas lancé une élimination de ses exélines.

Cet algorithme présente un inconvénient important : il ne garantit pas l'élimination de toutes les exélines si l'un des sites visités pendant l'algorithme d'élimination est en panne.

Deuxième méthode : élimination des exélines à l'échéance du délai d'exécution

La méthode d'élimination des exélines à l'échéance du délai d'exécution n'est applicable que pour les systèmes distribués dans lesquels les horloges de sites sont synchronisées. Elle nécessite la connaissance de la durée d'exécution d'une procédure ou tout au moins d'une borne supérieure de cette durée. Son principe est le suivant.

A chaque processus est associée une date limite d'exécution ou date d'échéance au delà de laquelle il ne peut poursuivre son exécution.

Les processus qui s'exécutent localement à un site ont une date limite d'exécution infinie tandis que les processus créés sur un site pour exécuter une procédure appelée à distance reçoivent de leur processus père une date d'échéance finie.

Un processus racine d'un appel de procédure à distance fixe une date limite à laquelle son appel ainsi que tous les appels qui y sont imbriqués doivent être terminés. Les processus exécutants des appels à distance imbriqués dans un autre appel à distance héritent de la date d'échéance du processus père de leur appel.

Un processus qui atteint sa date limite d'exécution est une exéline potentielle. Sur chaque site, un processus spécial de priorité très élevée est chargé d'éliminer les exélines locales. A intervalles de temps réguliers (Δ), il examine la date limite d'exécution des processus actifs de son site et détruit ceux qui l'ont dépassée. Ainsi, une exéline est éliminée au plus tard Δ unités de temps après sa date d'échéance.

En pratique, une des difficultés de cet algorithme est de choisir convenablement la date limite d'exécution. En effet, si celle-ci est trop courte, les appels dont la durée d'exécution est longue ne peuvent se terminer avant leur date limite d'exécution.

De même, le choix de la valeur de Δ est très délicat. Δ doit être plus long que le délai au bout duquel l'exception «échec» est signalée à l'appelant. En effet, si tel n'est pas le cas le processus appelant peut renouveler son appel (réémission du message *APPEL*) puisqu'il ignore l'élimination du processus (date limite d'exécution atteinte) exécutant l'appel. Par conséquent, dans cette situation un appel peut donner lieu à plusieurs exécutions et donc une seule exécution de la procédure appelée à distance n'est plus garantie en l'absence de panne.

En fait, la méthode d'élimination des exélines à l'échéance du délai d'exécution est en général utilisée en complément de l'algorithme d'élimination à l'initiative du site client pour assurer la disparition des exélines qui subsistent à une élimination compliquée par la panne d'un site.

La date d'échéance est choisie très grande. En cas d'échec de la méthode d'élimination, la phase de recouvrement ne prend fin qu'à l'issue d'un délai qui garantit que toutes les exélines ont atteint leur date d'échéance et ont par conséquent été éliminées.

Dans le cas d'une élimination lancée par un processus p ayant reçu l'exception «échec», l'exécution de p est suspendue jusqu'à ce que les processus qui exécutent l'appel de procédure à distance terminé anormalement aient atteint leur date d'échéance.

Lampson [Lamp81b] propose une variante de cette méthode d'élimination à l'échéance du délai d'exécution appelée élimination à l'échéance du délai d'exécution avec actualisation de l'échéance. Un processus qui atteint sa date d'échéance n'est pas détruit mais demande à son père de reporter sa date d'échéance. La demande de report est transmise le long de la chaîne d'appel jusqu'au processus racine. Si ce dernier est atteint et décide de reporter

la date d'échéance, le processus demandeur et les processus intermédiaires sur la chaîne des appels voient leur date d'échéance reportée et peuvent donc continuer leur exécution.

L'avantage de cet algorithme est de permettre de choisir une date d'échéance assez proche. En contre-partie, le coût de l'algorithme en nombre de messages échangés augmente.

En outre, le choix de la valeur de Δ est important. Si elle est trop faible, de nombreux rafraîchissements sont nécessaires. Si elle est trop élevée, la durée du recouvrement après panne augmente.

Troisième méthode : élimination des exélines à l'initiative du site serveur

Shrivastava suggère dans [Shri83] une méthode qui donne au serveur l'initiative de l'élimination des exélines. Chaque site S_i doit conserver une liste des sites en panne, que nous appelons *sites_en_panne*$_i$ dans la suite. Quand un site S_i détecte qu'un site S_j, qui ne se trouve pas encore dans *site_en_panne*$_i$, est en panne, il l'ajoute à sa liste et détruit tous les processus exécutant des appels initialisés par S_j. Lorsqu'un processus est détruit, s'il a un appel de procédure à distance en cours sur le site S_k, un message est envoyé à S_k pour que S_k détruise l'appel imbriqué (et ses descendants).

La liste *sites_en_panne*$_i$ est mise à jour sur chaque site toutes les Δ unités de temps. La panne d'un site est donc détectée par tous les autres sites après écoulement de la durée $\Delta + e$, e représentant l'erreur possible dans le calcul de la durée Δ. Cette méthode garantit que les exélines engendrées par la panne d'un site sont éliminées au bout de la durée $\Delta + e + t$, où t est le temps maximum nécessaire à un site pour éliminer toutes ses exélines.

La phase de recouvrement doit attendre l'écoulement de cette durée avant de se terminer pour que l'élimination des exélines soit assurée avant la reprise des applications. Cette méthode nécessite un mécanisme de détection des pannes de sites afin de gérer la liste *sites_en_panne*. Elle a l'inconvénient d'engendrer un nombre important de messages.

Quatrième méthode : élimination des exélines par la technique des compteurs de pannes

La technique proposée par Panzieri et Shrivastava [Shri82] pour éliminer les exélines engendrées par la panne d'un site est fondée sur le fait que tous les calculs commencés sur un site serveur avant la panne d'un site client sont détectés et détruits avant de commencer tout calcul initialisé après la panne. Dans le mécanisme d'appel de procédure à distance de Panzieri, un processus serveur est associé à chaque processus client et non pas à chaque appel de procédure à distance comme c'est le cas dans les mécanismes décrits précédemment. Un processus serveur qui a terminé l'exécution d'un appel reste inactif s'il ne reçoit pas un autre appel de son client. Plusieurs causes peuvent entraîner l'inactivité du processus

serveur : communication impossible entre le site client et le site serveur, panne du site client, terminaison de l'exécution du processus client, aucun appel de procédure à distance effectué par le processus client vers le site serveur.

Le principe de la technique des compteurs de panne est le suivant. Chaque site conserve un compteur de panne qui est incrémenté après chaque panne lors du recouvrement. Lorsqu'un processus serveur est créé pour exécuter les appels de procédure à distance d'un client, il reçoit la valeur du compteur de panne du site client. Pour un appel de procédure imbriqué, le processus serveur reçoit la valeur des compteurs de panne de tous les clients de la chaîne d'appel. Chaque site conserve dans une table globale, *tb_cpt*, la liste des compteurs de panne des sites clients de chaque processus serveur qui s'y exécute.

Un processus serveur S consulte cette table avant d'exécuter les appels de son client (situé sur le site C). Si un autre processus serveur S' exécute un appel pour le compte d'un client également situé sur le site C et que la valeur du compteur de panne reçu par S est inférieure à celle reçu par S' (conservée dans *tb_cpt*), S demande l'élimination de S' qui est une exéline.

Cependant, ce mécanisme ne permet pas d'éliminer toutes les exélines. En effet, si aucun appel n'est lancé à partir d'un site précédemment en panne et ayant repris son exécution, les processus serveurs des appels de S démarrés avant la panne ne les ont jamais détectés comme exélines.

Pour régler ce problème, tout processus serveur inoccupé arme un délai. A l'expiration du délai, le processus serveur se déclare exéline potentielle et met cette information dans le tableau des compteurs en panne.

Un processus système, présent sur chaque site S, vérifie à intervalles réguliers si les exélines potentielles sont de véritables exélines. Il demande à chaque site client C de l'exéline potentielle la valeur courante du compteur de panne. Si cette valeur est différente de celle mémorisée pour l'exéline potentielle, cette dernière est éliminée ainsi que tous les processus serveurs du site C s'exécutant sur S.

Ce mécanisme garantit que les exélines d'un site client C sur un site S sont éliminées avant l'exécution sur S de tout nouvel appel de procédure à distance initialisé par un processus de C.

Technique complémentaire de l'élimination des exélines : élimination des exélines par changement d'époque

L'algorithme d'élimination des exélines à l'initiative du site client ne garantit pas la disparition de toutes les exélines en cas de pannes de sites pendant le recouvrement après panne. Des techniques complémentaires doivent être employées pour éliminer les exélines laissées en vie par le premier algorithme d'élimination du paragraphe II.5.3. Ces exélines

doivent être éliminées avant de pouvoir interférer avec les nouveaux appels de leur site père.

Le mécanisme d'élimination des exélines par changement d'époque (*réincarnation*) utilise les ensembles *sites_appelants* et *sites_appelés* introduits au début du paragraphe II.5.3. Son principe est le suivant :

A la suite d'une élimination incomplète, c'est à dire contrariée par la panne d'un site, le site en phase de recouvrement change d'époque. Le changement d'époque consiste à éliminer tous les processus qui exécutent des procédures appelées par les sites présents dans *sites_appelants*. Pendant la phase de changement d'époque, aucune activité ne se déroule sur les sites. Chaque site connait l'époque dans laquelle il est et chaque message contient l'époque au cours de laquelle il est envoyé. Quand un site reçoit un message d'appel de procédure à distance d'une époque plus récente que la sienne, il change d'époque. Chaque message *APPEL* d'époque plus ancienne que celle du site qui le reçoit est ignoré mais reçoit comme réponse la nouvelle époque.

Cette méthode a pour conséquence l'élimination de tous les processus serveurs sur tous les sites impliqués dans un appel de procédure à distance. Cela est admissible dans la mesure où les époques changent très rarement (seulement lorsque l'élimination préalable échoue).

Une variante de cette technique, appelée élimination des exélines par changement d'époque en douceur (*gentle réincarnation*) est plus sélective. Un site ne détruit pas systématiquement tous les processus serveurs travaillant pour les sites présents dans l'ensemble *sites_appelants*. Lors de son changement d'époque, un site vérifie que tous les sites de *sites_appelants* sont également dans la nouvelle époque. Les appels provenant des sites pour lesquels la vérification s'avère positive sont préservés. Cependant la vérification d'un site entraîne par récursivité la vérification de ses sites pères et induit par conséquence beaucoup d'échanges de messages entre de nombreux sites.

Autres méthodes pour éliminer les exélines

Nous mentionnons dans ce paragraphe d'autres méthodes permettant l'élimination des exélines.

La méthode de changement d'époque en douceur peut être utilisée seule sans élimination préalable, en particulier dans les systèmes ne disposant pas d'horloges synchronisées.

Les exélines peuvent également être éliminées avec des méthodes simples fondées sur l'utilisation de la diffusion fiable. Chaque appel de procédure à distance racine reçoit un identificateur unique (uid) dont héritent tous ses appels imbriqués. Lors de la phase de recouvrement après panne, un site diffuse un message d'annulation contenant la liste des uid de tous les appels de procédure à distance en cours au moment de la panne. A la réception d'un message d'annulation, un site détruit tous les processus exécutant une procédure dont l'uid se trouve dans la liste reçue. A la terminaison de la diffusion toutes les exélines ont été

éliminées.

Cette méthode utilise une mémoire stable pour ranger les identificateurs uniques des appels en cours sur chaque site.

II.5.4 Récupération des exélines

Nous avons décrit dans le paragraphe précédent plusieurs méthodes visant à éliminer les exélines. Une autre méthode est de ne pas chercher à éliminer les exélines mais de les laisser s'exécuter en vue de les récupérer.

La récupération des exélines suppose l'existence d'un mécanisme de reprise après panne des processus. La reprise de l'exécution d'un processus après une panne est effectuée à partir d'informations représentant l'état du processus (point de reprise) sauvegardées périodiquement en mémoire stable. Lorsqu'un processus s'exécutant sur un site S appelle une procédure à distance, il est nécessaire de sauvegarder un point de reprise de manière que l'*appel* de procédure à distance ne soit pas réexécuté en cas de panne du site S. Outre le point de reprise, d'autres informations telles que les numéros de séquence identifiant les messages *APPEL* et *RETOUR* doivent être conservées en mémoire stable par le protocole d'appel de procédure à distance sur les sites client et serveur afin de leur permettre d'identifier les messages (émis avant ou après la panne) qui se rapportent à un même appel.

Sur le site serveur, les résultats d'un appel de procédure à distance doivent être sauvegardés en mémoire stable à la fin de l'exécution de la procédure pour qu'ils puissent être retransmis en réponse à un message *APPEL* réémis lors du recouvrement après panne du site client.

Le coût de cette méthode dépend des temps d'accès à la mémoire stable et de la taille des points de reprise sauvegardés.

II.5.5 Récapitulatif

Nous avons décrit plusieurs méthodes permettant de traiter les exélines. Nous avons montré tout d'abord qu'il est possible de prévenir les exélines engendrées par la réémission du message *APPEL*. En revanche, la panne d'un site client entraîne inévitablement la création d'exélines sur plusieurs sites (dans le cas général). Deux approches sont alors possibles face aux problèmes des exélines : l'élimination ou la récupération. La récupération des exélines nécessite la conservation d'un point de reprise lors de chaque appel de procédure à distance. Cette méthode peut donc s'avérer coûteuse à moins de disposer d'une mémoire stable rapide.

Les méthodes d'élimination sont nombreuses. Nous pouvons les comparer selon les critères suivants : quantité d'information à conserver sur chaque site, quantité de messages

échangés, nombre de sites touchés par l'algorithme, finesse de l'élimination (i.e. *reset* du système distribué ou bien élimination uniquement des exélines), présence d'une mémoire stable ou non.

La méthode d'élimination à l'initiative du site client est peu coûteuse, peu d'informations sont conservées sur chaque site. Une mémoire stable est nécessaire mais les mises à jour sont peu fréquentes. Seuls les sites sur lesquels se trouvent des exélines sont impliqués dans une élimination et seules les exélines sont éliminées. Cette méthode simple présente donc de nombreux avantages mais elle possède un inconvénient majeur : l'impossibilité de garantir l'élimination de toutes les exélines en cas de pannes de sites. Par conséquent, cette méthode ne peut être utilisée seule.

La méthode d'élimination des exélines à l'échéance du délai d'exécution est très simple et fonctionne même en cas de pannes de sites ou de partitionnement du réseau. Elle est peu coûteuse en place mémoire, chaque message *APPEL* contient cependant une date d'échéance. Cette méthode possède néanmoins deux contraintes : la nécessité d'avoir des horloges synchronisées et la difficulté de choisir les dates limites d'exécution. Le choix des dates limites d'exécution est délicat et influence notablement les performances de cet algorithme : beaucoup d'appels annulés inutilement pour une valeur trop faible, allongement de la phase de recouvrement pour les valeurs élevées. Pour la méthode d'élimination des exélines avec actualisation de l'échéance, le coût des communications dépend également très largement du choix des dates d'échéance. Des dates d'échéance trop petites impliquent un grand nombre de rafraîchissements et donc de nombreuses communications. Tous les sites du système surveillent périodiquement leurs propres processus. Lorsque cette méthode est employée en conjonction avec la méthode d'élimination, seules les exélines sont éliminées (les dates d'échéance ayant des valeurs élevées dans ce cas).

La méthode de changement d'époque en douceur a un coût de communication élevé. La propagation d'une nouvelle époque peut gagner tout le système. De plus, la vérification des ancêtres peut prendre beaucoup de temps surtout en cas de pannes de sites. Cette méthode entraîne une visite de tous les sites pour vérifier la présence d'exélines mais seules ces dernières sont éliminées. L'élimination des exélines par changement d'époque en douceur est une méthode acceptable pour des systèmes de taille réduite et a l'avantage de pouvoir être utilisée sans horloges synchronisées.

La méthode de changement d'époque quant à elle entraîne l'annulation de toutes les procédures s'exécutant à distance et équivaut à peu de chose près à un *reset* général du système distribué.

Les performances d'une méthode d'élimination des exélines utilisant une diffusion fiable dépendent très largement des performances de la diffusion fiable.

Il ressort de cette étude qu'en général ce n'est pas une seule méthode qui est utilisée pour éliminer les exélines mais plusieurs méthodes complémentaires : élimination à l'initiative

du site client et élimination à l'échéance du délai d'exécution ou élimination à l'initiative du site client et changement d'époque en douceur par exemple.

Les mécanismes d'appel de procédure à distance atomique n'ont pas besoin d'algorithme d'élimination des exélines. C'est le mécanisme d'action atomique intégré qui empêche la création d'exélines. Un appel de procédure à distance est une action qui est purement et simplement annulée en cas de terminaison anormale d'un appel ou en cas de panne d'un site client.

Enfin, il ne faut pas perdre de vue que les algorithmes d'élimination des exélines ne suppriment pas les effets de bord produits par les exélines avant qu'elles ne soient éliminées. Ils se limitent à la destruction de processus dont l'exécution est devenue inutile, les empêchant ainsi d'interférer avec les exécutions valides en cours.

II.6 Une mise en œuvre de l'appel de procédure à distance fondée sur la redondance active

[Yap86] décrit un mécanisme d'appel de procédure à distance tolérant les fautes fondé sur la redondance active. Un appel de procédure à distance est exécuté par un groupe de sites. Tous les sites du groupe forment une chaîne dont le site situé en tête est appelé site primaire. Les autres sites sont les sites secondaires. Le message d'appel de procédure est envoyé au site primaire qui le transmet au site suivant dans la chaîne. Le message d'appel est ainsi propagé le long de la chaîne. Tous les sites exécutent la procédure mais seul le site primaire envoie le résultat à l'appelant. Dans le cas d'un appel imbriqué, seul le site primaire réalise effectivement l'appel, les sites secondaires attendent le résultat. La figure II.6 montre le déroulement du protocole en l'absence de panne. En cas de défaillance du site primaire, le site secondaire qui le suit dans la chaîne joue le rôle de site primaire. La panne d'un site secondaire entraîne seulement une reconfiguration de la chaîne. Les défaillances peuvent engendrer des réémissions de messages ; les messages doubles sont détectés par le biais de numéros de séquence.

L'intérêt de cette approche est que le problème des orphelins ne se pose pas tant qu'il y a au moins un site secondaire dans la chaîne. En revanche, le coût de la redondance active est non négligeable.

II.7 Conclusion

Nous avons étudié dans ce chapitre plusieurs protocoles d'appel de procédure à distance. Le problème de fond est de savoir ce qu'il advient d'un appel de procédure à distance en cas de défaillance de l'un des sites client ou serveur c'est-à-dire connaître le nombre d'exécutions

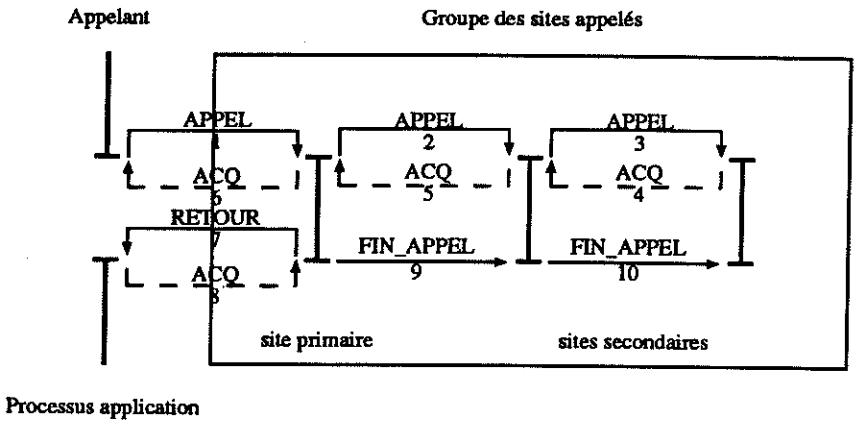


Figure II.6 Déroulement d'un appel de procédure

ayant eu lieu (sémantique d'appel) et détecter (et éliminer) les exélines (orphelins) qui ont pu être créées. Des mécanismes de numérotation des messages permettent facilement de résoudre les problèmes de perte ou de déséquencement des messages par le réseau de communication.

En ce qui concerne l'issue d'un appel de procédure à distance en présence de pannes, les protocoles les plus simples et les plus performants n'assurent pas une et une seule exécution de la procédure appelée. D'autres protocoles doivent alors être utilisés par les applications qui requièrent un degré de fiabilité élevé en plus du mécanisme d'appel de procédure à distance. Cette approche possède l'avantage de ne faire payer le prix de la fiabilité qu'aux applications qui le nécessitent. Des mécanismes d'appel de procédure à distance plus complexes et plus coûteux permettent d'obtenir plus de fiabilité c'est-à-dire de connaître plus précisément le nombre d'exécutions de la procédure en cas de panne. A l'extrême, les protocoles d'exécution atomique des appels de procédure à distance assurent exactement zéro ou une exécution de la procédure appelée. Ces protocoles mis en œuvre à l'aide d'actions atomiques sont coûteux mais permettent la réalisation d'applications robustes telles que la gestion de transactions bancaires.

Le problème des exélines peut être résolu de plusieurs manières. La création d'exélines est due soit à la retransmission du message *APPEL* ou à l'abandon d'un appel par le client suite au déclenchement du délai d'attente du message *RETOUR*, soit à la panne du site client. Dans le premier cas, la création d'exélines peut être évitée en numérotant les messages *APPEL* et en conservant quelques informations sur le site serveur. Dans le second cas, un protocole d'élimination ou de récupération des orphelins doit être mis en œuvre pour traiter les exélines. La méthode la plus simple et la moins coûteuse est l'élimination des exélines à l'initiative du site client lors du recouvrement après panne. Cependant, ce protocole présente l'inconvénient de ne pas détruire toutes les exélines si une panne survient pendant la phase de recouvrement. Un mécanisme complémentaire doit alors être utilisé pour y parvenir.

Cette étude nous amène à quelques réflexions sur l'intérêt que présente l'utilisation de mémoires stables rapides dans un protocole d'appel de procédure à distance.

Nous pensons que l'existence de mémoires stables *rapides* remet en cause l'idée que les protocoles d'appel de procédure à distance atomiques sont si coûteux qu'ils doivent être réservés aux applications nécessitant un degré de fiabilité extrêmement élevé. Le fait qu'il est maintenant possible de construire des mémoires stables pratiquement aussi rapides que des mémoires vives classiques a pour conséquence qu'efficacité et fiabilité ne sont plus antinomiques. Ainsi, les mémoires stables rapides permettent d'envisager des protocoles d'appel de procédure à distance à la fois très fiables et efficaces.

En outre, le fait de disposer de mémoires stables rapides n'est pas sans conséquences sur la conception même de certains algorithmes. Par exemple, l'algorithme d'élimination des exélines à l'initiative du site client proposé par Nelson nécessite la conservation de quelques données en mémoire stable. Compte tenu du coût élevé des mises à jour, la quantité d'informations sauvegardées et la fréquence des mises à jour sont réduites au maximum au prix d'une plus grande complexité du protocole. Ce protocole peut être considérablement simplifié par l'emploi de mémoires stables rapides car il devient alors concevable de mettre à jour des informations en mémoire stable lors de chaque appel de procédure à distance. La mise en œuvre effective d'un protocole de récupération des exélines devient envisageable dès lors que l'on dispose de mémoires stables rapides pour sauvegarder des points de reprise de processus. La présence de mémoires stables rapides permet même de se passer de protocoles de traitement des exélines en diminuant considérablement le coût des mécanismes d'appel atomique de procédure à distance.

Chapitre III

Diffusion fiable

III.1 Introduction

L'abstraction d'appel de procédure à distance permet la communication entre processus distants dans un système distribué. L'appel de procédure à distance offre des avantages pour l'écriture d'applications distribuées (nous les avons soulignés dans le chapitre précédent) mais présente néanmoins quelques limitations [Tane88a]. En particulier, la communication est de type client/serveur et est donc restreinte à deux processus.

Dans les systèmes distribués, les applications mettent souvent en jeu un ensemble de processus coopérants. Les communications n'ont plus seulement lieu entre deux processus mais aussi entre un processus et un groupe de processus. Citons quelques exemples.

- Un service, pour des raisons de sûreté de fonctionnement, peut être mis en œuvre par un groupe de processus (serveurs) s'exécutant sur des sites distincts. Un client du service adresse ses requêtes au groupe des serveurs.
- Une base de données peut être répliquée sur plusieurs sites de façon à en augmenter la disponibilité. Chaque copie est gérée par un gestionnaire. Un gestionnaire donné doit communiquer au groupe des gestionnaires les modifications qu'il apporte à sa copie locale de façon à les répercuter sur les autres sites.
- La localisation d'une ressource dans un sytème distribué peut entraîner l'envoi d'un message à tous les sites du système [Cher85].

Dans chacun de ces exemples apparaît l'utilité de disposer d'une primitive de diffusion c'est-à-dire d'envoi de messages d'un processus vers un ensemble de processus.

Plusieurs types de diffusion existent. Déjà, les réseaux de communication de la famille IEEE802 (réseau Ethernet, Token Ring, FDDI) permettent la transmission d'un message

vers un groupe de machines [Tane88b]. Cependant, ces primitives ne sont pas fiables au sens où elles ne garantissent pas que toutes les machines du groupe adressé reçoivent le message.

De nombreux travaux ont été conduits ces dernières années dans le domaine de la conception de protocoles de diffusion fiable tant dans les systèmes distribués asynchrones [Sega83, Chan84, Schn84, Baba85, Cris86, Birm87, Nava88, Garc89, Kaas89] que dans les systèmes distribués synchrones[Baba85, Baba88, Cris86, Cris89, Perr86]. Les divers protocoles proposés diffèrent sur les hypothèses de pannes ou sur la topologie du réseau qu'ils supposent, l'ordonnancement des messages diffusés, le caractère dynamique ou non des groupes de processus destinataires d'un message.

Dans les systèmes synchrones, des algorithmes de synchronisation d'horloges sont mis en œuvre de façon à obtenir le degré de synchronisation voulu à partir des horloges des sites [Srik87]. Alors que les protocoles de diffusion fiable dans les systèmes distribués asynchrones utilisent des messages explicites pour savoir que le message diffusé est arrivé à toutes ses destinations, les protocoles de diffusion fiable dans les systèmes distribués synchrones obtiennent cette même information à partir de l'écoulement du temps et de leur connaissance des délais qui bornent la durée des diverses opérations intervenant dans le système. Par conséquent, le coût de communication des protocoles de diffusion fiable synchrones est en général plus faible que pour les protocoles asynchrones.

Le système GOTHIC étant un système asynchrone, nous nous intéressons dans ce document uniquement aux travaux menés dans le cadre des systèmes distribués asynchones.

L'organisation de ce chapitre est la suivante. Nous précisons dans le paragraphe III.2 le type des pannes qui sont prises en compte dans les protocoles que nous présentons dans la suite. Nous donnons également toutes les définitions concernant les architectures matérielles et les systèmes distribués pris comme base de travail dans les protocoles étudiés. Dans le paragraphe III.3, nous classifions les divers protocoles de diffusion fiable en fonction des hypothèses qu'ils considèrent et des propriétés qui les caractérisent. La suite du chapitre est consacrée à la présentation des protocoles appartenant aux différentes catégories élaborées à partir des propriétés énoncées dans le paragraphe III.3. Les protocoles de diffusion atomique sont présentés dans le paragraphe III.4. Les protocoles de diffusion atomique qui garantissent un ordre sur les messages diffusés font l'objet du paragraphe III.5. Enfin, nous terminons par une comparaison des protocoles décrits au cours de ce chapitre.

III.2 Hypothèses sur le système et les pannes

Nous considérons dans la suite un système distribué comme étant un ensemble de sites (processeurs) interconnectés par des liens de communication. Nous ne faisons aucune hypothèse *a priori* sur la topologie du réseau. Sur chaque site s'exécutent un ou plusieurs processus application. Par hypothèse, il existe un ordre total sur les identités des sites. Les

sites ne partagent pas de mémoire ; toutes les communications se font par messages. Le réseau de communication offre une ou plusieurs des trois primitives d'envoi de messages suivantes :

- *envoyer_site (S, m)*
  Cette primitive permet l'envoi non fiable (en particulier, la préservation de l'ordre d'émission des messages n'est pas garantie) d'un message isolé m de taille limitée au site S. Cette primitive définit un service de type *datagramme*.

- *diffuser_groupe (G, m)*
  Cette primitive permet de diffuser le message m à tous les membres du groupe G. Elle est non fiable. La primitive *multicast* offerte par le réseau Ethernet est de ce type.

- *diffuser (m)*
  Cette primitive permet de diffuser (de façon non fiable) le message m à tous les sites du réseau. La primitive *broadcast* fournie par le réseau Ethernet est de ce type.

Ces primitives ne sont pas bloquantes.

En ce qui concerne la réception d'un message, nous distinguons les termes *recevoir* et *accepter*. Ces deux termes ont un sens différent lorsque le système de communication est constitué de plusieurs couches. Sur un site, le protocole de niveau i appelle la primitive *recevoir* pour obtenir du protocole de niveau i - 1 un message qui lui est destiné. Une primitive *accepter* est mise en œuvre par chacun des protocoles constituant le système de communication. Un message ne peut être transmis au protocole de niveau i + 1 par le protocole de niveau i que s'il est *accepté* par ce dernier. Un protocole n'accepte pas forcément tous les messages qu'il reçoit.

La primitive *recevoir* est bloquante si aucun message n'est disponible pour le site S.

La figure III.1 représente le système de communication de base que nous considérons dans la suite de ce chapitre. Sans précision contraire, nous supposons que le système de communication offre comme seule primitive d'émission la primitive *envoyer_site*.

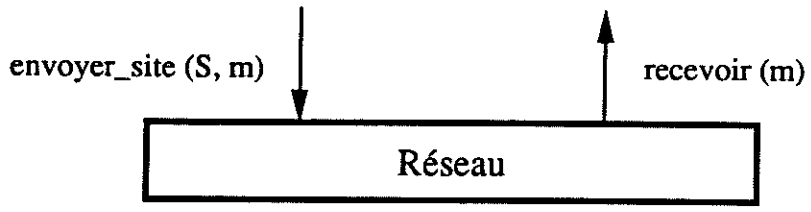


Figure III.1 Le système de communication de base

Le délai de transmission des messages sur le réseau est variable. Chaque site possède une horloge locale. Le système distribué est dit *synchrone* si les horloges de tous les sites sont

parfaitement synchronisées ou à défaut si la différence entre les valeurs des deux horloges n'excède pas ϵ unités de temps, la valeur fixée ϵ étant connue par tous les sites. Les systèmes distribués ne vérifiant pas une de ces deux hypothèses sont dits *asynchrones.*

Nous introduisons maintenant quelques définitions afin de préciser les hypothèses concernant les défaillances qui peuvent se produire dans le système tel que nous l'avons défini.

Nous appelons *composant* un processeur ou un lien de communication

Un composant est dit *correct* si en réponse au déclenchement d'occurrences d'événements il se comporte selon sa spécification [Cris86]. La spécification indique les transitions qui doivent se produire en réponse aux différents types d'événements et l'intervalle de temps prévu pour effectuer la transition.

Un composant est dit *défaillant* lorsqu'il ne se comporte pas selon ses spécifications. La cause d'une défaillance est une faute. On distingue plusieurs classes de fautes :

faute d'omission Un composant est victime d'une faute d'omission lorsqu'il ne répond pas au déclenchement d'un événement.

Des exemples de fautes d'omission sont : un processeur en panne, un lien de communication coupé, un processeur qui omet occasionnellement d'envoyer un message ou bien un lien de communication qui perd un message.

faute de délai Un composant est victime d'une faute de délai (*timing*) lorsqu'il répond soit trop tôt soit trop tard. Un composant est victime d'une faute de délai avec avance s'il répond plus tôt que prévu. Il est victime d'une faute de délai avec retard s'il répond plus tard que prévu. Par exemple, un processeur surchargé de travail peut commettre des fautes de délai avec retard.

faute byzantine Un composant est victime d'une faute byzantine s'il répond d'une façon non conforme à ses spécifications : l'altération d'un message pendant sa transmission (à cause d'un bruit électromagnétique), par exemple.

Les classes de fautes sont incluses les unes dans les autres comme le montre la figure III.2. Les fautes d'omission sont un sous-ensemble des fautes de délai qui constituent elles-même un sous-ensemble des fautes byzantines.

Un composant est dit *fail-stop* s'il se comporte correctement ou bien s'il cesse immédiatement et totalement de fonctionner. Les fautes qui provoquent ce type de défaillance forment un sous-ensemble propre des fautes d'omission.

Tous les protocoles que nous décrivons dans ce chapitre ne fonctionnent pas sous les mêmes hypothèses de pannes. La grande majorité d'entre-eux suppose l'absence de fautes byzantines [Chan84, Schn84, Birm87, Baba88, Garc88b, Garc88a, Luan88, Nava88] et la plupart ne considère que les fautes d'omission.

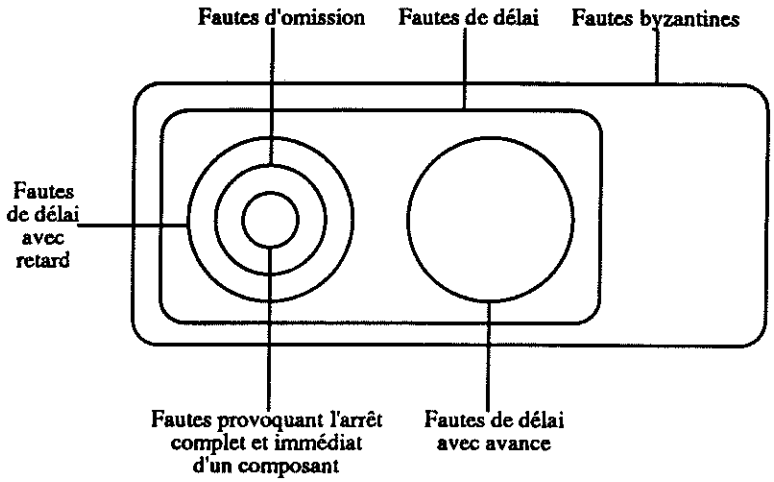


Figure III.2 Les différentes classes de fautes

Les protocoles proposés en univers asynchrone font toujours l'hypothèse que les processeurs possèdent la propriété *fail-stop* [Chan84, Schn84, Birm87, Baba88, Garc88a, Garc88c, Luan88, Nava88].

Nous avons défini au chapitre précédent la notion de partitionnement du réseau. La plupart des protocoles étudiés suppose que le réseau ne peut être partitionné [Cris86, Birm87, Baba88, Garc88c] avec toutefois l'exception de [Luan88].

Les mécanismes de détection de pannes diffèrent d'un protocole à l'autre mais généralement les pannes sont détectées en utilisant des délais [Garc88b, Nava88] ou à la suite d'une impossibilité de communiquer avec un site après N tentatives [Chan84]. Le système Isis [Birm87] met en œuvre un protocole de détection de pannes plus élaboré, appelé détecteur de fautes, au dessous des protocoles de communication. Ce protocole assure que tous les processeurs ont la même vision de l'état du système c'est-à-dire de l'ensemble des processeurs en panne et de l'ensemble des processeurs opérationnels. En outre, l'ordre dans lequel sont détectés les pannes et les recouvrements des processeurs est le même pour tous les processeurs.

III.3 Ebauche d'une classification des protocoles de diffusion fiable

Le terme diffusion fiable est très fréquemment employé dans la littérature mais ne recouvre pas toujours exactement la même chose. Une première distinction s'impose ; elle concerne la définition de l'ensemble des destinataires. Lorsque le message diffusé est adressé à tous les processus du système ou tous les sites du réseau (l'ensemble des destinataires est dans ce cas implicite), le terme *broadcast* que nous traduisons par *diffusion générale*

est généralement employé [Schn84, Baba88, Garc88a, Luan88]. Lorsque le message diffusé s'adresse à un groupe (ou une liste) déterminé de processus, le terme *multicast* est utilisé ; nous le traduisons par *diffusion* [Birm87, Garc88b, Nava88].

Tous les protocoles de diffusion fiable sont des protocoles de diffusion atomique c'est-à-dire qu'ils possèdent au moins la propriété d'atomicité définie comme suit [Chan84, Schn84, Cris86, Garc88b, Garc88c, Luan88, Nava88] :

> *Tous les destinataires corrects reçoivent le message diffusé ou aucun d'entre eux ne le reçoit (tout ou rien).*

Cependant, cette définition de l'atomicité d'un protocole de diffusion ne fait pas l'unanimité. Par exemple Cristian dans [Cris86] donne une autre définition pour un protocole de diffusion atomique dans le cadre des systèmes distribués synchrones. Sa définition de l'atomicité inclut le fait que les messages diffusés doivent être délivrés dans le même ordre à tous les destinataires.

Plusieurs protocoles [Lamp78, Chan84, Cris86, Birm87, Garc88c, Garc89, Luan88, Nava88, Rayn89, Schi89] garantissent en plus de la propriété d'atomicité, un ordre sur l'ensemble des messages diffusés. Garcia-Molina caractérise dans [Garc88c] trois propriétés d'ordre que nous énonçons :

(1) ordonnancement avec source unique (*single source ordering*)
Si un site diffuse deux messages m_1 et m_2 vers un groupe destinataire alors tous les processus destinataires reçoivent m_1 et m_2 dans le même ordre.

(2) ordonnancement avec sources multiples (*multiple source ordering*)
Si m_1 et m_2 sont destinés à un même groupe de processus, les processus destinataires recevront m_1 et m_2 dans un ordre identique même si m_1 et m_2 proviennent de deux sources différentes.

(3) ordonnancement avec groupes destinataires multiples (*multiple group ordering*)
Si deux messages m_1 et m_2 sont délivrés à deux processus P et P', ils sont délivrés dans un ordre identique même si m_1 et m_2 proviennent de sources différentes et sont destinés à des groupes différents (mais non disjoints).

Raynal dans [Rayn90] complète cette liste de propriétés d'ordre par la propriété d'ordre causal introduite par Birman [Birm87] et fondée sur la relation d'ordre partiel *se produire avant*, notée $\rightarrow$, [Lamp78] que nous rappelons :

> Si E_1 et E_2 sont deux événements (i.e. une émission de message, une réception de message ou une action locale à un site) alors $E_1 \rightarrow E_2$ si et seulement si :

- Les événements E_1 et E_2 se produisent sur le même site et E_1 précède E_2.
- E_1 est l'émission d'un message et E_2 est la réception du même message.
- Il existe un événement E_3 tel que $E_1 \rightarrow E_3$ et $E_3 \rightarrow E_2$.

L'ordre causal se définit alors par :

> Soient deux événements $E_1 = SEND\ (M_1)$ et $E_2 = SEND\ (M_2)$.
>
> L'ordre causal est respecté si la propriété suivante est vérifiée :
>
> si ($E_1 \rightarrow E_2$) et (M_1 et M_2 ont la même destination) alors *RECEIVE* $(M_1) \rightarrow$ *RECEIVE* (M_2).

Autrement dit, si deux émissions de messages vers une même destination sont liées par la relation *se produire avant* alors les réceptions de messages correspondantes sont liées par la même relation. Sur l'exemple de la figure III.3 le message M_1 doit être délivré avant le message M_3 sur le site S_3.

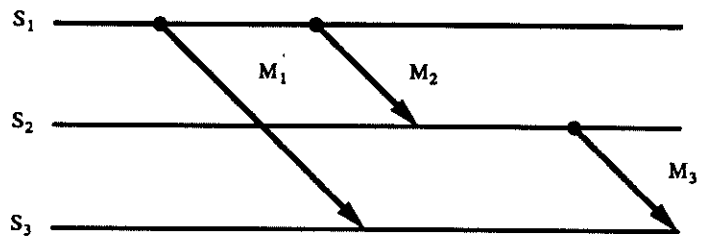


Figure III.3 Illustration de l'ordre causal

Nous pouvons établir une classification des protocoles de diffusion fiable en prenant comme critère les propriétés d'ordre qu'ils garantissent. Cette classification est présentée dans le tableau III.4.

Ordre causal	[Birm87, Rayn89, Schi89]
Ordre avec source unique	[Garc88a]
Ordre avec sources multiples	[Chan84, Luan88, Nava88]
Ordre avec groupes destinataires multiples	[Lamp78, Schn82, Birm87, Garc88c, Garc89]

Figure III.4 Classification des protocoles de diffusion fiable en fonction des propriétés d'ordre

Une autre façon de classer les protocoles de la littérature est de les regrouper selon la classe de fautes qu'ils tolèrent [Cris86]. Par exemple, les protocoles décrits dans [Lamp82] et

[Cris86] tolèrent les fautes byzantines tandis que ceux relatés dans [Perr86] et [Garc88a] ne considèrent que les fautes d'omission. Dans la suite du chapitre, la classe des fautes tolérées pour chaque protocole décrit est indiquée.

III.4 Protocoles de diffusion atomique

III.4.1 Introduction

La propriété la plus simple pour un protocole de diffusion fiable est l'atomicité, telle que nous l'avons définie dans le paragraphe précédent. L'intérêt de ce type de diffusion réside dans le fait qu'un processus qui reçoit un message diffusé peut agir comme si tous les autres destinataires avaient reçu le message. Les protocoles de diffusion atomique sont utiles pour mettre à jour un ensemble de données répliquées sur plusieurs sites.

A première vue, un protocole de diffusion atomique peut sembler trivial à mettre en œuvre quand on dispose d'un réseau de communication fiable.

Le processus source envoie le message diffusé à tous les sites sur lesquels se trouve au moins un processus destinataire. Chaque site destinataire délivre le message à tous les processus destinataires locaux.

Mais que se passe-t-il si le site source tombe en panne avant d'avoir envoyé le message à tous les sites destinataires ? Il est nécessaire qu'un des sites destinataires ayant reçu le message détecte la panne du site source et se charge de transmettre le message aux sites ne l'ayant pas reçu.

Cela suppose la conservation d'une copie du message sur les sites destinataires jusqu'à ce qu'ils acquièrent la certitude que tous les destinataires corrects ont reçu le message. Finalement, la mise en œuvre d'un protocole de diffusion atomique pose quelques difficultés.

Nous présentons dans la suite de ce paragraphe un protocole très simple de diffusion atomique qui fonctionne sous des hypothèses assez restrictives. Nous passons ensuite brièvement en revue quelques autres protocoles de diffusion atomique qui fonctionnent sous des hypothèses de pannes différentes.

III.4.2 Un protocole très simple fonctionnant en présence de processeurs fail-stop

Dans ce paragraphe, nous faisons les hypothèses suivantes :

- Le réseau de communication transmet correctement les messages (la primitive *envoyer_site* est fiable).

- *envoyer_site* est la seule primitive de communication.
- Les sites possèdent la propriété *fail-stop* (ils ne sont pas victimes d'autres types de défaillances).
- Il existe un mécanisme de détection des pannes de sites.

La figure III.5 donne un protocole simple qui met en œuvre la diffusion atomique [Schn86]. Quand un site reçoit un message m pour la première fois, il en transmet une copie

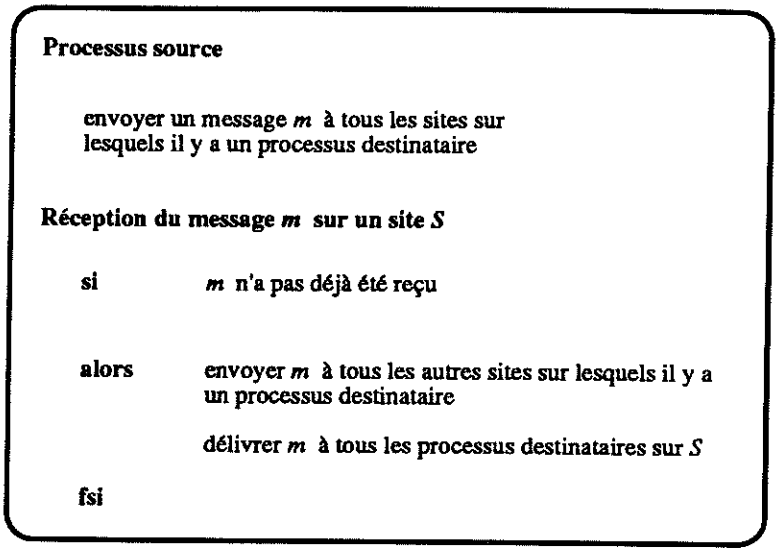


Figure III.5 Un protocole de diffusion atomique très simple

à tous les destinataires de m. Par conséquent, si un site reçoit un message et reste correct, tous les destinataires corrects reçoivent une copie du message : l'atomicité est assurée.

Les retransmissions du message diffusé entraînent un coût de communication très élevé. De plus, le message (ou tout au moins une partie de celui-ci) doit être conservé par tous les sites destinataires de façon à pouvoir détecter les doubles.

Ce protocole peut être amélioré en ne retransmettant les messages qu'en cas de panne du site source de manière à diminuer le coût de communication. Mais cette communication entraîne la conservation du message par tous les sites jusqu'à ce qu'il leur soit notifié que le message a été délivré à tous ses destinataires. Le coût en espace mémoire augmente et la notification de terminaison de la diffusion entraîne des échanges de messages supplémentaires.

L'utilisation d'une structure d'arbre pour propager un message vers tous ses destinataires permet de réaliser une diffusion atomique plus efficace en nombre de messages. En effet, un site qui reçoit un message le réémet seulement à destination de ses fils dans l'arbre. Un mécanisme d'acquittement dispense les nœuds de l'arbre de conserver indéfiniment les messages. Un tel protocole est proposé dans [Schn84]. Ses performances dépendent de la forme de l'arbre de diffusion.

III.4.3 Autres protocoles de diffusion atomique

Le protocole précédent est très simple mais est en pratique d'un intérêt limité du fait qu'il suppose l'existence d'un réseau de communication fiable.

Deux protocoles de diffusion générale qui tolèrent les fautes d'omission et le partitionnement du réseau de communication sont proposés dans [Garc88b] et dans [Garc88a]. Ces deux protocoles visent à optimiser les performances dans deux contextes distincts : le premier dans le cadre de réseaux longue distance offrant seulement la primitive *envoyer_site*, le second dans le cadre de réseaux locaux offrant la primitive *diffuser_groupe*. Ils utilisent tous les deux une structure d'arbre dynamique : le premier pour la diffusion des messages, le second uniquement pour les retransmissions en cas de pertes de messages.

Le protocole décrit dans [Garc88b] s'applique à un réseau constitué de plusieurs sous-réseaux (*clusters*) interconnectés. A l'intérieur d'un sous-réseau, le coût de transmission est faible. En revanche, entre deux sous-réseaux les communications sont coûteuses. Pour obtenir une diffusion efficace, l'arbre de diffusion doit être adapté à l'arbre des sous-réseaux. Le protocole est donc essentiellement constitué d'une procédure dont le rôle est de calquer dynamiquement (en fonction des changements de configuration et des pannes du réseau) l'arbre de diffusion sur l'arbre des sous-réseaux. Cette procédure dite d'attachement se déroule sur tous les sites et recherche en permanence le « meilleur » père (d'un point de vue efficacité) du site sur lequel elle s'exécute. Lorsque le père d'un nœud n'est pas le meilleur père possible, la procédure d'attachement modifie l'arbre de diffusion.

Le protocole décrit dans [Garc88a] tire partie de la primitive *diffuser_groupe* pour diminuer le nombre de messages émis lors des retransmissions prenant place à la fin d'un partitionnement du réseau de communication.

III.4.4 Discussion

Nous pouvons constater que les protocoles cités ci-dessus ne tolèrent qu'un ensemble restreint de fautes. Tous ces protocoles (à l'exception de [Garc88b] qui ne considère pas les pannes des processeurs) supposent des processeurs *fail-stop*. [Garc88b, Garc88a] fonctionnent en présence de fautes d'omission du réseau de communication. A notre connaissance, seuls existent dans les systèmes asynchrones des protocoles de diffusion atomique tolérant les fautes de délai.

La structure d'arbre est utilisée dans tous les protocoles que nous avons mentionnés sauf [Schn86]. L'arborescence permet de répartir sur tous les processeurs la responsabilité du transfert du message diffusé à tous ses destinataires. Cette méthode a l'avantage d'éviter l'engorgement d'un seul processeur qui serait chargé de toutes les retransmissions et de permettre l'acheminement du message vers tous ses destinataires même en cas de panne d'un

ou plusieurs processeurs. Bien sûr, le message doit être conservé sur tous les sites jusqu'à la terminaison du protocole. Cela suppose que les processeurs disposent d'une capacité mémoire suffisante.

Nous pouvons également remarquer que l'utilisation de la primitive *diffuser_groupe* (lorsqu'elle est offerte par le réseau) peut permettre d'améliorer l'efficacité des protocoles.

Enfin, le protocole de diffusion atomique décrit dans le paragraphe III.4.2 illustre bien l'interprétation donnée habituellement à la propriété d'atomicité appliquée aux diffusions. Ainsi, si un site reçoit un message diffusé et tombe en panne, il est possible que les autres sites destinataires ne reçoivent pas le message sans pour autant mettre en défaut la propriété d'atomicité. Le site tombé en panne n'est pas considéré *correct*. Par conséquent, la propriété d'atomicité telle que nous l'avons énoncée au début de ce chapitre est vérifiée. De plus, les sites destinataires en panne pendant la diffusion ne reçoivent jamais le message diffusé alors que celui-ci est reçu par les sites destinataires corrects. Nous pensons que cette propriété d'atomicité n'est pas suffisante. Il nous semble en effet important que les sites destinataires d'un message diffusé tombés en panne pendant la diffusion reçoivent lors de leur recouvrement après panne les messages qui leur étaient destinés. Par exemple, si le message diffusé contient une requête de mise à jour de données répliquées, il est crucial qu'un site tombé en panne mette à jour sa copie lorsqu'il reprend son exécution. Nous proposons dans le chapitre VII un protocole de diffusion atomique qui garantit qu'un message diffusé est délivré à tous ses destinataires, y compris ceux tombant en panne au cours de la diffusion. Nous montrons dans le chapitre V l'intérêt que présente une telle primitive pour la mise en œuvre du protocole d'appel de multiprocédure à distance.

III.5 Protocoles de diffusion atomique ordonnée

III.5.1 Introduction

Outre l'atomicité, plusieurs protocoles de diffusion atomique offrent des garanties sur l'ordre dans lequel les messages sont distribués à leurs destinataires. Un protocole de diffusion atomique ordonnée s'avère indispensable pour conserver une liste ordonnée sur plusieurs sites. De manière que les éléments de la liste soient rangés dans le même ordre sur tous les sites, les messages véhiculant les requêtes de modification de la liste doivent être acceptés dans un ordre identique sur tous les sites. Or si deux sites diffusent un message vers des destinations communes, il est possible que ces messages arrivent dans un ordre différent aux destinations communes. Le rôle d'un protocole de diffusion atomique ordonnée est de garantir un ordre sur l'ensemble des messages diffusés. Nous avons énoncé dans le paragraphe III.3 quatre propriétés concernant l'ordre des messages diffusés. L'ordonnancement des messages émis par une source unique peut être réalisé aisément en numérotant les messages diffusés en séquence.

Les messages sont alors délivrés dans l'ordre des numéros de séquence. Certains des protocoles de diffusion atomique présentés dans le paragraphe précédent [Garc88b, Schn84] utilisent des numéros de séquence pour détecter la perte de message. Ils pourraient être légèrement modifiés pour délivrer les messages à tous leurs destinataires dans l'ordre des numéros de séquence. De ce fait, nous concentrons notre attention uniquement sur les protocoles qui mettent en œuvre les trois autres propriétés d'ordre.

III.5.2 Ordonnancement des messages diffusés par des sources multiples

L'ordonnancement des messages émis par des sources différentes et destinés à un même groupe destinataire est réalisé par plusieurs protocoles [Chan84, Raja89, Nava88, Luan88, Sega83]. Dans ce paragraphe, nous nous intéressons plus spécialement au protocole de Chang et Maxemchuk [Chan84].

Le protocole de Chang et Maxemchuk

Dans le protocole de Chang et Maxemchuk, les sites opérationnels (c'est-à-dire qui ne sont pas en panne) forment un anneau défini par une liste appelée *liste_jeton* qui est conservée par tous les sites. L'un des sites de cette liste est appelé *site_jeton.*

Le protocole fonctionne sous les hypothèses suivantes : les sites possèdent la propriété *fail-stop* et le réseau de communication offre un service de type datagramme. Une défaillance est détectée lorsqu'un site ne parvient à communiquer avec un autre site après R tentatives. Elle peut être due soit à la panne d'un site soit à une défaillance du système de communication.

Décrivons le principe du protocole (cf. Fig. III.6).

Le site source S attribue au message diffusé un numéro unique qui est un couple *(S, n)*, où n est un numéro de séquence, et transmet le message à ses destinataires.

Le *site_jeton* attribue une estampille e au message et transmet un message d'acquittement contenant cette estampille. Les autres récepteurs du message n'acquittent pas le message. Cependant si l'un d'entre-eux détecte un trou dans la séquence des messages qu'il reçoit, il envoie une requête pour obtenir le message manquant. Le *site_jeton* sert les requêtes de retransmission. L'estampillage garantit que les messages sont délivrés dans le même ordre à tous leurs destinataires. Tous les sites de la *liste_jeton* doivent assumer chacun à leur tour la responsabilité de *site_jeton.* De cette manière les sites ne sont pas contraints de conserver indéfiniment les messages reçus en vue de possibles retransmissions. En effet, le protocole ci-dessus ne permet pas à un site de savoir que tous les autres sites ont reçu un message donné. De plus, si le *site_jeton* est défaillant, il peut avoir acquitté un message qui n'a été

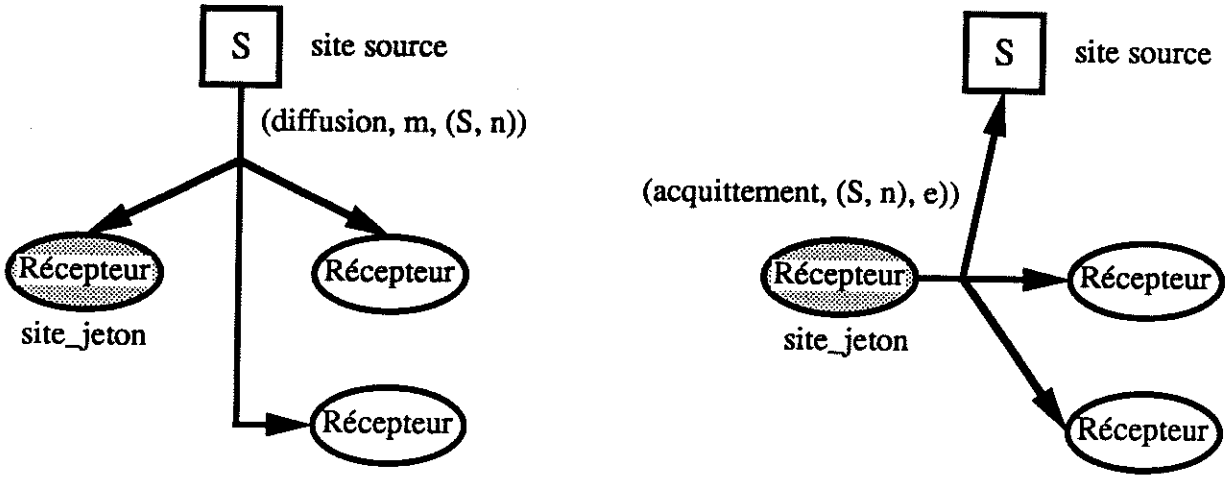


Figure III.6 Le protocole de Chang et Maxemchuk

reçu par aucun autre récepteur. Le changement périodique de *site_jeton* permet également de remédier à ce deuxième inconvénient.

Le jeton est transmis par le site S_i qui le possède soit à l'occasion de l'envoi d'un message d'acquittement soit par un message spécifique contenant la valeur de la prochaine estampille attendue par le site S_i.

Le site S_j à qui est envoyé le jeton ne l'accepte que s'il a reçu tous les messages estampillés. Si ce n'est pas le cas, il demande les messages qui lui manquent avant d'accepter le jeton. Cette politique de transmission du jeton assure que lorsque celui-ci revient au site qui a acquitté un message diffusé m, m est reçu par tous les sites. Lorsque la panne d'un des sites est détectée ou lorsqu'un site doit être réintégré dans l'anneau après une panne, le protocole entre en phase de reformation. Le but de cette phase est de reconstituer une *liste_jeton* et de choisir un nouveau *site_jeton*. Nous ne détaillons pas ici le protocole de reformation qui est quelque peu complexe du fait que plusieurs sites peuvent initialiser simultanément le protocole de reformation.

La fréquence de transmission du jeton influence les caractéristiques du protocole : nombre de messages échangés, capacité de stockage sur chaque site, délai de terminaison, tolérance aux fautes. Le délai de terminaison et la capacité de stockage diminuent quand la fréquence de transmission du jeton augmente. En contrepartie, le nombre de messages augmente. Le nombre de messages de contrôle nécessaires au protocole a tendance à diminuer lorsque le nombre de diffusion augmente puisque lorsqu'il y a beaucoup de diffusion en cours le jeton peut être transmis dans un message d'acquittement. Dans le cas limite où le jeton n'est jamais transmis, le protocole n'est plus tolérant aux fautes.

Les autres protocoles

Rajagopalan propose également un protocole fondé sur la transmission d'un jeton [Raja89] et qui fonctionne sous les mêmes hypothèses de panne que le protocole précédent. Le jeton contient des informations qui permettent aux sites d'une part de savoir quels sont les messages qui ont été reçus par tous les sites et d'autre part de détecter le fait que certains sites n'ont pas reçu tous les messages. Un site a la responsabilité d'émettre des requêtes pour obtenir les messages qui lui manquent. En cas de partitionnement, le protocole peut continuer de fonctionner dans la partition qui contient la majorité des sites (si une telle partition existe).

Le protocole décrit dans [Nava88] utilise pour l'estampillage des messages diffusés un mécanisme similaire à celui employé dans le protocole de Chang et Maxemchuk. En effet, pour chaque groupe de processus destinataires, un site particulier appelé site primaire est chargé d'attribuer une estampille au message. Tout message diffusé à un groupe de processus n'est en fait envoyé par le site source qu'au site primaire. Celui-ci le transmet à tous les autres sites sur lesquels se trouve au moins un membre du groupe (ces sites sont des sites secondaires) après lui avoir attribué une estampille.

Le site primaire joue en quelque sorte le rôle de *site_jeton*. Chaque site secondaire qui reçoit un message diffusé le délivre à tous les processus destinataires locaux et envoie un acquittement au site primaire. Lorsque le site primaire tombe en panne, les sites secondaires doivent élire un nouveau site primaire. Sur chaque site secondaire, un processus appelé veilleur est chargé de détecter la panne du site primaire. Lorsque le veilleur a détecté la panne du site primaire, un algorithme d'élection est lancé.

Ce protocole ne tolère pas le partitionnement du réseau de communication. L'efficacité du protocole est accrue si le réseau offre une primitive de diffusion. Cependant le nombre de messages transmis est important car les sites secondaires doivent acquitter les messages diffusés.

Nous n'avons pas été exhaustif dans ce paragraphe. D'autres protocoles permettent de mettre en œuvre l'ordonnancement des messages diffusés par des sources multiples. Nous pouvons citer par exemple celui de Segall et Awerbuch [Sega83] qui fiabilise un algorithme de routage pour obtenir un protocole de diffusion atomique ordonnée. Par ailleurs, Luan et Oligar [Luan88] ont développé un protocole de diffusion atomique ordonnée fondé sur un consensus majoritaire pour la validation de l'ordre dans lequel les messages diffusés sont délivrés. Il s'agit d'un protocole relativement complexe à trois phases qui fonctionne même en cas de partitionnement du réseau.

Discussion

Les protocoles décrits dans ce paragraphe doivent résoudre le problème d'obtenir un consensus de tous les sites destinataires sur l'ordre des messages diffusés. Cet ordre ne peut pas être déterminé lors de l'initialisation de la diffusion puisque les messages émanent de sources multiples. La solution la plus répandue [Chan84, Raja89, Nava88] est d'attribuer une estampille aux messages diffusés et de délivrer les messages dans l'ordre des estampilles. L'attribution de l'estampille est effectuée de manière centralisée par un site particulier, *site_jeton* dans le protocole de Chang et Maxemchuk, site primaire dans le protocole de Navaratman.

III.5.3 Ordonnancement des messages avec des groupes destinataires multiples

Introduction

Nous nous limitons dans ce paragraphe à la présentation du protocole ABCAST développé dans le cadre du projet Isis [Birm84]. Ce protocole assure la propriété d'ordonnancement avec groupes destinataires multiples, définie au paragraphe III.3.

Le système de communication de Isis

Le système de communication de Isis est organisé en plusieurs couches (fig. III.7).

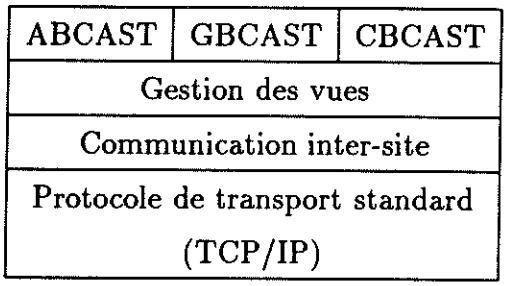


Figure III.7 Organisation du système de communication de Isis

Les protocoles de diffusion fiable (ABCAST,CBCAST et GBCAST) constituent le niveau supérieur. Les protocoles ABCAST et CBCAST réalisent un ordonnancement différent des messages diffusés : ordre total pour le protocole ABCAST et ordre causal pour le protocole CBCAST (nous étudions ce protocole dans le paragraphe III.5.4). Le protocole GBCAST permet d'ordonner des messages diffusés relativement à tous les autres messages diffusés quel que soit le type des diffusions. Il est essentiellement utilisé pour la gestion des défaillances et des modifications dans la composition des groupes.

Le protocole de gestion des vues a pour rôle d'assurer que tous les sites ont la même perception du système et perçoivent les pannes et les recouvrements de sites dans le même ordre.

Le protocole de communication inter-site assure le transfert des messages entre les différents sites et détecte les pannes. Ce protocole s'exécute au dessus d'un protocole de transport standard, TCP/IP par exemple.

Avant de présenter le protocole ABCAST, nous décrivons les protocoles de communication inter-site et de gestion des vues.

Le protocole de communication inter-site

Le protocole de communication inter-site convertit les défaillances en une abstraction appelée *vue* des sites. Il assure le transfert des messages en utilisant un mécanisme d'acquittement. De façon à détecter les défaillances, chaque site envoie périodiquement un message de type *j'existe* à tous les autres sites. Si un site S_i n'a pas reçu de message *j'existe* en provenance de S_j au bout d'un temps déterminé, il considère S_j en panne et démarre l'exécution du protocole de changement de vue. Avec ce mécanisme il est fort possible que S_j soit détecté en panne alors qu'il ne l'est pas. Dans ce cas, S_j doit exécuter le protocole de recouvrement (il est prévenu par un message en provenance d'un site ayant refusé un message dont S_j est l'émetteur). Un numéro, appelé numéro d'incarnation, est associé à chaque site. Chaque fois qu'un site effectue un recouvrement son numéro d'incarnation est incrémenté. La transmission du numéro d'incarnation au sein de tout message émis permet d'écarter tous les messages émis avant l'incarnation courante du site émetteur.

Le protocole de gestion des vues

Le protocole de gestion des vues permet à tous les sites du système d'observer les pannes et les recouvrements de sites dans un même ordre. Chaque site possède une vue des sites qui est la liste des couples (*site,numéro d'incarnation*) correspondant aux sites du système qu'il considère opérationnels.

La vue des sites change quand un site tombe en panne ou reprend son exécution après une panne. Une séquence de vues notée V_0, V_1, V_2,... est une séquence de vues de sites qui reflète ces changements.

Le protocole de gestion des vues assure que la séquence des vues est identique pour tous les sites. Il fonctionne de la façon suivante. Chaque site conserve une copie de la séquence des vues de sites (correctement initialisée au lancement initial du système). Dans une vue, les sites sont rangés selon un ordre unique. Le site le plus ancien selon cet ordre est appelé gestionnaire des vues. C'est lui qui initialise le protocole de gestion des vues lorsqu'il détecte

la panne ou le recouvrement d'un site.

Quand un site détecte que tous les sites plus anciens que lui sont en panne, il prend le rôle de gestionnaire des vues.

Le protocole de gestion des vues est fondé sur un protocole de validation à deux phases que nous décrivons maintenant. Soit $V_0, V_1, ..., V_t$ la séquence des vues courante. Lorsque le gestionnaire des vues détecte une panne ou un recouvrement, il calcule une extension de vue notée $V_{t+1}, V_{t+2}, V_{t+k}$ (en l'absence de panne pendant l'exécution de ce protocole k vaut 1). Il cesse d'accepter les messages émis dans des incarnations ne faisant pas partie de V_{t+k} et envoie la proposition d'extension de vue à tous les sites de V_{t+k}.

Lorsqu'un site reçoit cette proposition, il cesse d'accepter les messages émis dans des incarnations ne faisant pas partie de V_{t+k}. S'il n'a pas reçu d'autres propositions d'extension de vue ou si la proposition reflète tous les changements déjà mémorisés dans les autres extensions, il répond au gestionnaire des vues par un acquittement positif. Dans le cas contraire, il répond par un acquittement négatif incluant les événements manquants.

Le gestionnaire des vues collecte les acquittements. Si tous les acquittements sont positifs, il envoie un message de validation. Si de nouvelles défaillances ou de nouveaux recouvrements ont été détectés ou si des acquittements négatifs ont été reçus, le gestionnaire des vues met à jour l'extension de vue proposée et reprend l'exécution du protocole au début. Dans le cas où le gestionnaire des vues tombe en panne, un nouveau site devient gestionnaire des vues et agit de la manière suivante. S'il possède une extension de vue non validée, il est possible que l'ancien gestionnaire ait envoyé le message de validation avant de tomber en panne. Le nouveau gestionnaire ajoute une nouvelle vue des sites reflétant la panne de l'ancien gestionnaire et reprend l'exécution du protocole de gestion des vues au début. S'il a reçu une extension de vue validée, il est possible que d'autres sites n'aient pas reçu le message de validation correspondant. Il ajoute une vue à l'extension validée la plus récente et exécute le protocole de gestion des vues.

Ce protocole fonctionne correctement en l'absence de partitionnement du réseau à condition que tous les sites ne tombent pas en panne au même moment. En outre, la terminaison du protocole n'est garantie que si la fréquence des pannes n'est pas trop élevée.

Le protocole ABCAST

Le protocole ABCAST se déroule en deux phases : la première permet de choisir une estampille unique pour le message diffusé, la seconde sert à informer les destinataires du message de cette estampille. Les messages sont ensuite délivrés à leurs destinataires dans l'ordre des estampilles.

La primitive ABCAST comporte trois paramètres : l'ensemble des processus destinataires, une étiquette, le message. Seules les diffusions initialisées avec la même étiquette sont

ordonnées entre elles. Pour chaque processus le protocole gère une file d'attente par étiquette (voir figure III.8).

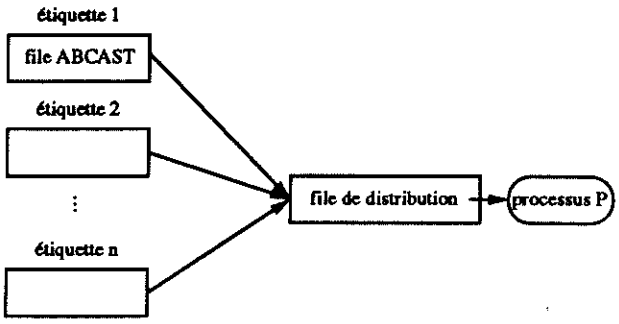


Figure III.8 Structures de données du protocole ABCAST

Tout d'abord, le message est envoyé à tous les sites destinataires. Quand un site reçoit un nouveau message, il ne le délivre pas de suite mais le range dans la file d'attente du processus destinataire correspondant à l'étiquette du message et lui affecte l'attribut *NON-LIVRABLE* qui indique que le message ne peut être délivré. Il choisit localement une estampille pour le message et envoie celle-ci au site source. L'estampille proposée est plus grande que tout autre estampille déjà proposée ou reçue par le site. Une estampille est composée d'un numéro de séquence local au site et d'une identité de site. L'unicité des estampilles est assurée du fait que les sites ont tous des numéros distincts. Le site source collecte les estampilles proposées par les destinataires et choisit la plus grande d'entre elles, selon l'ordre défini dans [Lamp78], comme estampille définitive du message. Il l'envoie à tous les destinataires. Quand un site destinataire reçoit l'estampille définitive d'un message, il l'affecte au message correspondant de la file d'attente et déclare le message *LIVRABLE*. La file d'attente est réordonnée dans l'ordre des estampilles croissantes.

Si le message qui se trouve en tête de file est *LIVRABLE*, il est délivré (i.e. déposé dans la file de distribution). Cette opération est répétée jusqu'à ce que la file soit vide ou que le message situé en tête possède l'attribut *NON-LIVRABLE*.

Cet algorithme assure que les messages diffusés de même étiquette sont délivrés dans le même ordre à tous leurs destinataires : l'ordre des estampilles définitives.

Considérons les cas de pannes. Si l'un des sites destinataires tombe en panne, il est ignoré pour la suite du protocole qui termine la diffusion du message à tous les autres destinataires. Si le site source tombe en panne avant la fin du protocole, l'un des sites destinataires doit le remplacer et assurer la terminaison du protocole.

L'ordonnancement des messages se fait de façon centralisée sur le site source. Du fait que le protocole se déroule en deux phases, le délai de terminaison est toujours au moins

égal au temps nécessaire pour trois transferts de message. Lorsque le réseau de communication offre une primitive de diffusion, l'envoi du message diffusé et l'envoi de l'estampille du message ne nécessitent qu'un envoi de message.

Modification dynamique des groupes de processus

Une modification de la composition des groupes de processus peut intervenir à tout moment. Dans le système Isis, un protocole appelé GBCAST permet à tous les processus d'un groupe d'observer les modifications du groupe dans le même ordre. De plus, les modifications du groupe sont ordonnées par rapport aux diffusions en cours (de type ABCAST, CBCAST ou GBCAST). Ainsi, Isis garantit que si la composition d'un groupe est modifiée alors qu'une diffusion est en cours pour ce groupe, le message diffusé est délivré soit à tous les processus qui faisaient partie du groupe avant la modification soit à tous ceux qui en font partie après la modification. En d'autres termes, il n'est pas possible que certains processus d'un groupe G reçoivent un message diffusé avant d'être avertis de la modification du groupe G tandis que d'autres percevraient la modification de G avant de recevoir le message diffusé.

Dans Isis, tout processus p peut se joindre à un groupe G ou quitter un groupe G en faisant appel à la primitive GBCAST(*action*, p, G) où *action* indique l'arrivée ou le départ de p du groupe G. La primitive GBCAST entraîne la diffusion atomique d'un message à tous les processus du groupe G. Chaque processus membre d'un groupe possède une image de la composition du groupe qu'il met à jour lorsqu'il reçoit un message de modification.

Le protocole GBCAST est également employé pour traiter les défaillances. Le protocole de communication inter-site initialise une diffusion de type GBCAST pour chaque processus qui s'exécutait sur le site défaillant. Lorsque le protocole GBCAST est utilisé pour une défaillance, il assure en plus des propriétés énoncées plus haut que tous les messages diffusés par le processus défaillant sont remis à leurs destinataires avant la prise en compte du message de défaillance diffusé par le protocole GBCAST. A la réception d'un message de défaillance, un site met à jour l'image des groupes modifiés. Les défaillances et recouvrements apparaissent donc comme de simples modifications de la composition des groupes.

Nous ne décrivons pas le protocole GBCAST (qui présente quelques similitudes avec le protocole ABCAST). Le lecteur intéressé pourra se reporter à [Birm87] où il en trouvera une description détaillée.

III.5.4 Ordonnancement des messages selon l'ordre causal

Introduction

L'ordonnancement des messages selon l'ordre causal est moins contraignant que l'ordonnancement avec groupes multiples. En effet, l'ordre causal n'est pas un ordre total sur l'ensemble des messages diffusés. Seuls les messages liés par la relation de dépendance causale sont ordonnés ; les autres messages sont délivrés dans un ordre quelconque à chacun de leurs destinataires.

Pour illustrer l'intérêt d'un protocole de diffusion atomique avec ordonnancement des messages selon l'ordre causal, nous pouvons considérer l'exemple d'un ensemble de données répliquées sur plusieurs sites. Chaque site possède une copie et peut la mettre à jour. La circulation d'un jeton permet de garantir que les mises à jour sont réalisées en exclusion mutuelle : seul le site qui possède le jeton peut mettre à jour sa copie. Une opération d'écriture par l'un des sites entraîne la diffusion d'un message de mise à jour à tous les autres sites. L'ordre causal assure un ordre identique des mises à jour sur tous les sites, garantissant ainsi la consistance des données. La mise à jour d'une donnée répliquée indépendante de la première est contrôlée par un jeton indépendant. L'ordre causal est un ordre partiel. Par conséquent aucun ordre n'est garanti entre les mises à jour des données indépendantes. D'autres exemples démontrent l'utilité de ce type d'ordonnancement dans les systèmes distribués : observation d'un système destinataire, allocation de ressources par exemple [Rayn90].

Le premier protocole de diffusion fiable qui garantit la distribution des messages selon l'ordre causal a été mis en œuvre dans le système Isis sous le nom CBCAST [Birm87]. Tout récemment un nouveau protocole CBCAST a été expérimenté dans le système Isis [Birm90]. Ce nouveau protocole s'inspire de celui décrit dans [Schi89] et possède de bien meilleures performances que l'ancien protocole CBCAST.

Nous décrivons le premier protocole CBCAST de Isis puis nous exposons très brièvement les idées sur lesquelles est fondé le nouveau protocole CBCAST de Isis.

Le premier protocole CBCAST du système Isis

L'idée de base est la suivante : lorsqu'un message m est transmis, tous les messages envoyés avant m (et connus sur le site émetteur de m) sont transmis en même temps que m. Pour mettre en œuvre cette idée, chaque site S_i gère un tampon de messages (noté MSG_i) qui contient, dans l'ordre, tout message reçu ou envoyé par S_i. L'envoi d'un message m du site S_i vers le site S_j comporte les actions suivantes :

- insertion de m en queue de MSG_i,
- composition d'un message M contenant tous les messages présents dans MSG_i,

- envoi de M au site S_j.

Lorsque le message M arrive sur le site S_j, les actions suivantes sont exécutées pour tout message m contenu dans M :

- si m se trouve dans MSG_j (un identificateur unique est attribué à chaque message), m a déjà été pris en compte par S_j et est ignoré. Dans le cas contraire, m est inséré dans MSG_j.
- si S_j est le destinataire de m, m est délivré à son destinataire. Les messages sont traités dans l'ordre du tampon. Ils sont donc délivrés dans l'ordre voulu.

L'exemple de la figure III.9 illustre le comportement de ce protocole. Les messages M_a et M_c sont envoyés à S_3 respectivement par les sites S_1 et S_2. Le message M_b est envoyé à S_2 par S_1. Deux chemins mènent $M_a$vers S_3 :

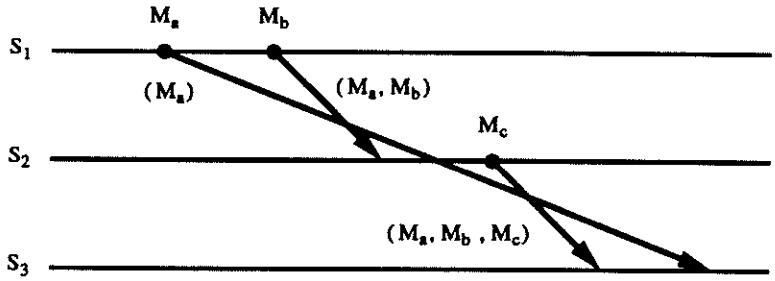


Figure III.9 L'ordonnancement des messages par le protocole CBCAST

(1) M_a est envoyé directement à S_3 par S_1,

(2) M_a prend le chemin $S_1S_2S_3$. Quel que soit l'ordre d'arrivée des messages (M_a) et (M_a,M_b,M_c) sur le site S_3, le message M_a est toujours délivré avant M_c. L'ordre causal sur les messages diffusés est donc respecté.

Ce protocole doit être complété de façon à éliminer les messages devenus inutiles dans les tampons. Sinon, la taille de ces tampons augmente indéfiniment. Nous présentons ce protocole de manière simplifiée. Périodiquement, chaque site S_i diffuse à tous les autres sites un message M contenant tous les identificateurs des messages qui se trouvent dans MSG_i. Quand un site S_j reçoit M, il mémorise le fait que les messages m dont l'identifiateur se trouve dans M ne doivent plus être transmis au site S_i. S_j acquitte le message M. Lorsque S_i a reçu tous les acquittements de M, il peut ôter de MSG_i tous les messages dont l'identificateur est dans M.

Tout site S_i qui reçoit un message m le met dans MSG_i. Puisque tous les messages présents dans MSG_i sont envoyés au bout d'un temps fini, si un site a reçu un message m et ne tombe pas en panne alors tous les autres sites opérationnels reçoivent le message au bout d'un temps fini. Par conséquent, le protocole décrit ci-dessus est atomique. Notons que dans Isis aucune distinction n'est faite entre les deux situations suivantes : le site S reçoit le message m et tombe en panne, le site S tombe en panne avant de recevoir m. La correction de ce protocole est prouvée dans [Birm85].

Le nouveau protocole CBCAST du système Isis

Depuis peu de temps, un nouveau protocole CBCAST (*bypass protocol*) est expérimenté dans le système Isis [Birm90]. Il présente un point commun avec le protocole [Schi89] : l'utilisation du temps logique introduit simultanément dans [Fidg88] et [Matt88] pour ordonner les messages.

Ce temps logique définit un ordre partiel sur les événements d'un système distribué. Il est défini par un vecteur de longueur N appelé *vecteur_temps* (N est le nombre de sites dans le système).

Dans le nouveau protocole CBCAST, un vecteur_temps est associé à chaque groupe de processus.

Le protocole CBCAST est en fait la combinaison de deux protocoles : un protocole nommé *VT* qui gère les vecteurs_temps et un protocole appelé *LT* qui gère des horloges logiques à la Lamport [Lamp78]. Tous les vecteurs_temps sont normalement transmis avec chaque message diffusé. Cependant, pratiquement, cela conduit à des messages de taille trop importante. C'est pourquoi, le protocole met en œuvre un mécanisme qui vise à réduire dans de nombreux cas la quantité d'information transmise avec chaque message. Ces différents cas sont définis par un graphe reflétant la structure des groupes de processus.

Ce nouveau protocole CBCAST tolère les pannes de sites. Les processus d'un groupe conservent les messages qu'ils reçoivent de manière à pouvoir terminer une diffusion inachevée suite à la panne du site source. Les messages conservés par S_i peuvent être détruits lorsque le site S_i reçoit un message du site source de la diffusion indiquant la terminaison de celle-ci. L'intérêt de ce nouveau protocole est qu'il est significativement plus efficace que l'ancien protocole CBCAST.

Discussion

Le premier protocole CBCAST entraîne le transfert de messages de taille importante et la conservation de nombreux messages. Il nécessite l'échange de messages de contrôle pour éliminer les messages devenus inutiles. Le nouveau protocole CBCAST fait la synthèse des

avantages de l'ancien protocole CBCAST et du protocole décrit dans [Schi89] : message de taille réduite par rapport à ceux du protocole CBCAST initial en utilisant les vecteurs-temps et traitement simple des pannes de sites.

III.6 Récapitulatif

Au cours de ce chapitre, nous avons présenté plusieurs protocoles de diffusion fiable qui nous semblent représentatifs des travaux dans le domaine. Sans vouloir être exhaustifs, nous avons aussi mentionné l'existence de nombreux autres protocoles. Face à l'abondance des protocoles relatés dans la littérature, il nous semble important de faire ressortir les ressemblances et les différences entre les protocoles existants. Nous avons dressé un bref récapitulatif des protocoles (conçus pour des systèmes asynchrones) cités dans ce chapitre (voir figure III.10). Le but de ce tableau est à la fois de montrer les caractéristiques communes à plusieurs protocoles qui permettent de les classer et les différences qui font l'originalité d'un protocole par rapport à un autre.

Il est intéressant de classer les protocoles par rapport à l'ordonnancement réalisé sur les messages diffusés. C'est le critère qui nous semble le plus significatif pour le programmeur d'applications distribuées. Une fois le type d'ordonnancement choisi, le choix d'un protocole pourra se faire en fonction des performances et du modèle de pannes considéré.

En ce qui concerne le modèle de pannes, tous les protocoles (à l'exception de ceux qui ne tolèrent pas la panne des processeurs) font l'hypothèse de processeurs *fail-stop*. Par ailleurs, si la plupart des protocoles sont applicables en présence d'un réseau non fiable, rares sont ceux qui tolèrent le partitionnement du réseau.

Le traitement des pannes des processeurs est généralement complexe dans les protocoles présentés. L'occurrence d'une panne entraîne une reconfiguration du système. Cette reconfiguration permet d'attribuer un rôle (par exemple, site primaire ou site secondaire) à chaque processeur du système afin de garantir le bon fonctionnement du protocole malgré la panne.

Enfin, nous pouvons constater qu'aucun protocole ne fait usage d'une mémoire stable. Dans la plupart des algorithmes présentés, le recouvrement des sites en panne n'est pas traité. Les protocoles garantissent seulement que les sites restés opérationnels coopèrent pour que tous reçoivent les messages diffusés si l'un d'entre eux les a reçus. L'usage d'une mémoire stable permettrait de traiter plus simplement le recouvrement d'un site après une panne. En effet, un site qui reprend son exécution après une panne doit récupérer auprès des autres sites l'état courant du système de communication. Dans la plupart des protocoles, un site particulier est responsable de la retransmission des messages. L'intérêt de la mémoire stable serait donc de répartir sur tous les sites cette responsabilité.

Protocole	Type de diffusion	Ordre	Hypothèses sur les processeurs	Hypothèses sur le réseau	Autres caractéristiques du protocole
BIRM87 ABCAST	M	MGO	Fail-stop	pas de partitionnement *envoyer_site*	protocole à 2 phases
CHAN84	M	MSO	Fail-stop	*envoyer_site* ou *diffuser_groupe*	utilisation d'un jeton sites sur un anneau
GM88a	M	aucun	pas de panne	partitionnement possible *envoyer_site* (fautes d'omission)	diffusion sur un arbre réseau longue distance
GM88b	B	SSO	Fail-stop	*diffuser_groupe* (fautes d'omission)	
GM88c GM89	M	MGO	Fail-stop	pas de partitionnement	construction d'un arbre en fonction des groupes de destinataires possibles
LUAN88	B	MSO	Fail-stop	partitionnement possible	protocole de consensus majoritaire à 3 phases
NAVA88	M	MSO	Fail-stop	pas de partitionnement efficacité accrue si *diffuser_groupe*	algorithme d'élection
RAJA89	M	MSO	Fail-stop	*envoyer_site*	utilisation d'un jeton
SCHN84	B	aucun	Fail-stop	réseau fiable point à point	diffusion sur un arbre
SCHN86	B	aucun	Fail-stop	réseau fiable point à point	
BIRM87 CBCAST	M	causal	Fail-stop	pas de partitionnement *envoyer_site*	
BIRM90	M	causal	Fail-stop	*envoyer_site*	utilisation de l'ordre de Fidge et Mattern et de l'ordre de Lamport
RAYN89	M	causal	pas de panne	réseau fiable	
SCHI89	M	causal	pas de panne	réseau fiable	utilisation de l'ordre de Fidge et Mattern

Légende :

M : diffusion
B : diffusion générale

SSO : ordonnancement avec source identique
MSO : ordonnancement avec sources multiples
MGO : ordonnancement avec groupes destinataires multiples

Figure III.10 Tableau récapitulatif des protocoles de diffusion asynchrones

Chapitre IV

De l'appel multiple à l'appel de multiprocédure

IV.1 Introduction

Nous avons étudié au chapitre II plusieurs protocoles d'appel de procédure à distance. Une des limitations de l'appel de procédure à distance est que la communication est restreinte au schéma client/serveur. Dans le contexte d'applications distribuées mises en œuvre par un ensemble de processus coopérant d'autres formes de communication non limitées à deux protagonistes s'avèrent très utiles. Nous avons décrit au chapitre III plusieurs protocoles de diffusion fiable qui mettent en œuvre une communication de type 1 vers N. Ces travaux constituent une première approche. Une deuxième approche est d'étendre le mécanisme d'appel de procédure à distance afin de permettre l'exécution d'appels de procédure multiples en parallèle. Nous pouvons diviser les travaux concernant les appels de procédure multiples en deux groupes qui correspondent chacun à une motivation différente.

D'un côté, la motivation est d'augmenter le parallélisme en permettant l'exécution d'une même procédure sur plusieurs sites en parallèle. Par exemple, il est intéressant de pouvoir exécuter en parallèle une procédure de recherche d'un fichier dans un répertoire réparti.

De l'autre côté, la motivation est de fiabiliser les applications distribuées en exécutant une même procédure simultanément sur plusieurs sites. L'application peut poursuivre son exécution malgré l'occurrence de pannes aussi longtemps que l'un des sites demeure opérationnel.

Dans la suite de ce chapitre nous décrivons tout d'abord des mécanismes d'appels multiples qui visent à augmenter le parallélisme. Nous nous intéressons dans un deuxième temps à deux protocoles d'appels multiples proposés dans Circus et dans Isis pour rendre des applications tolérantes aux fautes. Nous terminons par une discussion.

IV.2 Appels de procédure multiples et parallélisme

IV.2.1 Introduction

Dans ce paragraphe, nous décrivons brièvement deux mécanismes d'appels de procédure à distance multiples qui permettent l'exécution d'une procédure sur plusieurs sites en parallèle. Ces deux mécanismes sont tout à fait similaires et ont été tous les deux étudiés dans le cadre de systèmes de gestion de fichiers distribués. Ce type de protocole est utilisé par exemple pour faire une recherche de fichier dans un répertoire distribué.

IV.2.2 L'appel de procédure à distance parallèle

Le mécanisme d'appel de procédure à distance parallèle [Mart86] a été développé dans le cadre du projet Gemini dont le but est de mettre en œuvre un système de gestion de fichiers répliqué sur un réseau local de machines gérées avec le système Unix.

Un appel de procédure à distance parallèle donne lieu à l'exécution d'une procédure sur N sites en parallèle. La liste des sites sur lesquels doit s'exécuter la procédure est passée en paramètre de la procédure appelée. L'appelant est bloqué pendant l'exécution des N procédures. Les résultats de l'exécution des procédures lui sont fournis au fur et à mesure de leur arrivée. Chaque fois qu'un résultat devient disponible, l'appelant est débloqué et exécute une procédure de traitement des résultats, nommée *trait_résul*. Si tous les résultats ne sont pas encore arrivés, l'appelant se bloque à l'issue de l'exécution de la procédure *trait_résul*. Toutefois, l'appelant peut ne pas attendre tous les résultats. Dans ce cas, les résultats non encore reçus seront indisponibles.

Les paramètres de la procédure appelée peuvent être soit des paramètres d'entrée soit des paramètres résultats. Dans le cas d'une procédure sans paramètres résultats, l'appelant poursuit son exécution immédiatement après l'appel sans se bloquer. L'appel de procédure à distance parallèle dans cette situation particulière est une diffusion (*multicast*).

L'appel de procédure parallèle est mis en œuvre dans un environnement Unix par la notion de *processus réseau*. Un processus réseau est un ensemble de processus Unix. L'un d'entre eux est appelé *client*, les autres sont appelés *serveurs*. Les communications inter machines utilisent le protocole TCP/IP. Si le client est défaillant le processus réseau est considéré défaillant dans son ensemble. Si un appel de procédure à distance sur un des serveurs n'aboutit pas, ce serveur est ignoré dans la suite de l'appel de procédure à distance parallèle qui est poursuivi avec les autres serveurs. Ce type de comportement s'apparente à la sémantique *peut-être une fois* puisque la procédure peut ne pas être exécutée sur certains sites. Un tel choix est motivé par le type des applications visées qui ne requièrent pas un niveau de fiabilité élevé.

IV.2.3 MultiRPC

MultiRPC [Saty86b] est une extension du mécanisme d'appel de procédure à distance RPC2 [Saty86a] qui est mis en œuvre dans le cadre du système Andrew. MultiRPC permet à un client d'appeler simultanément à distance une procédure sur plusieurs sites serveurs. Chacun des appels de procédure à distance au sein d'un multiRPC possède la sémantique *exactement une fois* en l'absence de panne. Comme pour les appels de procédure parallèles présentés dans le paragraphe précédent, le client reçoit les réponses des serveurs au fur et à mesure de leur arrivée. De même le client peut terminer un appel avant d'avoir reçu toutes les réponses. Ainsi, dans l'exemple de la recherche d'un fichier en parallèle dans un répertoire distribué, le client cesse d'attendre les réponses dès que le fichier recherché est localisé.

Les serveurs ne font pas la distinction entre un appel de procédure à distance RPC2 et un appel faisant partie d'un MultiRPC. Ces deux types d'appels sont traités strictement de la même façon par les serveurs. Dans le cas où une erreur se produit sur une des connexions avec un serveur, l'appel est poursuivi mais le client est averti de l'erreur. La connexion défaillante est simplement ignorée pour la suite de l'appel.

IV.3 Appels multiples et tolérance aux fautes

IV.3.1 Introduction

Nous présentons dans ce paragraphe deux méthodes permettant de rendre des applications distribuées tolérantes aux fautes. La première méthode proposée par Cooper dans Circus [Coop85] est fondée sur le principe de la redondance active. Un appel de procédure à distance est exécuté simultanément sur plusieurs sites. La deuxième méthode, mise en œuvre pour les objets résilients dans le cadre du système Isis [Birm86], repose sur le principe de la redondance passive. Un appel de procédure à distance est transmis à plusieurs sites mais un seul, appelé *site primaire* exécute la procédure. Les autres sites interviennent seulement en cas de panne du site primaire.

IV.3.2 Appels de procédures répliqués

Circus [Coop85] a été développé en vue de permettre l'exécution d'applications distribuées malgré l'occurence de pannes de sites dans le système distribué. La tolérance des pannes est mise en œuvre par réplication.

Une application distribuée est constituée de modules interagissant répliqués sur plusieurs sites. L'ensemble des réplica d'un module est appelé une *troupe*. Un appel de procédure répliqué est exécuté une fois et une seule par tous les membres de la troupe (fig. IV.1).

Tant qu'il existe au moins un membre dans une troupe, l'appel répliqué peut être exécuté. Les membres d'une troupe sont indépendants et ne communiquent pas entre eux. Chaque membre d'une troupe agit exactement comme si les autres replica n'existaient pas. Le degré de réplication peut donc varier dynamiquement.

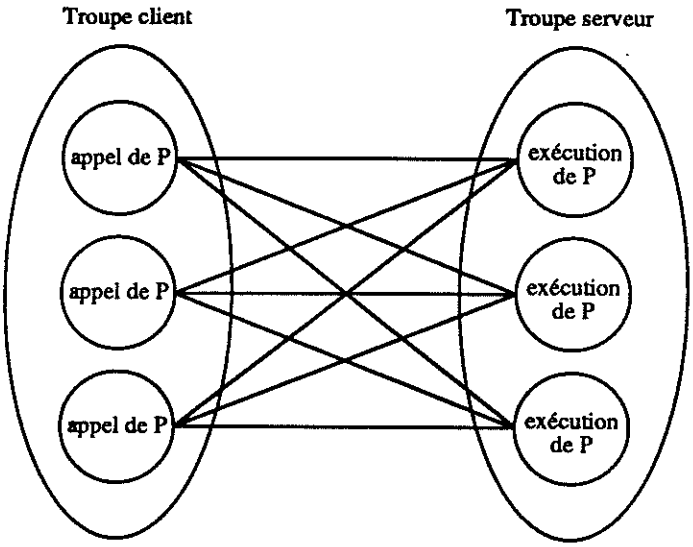


Figure IV.1 Un appel de procédure répliqué

Circus s'exécute sur le système Unix Berkeley 4.2 BSD. Le protocole d'appel de procédure répliqué est mis en œuvre au dessus d'un protocole de communication fiable par paires de messages, similaire au protocole d'appel de procédure à distance décrit dans [Birr84].

Il se décompose en cinq étapes qui sont les suivantes :

(1) Chaque membre de la troupe client envoie un message de type *APPEL* à tous les membres de la troupe serveur.

(2) Chaque membre de la troupe serveur reçoit un message *APPEL* en provenance de chacun des membres de la troupe client.

(3) Chaque membre de la troupe serveur exécute la procédure demandée.

(4) Chaque membre de la troupe serveur envoie un message de type *RETOUR* à tous les membres de la troupe client.

(5) Chaque membre de la troupe client collecte les messages *RETOUR* provenant des membres de la troupe serveur.

Ce protocole peut être séparé en deux parties : le protocole 1-N (étapes 1 et 5) et le protocole N-1 (étapes 2, 3 et 4).

Le protocole 1-N est exécuté par chaque membre de la troupe client. Il garantit que chaque membre de la troupe serveur exécute l'appel.

Les membres de la troupe serveur exécutent le protocole N-1. Ce protocole permet à chaque membre de la troupe serveur d'une part de distinguer les divers messages d'appel qui lui parviennent de façon à n'exécuter l'appel qu'une et une seule fois et d'autre part de connaître le nombre de messages d'appel qu'il doit recevoir pour un appel donné. En ce qui concerne ce dernier point, plusieurs politiques peuvent être appliquées : exécution d'un appel seulement quand le message d'appel de chaque membre de la troupe client est arrivé, exécution d'un appel dès réception du premier message d'appel qui arrive, exécution d'un appel une fois qu'un nombre minimal (fixé) de messages d'appel est arrivé.

Les protocoles 1-N et N-1 reposent d'une part sur l'attribution d'identificateurs uniques (uid) à chaque troupe, à chaque membre d'une troupe et à chaque appel et d'autre part sur des tests appropriés de ces identificateurs. Dans le cas particulier de troupes contenant un membre unique, un appel de procédure répliqué est un simple appel de procédure à distance dont la mise en œuvre est proche de celle de Birrell et Nelson [Birr84].

Dans Circus, l'exécution d'un programme est représentée par un *thread*. Le mécanisme que nous venons de décrire garantit que tous les membres d'une troupe sont dans le même état (consistance de la troupe) pour des programmes distribués déterministes constitués d'un seul *thread*. Dans le cas de programmes distribués composés de plusieurs *threads* s'exécutant en parallèle, il est nécessaire de compléter le protocole précédent pour assurer que tous les membres d'une troupe serveur sérialisent les appels de procédure répliqués de manière identique. Pour ce faire, un protocole de diffusion fiable ordonné très proche du protocole ABCAST décrit dans le paragraphe III.5.3 est employé d'une part par les membres d'une troupe client pour diffuser le message d'appel à tous les membres de la troupe serveur et d'autre part par les membres d'une troupe serveur pour envoyer le résultat à tous les membres de la troupe client.

IV.3.3 Objets résilients

Dans le système Isis, les applications distribuées tolérantes aux fautes sont programmées à l'aide des objets résilients. Un objet résilient possède la particularité d'être répliqué sur plusieurs sites. L'une des copies de l'objet est appelée *composant primaire* (*coordinator*). Toutes les autres copies de l'objet sont appelées *composants secondaires* (*cohorts*).

L'appel à une méthode d'un objet résilient est un appel de procédure à distance. Un appel de procédure à distance se déroule de la façon suivante. L'appelant diffuse un message *APPEL* à l'aide du protocole CBCAST (étudié dans le paragraphe III.5.4) à tous

les composants de l'objet puis attend le résultat de l'appel.

A la réception d'un message *APPEL*, seul le composant primaire de l'objet exécute la procédure appelée. (Celle-ci n'est d'ailleurs exécutée que si l'appel n'a pas encore donné lieu à une exécution. Si une exécution a déjà été effectuée, le composant a conservé le résultat obtenu et est donc en mesure de le renvoyer à l'appelant.) Une fois l'exécution terminée, le résultat est transmis par une diffusion de type CBCAST à l'appelant et à tous les composants secondaires.

Les composants secondaires détectent la défaillance du composant primaire lorsqu'ils reçoivent un message de défaillance diffusé par le protocole GBCAST (voir paragraphe III.5.3). Dans ce cas, un nouveau composant primaire est choisi parmi les composants secondaires.

IV.4 Discussion

Il existe plusieurs extensions du mécanisme d'appel de procédure à distance qui permettent l'exécution en parallèle d'une procédure sur plusieurs sites.

L'intérêt des propositions faites par Satyanarayanan et par Martin est de remplacer N appels consécutifs à une même procédure par un seul appel réalisant les N exécutions en parallèle. Cependant, le traitement des pannes n'est pas très satisfaisant. Si l'un des serveurs tombe en panne, la procédure est quand même exécutée par les autres serveurs mais aucun recouvrement n'est prévu pour le serveur tombé en panne. La panne du client entraîne la création d'orphelins. Aucun protocole d'élimination des orphelins n'est décrit dans [Saty86b, Mart86].

Cooper propose les appels de procédures répliqués dans un but différent puisqu'il s'agit dans Circus de rendre des applications distribuées tolérantes aux fautes. Dans Circus, le parallélisme est utilisé à des fins de tolérance aux fautes de façon transparente aux applications.

Le concept de multiprocédure introduit dans le langage Polygoth [Bana86] est une extension de la procédure qui permet de structurer simplement des applications parallèles. Une multiprocédure est composée de plusieurs composants qui s'exécutent en parallèle. On désigne par appel N-P un appel à une multiprocédure constituée de P composants effectué par une multiprocédure comportant N composants. Les communications sont fondées comme pour les procédures sur le passage de paramètres et résultats. Un appel 1-1 de multiprocédure correspond à un appel de procédure classique.

Il est intéressant de comparer les appels multiples, les appels répliqués et les appels de multiprocédure.

Le concept de multiprocédure permet d'exprimer le parallélisme dans sa forme la plus générale. On peut donc voir les appels multiples de Martin ou de Satyanarayanan comme le cas particulier d'un appel 1-N de multiprocédure dont tous les composants sont identiques.

Un appel de multiprocédure est exécuté exactement une fois en l'absence de panne et donne lieu à exactement 0 ou 1 exécution en cas de panne. Un tel comportement n'est pas garanti pour les appels multiples.

Les règles de synchronisation sont strictes pour les appels de multiprocédure pour lesquels le contrôle est rendu aux composants appelants qu'une fois l'exécution de tous les composants appelés terminés. Dans le cas des appels multiples, l'appelant peut reprendre son exécution dès la terminaison d'un des appels parallèles.

L'exemple de la recherche de fichier dans un répertoire semble *a priori* être réalisé plus efficacement par un appel de procédure multiple plutôt que par un appel de multiprocédure. En revanche, la mise à jour de données répliquées peut être mise en œuvre par un simple appel 1-N de multiprocédure alors que le programmeur devra gérer explicitement les cas de panne s'il utilise un appel multiple.

Comparé aux protocoles d'appel de procédure multiple, les principaux intérêts d'un protocole d'appel de multiprocédure à distance est d'une part d'offrir une forme de communication plus générale (communications inter-groupales) et d'autre part de décharger le programmeur de la gestion des pannes. La deuxième partie de ce document est consacrée à l'étude du protocole d'appel de multiprocédure à distance.

Deuxième Partie

Le protocole d'appel de multiprocédure à distance (RMPC)

Cette seconde partie a pour but de présenter le protocole d'appel de multiprocédure à distance, appelé protocole RMPC. Nous y présentons tout d'abord brièvement le langage PolyGoth développé dans le cadre du projet Gothic en insistant sur le concept de multiprocédure qui est l'une de ses originalités. Nous étudions ensuite le protocole RMPC que nous situons par rapport aux travaux présentés dans la première partie de ce document.

Chapitre V

Le protocole d'appel de multiprocédure à distance

V.1 Polygoth : le langage de Gothic

V.1.1 Introduction

Le système distribué GOTHIC [Bana88b] est conçu pour être adapté à l'exécution des programmes écrits dans le langage POLYGOTH. Ce langage est orienté objet au sens Simula [Dahl68]. Une des originalités de ce langage, développé dans le cadre du projet GOTHIC, réside dans l'introduction d'un nouveau concept : la multiprocédure [Bana86, Bana89b]. Ce nouveau concept ainsi que les notions de classe et d'objet ont été introduites dans le langage Modula-2 [Wirt84].

Nos travaux consistent à mettre en œuvre un protocole d'appel de multiprocédure à distance dans le système GOTHIC. Par conséquent, nous nous contentons de présenter le concept de multiprocédure dans le paragraphe suivant. Le lecteur intéressé par une discussion du concept de multiprocédure et des exemples de programmation à l'aide de ce concept pourra se reporter à [Bana91a]. Le langage POLYGOTH et la réalisation de son compilateur sont décrits plus en détail dans la thèse de Philippe Lecler [Lecl89].

V.1.2 Le concept de multiprocédure

Le concept de procédure est une notion bien comprise pour structurer les programmes. Les procédures offrent un mode de communication par passage de paramètres et résultats. Le concept de multiprocédure [Bana87], introduit dans le langage POLYGOTH, est une extension de la procédure. Il permet de structurer les applications parallèles. La procédure est une abstraction du bloc ; la multiprocédure est une abstraction de la proposition parallèle.

Il existe deux types d'appel d'une multiprocédure : l'appel simple et l'appel coordonné. Nous décrivons chaque type d'appel sur l'exemple de la multiprocédure définie sur la figure V.1.

```
multiproc mf (p1 : t1 , p2 : t2) : (res1 : tr1 , res2 : tr2 , res3 : tr3) ;
cobegin
    begin (p1) : res1
      /* corps du composant (1) */
    end //
    begin (p2) : res2
      /* corps du composant (2) */
    end //
    begin (p1 , p2) : res3
      /* corps du composant (3) */
    end
coend mf ;
```

Figure V.1 Définition de la multiprocédure *mf*

La multiprocédure *mf* est constituée de trois composants. Le composant (1) prend un paramètre en entrée (p_1) et produit le résultat res_1 ; le composant(2) prend en entrée le paramètre p_2 et rend res_2 en résultat ; le composant (3) prend p_1 et p_2 comme paramètres d'entrée et produit res_3. Un appel simple est l'invocation de la multiprocédure par un seul appelant : une procédure ou un composant d'une multiprocédure. Un appel simple est dit de type 1-P car il fait intervenir un appelant et P appelés (les P composants de la multiprocédure appelée). Connaissant la définition de la multiprocédure *mf* et supposant la déclaration des variables *m*, *n*, *a*, *b*, *c*, un appel simple (1-3) à *mf* s'écrit :

$$(a,\ b,\ c) := mf\ (m,\ n).$$

Dans cet appel, *m* et *n* sont les paramètres effectifs en entrée tandis que *a*, *b* et *c* sont les variables destinées à recevoir les résultats élaborés pendant l'exécution de *mf*. L'éxécution de l'appel simple de la multiprocédure *mf* peut être représentée comme sur la figure V.2.

L'appel simple de la multiprocédure *mf* se déroule de la manière suivante :

- distribution des paramètres d'entrée aux divers composants,
- exécution parallèle des composants,
- synchronisation des composants pour la construction et le transfert du résultat à l'appelant,

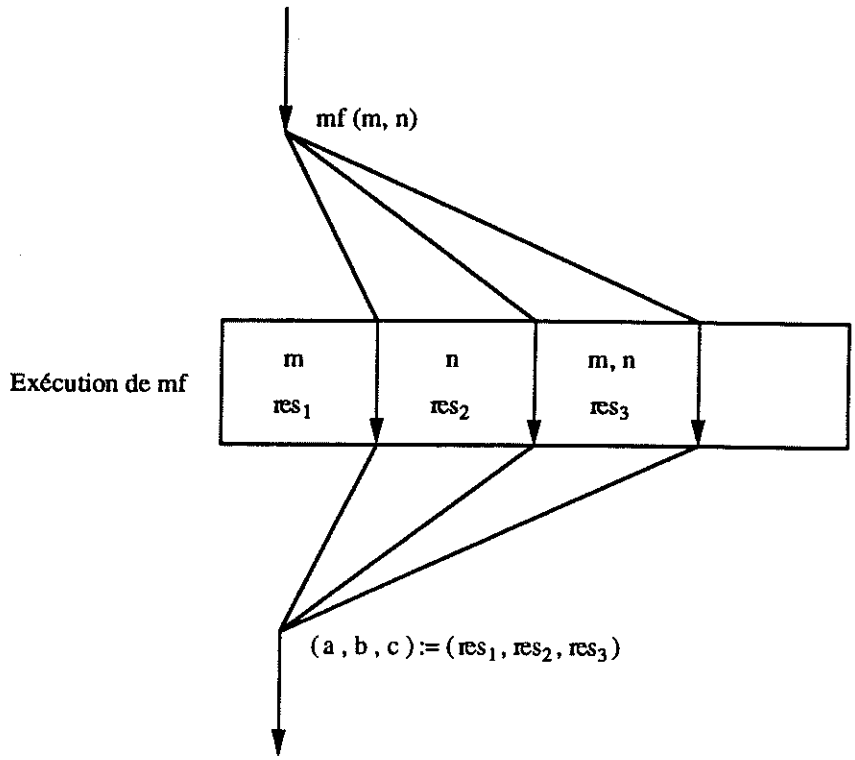


Figure V.2 Schéma de l'exécution d'un appel simple de multiprocédure

- reprise de l'exécution de l'appelant.

Une multiprocédure peut également être appelée par une autre multiprocédure. On parle alors d'appel coordonné ou d'appel de type N-P où N est le nombre de composants de la multiprocédure appelante et P le nombre de composants de la multiprocédure appelée. L'appel coordonné permet d'exprimer l'imbrication de propositions parallèles sous la forme la plus générale. Nous définissons sur la figure V.3 un programme réalisant un appel coordonné 2-3 à la multiprocédure *mf*. Le déroulement de cet appel coordonné est représenté sur la figure V.4. Les composants (1) et (2) du programme appelant se synchronisent pour le transfert des paramètres. Ces derniers sont distribués aux trois composants de la multiprocédure appelée *mf*. Ces trois composants s'exécutent en parallèle puis se synchronisent pour la construction et le transfert du résultat. Finalement, le résultat est distribué aux deux composants du programme appelant.

```
cobegin
  /* Déclarations */
  a : tr1 ; b : tr2 ; c : tr3 ;
  begin
     var m : t1 ;
     ( a , b ) := mf ( p1 = m ) . ( res1 , res2 )
  end //
  begin
     var n : t2 ;
     ( c ) := mf ( p2 = n ) . ( res3 )
  end
coend programme ;
```

Figure V.3 Un appel coordonné 2-3 de multiprocédure

V.2 Le protocole RMPC

V.2.1 Introduction

Le langage POLYGOTH est le langage de programmation des applications distribuées destinées à s'exécuter sur le système GOTHIC. Notre travail a consisté à concevoir et mettre en œuvre dans le cadre de GOTHIC un protocole qui respecte le schéma de contrôle de la multiprocédure. Dans la suite de ce document, ce protocole est appelé protocole d'appel de multiprocédure à distance ou sous forme abrégée protocole RMPC (*Remote MultiProcedure Call*).

V.2.2 Modèle de système et hypothèses sur les pannes

Pour la description des protocoles qui suivent, nous considérons un système distribué composé d'un ensemble de sites reliés par un réseau de communication. Chaque site est constitué d'un processeur associé à une mémoire stable et à une mémoire locale (voir figure V.5).

Les pannes peuvent affecter tant les sites que le réseau de communication. Les sites possèdent la propriété *fail-stop*. La détection d'une erreur matérielle entraîne l'arrêt immédiat du processeur. Aucune instruction erronée ne peut donc être exécutée aléatoirement par un processeur défaillant. Un exemple simple de panne de site est la coupure de courant. De même, le réseau de communication peut être défaillant rendant alors la communication impossible entre certains sites. Dans une telle situation, le réseau est dit partitionné. Chaque partition est donc un sous-ensemble de sites pouvant communiquer entre eux.

Le réseau de communication ne garantit pas une parfaite transmission des messages. Un message peut être perdu c'est-à-dire ne jamais parvenir à son destinataire. Un message

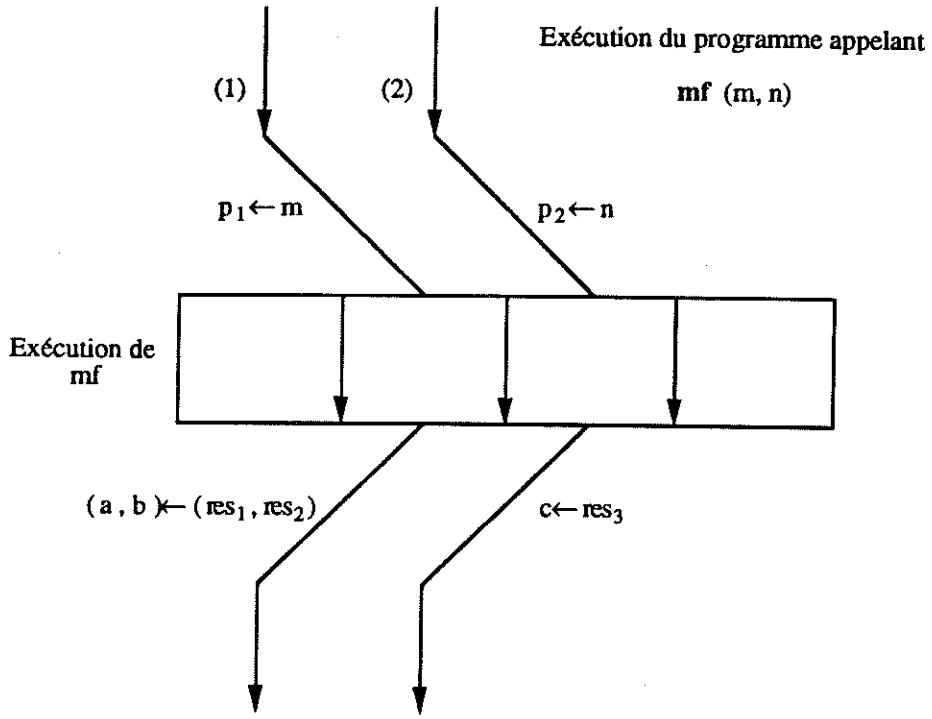


Figure V.4 Déroulement d'un appel coordonné 2-3 de multiprocédure

peut être dupliqué. En d'autres termes, il peut être reçu sans avoir jamais été envoyé. Les messages peuvent être déséquencés. La séquence de messages (m_1, m_2, m_3, m_4) émise dans cet ordre pourra être reçue dans un ordre différent par son destinataire (par exemple (m_2, m_4, m_3, m_1)). Par contre, les messages ne sont jamais altérés au cours de leur transmission. En effet, même s'ils peuvent l'être, l'altération d'un message est toujours détectée par le site destinataire par une vérification systématique du code de contrôle, *checksum*, faisant partie de la couche de communication dite de liaison. Le code de contrôle calculé sur le message à son arrivée est comparé à celui calculé par l'émetteur et transmis en même temps que le message. La non égalité entre les deux valeurs indique que le message est altéré. Il est alors simplement écarté assimilant ainsi pour les couches de niveau supérieur l'altération d'un message à sa perte. Enfin, le temps de transfert d'un message n'est pas borné a priori.

V.2.3 Les propriétés du protocole RMPC

Nous devons définir une sémantique pour le protocole d'appel de multiprocédure à distance comme cela est fait pour les protocoles d'appel de procédure à distance. Notre étude des protocoles d'appel de procédure à distance (voir chapitre II) montre que pour la construction d'applications distribuées très fiables la sémantique *au plus une fois*, choisie notamment dans le système Argus [Lisk87], s'avère la mieux adaptée. En effet, les traitements

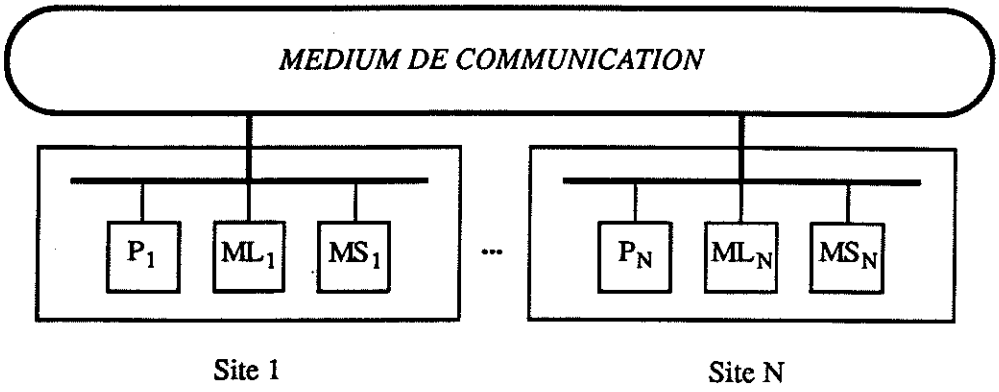


Figure V.5 Modèle du système distribué

d'exception sont considérablement simplifiés puisqu'en cas de panne la procédure appelée n'est exécutée que 0 ou 1 fois et le nombre d'exécutions réalisées est connu. Le principal inconvénient des protocoles d'appel de procédure à distance qui respectent la sémantique *au plus une fois* est leur coût élevé dû à la mise en œuvre de transactions. Nous pensons que cet inconvénient peut être diminué dans le système GOTHIC puisque nous disposons de mémoires stables rapides. C'est pourquoi, nous avons choisi de mettre en œuvre dans le protocole RMPC la sémantique *au plus une fois* ou plus exactement *only once type 2* dans la classification de Spector [Spec82].

Nous précisons la définition de la sémantique *au plus une fois* appliquée à l'appel de multiprocédure à distance. Le protocole d'appel de multiprocédure à distance respecte la sémantique *au plus une fois* si et seulement si les propriétés suivantes sont vérifiées :

(1) En l'absence de panne, la multiprocédure appelée est exécutée exactement une fois c'est-à-dire que tous ses composants s'exécutent exactement une fois.

(2) En présence de panne, la multiprocédure appelée est exécutée exactement 0 ou 1 fois et les composants de la multiprocédure appelante connaissent le nombre d'exécutions ayant eu lieu. 0 exécution de la multiprocédure appelée signifie qu'aucun de ses composants ne s'est exécuté.

Il est intéressant de noter que le cas particulier d'un appel 1-1 de multiprocédure à distance est équivalent à un appel de procédure à distance. Par ailleurs, le cas particulier d'un appel 1-N d'une multiprocédure dont tous les composants sont identiques s'apparente à un appel de procédure parallèle [Mart86] ou à un MultiRPC [Saty86b] mais à la différence près de la sémantique. La propriété 2 énoncée ci-dessus n'est garantie ni pour un appel de procédure parallèle ni pour un MultiRPC. Il ne faut pas non plus négliger le fait qu'un mode d'imbrication général est défini pour les multiprocédures alors que ce n'est pas le cas pour l'appel de procédure parallèle et le MultiRPC.

Le protocole RMPC est tolérant aux fautes dans le sens où il assure la résilience des applications. La propriété de résilience définie dans [Svob84b] peut s'exprimer ainsi :

> *Une application est résiliente si elle peut terminer son exécution en dépit des pannes qui peuvent survenir dans les divers composants du système.*

Une méthode pour rendre une application résiliente est de sauvegarder périodiquement l'état courant de l'application dans un point de reprise. L'application peut ainsi en cas de panne reprendre son exécution à partir de son dernier point de reprise soit sur un autre site soit sur le site tombé en panne après le recouvrement. Les calculs effectués avant le dernier point de reprise n'ont pas besoin d'être réexécutés.

Dans GOTHIC, nous avons suivi cette approche. Le protocole RMPC est exécuté au dessus du noyau de processus stables. Un processus stable est un processus dont l'état est sauvegardé en mémoire stable. Le protocole RMPC sauvegarde des points de reprise des processus communiquant de façon cohérente et garantit la résilience des applications. La sauvegarde de points de reprise et l'hypothèse de pannes de durée finie nous permettent de traiter le problème des exélines de façon simple. En effet, puisque tout processus reprend son exécution au bout d'un temps fini à partir d'un point de reprise qui a été sauvegardé de façon consistante un processus n'est orphelin que pendant la durée de la panne d'un des sites clients (dans le cas des multiprocédures, il peut y avoir plusieurs sites clients puisque les multiprocédures peuvent être imbriquées). Toute exéline est récupérée lors du recouvrement. Aucun algorithme d'élimination n'est nécessaire. La récupération des exélines est automatique lors du recouvrement après panne.

V.2.4 Le transfert de contrôle aux composants appelés

Nous considérons dans ce paragraphe un appel simple de multiprocédure à distance. Le transfert de contrôle pour un appel coordonné de multiprocédure se déroule de la même façon que pour un appel simple mais il est précédé par une phase de synchronisation des composants de la multiprocédure appelante, décrite dans la suite.

Dans un appel de procédure à distance, le transfert de contrôle de l'appelant vers l'appelé se fait par l'envoi d'un message de type *APPEL*. Le protocole d'appel de procédure à distance assure qu'un seul message *APPEL* donne lieu à l'exécution de la procédure appelée pour un appel donné en dépit des retransmissions indispensables du fait de la non fiabilité des sites et du réseau de communication.

Pour un appel de multiprocédure à distance, un message de type *APPEL_MP* (analogue au message de type *APPEL* de l'appel de procédure à distance) est transmis à tous les sites sur lesquels doivent s'exécuter les composants de la multiprocédure appelée (*sites appelés*). Afin de respecter la sémantique que nous avons définie pour le protocole RMPC, en présence

de pannes, le message *APPEL_MP* doit être reçu par tous les sites appelés ou par aucun d'entre eux. Le transfert de contrôle aux composants de la multiprocédure appelée est donc réalisé par une diffusion atomique.

Cette propriété de la diffusion n'est cependant pas suffisante. Les messages *APPEL_MP* ne doivent pas en effet être delivrés dans n'importe quel ordre aux sites appelés. Un exemple simple permet de nous en convaincre (voir figure V.6).

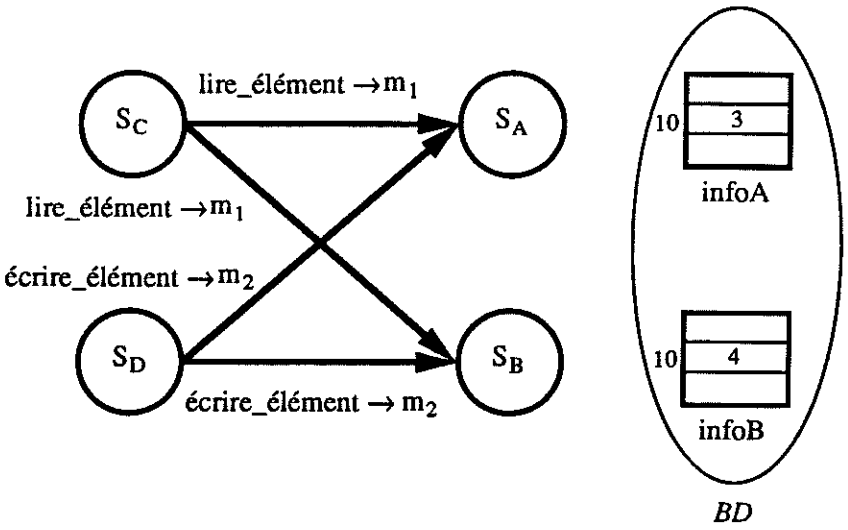


Figure V.6 Manipulation de la base de données *BD*

Considérons une base de données *BD*. Un élément de *BD* est composé de deux parties *infoA* et *infoB*. La partie *infoA* de chaque élément réside sur le site S_A tandis que la partie *infoB* réside sur le site S_B. Par exemple, pour l'élément 10 de la base de données, *infoA* vaut 3 et *infoB* vaut 4.

L'accès à un élément en lecture (respectivement en écriture) s'effectue à l'aide de la multiprocédure *lire_élément* (respectivement *écrire_élément*) (voir figure V.7) ; *lire_élément* (respectivement *écrire_élément*) est constituée de deux composants, l'un s'exécutant sur S_A et l'autre sur S_B. Chaque composant réalise la lecture (respectivement l'écriture) de la partie d'élément sur son site.

L'appel à *lire_élément* par un processus s'exécutant sur S_C entraîne la diffusion du message m_1, contenant le paramètre d'entrée (10) de la multiprocédure *lire_élément*, à l'ensemble des sites S_A et S_B.

L'appel à *écrire_élément* par un processus s'exécutant sur S_D entraîne la diffusion du message m_2, contenant les paramètres d'entrée (10,0,0) de la multiprocédure *écrire_élément*, à l'ensemble des sites S_A et S_B. L'ordre de réception des messages m_1 et m_2 par S_A et S_B n'est pas anodin. En effet, pour que le résultat de l'exécution de chacune des multiprocédures

```
multiproc lire_élément (élém : ent) : (a : t_infoA, b : t_infoB) ;
cobegin
  begin (élém) : a
    a := infoA[élém]
  end //
  begin (élém) : b
    b := infoB[élém]
  end
coend lire_élément ;

multiproc écrire_élément (élém : ent, a : t_infoA, b : t_infoB) ;
cobegin
  begin (élém, a)
    infoA[élém] := a
  end //
  begin (élém, b)
    infoB[élém] := b
  end
coend écrire_élément ;
```

Figure V.7 Définition des multiprocédures *lire_élément* et *écrire_élément*

soit cohérent, il importe que les deux sites S_A et S_B reçoivent m_1 et m_2 dans le même ordre.

Si l'ordre d'arrivée des messages est (m_1, m_2) sur S_A et (m_2, m_1) sur S_B, le résultat (3,0) de l'exécution de la multiprocédure *lire_élément* n'est pas celui escompté. Le champ *infoA* contient sa valeur avant écriture (3) tandis que le champ *infoB* contient sa valeur après écriture (0).

Il en ressort que le protocole de diffusion fiable utilisé par le protocole RMPC doit garantir que les messages diffusés sont reçus dans le même ordre par tous les destinataires communs. Autrement dit :

> *Si deux messages m_1 et m_2 sont diffusés à deux sites S_j et S_k, alors S_j et S_k recevront les messages m_1 et m_2 dans un ordre identique.*

Ainsi, si cette propriété est respectée, le résultat de *lire_élément* dans l'exemple est soit (3,4) soit (0,0) suivant que l'ordre d'arrivée des messages est (m_1, m_2) ou (m_2, m_1).

Une propriété supplémentaire est également indispensable pour assurer l'ordre des messages diffusés.

> *Si deux messages m_1 et m_2 sont diffusés dans cet ordre par un site S_i alors m_1 sera reçu avant m_2 par tous les sites destinataires de m_1 et de m_2.*

Nous en donnons la justification en nous appuyant sur l'exemple précédent. Supposons que *lire_élément* et *écrire_élément* soient exécutées par deux processus localisés sur un même

site. Faisons en outre l'hypothèse que la synchronisation entre deux processus soit telle que l'appel à *écrire_élément* précède l'appel à la multiprocédure *lire_élément*. Le résultat rendu par *lire_élément* doit être la valeur écrite par *écrire_élément*. Les messages m_1 et m_2 diffusés dans l'ordre (m_2, m_1) doivent être reçus dans cet ordre par S_A et S_B.

En résumé, le transfert de contrôle aux composants de la multiprocédure appelée est réalisé par une diffusion fiable qui possède les propriétés suivantes :

(1) atomicité : en présence de pannes, un message diffusé est reçu soit par tous ses destinataires soit par aucun d'entre eux,

(2) les messages diffusés par un même site sont reçus dans l'ordre d'émission par tous les destinataires communs,

(3) les messages diffusés sont reçus dans un même ordre par leurs destinataires communs.

V.2.5 La synchronisation des composants d'une multiprocédure

Au cours d'un appel de multiprocédure à distance, la synchronisation des composants soit de la multiprocédure appelante soit de la multiprocédure appelée est nécessaire. Dans le cas d'un appel coordonné, les composants de la multiprocédure appelante se synchronisent avant l'envoi par l'un d'entre eux du message *APPEL_MP*. De même à l'issue de l'exécution des composants de la multiprocédure appelée (quel que soit le type d'appel), ceux-ci se synchronisent pour la construction du résultat et son transfert par l'un d'entre eux aux composants de la multiprocédure appelante.

Nous décrivons dans ce paragraphe le protocole de synchronisation des composants d'une multiprocédure. L'ensemble des composants d'une multiprocédure forme un groupe dont l'un des membres est appelé *coordinateur*. L'identité du coordinateur est connue de tous les membres du groupe. Lorsque les composants d'une multiprocédure doivent se synchroniser, ils envoient un message de synchronisation au coordinateur de leur groupe. Le coordinateur collecte les messages de synchronisation. Il doit recevoir un et un seul message de chacun des autres composants pour une synchronisation donnée. Autrement dit, le transfert des messages de synchronisation doit être fiable c'est-à-dire que ces messages ne doivent pas être perdus ou dupliqués malgré l'occurence de pannes. Un protocole de communication fiable est employé pour assurer ces deux propriétés.

Dans le protocole RMPC, le message de synchronisation est de type *APPEL_PARTIEL* lorsqu'il s'agit de synchroniser les composants de la multiprocédure appelante lors d'un appel coordonné et de type *RESUL_PARTIEL* lorsqu'il s'agit de synchroniser les composants de la multiprocédure appelée pour la construction du résultat.

V.2.6 Le retour du résultat aux composants de la multiprocédure appelante

Une fois que tous les composants de la multiprocédure appelée sont synchronisés (le coordinateur a reçu un message de type *RESUL_PARTIEL* de chacun des autres composants), le coordinateur construit un message de type *RESUL_MP* analogue au message de type *RETOUR* utilisé dans les protocoles d'appel de procédure à distance. A la différence de l'appel de procédure à distance, ce message doit être transmis à tous les composants de la multiprocédure appelante dans le cas d'un appel coordonné. La diffusion du message *RESUL_MP* à tous les composants de la multiprocédure appelante doit être atomique. Aucun ordre particulier ne doit être assuré sur l'ensemble des messages de type *RESUL_MP*.

Dans le cas d'un appel simple, le message de type *RESUL_MP* est transmis par le protocole de communication fiable par message (également employé pour les messages de synchronisation) qui n'est qu'un cas particulier du protocole de diffusion atomique.

V.2.7 L'appel simple de multiprocédure

Un appel simple de multiprocédure à distance se déroule en quatre étapes (fig. V.8) :

(1) Transmission du contrôle et des paramètres aux composants de la multiprocédure appelée par la diffusion atomique ordonnée du message de type *APPEL_MP* puis attente du résultat de l'exécution de la multiprocédure.

(2) Exécution des composants de la multiprocédure appelée.

(3) Synchronisation des composants de la multiprocédure appelée pour la construction du résultat. Le coordinateur collecte les résultats partiels calculés par les autres composants de la multiprocédure appelée. Ceux-ci lui envoient un message fiable de type *RESUL_PARTIEL* contenant la partie du résultat qu'ils ont calculée.

(4) Retour du résultat à l'appelant. Le coordinateur lui envoie un message fiable de type *RESUL_MP*. A la réception du message *RESUL_MP*, l'appelant reprend son exécution.

V.2.8 L'appel coordonné de multiprocédure

L'appel coordonné, préalablement à la diffusion fiable du message *APPEL_MP*, comporte une phase de synchronisation des composants de la multiprocédure appelante (voir figure V.9). Le coordinateur attend un message de type *APPEL_PARTIEL* de chacun des

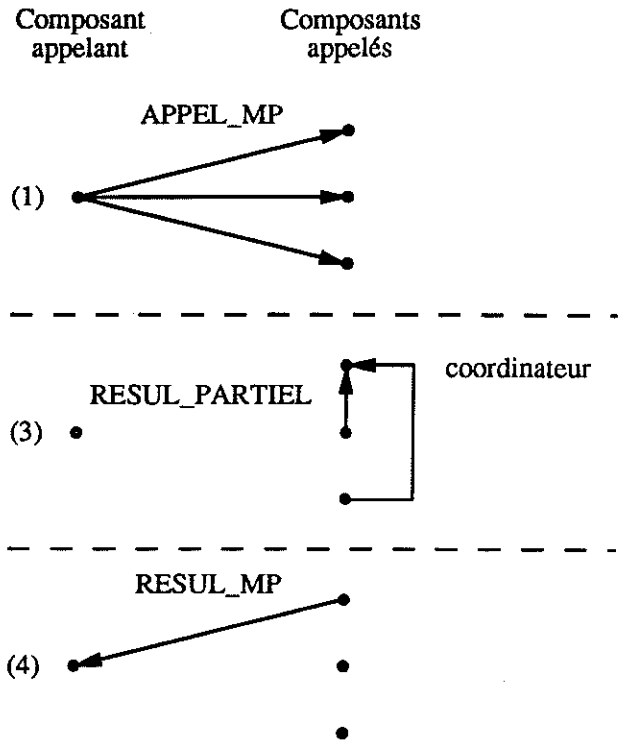


Figure V.8 Un appel simple de multiprocédure

autres composants appelants. Un tel message contient les paramètres d'entrée fournis par le composant émetteur.

Une fois collectés tous les messages de type *APPEL_PARTIEL*, le coordinateur de la multiprocédure appelante envoie (par une diffusion atomique ordonnée) à tous les sites sur lesquels vont s'exécuter les composants de la multiprocédure appelée un message de type *APPEL_MP* comme dans le cas de l'appel simple.

Lorsque les composants appelés ont terminé leur exécution, ils se synchronisent comme dans le cas d'un appel simple pour la construction du résultat. Le coordinateur de la multiprocédure appelée diffuse alors atomiquement le message de type *RESUL_MP* à tous les composants de la multiprocédure appelante.

V.3 Conclusion

Nous avons présenté dans ce chapitre le protocole RMPC. Trois types de primitives de communication sont utilisées dans ce protocole :

- l'envoi d'un message fiable m à un site S,

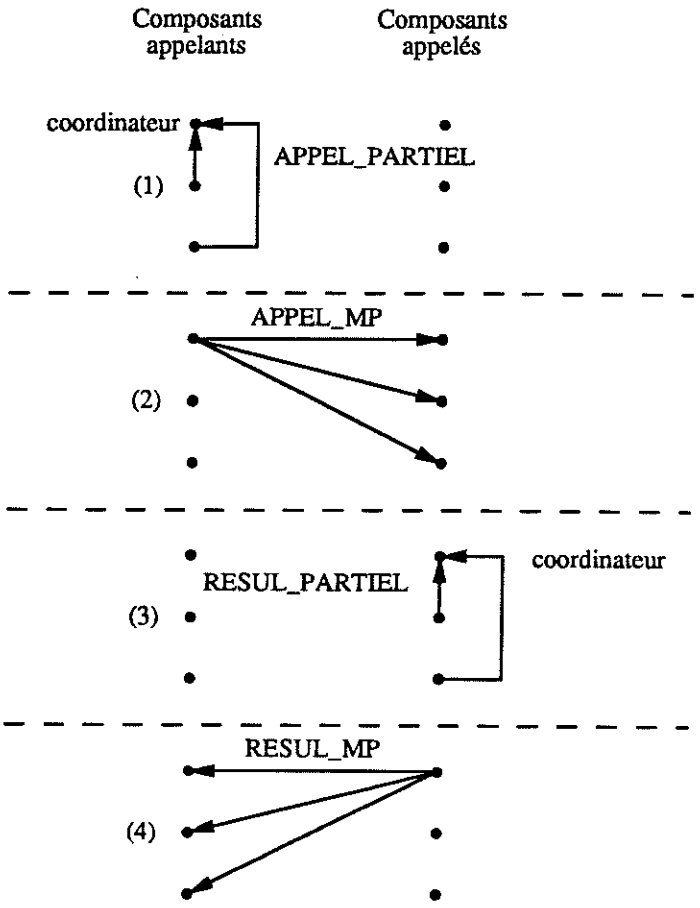


Figure V.9 Un appel coordonné de multiprocédure

- la diffusion atomique d'un message à un ensemble de sites,
- la diffusion atomique ordonnée d'un message à un ensemble de sites.

Ces primitives de communication fiable ont été mises en œuvre dans le cadre du système GOTHIC. La troisième partie de ce document est consacrée au système de communication fiable de GOTHIC. Nous y décrivons deux protocoles, les protocoles RMC et RBC. Le protocole RMC (*Reliable Message Communication*) est un protocole de communication fiable par messages. Ce protocole permet aussi de diffuser atomiquement un message à un ensemble de sites. Le protocole RBC (*Reliable Broadcast Communication*) est un protocole de diffusion atomique ordonnée qui est mis en œuvre au dessus du protocole RMC. L'ordre sur les messages diffusés est le même que celui assuré par le protocole ABCAST [Birm87]. Les protocoles RMC et RBC sont originaux par rapport à ceux que nous avons décrits dans la première partie de ce document dans la mesure où ils reposent sur l'utilisation de mémoires stables rapides.

Troisième Partie

Conception de protocoles de communication fiable pour la mise en œuvre du protocole RMPC

Cette troisième partie décrit les protocoles RMC et RBC introduits dans la présentation du protocole RMPC. Ces protocoles ont été mis en œuvre dans le cadre du système GOTHIC. Le chapitre VI est consacré à la présentation des aspects matériels et logiciels du système GOTHIC. Nous décrivons ensuite le protocole RMC dans le chapitre VII puis le protocole RBC dans le chapitre VIII. Enfin, le chapitre IX est réservé à la mise en œuvre et à l'analyse des performances des protocoles RMC et RBC.

Chapitre VI

Présentation de Gothic

VI.1 Organisation du système Gothic

L'organisation générale du système GOTHIC est représentée sur la figure VI.1.

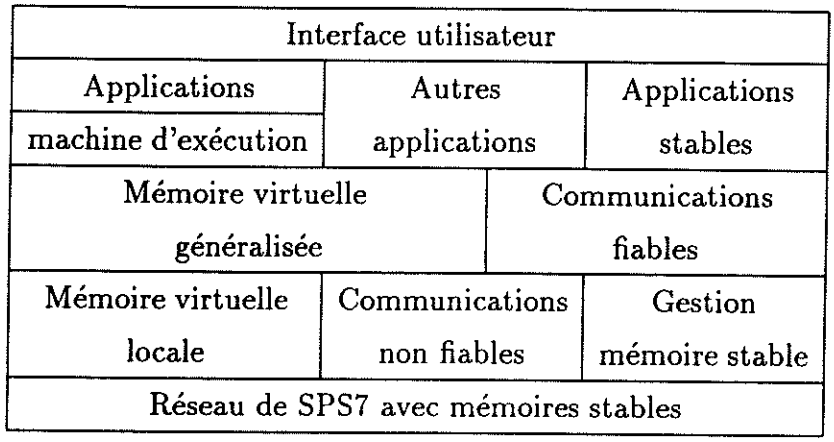


Figure VI.1 Organisation du système Gothic

Au niveau matériel, on trouve le réseau de multiprocesseurs tolérant les fautes ; cette tolérance aux fautes est obtenue par association d'une mémoire stable à chaque processeur composant un multiprocesseur.

Au niveau 2, on trouve les services de base du noyau GOTHIC, à savoir, la mémoire virtuelle locale à un processeur, les communications non fiables (datagrammes Ethernet, boîtes aux lettres) et la gestion de la mémoire stable.

Au niveau 3, se trouvent la mémoire virtuelle généralisée, ainsi que le système de communication fiable construit sur la mémoire stable.

Le niveau 4 est constitué d'une machine d'exécution POLYGOTH et d'applications POLYGOTH ainsi que d'applications utilisant plus directement les facilités offertes par les

niveaux inférieurs.

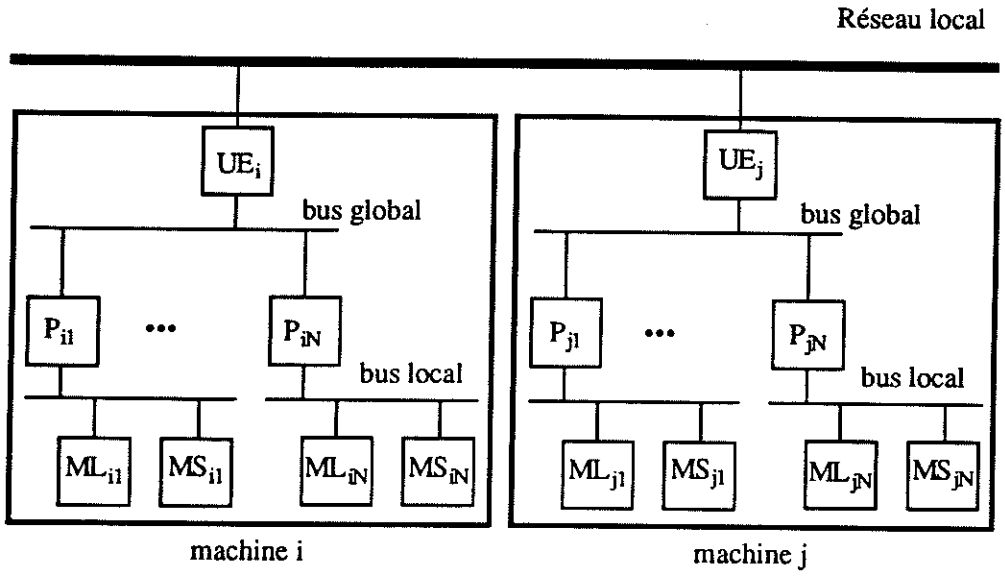
Enfin, le niveau 5 contient l'interface offerte à l'utilisateur.

Dans le paragraphe VI.2, nous nous intéressons à l'architecture matérielle du système GOTHIC et spécialement à la mémoire stable que nous utilisons dans les protocoles RMC et RBC. Les aspects logiciels sont étudiés dans le paragraphe VI.3. Nous concentrons notre attention sur la gestion de la mémoire stable et des communications.

VI.2 Architecture matérielle

VI.2.1 Vue d'ensemble

L'architecture matérielle de GOTHIC est constituée d'un ensemble de machines multiprocesseurs interconnectées par un réseau local (fig. VI.2).



P : processeur
ML : mémoire locale
MS : mémoire stable
UE : unité d'échange Ethernet

Figure VI.2 L'architecture matérielle de GOTHIC

Chaque processeur peut accéder sur son bus local une mémoire locale et une mémoire stable rapide. La description de la mémoire stable fait l'objet du paragraphe suivant. Les

processeurs d'une même machine sont interconnectés par le bus global. Les machines sont des machines BULL SPS7. Elles peuvent comporter jusqu'à huit processeurs, de type MOTOROLA 68020. Dans notre configuration, les processeurs possèdent entre 4 et 12 Mega-octets de mémoire locale. Le réseau local est un réseau Ethernet [Metc76]. L'accès d'une machine au réseau Ethernet se fait par l'intermédiaire d'une unité d'échange Ethernet [Bull86].

VI.2.2 La mémoire stable rapide

L'originalité de l'architecture que nous venons de décrire réside dans l'association d'une mémoire stable à chaque processeur. La mémoire stable rapide, développée dans le cadre du projet GOTHIC [Bana88a], est composée de deux bancs de mémoire vive rapide. L'alimentation de la mémoire stable, distincte de l'alimentation du reste la machine, est secourue par une batterie pour masquer les coupures de courant. La vérification de la validité des accès aux objets et l'autoprotection sont mis en œuvre par un contrôleur intelligent fonctionnant indépendamment du processeur (figure VI.3).

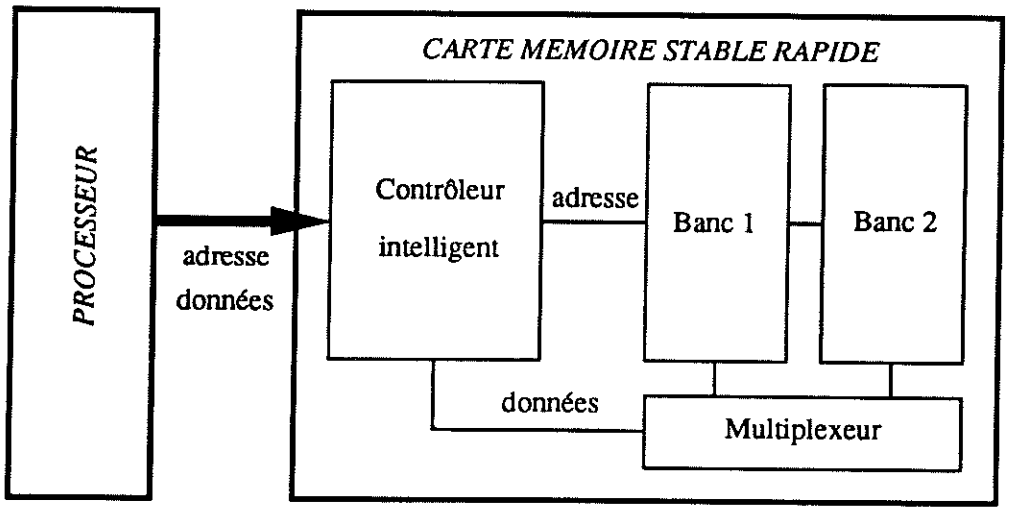


Figure VI.3 Schéma de la carte mémoire stable rapide

La capacité utile de la mémoire stable est de 1 Mega-octet (compte-tenu des progrès de la technologie, une capacité 4 fois plus importante serait désormais envisageable). La mémoire stable est accédée par mots de 32 bits. En résumé, la mémoire stable possède les propriétés suivantes :

(1) non volatilité,

(2) mise à jour atomique (par un protocole à deux phases) des informations (objets) qui y sont conservées,

(3) protection contre les fautes externes,

(4) temps d'accès de l'ordre des temps d'accès à la mémoire vive.

La thèse de Gilles Muller [Mull88] décrit la réalisation d'un noyau de système sûr de fonctionnement utilisant ce type de mémoires. Le lecteur intéressé y trouvera une description détaillée de la mémoire stable.

VI.2.3 Les média de communication

La communication entre deux processseurs utilise des média de communication différents selon que les processeurs sont situés dans la même machine multiprocesseur ou dans deux machines distinctes. La communication à l'intérieur d'une machine se fait via le bus global de la machine [Bull85] ; les communications intermachines utilisent un réseau local de type Ethernet [Metc76].

VI.3 Architecture logicielle

VI.3.1 Le noyau Gothic

Le noyau GOTHIC est conçu de façon modulaire. Il est constitué d'un noyau interne autour duquel viennent se greffer des agences (fig. VI.4).

Le noyau interne respecte la norme SCEPTRE [Brow84]. Chaque agence offre un service. L'entité de base de GOTHIC étant l'objet, chaque agence met en œuvre une classe d'objets. Elle fournit aux utilisateurs un ensemble de primitives qui leur permettent de créer, manipuler ou détruire les objets de la classe. Une agence a accès aux données du noyau interne et utilise les fonctions de celui-ci. Les agences de GOTHIC sont :

- agence de gestion des processus,
- agence de gestion des programmes (objet contenant le code d'un processus),
- agence de gestion des segments mémoire,
- agence de gestion des boîtes aux lettres,
- agence de gestion des sémaphores,
- agence de gestion des délais,
- agence de gestion des entrées/sorties,

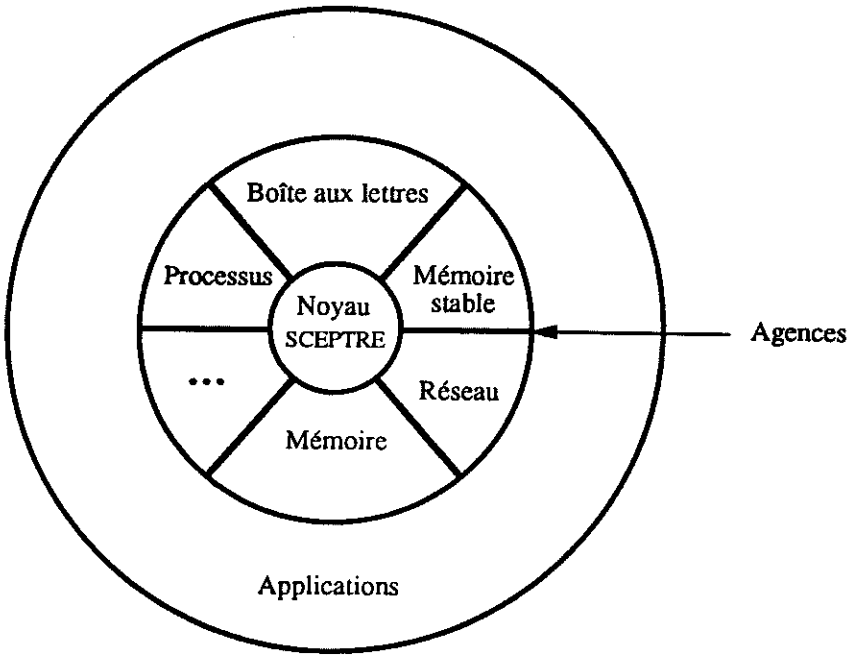


Figure VI.4 Structure du noyau GOTHIC

- agence de gestion de la mémoire stable,
- agence de gestion des noms symboliques des objets,
- agence de gestion du réseau.

Cette conception modulaire du noyau GOTHIC rend aisés les modifications et les ajouts de code. Ainsi, la réalisation de la mémoire virtuelle de GOTHIC a consisté principalement à modifier l'agence mémoire [Mich89]. L'intégration de la mémoire stable s'est traduite par l'ajout d'une nouvelle agence dans GOTHIC [Mull88]. Nous décrivons un peu plus en détail certaines agences dont nous utilisons les primitives dans la mise en œuvre de l'exécution de l'appel de multiprocédure. Ainsi, nous nous intéressons tout d'abord à l'agence mémoire stable puis aux agences traitant les communications : l'agence boîte aux lettres et l'agence réseau.

VI.3.2 L'agence mémoire stable

L'agence mémoire stable assure la gestion de la ressource mémoire stable rapide. Elle offre deux classes de services : des services de bas niveau offrant un accès direct aux fonctions de la mémoire stable et des services de haut niveau facilitant la mise en œuvre d'applications sûres de fonctionnement.

Les services de bas niveau

L'utilisateur qui emploie la mémoire stable pour mettre en œuvre un service sûr de fonctionnement doit rendre «visible» dans son espace d'adressage logique une zone de la mémoire stable. La gestion de la mémoire dans GOTHIC est segmentée. La notion de segment a donc été étendue par la notion de segment stable. Les quatre primitives atomiques de manipulation des segments stables sont :

- *créer_segment_stable* pour créer un segment en mémoire stable,
- *détruire_segment_stable* pour détruire un segment en mémoire stable,
- *accéder_segment_stable* pour introduire un segment de mémoire stable dans un espace d'adressage,
- *déaccéder_segment_stable* pour retirer un segment de mémoire stable d'un espace d'adressage.

A l'intérieur d'un segment stable, l'utilisateur peut créer des objets stables qu'il doit manipuler à l'aide des primitives de base d'accès à la mémoire stable. Ces opérations sont la création, l'écriture, la lecture et la destruction d'un objet stable et enfin la mise à jour atomique d'un groupe d'objets stables. Toutes ces primitives sont mises en œuvre par matériel dans la carte mémoire stable. Nous en donnons la définition précise :

- *créer_objet_stable ([in] taille, [in] deb_stb).*

  Une zone mémoire de la taille passée en paramètre est réservée en mémoire stable à partir de l'adresse de début *deb_stb.*

- *détruire_objet_stable ([in] deb_stb, [in] taille).*

  La zone mémoire dont l'adresse de début et la taille sont passées en paramètre est libérée.

- *lire_objet_stable ([in] deb_stb, [in] taille, [in] deb_ram).*

  L'objet stable situé à l'adresse *deb_stb* et de taille *taille* est recopié en mémoire vive à partir de l'adresse *deb_ram.*

- *écrire_objet_stable ([in] deb_ram, [in] taille, [in] deb_stb).*

  La zone de mémoire vive commençant à l'adresse *deb_ram* (objet *OB*) et dont la taille est passée en paramètre est recopiée à partir de l'adresse *deb_stb* en mémoire stable.

  Cette opération atomique comporte deux phases (voir figure VI.5) :

- phase 1 : écriture de l'objet *OB* sur le banc 1 (dans *OB1*)
- phase 2 : si et seulement si la phase 1 est terminée et s'est correctement déroulée (vérification par matériel), l'objet est recopié du banc 1 vers le banc 2 (de *OB1* vers *OB2*).

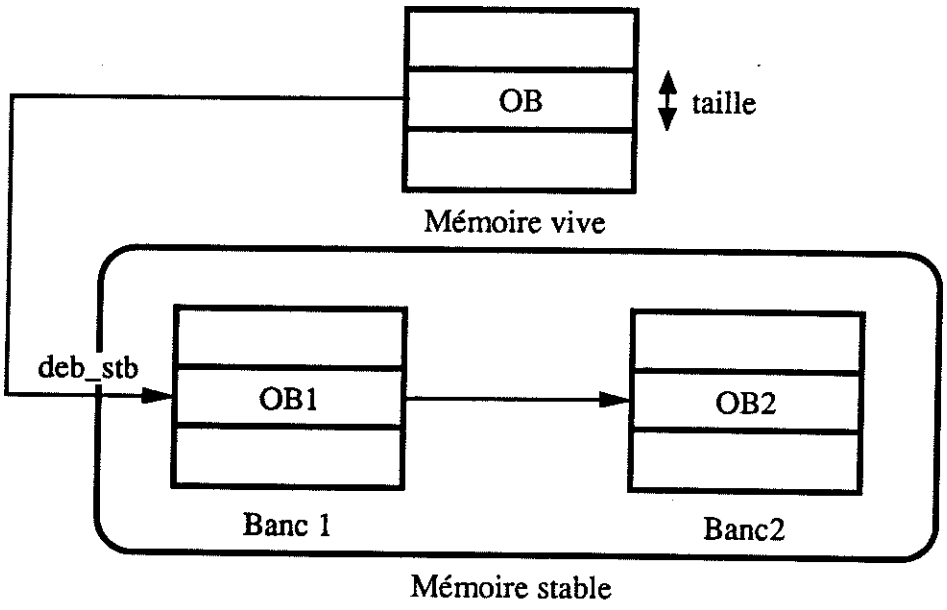


Figure VI.5 Ecriture atomique d'un objet en mémoire stable

- *maj_atomique_groupe_objet ([in] liste_desc_obj).*

  Le paramètre d'entrée est une liste de descripteurs d'objets. Un descripteur comporte l'adresse en mémoire vive de l'objet, son adresse de début en mémoire stable et sa taille.

  Cette primitive comporte deux phases comme la primitive *écrire_objet_stable*, chaque phase étant répétée pour chaque objet. Lors de la première phase, tous les objets sont recopiés sur le premier banc. Si cette phase s'est déroulée correctement tous les objets sont recopiés sur le banc 2 lors de la seconde phase.

Nous soulignons l'importance de la primitive d'écriture atomique d'un groupe d'objets stables. L'utilisation de cette primitive est indispensable pour la mise en œuvre des protocoles de communication présentés par la suite. Elle y est très largement employée. A l'issue de son exécution et en l'absence de défaillance, tous les objets du groupe sont mis à jour. En cas de défaillance survenant pendant l'exécution, tous les objets du groupe conservent leur état initial. Il n'y a aucun autre état possible.

A chaque type d'opération atomique correspond un automate intégré au contrôleur de la mémoire stable. Chaque phase d'une opération atomique correspond à un état de l'automate associé. Avant d'exécuter une des phases, le processeur écrit dans le registre de commande de la mémoire stable l'état correspondant. La transition entre phases n'est acceptée que si l'automate valide la transition entre les deux états associés. Les automates correspondant aux primitives *écrire_objet_stable* et *maj_atomique_groupe_objet* sont donnés dans l'annexe 1.

Les services de haut niveau

Les services de haut niveau offrent des primitives pour la construction d'applications sûres de fonctionnement. Ces primitives affranchissent l'utilisateur de la gestion directe des objets en mémoire stable. Deux types de services de haut niveau sont disponibles : la gestion des processus stables et la gestion des consoles séries.

Un processus est dit stable à partir du moment où un point de reprise existe pour ce processus. Un point de reprise est un ensemble d'objets stables contenant toutes les informations qui permettent de redémarrer un processus après le traitement d'une faute. La primitive *sauver_point_de_reprise* (sans paramètre) permet la création et la sauvegarde d'un point de reprise. Elle est mise en œuvre à l'aide de la primitive *maj_atomique_groupe_objet*. Les points de reprise dans GOTHIC sont utilisés lors du redémarrage du système suite à une défaillance.

Sur une console série, les affichages effectués par un processus stable doivent se poursuivre correctement à la reprise du processus après une panne. Cependant, le bon fonctionnement de la voie série dépend de paramètres tels que la vitesse qui sont fixés à l'ouverture de la voie. Ces paramètres sont conservés en mémoire stable afin de permettre la réinitialisation de la voie série lors de la reprise.

VI.3.3 L'agence boîtes aux lettres

La communication interprocessus à l'intérieur d'une machine multiprocesseur se fait par l'intermédiaire de boîtes aux lettres. Une boîte aux lettres est une file FIFO destinée à contenir des messages de taille fixe, associée à des mécanismes de synchronisation. Les primitives de l'agence boîte aux lettres sont :

- *créer_bal ([in]attribut, [in] msg_nb, [out]bal_id, [out]état)*
  Cette primitive permet de créer une boîte aux lettres de taille fixe (exprimée en nombre de messages par le paramètre *msg_nb*) soit en mémoire locale, soit en mémoire globale selon la valeur du paramètre *attribut*. La primitive rend comme résultat l'identificateur de la boîte aux lettres créée (*bal_id*) et un compte rendu d'erreur.

- *détruire_bal ([in]bal_id, [out]état)*
  La boîte aux lettres identifiée par *bal_id* est détruite.

- *receive ([in]bal_id, [in]délai, [out]message, [out]état)*
  Le message situé en tête de la file FIFO de la boîte aux lettres *bal_id* est délivré au processus appelant cette primitive. Si aucun message n'est présent, le processus est mis en attente pendant le délai passé en paramètre dont la valeur est comprise entre zéro et l'infini.

- *send([in]message, [in]bal_id,[in]délai, [out]état)*
  Le message *message* est envoyé vers la boîte aux lettres *bal_id.* L'envoi peut être mis en attente pendant le délai passé en paramètre (compris entre zéro et l'infini) dans le cas où la boîte aux lettres est pleine.

VI.3.4 L'agence réseau

Les communications intermachines sont mises en œuvre par l'agence réseau. Celle-ci offre l'interface IP [Post81a], l'interface UDP [Post80] et l'interface TCP [Post81b]. Le protocole IP permet d'acheminer des datagrammes à travers un réseau, les machines source et destinataire étant identifiées par une adresse Internet. Le protocole UDP offre la notion de port de façon à ce que plusieurs programmes puissent utiliser UDP en même temps. Les datagrammes UDP sont acheminés par le protocole IP. Le protocole TCP permet de transmettre des messages de taille supérieure à la taille d'un datagramme. Le message est fractionné en plusieurs datagrammes (transmis par le protocole IP) que TCP réassemble à la réception. En outre, TCP gère la perte des messages en assurant la retransmission des datagrammes quand cela est nécessaire. Une bonne introduction à ces protocoles peut être trouvée dans [Hedr87].

L'agence réseau est réalisée au dessus du logiciel de communication de base SC7 conçu pour machine BULL SPS7 [Bull88].

VI.3.5 L'agence des communications fiables

Les protocoles de communication que nous mettons en œuvre dans GOTHIC en vue de l'exécution atomique des appels de multiprocédure (protocole RMPC) sont regroupés dans une nouvelle agence. Cette agence est intégrée au noyau GOTHIC. Le système de communication fiable que nous proposons dans GOTHIC est structuré en six couches (fig. VI.6).

Le niveau (1) est la couche matérielle du système de communication. Il comporte les deux média de communication de l'architecture matérielle de GOTHIC : le bus global SM-BUS et le réseau local Ethernet. Le niveau (2) est une couche de logiciel existant dans le

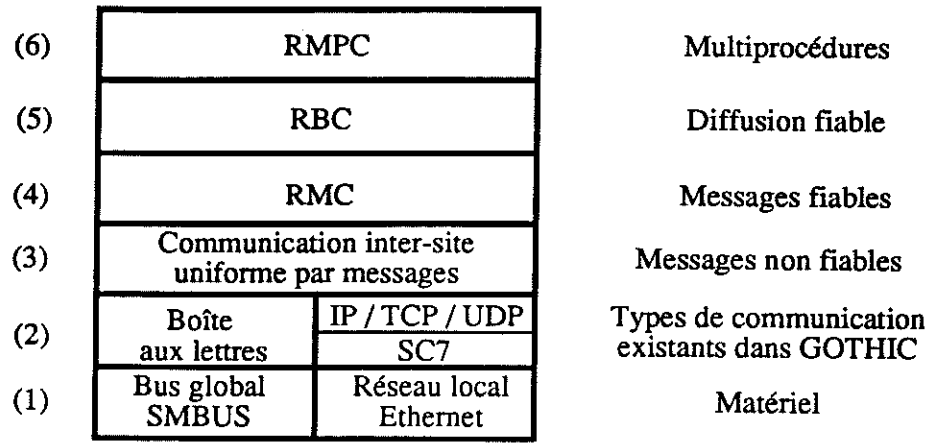


Figure VI.6 Structure du système de communication fiable

noyau GOTHIC. Il se subdivise en deux parties : l'agence boîte aux lettres et l'agence réseau. La première d'entre elles, l'agence boîte aux lettres, offre les primitives permettant à deux processus situés à l'intérieur d'une même machine de communiquer via le bus global. La seconde fournit aux processus situés dans des machines distinctes des primitives de communication via le réseau Ethernet. Les protocoles de communication réseau sont standard de type TCP/IP et UDP. Afin de rendre uniformes les communications interprocessus, nous avons jugé nécessaire de construire un niveau de logiciel supplémentaire (niveau (3)). Ce niveau utilise directement les primitives offertes par le niveau (2) après avoir déterminé la localisation relative des processus voulant communiquer : même machine, machines différentes. Une description en est donnée dans le paragraphe IX.3.1.

Le niveau (4) est la première couche de communication fiable. Elle offre un service de communication fiable par messages. La description de ce service fait l'objet du chapitre suivant. Le niveau (5) offre une primitive de diffusion fiable atomique ordonnée permettant à un processus d'envoyer un message vers N destinataires. Le chapitre VIII est consacré à ce protocole. Enfin, le dernier niveau (6) réalise le protocole d'appel de multiprocédure à distance, qui a été présenté dans le chapitre V.

Chapitre VII

Le protocole de communication fiable par messages (RMC)

VII.1 Introduction

Nous présentons dans ce chapitre le protocole de communication fiable par messages (RMC) mis en œuvre dans le système GOTHIC. L'originalité de ce protocole réside dans l'utilisation de la mémoire stable [Mori90].

Nous exposons tout d'abord le principe de fonctionnement du protocole RMC puis nous donnons les primitives qui constituent son interface. Le protocole RMC comporte deux parties : la première assure l'acheminement des messages présents dans la mémoire stable et la seconde permet aux usagers du service RMC de déposer des messages dans la mémoire stable ou de les retirer. Ces parties font l'objet des paragraphes VII.4 et VII.5 respectivement. Le comportement du protocole dans les cas de panne est étudié dans le paragraphe VII.6. Nous terminons ce chapitre par une vérification du protocole.

VII.2 Principe du protocole RMC

L'objectif du protocole RMC est d'offrir un service de communication fiable pour la mise en œuvre du protocole RMPC dans le système GOTHIC. L'exécution de chaque composant d'une multiprocédure est réalisée par un processus stable géré par le noyau sûr de fonctionnement de GOTHIC. Un processus stable est un processus qui peut être relancé après l'occurrence d'une faute à partir d'un point de reprise précédemment mémorisé. Les points de reprise des processus stables sont rangés en mémoire stable. Chaque processus stable possède un et un seul point de reprise. La sauvegarde d'un point de reprise détruit le point de reprise précédent.

Les clients du service RMC sont donc les processus stables. Les processus stables peuvent s'échanger des messages par l'intermédiaire de *ports stables*, gérés par le protocole RMC.

Un port stable est une structure de données (analogue à une boîte aux lettres) rangée en mémoire stable. Un port stable est créé par un processus stable, dit propriétaire du port stable, qui est le seul processus pouvant lire les messages présents dans le port stable. Un port stable ne peut être détruit que par son propriétaire. Un port stable est identifié de manière unique dans le système par un *uid*, qui est un couple de la forme (numéro de site, numéro local). Les ports stables peuvent aussi être désignés par un nom symbolique. Un mécanisme de désignation permet de retrouver le nom interne d'un port stable à partir de son nom symbolique (cf. annexe 3). Si un processus P_1 veut communiquer avec un processus P_2, il doit désigner un port stable dont P_2 est le propriétaire. Les messages déposés dans un port stable sont gérés en liste FIFO. Un message qui arrive est déposé en queue de liste. Quand, le propriétaire d'un port stable demande à lire un message, c'est le message en tête de liste qui lui est délivré.

Le service RMC garantit la propriété qui suit :

La séquence de messages (m_1, m_2, ..., m_i, ..., m_n) reçue par le propriétaire du port stable p est exactement la séquence de messages émise à destination du port stable p.

Cette propriété résume le fait que le service RMC évite la perte, la duplication et le déséquencement des messages. Le dépôt d'un message dans son port stable destinataire p peut être retardé par la panne de l'un des deux sites émetteur ou récepteur ou bien encore par une défaillance du réseau de communication.

Le protocole RMC est mis en œuvre sur chaque site S_i par deux processus serveurs appelés em_ser_i (serveur d'émission) et rec_ser_i (serveur de réception). Ces deux processus partagent des données rangées en mémoire stable qui représentent l'état du système de communication fiable. Il est utile de préciser dès maintenant qu'un numéro de séquence unique est attribué à chaque message lors de son émission. Les données stables du protocole RMC sur le site S_i sont pour chaque site distant S_j la liste des messages à envoyer à S_j, le numéro de séquence du dernier message envoyé à S_j par S_i et le numéro de séquence du dernier message reçu par S_i en provenance de S_j.

Le processus em_ser_i est chargé sur le site S_i de l'envoi vers le réseau de communication de tous les messages présents dans les listes de messages à envoyer aux sites distants. Les messages envoyés séjournent en mémoire stable jusqu'à ce qu'ils soient acquittés par le site destinataire. Un message m qui arrive sur un site S_i est traité par le processus rec_ser_i qui décide de sa prise en compte ou pas et qui le cas échéant dépose m dans le port stable destinataire et enfin envoie un acquittement au site émetteur. Le protocole RMC comporte en outre une procédure dite de recouvrement après panne qui est exécutée par le processus

serveur en émission lors de la réinitialisation du système après une panne.

La coopération des serveurs d'émission em_ser_i et em_ser_j et des serveurs de réception rec_ser_i et rec_ser_j assure que tous les messages présents dans la liste des messages à envoyer au site S_j sont déposés au bout d'un temps fini dans le port stable destinataire dans l'ordre des numéros de séquence (pourvu que le port stable destinataire existe).

Cette première partie du protocole suppose la présence en mémoire stable des messages à transmettre. La seconde partie du protocole a pour but d'assurer que tout message émis par un processus stable est déposé de façon atomique dans la liste des messages à envoyer adéquate et que ce message est délivré à son destinataire une fois et une seule.

L'envoi d'un message m par un processus stable ne provoque donc pas directement l'émission de m sur le réseau de communication mais a pour effet de déposer m en mémoire stable dans la liste des messages à envoyer au site destinataire. La solution au problème de l'atomicité du dépôt ou du retrait d'un message en mémoire stable consiste à sauvegarder un point de reprise du processus stable client du service RMC lors de toute opération de communication (envoi ou réception d'un message).

VII.3 Primitives du service RMC

L'interface du service RMC est composée de quatre primitives dont nous décrivons les paramètres et le comportement.

- Création d'un port stable

  create_port([out] port_id, [out] état)

  Cette primitive permet à un processus stable de créer un port stable en mémoire stable. Le processus créateur est le propriétaire du port stable. La primitive rend l'identificateur local du port stable dans le paramètre *port_id*. S'il n'y a plus de port stable disponible en mémoire stable, une erreur est signalée.

- Destruction d'un port stable

  delete_port([in] port_id, [out] état)

  Cette primitive permet à un processus stable de détruire un port stable dont il est le propriétaire. Le paramètre *port_id* contient l'identificateur local du port stable à détruire. Un port stable peut être détruit même s'il contient encore des messages. Une erreur est retournée si le processus qui appelle la primitive n'est pas le propriétaire du port stable ou si le port stable est inexistant.

- Envoi d'un message sur un port stable

rlb_send([in] port_id, [in] mg, [in] taille, [in] délai, [out] état)

Le message *mg* de longueur *taille* est envoyé sur le port stable identifié par *port_id*. Le processus utilisateur attend pendant le temps *délai* le compte rendu de la primitive. Le compte rendu peut prendre l'une des trois valeurs suivantes :

- *ok* : Le message est bien arrivé dans le port stable destinataire.
- *inex* : Le port stable destinataire est inexistant.
- *nok* : Le délai fixé par l'utilisateur est trop court pour permettre au service RMC de savoir si le message est bien arrivé à destination. Une défaillance soit du réseau de communication soit du site destinataire a pu survenir. Le service RMC ne peut garantir un délai pour la remise du message émis à son destinataire. Cependant, le message parviendra à son destinataire dès que possible

- Réception d'un message sur un port stable

rlb_receive ([in] port_id, [out] mg, [out] taille, [in] délai, [out] état)

Cette primitive, appelée par le processus propriétaire du port stable *port_id*, rend le premier message de la file FIFO dans *mg* et sa longueur dans *taille*. Si le processus appelant n'est pas le propriétaire du port stable *port_id* ou si le port stable est inexistant, une erreur est retournée (paramètre *état*). *délai* indique le délai pendant lequel le processus souhaite attendre un message si aucun message n'est présent dans le port stable au moment de l'appel de la primitive. Si le délai est nul, il n'y a pas d'attente. En cas de délai infini, le processus reste bloqué jusqu'à l'arrivée d'un message.

VII.4 Transmission fiable des messages présents en mémoire stable

VII.4.1 Introduction

Nous commençons par introduire les notations utilisées dans la description algorithmique des protocoles RMC et RBC (dans le chapitre suivant) puis nous précisons les structures de données rangées en mémoire stable et le type des messages employés. Nous expliquons ensuite le comportement des serveurs em_ser_i et rec_ser_i.

VII.4.2 Notations

Dans la description algorithmique des protocoles RMC et RBC, nous employons plusieurs primitives classiques de gestion de liste : *ôter_premier_élément* (*liste*, *élément*) pour

ôter le premier élément de la liste *liste* et mettre sa valeur dans *élément*, *ôter_premier* (*liste*) pour ôter le premier élément de la liste passée en paramètre, *insérer_en_tête* (*liste*, *élément*) pour insérer un élément en tête d'une liste, *insérer_en_queue* (*liste*, *élément*) pour insérer un élément en queue de liste, *premier_élément* (*liste*, *élément*) pour lire la valeur du premier élément de la liste *liste* et la mettre dans *élément*.

Les primitives suivantes sont employées pour la manipulation des listes de messages utilisées dans les divers algorithmes présentés dans la suite. Ces primitives supposent qu'il existe une notion d'ordre sur les éléments d'une liste. Les éléments des listes $msg\_env_i[j]$ et des listes associées aux ports stables (utilisées dans le protocole RMC) sont des descripteurs de messages. Ces listes sont ordonnées selon les numéros de séquence des messages (contenus dans les descripteurs). Les listes $desc\_diff_i$ et msg_diff_i (employées par le protocole RBC) sont ordonnées selon le champ *ndiff* des descripteurs qu'elles contiennent.

- *présent*(*liste*, *no_ordre*)
  Cette fonction à résultat booléen rend vrai si l'élément de numéro d'ordre passé en paramètre est présent dans la liste *liste*.

- *chercher* (*liste*, *élément*, *no_ordre*)
  Cette procédure copie dans *élément* la valeur de l'élément de la liste *liste* dont le numéro d'ordre est donné en paramètre.

- *ôter_élément* (*liste*, *no_ordre*)
  L'élément de numéro d'ordre *no_ordre* est ôté de la liste considérée.

- *ôter_élément_inf* (*liste*, *no_ordre*)
  Tous les éléments dont le numéro d'ordre est inférieur ou égal à celui passé en paramètre sont ôtés de la liste *liste*.

- *remplacer* (*liste*, *no_ordre*, *élément*)
  L'élément de numéro d'ordre *no_ordre* de la liste passée en paramètre est remplacé par l'élément *élément*.

- *réémettre_msg_sup* (*liste*, *no_ordre*)
  Tous les messages de la liste *liste* dont le numéro d'ordre est supérieur à *no_ordre* sont réémis à l'aide de la primitive *envoyer_site* dans l'ordre croissant des numéros d'ordre.

- *présent_premier* (*liste*, *type_msg*)
  Cette fonction rend vrai si un message de type *type_msg* est présent en tête de la liste considérée.

- *tout_réémettre* (*liste*)
  Cette primitive réémet dans l'ordre croissant des numéros de séquence à l'aide de la primitive *envoyer_site* tous les messages contenus dans la liste *liste*.

Enfin, nous utilisons la fonction *extraire_site* (*liste_port*) qui rend comme résultat la liste des sites sur lesquels sont situés les ports (identifiés par leur nom interne) de la liste *liste_port*.

Dans les algorithmes qui suivent, le mot clé *stable* dans une déclaration indique que la structure de donnée correspondante est rangée en mémoire stable.

Afin de ne pas alourdir les algorithmes, nous n'utilisons pas directement les primitives offertes par la mémoire stable dans la description algorithmique des protocoles. Nous préférons introduire les instructions *début_maj_atomique* et *fin_maj_atomique* pour regrouper les opérations de mise à jour de la mémoire stable qui doivent être globalement atomiques. L'exécution de l'instruction *début_maj_atomique* permet au processus courant d'accéder à la mémoire stable en exclusion mutuelle. La mémoire stable ne devient de nouveau accessible aux autres processus du système qu'à la fin de l'exécution de l'instruction *fin_maj_atomique*.

L'instruction *début_maj_atomique* provoque la lecture de toutes les données rangées en mémoire stable utilisées entre les deux instructions *début_maj_atomique* et *fin_maj_atomique*. A l'issue de l'instruction *début_maj_atomique* une copie des données lues est disponible en mémoire vive. Les opérations concernant les données stables sont effectuées sur ces copies en mémoire vive. La mise à jour des copies en mémoire stable (y compris la sauvegarde d'un point de reprise le cas échéant) est réalisée en une seule opération atomique (mise en œuvre par la primitive *maj_atomique_groupe_objet* de la mémoire stable) lors de l'instruction *fin_maj_atomique*. Toutes les opérations de mise à jour de la mémoire stable comprises entre les instructions *début_maj_atomique* travaillent sur la copie des données en mémoire vive.

VII.4.3 Structures de données en mémoire stable

De façon à gérer les problèmes de perte, duplication, et déséquencement de messages en présence de pannes, un numéro de séquence est associé à chaque message. Ainsi, les messages sont identifiés de manière unique. Plusieurs informations sont conservées en mémoire stable (voir figure VII.1). Nous en donnons la liste pour un site S_i quelconque :

$\forall$ *j stable num_sqce*$_i[j]$	init 0	: entier ;
$\forall$ *j stable dern_msg_reçu*$_i[j]$	init 0	: entier ;
$\forall$ *j stable msg_env*$_i[j]$	init ()	: liste de messages ;

Figure VII.1 Données stables du protocole RMC

- Une liste *msg_env*$_i[j]$ pour chaque site distant S_j. Cette liste est destinée à contenir les messages émis par les clients du service RMC du site S_i à destination d'un port stable

localisé sur le site S_j. Les messages y séjournent tant qu'ils ne sont pas encore acquittés par le site S_j. Ces files sont initialement vides.

- Le numéro de séquence $num\_sqce_i[j]$ du dernier message envoyé par le site S_i à chacun des sites distants S_j. Quel que soit S_j, $num\_sqce_i[j]$ est nul à l'initialisation du système.
- Le numéro de séquence du dernier message en provenance du site S_j, reçu et acquitté par le site S_i. Sa valeur, désignée par $dern\_msg\_reçu_i[j]$ est nulle lors de l'initialisation du système.

Tout message destiné à un processus p s'exécutant sur le site S_i est déposé à son arrivée sur S_i en mémoire stable dans l'un des ports stables de p.

En outre, chaque site S_i conserve en mémoire vive une information concernant l'état $état_i[j]$ de chaque site distant S_j tel qu'il le perçoit ; $état_i[j]$ contient l'une des deux valeurs : *opérationnel* ou *en panne*. Au lancement du système, tous les sites distants sont supposés opérationnels.

VII.4.4 Messages employés par le protocole RMC

Cinq types de messages sont employés par le protocole :

- (*msg_fiable*, m, sq, S_i, p)
  Ce type de message sert à véhiculer les messages émis par les clients du service RMC ; m contient le message du client, sq est le numéro de séquence de m, S_i est le site émetteur et p le port stable destinataire du message.
- (*acquittement*, sq, S_i, *type*)
  Ce type de message est utilisé par le protocole pour acquitter les messages de type *msg_fiable*. sq est le numéro de séquence du message acquitté par le site S_i. Un acquittement est de type *positif* pour indiquer qu'un message a bien été déposé dans le port stable destinataire. Un acquittement de type *négatif* est envoyé lorsque le port stable destinataire est inexistant.
- (*recouvrement*, sq, S_i)
  Un tel message est envoyé par un site S_i qui reprend son exécution après une panne à tous les autres sites S_j ; sq est le numéro de séquence du dernier message que S_i a reçu en provenance de S_j.
- (*acq_recouvrement*, S_i)
  Ce message sert à acquitter un message de type *recouvrement* ; S_i est l'émetteur de ce message.

- (*hello*, S_i)
  Un message *hello* est envoyé périodiquement par un site S_i aux autres sites du système pour signifier son existence. Ce message permet de traiter la panne du réseau de communication.

VII.4.5 Le serveur responsable des émissions

Le serveur em_ser_i chargé sur le site S_i de l'émission des messages fiables scrute continuellement les listes $msg\_env_i[j]$. Si la liste examinée est vide, il considère la liste suivante. Les actions qu'il exécute lorsqu'il découvre une liste $msg\_env_i[j]$ non vide sont :

(1) Si le site S_j est en panne, la liste suivante est examinée.

(2) Si le site S_j est opérationnel, le message situé en tête de la liste considérée est envoyé. A chaque message est associé un compteur du nombre de retransmissions initialisé à zéro lors du dépôt du message dans la liste. Le compteur est incrémenté. S'il dépasse la valeur maximale fixée par le protocole, le site S_i considère le site S_j en panne et met à jour $état_i[j]$ en conséquence.

La mise à jour des structures de données stables modifiées pendant la phase (2) est atomique.

La description algorithmique du serveur em_ser_i est donnée sur la figure VII.2.

VII.4.6 Le serveur chargé de la réception des messages

Les messages reçus par le site S_i en provenance des autres sites sont déposés temporairement dans la liste $msg\_reçu_i$ (en mémoire vive) par les protocoles de niveau inférieur dans la hiérarchie représentée sur la figure VI.6. Le processus rec_ser_i chargé de la réception des messages sur le site S_i scrute en permanence cette liste. Si elle n'est pas vide, il traite le message qui se trouve en tête. Tout d'abord, il met à jour le cas échéant l'état $état_i$ du site émetteur du message. Lorsqu'un site S_i reçoit un message en provenance d'un site S_j, il considère qu'*a priori* S_j n'est pas en panne. Ensuite, le traitement du message est effectué selon son type.

Lors de la réception d'un message de type *msg_fiable* en provenance du site S_j, le numéro de séquence *sq* du message reçu est comparé au numéro de séquence du dernier message précédemment reçu de S_j : $dern\_msg\_reçu_i[j]$. S'il s'agit d'un message qui a déjà été traité ($sq \leq dern\_msg\_reçu_i[j]$), le processus rec_ser_i envoie un message de type *acquittement* contenant *sq* au site S_j.

```
processus em_ser_i =
var
   k : entier ;
   msg : message ;
début
   k := 1 ;
   faire pour toujours
   début
      début_maj_atomique
         si  (i ≠ k ∧ état_i[k] = opérationnel ∧ msg_env_i[k] ≠ ())
         alors
            co Envoi d'un message au site distant S_k considéré opérationnel par S_i fco
            premier_élément (msg_env_i[k], msg) ;
            msg.nb_retransmission +:= 1 ;
            si  msg.nb_retransmission < nb_max_retransmission
            alors
               envoyer_site (msg, S_k)
            sinon
               co S_i considère S_k en panne car le nombre maximal de retransmissions
               est dépassé fco
               état_i[k] := en_panne ;
               msg.nb_retransmission := 1
            fsi
         fsi
      fin_maj_atomique
      périodiquement
         co Envoi périodique du message hello aux autres sites fco
         envoyer_site ((hello, S_i), S_k) ;
      fin_périodiquement
      co Passage à la liste suivante fco
      k := (k + 1) modulo N ;
   fin
   fait pour toujours
fin
```

Figure VII.2 Description algorithmique du serveur em_ser_i

S'il s'agit d'un nouveau message dont le numéro de séquence *sq* est immédiatement supérieur à celui mémorisé ($sq = dern\_msg\_reçu_i[j] + 1$), il est inséré dans le port stable *p*.

Si le port stable *p* est inexistant, un message d'acquittement négatif est retourné au site S_j. Dans tous les autres cas, le message reçu est ignoré.

Lorsque le message reçu par le site S_i en provenance du site S_j est de type *acquittement*, le processus rec_ser_i cherche le message acquitté dans la liste $msg\_env_i[j]$ des messages envoyés au site S_j et l'ôte de cette liste. Si le processus émetteur du message acquitté est en attente du compte rendu d'erreur, il est débloqué. La valeur *ok* lui est retournée dans le cas d'un acquittement positif et la valeur *inex* dans le cas d'un acquittement négatif.

Lorsque le site S_i reçoit un message de type *recouvrement* en provenance du site S_j, il déclare le site S_j opérationnel et envoie un message de type *acq_recouvrement* au site S_j. Il réémet les messages dont le numéro de séquence est supérieur au numéro de séquence *q* reçu dans le message de recouvrement. Les messages dont le numéro de séquence est inférieur ou égal à *q* sont des messages qui ont été acquittés par le site S_j et sont donc ôtés de $msg\_env_i[j]$.

Lorsque le site S_i reçoit un message de type *acq_recouvrement* en provenance du site S_j, il ôte le message de type *recouvrement* situé en tête de $msg\_env_i[j]$.

Lorsque le site S_i reçoit un message de type *hello* en provenance du site S_j, il met à jour $état_i[j]$ avec la valeur *opérationnel* dans le cas où S_i considérait le site S_j en panne. Il réémet alors tous les messages non encore acquittés qui sont contenus dans $msg\_env_i[j]$.

Dans chacun des cas, toutes les données stables modifiées sont mises à jour en une seule opération atomique. L'algorithme du serveur rec_ser_i est donné sur la figure VII.3.

```
processus rec_ser_i =
début
   faire pour toujours
   début
      attendre (msg_reçu_i ≠ ()) ;
      co Réception d'un message en provenance du réseau fco
      ôter_premier_élément(msg_reçu_i, msg);
      co Mise à jour de l'état supposé du site distant fco
      si  état_i[msg.émetteur] = en_panne
      alors
         état_i[msg.émetteur] := opérationnel
      fsi
```

```
cas msg.type :
  msg_fiable :
  début
    début_maj_atomique
      si  msg.num_sqce = dern_msg_reçu_i[msg.émetteur] + 1
      alors
        co Arrivée du message attendu qui est déposé dans le port stable
        destinataire.
        Nous avons omis la description du cas où le port destinataire est
        inexistant fco
        dern_msg_reçu_i[msg.émetteur] +:= 1;
        insérer_en_queue (msg.port_dest,msg.données);
      fsi
    fin_maj_atomique
    si  msg.num_sqce ≤ dern_msg_reçu_i[msg.émetteur]
    alors
      co Envoi d'un acquittement à l'émetteur du message reçu fco
      envoyer_site((acquittement,msg.num_sqce, S_i, positif), msg.émetteur) ;
    fsi
    fin
 acquittement :
 début
    début_maj_atomique
    co Le message acquitté est ôté de la liste des messages à envoyer
    correspondante fco
      si  présent (msg_env_i[msg.émetteur], msg.num_sqce)
      alors
        ôter_élément (msg_env_i[msg.émetteur],msg.num_sqce)
      fsi
    fin_maj_atomique
 fin
 recouvrement :
 début
    co Les messages acquittés par S_j sont otés de la liste des messages
    à envoyer à S_j.
    Les autres messages destinés à S_j sont réémis.fco
    début_maj_atomique
      ôter_élément_inf (msg_env_i[msg.émetteur],msg.num_sqce) ;
      reémettre_msg_sup(msg_env_i[msg.émetteur],msg.num_sqce)
    fin_maj_atomique
```

```
        envoyer_site ((acq_recouvrement, S_i),msg.émetteur);
      fin
      acq_recouvrement :
      début
        co Le message de recouvrement est oté de la liste des messages à envoyer
        correspondante fco
        début_maj_atomique
          si présent_premier(msg_env_i[msg.émetteur],recouvrement)
          alors
            ôter_premier (msg_env_i[msg.émetteur])
          fsi
        fin_maj_atomique
      fin
      hello :
      début
        co Réémission de tous les messages destinés au site émetteur du message
        hello fco
        début_maj_atomique
          tout_reémettre ((msg_env_i[msg.émetteur]) ;
        fin_maj_atomique
      fin
    fcas
  fait pour toujours
fin
```

Figure VII.3 Description algorithmique du serveur rec_ser_i

VII.4.7 Le recouvrement après panne

Les deux processus em_ser_i et rec_ser_i sont relancés chaque fois que le système est réinitialisé. Aucun point de reprise n'est sauvegardé pour ces deux processus du système car les traitements qu'ils effectuent sont atomiques et laissent les données stables dans un état cohérent.

La procédure de recouvrement après panne, dont l'algorithme est donné sur la figure VII.4, est exécutée par le processus em_ser_i au moment de la réinitialisation du système de communication après une panne du site S_i. Elle a pour effet de déposer en tête de chaque liste $msg\_env_i[j]$ un message de type *recouvrement*. Le message déposé dans $msg\_env_i[j]$ contient le numéro de séquence du dernier message reçu en provenance du site S_j par le site S_i

(cette information est conservée dans la variable stable $dern\_msg\_reçu_i[j]$). Un message de recouvrement après panne est donc envoyé à tous les sites distants S_j. Ce message n'est ôté de la liste $msg\_env_i[j]$ qu'à la réception du message d'acquittement de type *acq_recouvrement.*

```
procédure recouvrement
var
  j : entier ;
début
  faire pour j := 1 à N
  début
    si j ≠ i
    alors
      début_maj_atomique
        insérer_en_tête (msg_env_i[j],(recouvrement, dern_msg_reçu_i[j], S_i) ) ;
      fin_maj_atomique
      envoyer_site((recouvrement, dern_msg_reçu_i[j], S_i), S_j);
    fsi
  fin
fin
```

Figure VII.4 Description algorithmique de la procédure de recouvrement

VII.5 Envoi et réception de messages

VII.5.1 Envoi d'un message

L'envoi d'un message m vers le port stable p localisé sur le site S_j par le processus P_i s'exécutant sur le site S_i est réalisé par l'appel :

rlb_send (p, m, *longueur*(m), *délai*, *cr*).

Le déroulement de l'exécution de la primitive *rlb_send* comprend quatre étapes :

(1) Incrémentation de $num\_sqce_i[j]$

(2) Constitution d'un message de type *msg_fiable.*
Le message (*msg_fiable*, m, $num\_sqce_i[j]$, i, p) est déposé en queue de la liste $msg\_env_i[j]$.

(3) Sauvegarde d'un point de reprise de P_i.

(4) Si le délai est non nul, P_i se bloque en attente du compte rendu d'erreur.

La séquence des phases (1), (2) et (3) est exécutée atomiquement.

En effet, supposons que la phase (3) ne soit pas exécutée atomiquement avec les phases (1) et (2). L'exécution des phases (1) et (2) a pour effet d'attribuer un numéro de séquence au message m envoyé par P_i. Si une panne survient entre les phases (2) et (3), P_i reprend son exécution à partir d'un point de reprise antérieur à l'exécution de la primitive *rlb_send*. L'appel à *rlb_send* est donc réexécuté. Un nouveau numéro de séquence est attribué au même message m et ce dernier est déposé une seconde fois dans $msg\_env_i[j]$. Deux numéros de séquence distincts ont été attribués à m. Pour le serveur em_ser_i, il s'agit de deux messages distincts. Par conséquent le message m sera émis deux fois. Pour éviter une telle situation, il est indispensable d'exécuter atomiquement les phases (1), (2) et (3).

La description algorithmique du cœur de la primitive *rlb_send* est présentée sur la figure VII.5. Volontairement, nous avons omis la description de l'étape (4) qui n'est pas utile à la compréhension du protocole.

début_maj_atomique
  $num\_sqce_i[j]$ +:= 1 ;
  msg := (msg_fiable, m, $num\_sqce_i[j]$,S_i, p) ;
  insérer_en_queue ($msg\_env_i[j]$, msg) ;
  sauver_point_de_reprise (*ps_courant*) ;
fin_maj_atomique

Figure VII.5 Algorithme d'envoi de messages

VII.5.2 Réception d'un message

La réception d'un message par le processus P_i sur le port stable p dont il est propriétaire est réalisé par l'appel :

rlb_receive (p, m, *taille*, *délai*, *cr*).

Le déroulement de l'exécution de la primitive *rlb_receive* comporte les étapes suivantes :

(1) Si le port stable p contient au moins un message, le message de tête est ôté de la liste des messages de p. Les paramètres résultats sont mis à jour. Dans le cas contraire,

le processus P_i se bloque en attente d'un message pendant le temps indiqué dans le paramètre *délai* si ce dernier est non nul.

(2) Sauvegarde d'un point de reprise de P_i.

La mise à jour des variables stables modifiées dans l'étape (1) est effectuée atomiquement avec l'étape (2).

VII.6 Fonctionnement du protocole dans les cas de panne

Les deux types de panne qui peuvent se produire sont la panne d'un site ou la panne du réseau de communication. Le protocole présenté dans les paragraphes précédents traite, grâce à la gestion des numéros de séquence, la perte, la duplication et le déséquencement des messages. La panne d'un site S_j est détectée par les autres sites S_i qui tentent de lui envoyer un message par le fait que le message émis n'est pas acquitté. Les messages non acquittés étant conservés en mémoire stable, ils peuvent toujours être réémis à destination du site S_j lorsque ce dernier signale qu'il est de nouveau opérationnel par l'envoi d'un message de reprise après panne. Si le site S_j émetteur d'un message m tombe en panne avant d'avoir reçu l'acquittement de m, il réémettra le message m. Ce dernier sera acquitté par le serveur rec_ser_i mais ne sera pas transmis aux couches supérieures du système de communication s'il avait déjà été pris en compte.

La panne du réseau de communication reliant deux sites S_i et S_j ne peut pas être différenciée de la panne du site S_j du point de vue du site S_i. Dans les deux cas, toute communication est impossible entre S_i et S_j et la variable $état_i[j]$ contient la valeur *en panne*. Tant que cette valeur ne change pas, le site S_i ne tente pas de transmettre au site S_j les messages contenus dans $msg\_env_i[j]$.

Dans le cas où l'impossibilité de communication est due à la panne du site S_j, la variable $état_i[j]$ est mise à jour avec la valeur *opérationnel* à la réception par le site S_i du message de recouvrement émis par le site S_j. Le message de type *recouvrement* indique au site S_i la fin de la panne du site S_j.

En revanche, un mécanisme doit être mis en place pour permettre au site S_i de détecter la fin de la panne du réseau de communication et par conséquent pour éviter le blocage du système. Cet objectif est atteint grâce à l'envoi périodique par tout site S_j d'un message de type *hello* vers tous les autres sites S_i du système. A la réception d'un tel message, si sur le site S_i, la variable $état_i[j]$ contient la valeur *en panne*, S_i la met à jour avec la valeur *opérationnel* et réémet les messages non acquittés à destination du site S_j.

VII.7 Optimisation de l'envoi d'un message vers N destinataires

Il est fréquent qu'un message fiable m doive être diffusé à plusieurs destinataires. Une mise en œuvre immédiate de la diffusion est de faire appel N fois à la primitive *rlb_send* du service RMC. Cette méthode simple présente l'inconvénient d'être peu performante. En effet, un point de reprise est sauvegardé lors de chaque appel à *rlb_send*. De façon à éviter une telle dégradation des performances, nous introduisons dans le protocole RMC une nouvelle primitive permettant de réaliser une diffusion atomique, appelée *multi_send* dont l'interface est :

multi_send([in] liste_dest, [in] mg, [in] taille, [out] état).

mg contient le message à envoyer, *taille* sa taille. *liste_dest* est la liste des ports destinataires du message. *état* est le compte rendu de l'exécution de la primitive.

Le déroulement de cette primitive sur le site S_i est le suivant :

(1) Pour chaque port p localisé sur le site S_j et contenu dans la liste *liste_dest* :

- incrémentation de $num\_sqce_i[j]$,
- constitution d'un message de type *msg_fiable*.
  Ce message (*msg_fiable*, *mg*, $num\_sqce_i[j]$, i, p) est déposé en queue de $msg\_env_i[j]$.

(2) Sauvegarde d'un point de reprise du processus P_i.

Ces deux phases sont exécutées en une seule opération atomique.

Un appel à la primitive *multi_send* avec N ports destinataires revient à effectuer N appels à la primitive *rlb_send* avec un délai nul mais avec la sauvegarde d'un seul point de reprise au lieu de N.

La primitive *multi_send* est une primitive de diffusion atomique mais elle ne garantit aucun ordre sur les messages diffusés par des sources différentes.

VII.8 Justification de la correction du protocole RMC

VII.8.1 Introduction

Nous donnons dans ce paragraphe quelques éléments de preuve pour le protocole RMC. Cette vérification (relativement informelle) a pour but de montrer que le protocole possède

quelques propriétés essentielles à son bon fonctionnement. Elle n'a pas la prétention d'être exhaustive mais nous pensons qu'elle contribue à augmenter le degré de confiance que nous pouvons accorder au protocole RMC.

VII.8.2 Énoncé des propriétés à vérifier

Nous nous sommes attachés à définir un ensemble de propriétés qu'il est nécessaire (mais certainement pas suffisant) de vérifier pour se convaincre du bon fonctionnement du protocole RMC. Nous les énonçons :

Propriété 1 Tout message m envoyé par le site S_i à l'aide de la primitive *rlb_send* (m est appelé message fiable) arrive à destination au bout d'un temps fini pourvu que les pannes aient une durée finie.

Propriété 2 Aucun message fiable n'est dupliqué.

Propriété 3 Les messages fiables ne sont pas déséquencés.

Propriété 4 Le protocole RMC résiste aux défaillances du réseau.

Propriété 5 Le protocole RMC résiste aux défaillances des sites.

Dans les paragraphes qui suivent, nous tentons de vérifier ces propriétés.

VII.8.3 Etude de la propriété 1

Pour vérifier la propriété 1, nous montrons tout d'abord que l'intégrité des données stables du protocole RMC est préservée en présence de panne pendant l'exécution de la primitive *rlb_send*. Nous montrons ensuite que le message m, émis par le site S_i est déposé dans son port stable destinataire p sur le site S_j.

Le fonctionnement du protocole RMC repose sur la numérotation en séquence des messages. Tout message émis par un site S_i doit posséder un numéro de séquence supérieur de un au numéro de séquence attribué au message émis immédiatement avant lui sur le site S_i. Cette numérotation des messages est correctement effectuée puisque $num\_sqce_i[j]$ et $msg\_env_i[j]$ sont mis à jour atomiquement. En outre, le serveur em_ser_i et le processus P_i doivent avoir une vision consistante de l'envoi de m en cas de panne : soit P_i et em_ser_i considèrent que m est pris en compte, soit ils considèrent que m n'a pas été pris en compte. Cela est assuré par le fait qu'un point de reprise de P_i est sauvegardé en mémoire stable atomiquement avec la mise à jour de $num\_sqce_i[j]$ et $msg\_env_i[j]$. En cas de panne entre les instructions *début_maj_atomique* et *fin_maj_atomique*, les données stables restent dans leur état initial.

Montrons qu'un message m émis par P_i sur S_i est bien déposé dans son port stable destinataire p sur le site S_j.

A l'issue de l'exécution de la primitive *rlb_send*, le message m se trouve dans la liste $msg\_env_i[j]$. Le processus em_ser_i scrute continuellement les listes msg_env_i. La liste $msg\_env_i[j]$ étant non vide (nous supposons que m est en tête de cette liste) , deux cas peuvent se présenter :

(1) Le site S_i considère le site S_j en panne ($état_i[j] = en\ panne$).

(2) Le site S_i considère le site S_j opérationnel ($état_i[j] = opérationnel$).

Nous traitons chacun de ces deux cas séparément.

Le site S_j est considéré en panne par le site S_i

Le site S_j peut être considéré en panne par S_i pour l'une des deux raisons suivantes : S_j est effectivement en panne ou le lien de communication qui relie S_i et S_j est défaillant. Dans le premier cas, la procédure de recouvrement exécutée par S_j après la panne assure que m est déposé dans le port stable p. La vérification en est donnée dans le paragraphe VII.8.7. Dans le second cas, l'émission périodique d'un message de type *hello* permet d'assurer le bon fonctionnement du protocole.

Le site S_j est considéré opérationnel par le site S_i

Le message m est envoyé au site S_j si toutefois le nombre de retransmissions de m n'exède pas *nb_max_retransmission* auquel cas le site S_j est déclaré en panne (voir paragraphe précédent). Nous considérons que le site S_j reçoit le message m. En effet, si m est perdu par le réseau de communication, il est retransmis lors de l'examen suivant de la liste $msg\_env_i[j]$.

Montrons que le message m est accepté par le site S_j.

m est accepté par S_j si l'égalité suivante est vérifiée :

$$sq_m = dern\_msg\_reçu_j[i] + 1$$

Raisonnons par l'absurde.

Supposons que $sq_m \neq dern\_msg\_reçu_j[i] + 1$.

Montrons dans un premier temps que $\neg\ (sq_m \leq dern\_msg\_reçu_j[i])$.

Si $sq_m \leq dern\_msg\_reçu_j[i]$, nous pouvons en déduire que nécessairement m a déjà été accepté par S_j. Le site S_j a donc envoyé un acquittement du message m à S_i. Comme m se

trouve toujours dans $msg\_env_i[j]$, S_i n'a pas reçu l'acquittement. Deux interprétations sont possibles : l'acquittement a bien été envoyé mais a été perdu par le réseau de communication ou l'acquittement n'a pas été envoyé. Comme le message m est réémis par em_ser_i, l'acquittement est réémis également. Compte tenu des propriétés du réseau local, l'acquittement ne peut pas se perdre indéfiniment. Par conséquent, la présence de m dans $msg\_env_i[j]$ implique que l'acquittement n'a pas été envoyé. L'inégalité $sq_m > dern\_msg\_reçu_j[i]$ est donc vérifiée.

Montrons dans un deuxième temps que $\neg\ (sq_m > dern\_msg\_reçu_j[i] + 1)$. Comme m est en tête de $msg\_env_i[j]$, tous les messages de numéro de séquence q tel que $q \leq sq_m$ - 1 ont été acceptés par S_j. Par conséquent, $dern\_msg\_reçu_j[i]$ est au moins égal à sq_m - 1 c'est-à-dire :

$$dern\_msg\_reçu_j[i] \geq sq_m - 1.$$

En regroupant nos résultats, on obtient :

$$sq_m > dern\_msg\_reçu_j[i]$$
$$\text{et } sq_m \leq dern\_msg\_reçu_j[i] + 1.$$

soit

$$sq_m = dern\_msg\_reçu_j[i] + 1.$$

Le message m est accepté par le site S_j et donc inséré dans le port stable p.

Pour achever la vérification de la propriété 1 en l'absence de panne de site, montrons que le message m n'est pas acquitté par le site S_j avant d'être inséré dans le port stable p. Supposons que S_j envoie un acquittement pour m alors que m n'a pas été déposé dans le port stable p. L'inégalité $sq_m \leq dern\_msg\_reçu_j[i]$ est donc vérifiée. Cela implique que le message de numéro de séquence sq_m a déjà été accepté par le site S_j. $dern\_msg\_reçu_j[i]$ est en effet égal au numéro de séquence du dernier message accepté par S_j et n'a pu être incrémenté que si les messages correspondants ont été déposés dans leurs ports stables destinataires respectifs (atomicité du traitement effectué par rec_ser_j dans le cas où un message de type *acquittement* est reçu par S_j).

VII.8.4 Etude de la propriété 2

Nous vérifions dans ce paragraphe que les messages fiables ne sont pas dupliqués. Nous montrons qu'un message m de numéro de séquence sq_m n'est pas déposé plus d'une fois dans son port stable destinataire p c'est-à-dire que m n'est pas accepté une deuxième fois par rec_ser_j. Lorsque m est accepté la première fois on a l'égalité :

$$sq_m = dern\_msg\_reçu_j[i] + 1.$$

Après l'acceptation de m, $dern\_msg\_reçu_j[i]$ vaut sq_m puisque $dern\_msg\_reçu_j[i]$ est incrémenté atomiquement avec le dépôt de m dans p. Lorsque S_j reçoit de nouveau m, l'égalité $sq_m = dern\_msg\_reçu_j[i]+ 1$ n'est donc plus vérifiée.

Nous pouvons en déduire que les messages fiables ne sont pas dupliqués.

VII.8.5 Etude de la propriété 3

Nous vérifions que les messages fiables ne sont pas déséquencés. Soient m et n deux messages de numéros de séquence respectifs sq_m et sq_n émis par le site S_i à destination du site S_j tels que $sq_m < sq_n$.

Montrons que m est accepté avant n par le serveur rec_ser_j. Il nous faut seulement considérer le cas où n est reçu avant m par S_j. Pour que n soit accepté, il faut que l'égalité suivante soit vérifiée :

$$dern\_msg\_reçu_j[i] = sq_n - 1.$$

Or, on a $sq_m < sq_n$ c'est-à-dire $sq_m \leq sq_n - 1$. Par conséquent, n ne peut pas être accepté avant m. Si n est reçu avant m, il est simplement ignoré par rec_ser_j.

VII.8.6 Résistance du protocole RMC au partitionnement du réseau de communication

Nous montrons dans ce paragraphe que le protocole RMC tolère le partitionnement du réseau de communication.

Soient S_i et S_j deux sites tels que le lien de communication l_{ij} qui les relie est défaillant. La défaillance de l_{ij} n'est pas détectée par le protocole RMC. Par contre, s'il existe un message m dans $msg\_env_i[j]$ qui n'est pas acquitté au bout de $nb\_max\_retransmission$ retransmissions à cause de la défaillance de l_{ij}, S_i détecte par défaut que le site S_j est en panne et met à jour $état_i[j]$. Le site S_j étant considéré en panne par S_i, ce dernier n'émet plus aucun message à destination de S_j.

Chaque site envoie périodiquement un message de type *hello* aux autres sites. Lorsque le partitionnement prend fin, les messages *hello* que S_j émet à destination de S_i parviennent à S_i qui considère alors S_j de nouveau opérationnel. Les messages destinés à S_j sont réémis. Nous avons donc bien vérifié que la détection par S_i de la panne de S_j, alors que ce dernier est opérationnel, n'entraîne pas le blocage du protocole RMC.

Le partitionnement du réseau de communication peut également empêcher un acquittement de parvenir à son destinataire. Le bon fonctionnement du protocole RMC est cependant

assuré puisque tout message non acquitté est conservé par son site émetteur et est réémis à la réception d'un message *hello*.

VII.8.7 Résistance du protocole RMC aux pannes de sites

Nous avons étudié dans le paragraphe VII.8.3 l'effet d'une panne de site sur l'exécution de la primitive *rlb_send*. Nous nous intéressons maintenant au comportement des serveurs *em_ser* et *rec_ser* en présence de pannes de sites.

Nous vérifions dans un premier temps que la panne d'un site n'entraîne pas le blocage du protocole RMC. Dans un deuxième temps, nous montrons qu'un site récupère lors de son recouvrement après une panne tous les messages qui lui ont été envoyés par les autres sites pendant sa panne.

La panne du site S_j destinataire du message m émis par le site S_i n'entraîne pas le blocage du site S_i. Le comportement du serveur rec_ser_i n'est pas altéré par la panne de S_j puisque rec_ser_i n'attend pas les messages en provenance d'un site particulier mais en provenance d'un site quelconque. Le serveur em_ser_i cesse simplement de scruter la liste $msg\_env_i[j]$ à partir du moment où la panne de S_j est détectée. Le compteur du nombre de retransmissions est réinitialisé afin de permettre la réémission, après le recouvrement de S_j, du message situé en tête de la liste $msg\_env_i[j]$.

Pour terminer, montrons que la procédure de recouvrement après panne permet à un site de recevoir les messages qui lui ont été envoyés pendant sa panne. Lors du recouvrement après panne, S_j envoie à chaque site S_i un message contenant le numéro de séquence du dernier message reçu en provenance de S_i : $dern\_msg\_reçu_j[i]$. A la réception de ce message, S_i met à jour $état_i[j]$ et réémet tous les messages présents dans $msg\_env_i[j]$. Le premier de ces messages possède nécessairement un numéro de séquence q tel que $q \leq dern\_msg\_reçu_j[i] + 1$ compte tenu des vérifications déjà effectuées (un message ne peut pas avoir été ôté de $msg\_env_i[j]$ sans avoir été accepté par S_j). Notons que $q = dern\_msg\_reçu_j[i] + 1$ si S_i a reçu l'acquittement du message de numéro de séquence $dern\_msg\_reçu_j[i]$ et que $q \leq dern\_msg\_reçu_j[i]$ si des messages d'acquittement ont été perdus.

Chapitre VIII

Le protocole de diffusion fiable (RBC)

VIII.1 Introduction

La mise en œuvre du protocole RMPC repose sur l'existence d'une primitive de diffusion atomique ordonnée. Nous avons vu au chapitre V qu'un ordre total doit en effet être établi sur l'ensemble des messages diffusés de type *APPEL_MP*. Dans le cadre du système GOTHIC, nous avons mis en œuvre le protocole RBC qui est un protocole de diffusion atomique ordonné assurant un ordre total sur les messages diffusés à différents groupes de destinataires (*multiple group ordering* dans la classification proposée par Garcia Molina dans [Garc88c]).

La primitive de diffusion atomique ordonnée offerte par le protocole RBC est ainsi définie :

rlb_multicast (message, taille, liste_destinataire, délai, état)

L'utilisateur donne le message à diffuser dans *message* et indique sa taille dans *taille*. La liste des destinataires contient les noms internes des ports stables destinataires.
Le délai peut varier de zéro à l'infini. S'il est nul, la primitive n'est pas bloquante et le destinataire n'est pas informé de l'issue de la primitive. Une diffusion initialisée avec un délai nul ne peut être annulée. Si le délai est infini, le destinataire est bloqué jusqu'à ce que le système de communication lui transmette le compte rendu de l'exécution de la primitive. L'utilisateur peut choisir tout délai intermédiaire. A l'expiration du délai ou à la terminaison de la diffusion (avec succès ou échec), un compte rendu lui est fourni :

- *ok* : La diffusion est terminée ; le message est parvenu à tous les sites destinataires.
- *nok* : La diffusion a échoué car l'un des sites destinataires ne peut pas être atteint.

- *en_cours* : La diffusion est en cours. Le système de communication n'a pas encore la confirmation que le message a été remis à tous ses destinataires mais garantit que la diffusion se terminera avec succès. Aucun délai de terminaison ne peut être avancé.

Nous décrivons la structure générale du protocole dans le paragraphe VIII.2. Une présentation détaillée du protocole est donnée dans le paragraphe VIII.3. Le paragraphe VIII.3.5 est consacré à l'étude du fonctionnement du protocole dans les cas de pannes.

VIII.2 Structure générale du protocole RBC

La mise en œuvre du protocole RBC repose sur l'utilisation de la mémoire stable et utilise les primitives offertes par le service RMC.

En ce qui concerne l'ordonnancement des messages selon l'ordre défini par les propriétés (2) et (3) énoncées dans le paragraphe V.2.4, notre protocole s'inspire du protocole ABCAST qui est mis en œuvre dans le système Isis [Birm88] et dont une variante est utilisée dans la mise en œuvre des procédures répliquées [Coop85]. Nous l'avons présenté dans le paragraphe III.5.3. Le protocole ABCAST est en pratique le seul protocole à notre connaissance qui soit applicable dans un système asynchrone et qui respecte la propriété d'ordonnancement total des messages avec groupes de destinataires multiples sous des hypothèses de pannes identiques à celles considérées dans le système GOTHIC. L'ordonnancement des messages est réalisé selon un ordre total défini sur l'ensemble des messages à partir de l'identité des sites et de la valeur d'une horloge locale à chaque site [Lamp78]. A chaque diffusion est associée une estampille et les messages sont délivrés dans l'ordre des estampilles.

Dans le protocole ABCAST, l'attribution d'une estampille à une diffusion se fait en deux phases. Au cours de la première phase, les destinataires d'un message diffusé attribuent une estampille provisoire au message reçu et la proposent au site émetteur de la diffusion. Une fois que ce dernier a reçu une proposition de chacun des sites destinataires, il attribue une estampille finale unique au message qu'il transmet à tous ses destinataires dans la deuxième phase du protocole. Un message diffusé ne peut pas être délivré avant que ne lui soit attribuée son estampille finale.

Comme les messages sont délivrés dans l'ordre de leurs estampilles finales, ils sont délivrés dans le même ordre à tous leurs destinataires.

Dans le protocole RBC, une transaction est associée à chaque diffusion. Le but de la transaction est de déterminer l'estampille finale du message diffusé. Ces transactions sont traitées d'une façon un peu particulière puisque les messages classiques de type *prêt_à_valider* et de validation sont utilisés pour transmettre les estampilles finale et provisoires de la diffusion associée. Les transactions sont gérées sur chaque site S_i par un processus serveur appelé serveur de diffusion ou bien encore ser_diff_i. Ces serveurs de diffusion coopèrent afin

d'assurer la terminaison des transactions créés par la primitive *rlb_multicast* d'initialisation des diffusions atomiques ordonnées.

VIII.3 Description détaillée du protocole RBC

VIII.3.1 Introduction

L'organisation de ce paragraphe est la suivante. Les structures de données conservées en mémoire stable et gérées par le serveur de diffusion de chaque site sont décrites dans le paragraphe suivant. Nous présentons ensuite les messages échangés par les différents serveurs de diffusion. Enfin, nous décrivons assez précisément le déroulement du protocole RBC.

VIII.3.2 Structures de données

Nous présentons les différentes structures de données rangées en mémoire stable, qui sont employées par chaque site S_i lorsqu'il est impliqué dans une diffusion soit en tant qu'émetteur soit en tant que récepteur (voir figure VIII.1).

Chaque site S_i conserve en mémoire stable des informations sur les diffusions dont il est l'initiateur. Le descripteur associé à une diffusion comporte les champs suivants : *ndiff*, *message*, *liste_destinataire*, *nb_destinataire*, *estampille*, *état*.

ndiff est l'identificateur unique de la diffusion à laquelle est associé le descripteur. En effet, afin de discerner les différentes diffusions en cours, un identificateur unique est attribué à chaque diffusion initialisée par un processus s'exécutant sur le site S_i. Cet identificateur unique *ndiff* est un couple (S_i, *nbdiff*) où *nbdiff* est produit à l'aide d'un compteur num_diff_i rangé en mémoire stable.

message est le message diffusé, *liste_destinataire* est la liste des ports stables destinataires du message ; *nb_destinataire* est le nombre de destinataires.

A tout message diffusé, le protocole associe une estampille e qui permet de construire un ordre total sur l'ensemble des messages diffusés dans le système. Cette estampille est un couple (S_i, h) ; S_i est un numéro de site et h une valeur générée à partir d'une horloge h_i locale au site S_i. Sur chaque site S_i, l'horloge h_i est un compteur rangé en mémoire stable.

L'ordre total est défini sur les couples (site, heure) de la façon suivante :

$$(S_i, h_i) < (S_j, h_j) <=> [(h_i < h_j) \text{ ou } ((h_i = h_j) \text{ et } (i < j))].$$

Notons d'emblée que *ndiff* et l'estampille e associée à une diffusion ne font pas double emploi. En effet, l'estampille ne peut pas servir à identifier de manière unique une diffusion

type *desc_diff* = **struct**
ndiff : **struct** *site* : entier, *h* : entier **fstruct**
message : msg
liste_destinataire : liste de port
nb_destinataire : entier
compteur : entier
estampille : **struct** *site* : entier, *h* : entier **fstruct**
état : [*en_cours*,*validée*,*annulée*]
fstruct
stable *desc_diff*$_i$: liste de *desc_diff*
type *msg_diff* = **struct**
ndiff : **struct** *site* : entier, *h* : entier **fstruct**
message : msg
liste_destinataire : liste de port
estampille : **struct** *site* : entier, *h* : entier **fstruct**
état : [*validée*,*prête_à_valider*]
fstruct
stable *msg_diff*$_i$: liste de *msg_diff*
stable *num_diff*$_i$: entier
stable h_i : entier

Figure VIII.1 Structures de données stables du protocole RBC

puisqu'initialement aucune estampille n'est attribuée à une diffusion et que celle-ci résulte d'un calcul effectué à partir de valeurs proposées par les sites destinataires.

Le champ *état* indique l'état d'avancement de la transaction (identifiée par *ndiff*) associée à la diffusion. Il est aussi utilisé pour déterminer le compte rendu pour l'appelant de la primitive *rlb_multicast*. Il peut prendre l'une des trois valeurs suivantes : *en_cours*, *validée*, *annulée*.

Les descripteurs de toutes les diffusions initialisées par le site S_i sont rangés en mémoire stable dans une liste appelée *desc_diff*$_i$.

Lorsqu'un message diffusé arrive sur un site destinataire S_i, il ne peut pas être déposé de suite dans le port stable destinataire. En attendant de pouvoir être délivré à son destinataire, le message ainsi que quelques informations de contrôle sont conservés en mémoire stable dans une liste nommée *msg_diff*$_i$. Un élément de la liste *msg_diff*$_i$ comporte les champs suivants : *ndiff*, *message*, *liste_destinataire*, *estampille*, *état*. Le champ *état* représente l'état local de la transaction et peut prendre l'une des deux valeurs : *validée* ou *prête_à_valider*.

Tant que l'état de la transaction n'est pas égal à *validée*, le message concerné ne peut pas être délivré. Une fois que la transaction est validée, le message peut être délivré selon l'ordre des estampilles finales à son destinataire s'il n'est pas précédé par un message dont la transaction associée n'est pas encore validée. Le champ *liste_destinataire* contient la liste des ports stables destinataires locaux au site S_i.

La liste msg_diff_i est ordonnée suivant l'ordre défini sur les estampilles.

VIII.3.3 Les différents types de messages

Cinq types de messages sont utilisés par les serveurs de diffusion dans le protocole RBC. Nous les examinons en donnant leur structure et en indiquant brièvement la façon dont ils sont employés :

- (*init_transaction*, *ndiff*, m, S_i, *dest*)

  Un tel message sert à initialiser sur les sites destinataires de m la transaction associée à la diffusion du message m ; *ndiff* est le numéro unique attribué à la diffusion et S_i est le site initiateur de la diffusion ; *dest* est la liste des destinataires du message m, chaque destinataire étant désigné par un de ses ports stables.

- (*prêt_à_valider*, *ndiff*, e)

  Ce message indique que le site qui l'émet est prêt à valider la transaction identifiée par *ndiff*. e est la date qui est proposée par le site S_i, émetteur de ce message, pour délivrer à son destinataire le message m qui fait l'objet de la diffusion *ndiff*.

- (*validation*, *ndiff*, e)

  Ce message valide la transaction associée à la diffusion *ndiff* et permet d'informer les destinataires du message objet de la diffusion *ndiff* de la date définitive e à laquelle ce message pourra être délivré.

- (*acquittement*, *ndiff*)

  Un tel message est envoyé au site initiateur de la diffusion *ndiff* par les sites destinataires une fois le message diffusé délivré. Ce message est destiné à élaborer le compte rendu de la primitive *rlb_multicast*.

- (*annulation*, *ndiff*)

  Lorsque la transaction *ndiff* doit être annulée, le site initiateur de la diffusion diffuse un message de ce type pour informer tous les sites destinataires de l'annulation.

VIII.3.4 Déroulement du protocole

Faisons deux remarques préliminaires avant de décrire le protocole RBC.

De façon à simplifier l'explication du protocole, nous faisons l'hypothèse que tous les processus destinataires d'un message diffusé s'exécutent sur des sites distincts. Si plusieurs destinataires d'un message diffusé sont situés sur le même site, le traitement décrit pour un destinataire est appliqué à chacun des destinataires du site considéré.

Tous les messages émis par le protocole RBC sont des messages fiables traités par le protocole RMC. Les primitives *rlb_send* et *multi_send* sont utilisées avec un délai nul.

Diffusion d'un message m

Un processus utilisateur P_i, sur le site S_i, lance une diffusion par l' appel :

rlb_multicast (*m*, *longueur*(*m*), *liste_destinataire*, *délai*, *cr*).

Cet appel comporte les phases suivantes :

(1) Attribution d'un numéro unique *ndiff* à la diffusion ; *ndiff* est un couple constitué de S_i et de la valeur de num_diff_i incrémentée de 1.

(2) Initialisation des informations concernant la diffusion et insertion du descripteur correspondant dans la liste $desc\_diff_i$. En particulier, le champ *état* du descripteur représentant l'état de la transaction associée à la diffusion est initialisé à *en_cours*.

(3) Envoi du message (*init_transaction*, *ndiff*, *m*, S_i, *liste_destinataire*) à tous les sites destinataires (c'est-à-dire les sites sur lesquels est localisé au moins un des ports stables contenus dans *liste_destinataire*).

(4) Attente de la terminaison de la diffusion durant le délai passé en paramètre.

Les phases (1), (2) et (3) sont exécutées en une seule opération atomique. Notons qu'un point de reprise de P_i est sauvegardé à l'étape 3 dans la primitive de communication du service RMC. La procédure *rlb_multicast* est présentée sous forme algorithmique sur la figure VIII.2.

```
procédure rlb_multicast (m, taille(m), liste_port_dest, délai)
début
   liste_site_dest := extraire_site(liste_port_dest) ;
   début_maj_atomique
   co Attribution d'un idenficateur unique à la diffusion fco
      num_diff_i +:= 1 ;
      ident_diffusion := (S_i, num_diff_i) ;
      coInitialisation d'un descripteur de diffusion fco
      desc_diff.ndiff := ident_diffusion ;
      desc_diff.message := m ;
      desc_diff.liste_destinataire := liste_port_dest ;
      desc_diff.nb_destinataire := taille(liste_port_dest) ;
      desc_diff.compteur := 0 ;
      desc_diff.estampille := (0,0) ;
      desc_diff.état := en_cours ;
      insérer_en_queue (desc_diff_i, desc_diff) ;
      multi_send ((init_transaction, desc_diff.ndiff, m, liste_site_dest), liste_site_dest)
   fin_maj_atomique
   attendre (fin_diffusion) pendant délai
   co Annulation de la diffusion si le délai est écoulé fco
   si non_arrivé (fin_diffusion)
   alors
     début_maj_atomique
        chercher(desc_diff_i,desc_diff,ident_diffusion) ;
        si desc_diff.état = en_cours
        alors
           desc_diff.état := annulée ;
           multi_send ((annulation, ident_diffusion),liste_site_dest)
        fsi
     fin_maj_atomique
  fsi
fin
```

Figure VIII.2 Description algorithmique de la procédure *rlb_multicast*

Attribution d'une date de délivrance au message m

Pour qu'un message diffusé puisse être remis à l'ensemble de ses destinataires, les gestionnaires de diffusion ser_diff_k des sites destinataires doivent au préalable déterminer la

date à laquelle le message pourra être délivré sans enfreindre l'ordre total sur les messages diffusés.

La figure VIII.3 montre le déroulement du protocole sur l'exemple de la diffusion d'un message m depuis le site S_0 vers 2 ports stables destinataires p_1 et p_2 situés sur les sites S_1 et S_2 (*ndiff* vaut (S_0,3) pour cette diffusion). Sur cette figure, seules les valeurs utiles à la compréhension du protocole sont représentées.

Dans le champ *état*, *PV* est synonyme de *prête_à_valider* et *V* est synonyme de *validée*.

Lorsque le processus ser_diff_k reçoit un message de type (*init_transaction*, *ndiff*, m, S_i, *dest*), il incrémente son horloge locale h_k de 1 et affecte provisoirement l'estampille (S_k, h_k) à la diffusion *ndiff*. Le message m, estampillé par (S_k, h_k) est inséré selon l'ordre des estampilles dans la liste msg_diff_k. L'état de la transaction est *prête_à_valider*.

Ainsi, dans l'exemple, h_1 passe de la valeur 3 à la valeur 4. Le message m est inséré à son ordre dans la liste msg_diff_1 avec comme estampille proposée par S_1 : (S_1, 4). De même h_2 passe de la valeur 5 à la valeur 6. Le message m est inséré dans msg_diff_2 avec (S_2, 6) comme estampille.

ser_diff_k insère l'estampille (S_k, h_k) dans le message (*prêt_à_valider*, *ndiff*, k, h_k) envoyé au processus ser_diff_i pour l'informer que la transaction peut être validée (S_i étant le site initiateur de la diffusion *ndiff*).

Ainsi, dans l'exemple, S_1 et S_2 envoient respectivement les messages (*prêt_à_valider*, 3, (S_1, 4)) et (*prêt_à_valider*, 3,(S_2, 6)) à S_0.

Le processus ser_diff_i collecte tous les messages de type *prêt_à_valider* et choisit comme date définitive pour délivrer le message m le maximum des dates proposées soit (S_m, h_m). la transaction correspondante est validée et l'estampille finale est envoyée à chacun des sites destinataires de m dans le message de validation de la transaction (*validation*, *ndiff*, S_m, h_m). L'état de la diffusion passe à *validée*.

Dans l'exemple, la date définitive calculée par S_0 pour le message m est (S_2, 6). Le message (*validation*, 3, (S_2, 6)) est envoyé à S_1 et S_2.

A la réception de ce message, le processus ser_diff_k remplace l'estampille provisoire qu'il avait attribuée au message m par l'estampille finale (S_m, h_m) et reclasse le message dans la liste msg_diff_k compte tenu de sa nouvelle estampille. La transaction est validée. De plus, l'horloge locale prend comme valeur le maximum entre son ancienne valeur et h_m.

Dans l'exemple, S_1 remplace l'estampille (S_1, 4) de m par (S_2, 6). Cette modification entraîne le reclassement de m dans msg_diff_1 puisque (S_2, 6) est supérieure à (S_4, 5). L'état de la transaction passe de *PV* à *V*. h_1 prend la valeur 6. Pour le site S_2, seul l'état de la transaction est modifié puisque l'estampille définitive attribuée à m est celle qui avait été proposée par S_2. h_2 conserve la valeur 6.

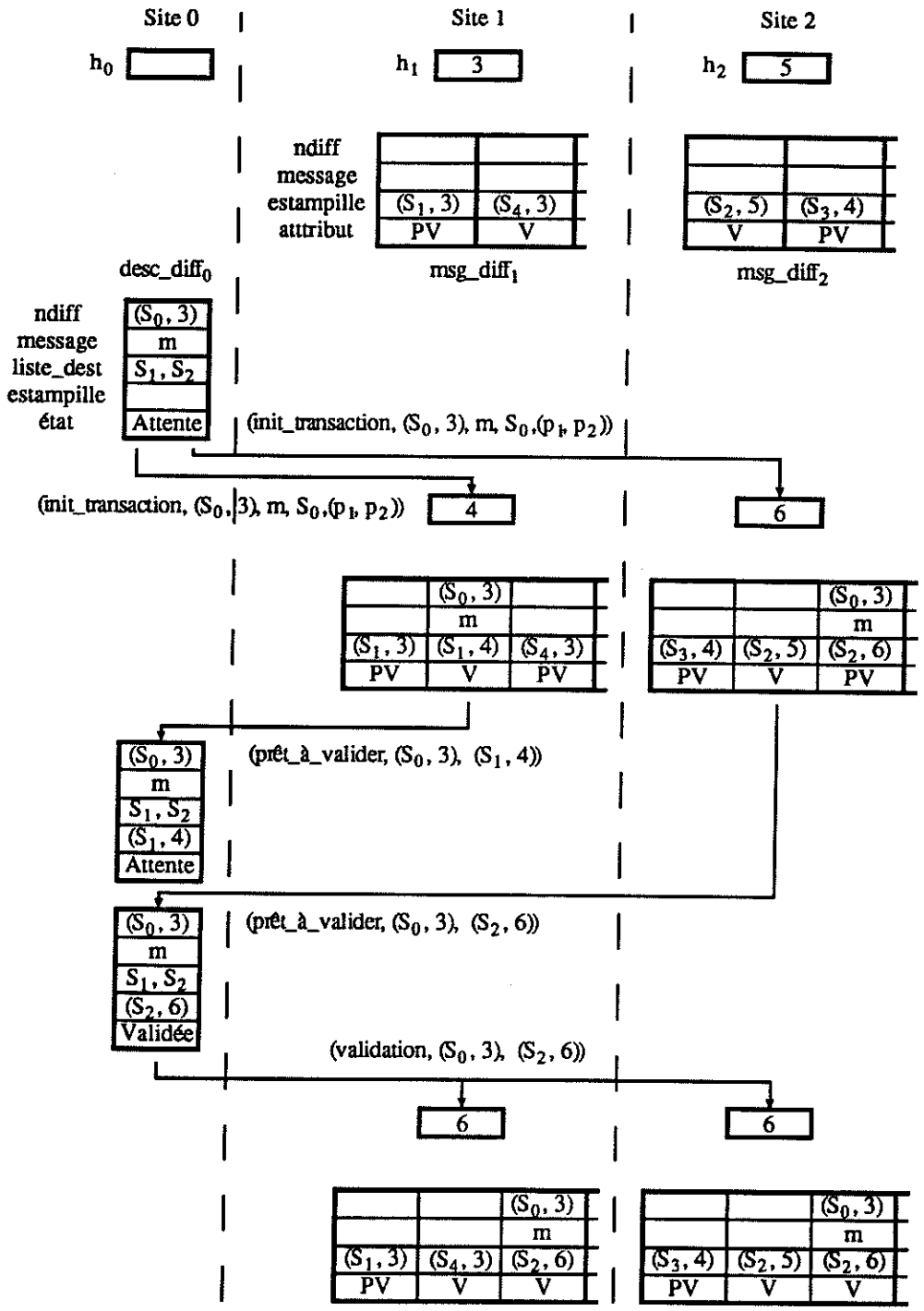


Figure VIII.3 Déroulement du protocole RBC

Le message situé en tête de msg_diff_k est déposé dans le port stable destinataire si l'état de la transaction correspondante est *validée*. Le même traitement est appliqué aux messages suivants jusqu'à ce que la transaction associée au message situé en tête de la liste msg_diff_k soit dans l'état *prête_à_valider*. Lorsqu'un message est ôté de la liste msg_diff_k pour être déposé dans un port stable destinataire, un message d'acquittement (de type *acquittement*) est envoyé au site initiateur de la diffusion. Le seul rôle du message de type *acquittement* est de fournir un compte rendu de terminaison à l'émetteur de la diffusion.

Dans l'exemple, un message dont la transaction associée est dans l'état *prête_à_valider* se trouve en tête des listes msg_diff_1 et msg_diff_2. Aucun message ne peut donc être délivré à son destinataire.

Une diffusion est terminée lorsque le message est délivré à tous ses destinataires. Cette propriété se traduit dans le protocole par la réception de tous les messages de type *acquittement* par le processus *serdiff* du site initiateur de la diffusion. La panne d'un site impliqué dans une diffusion peut retarder la terminaison d'autres diffusion en cours. Un exemple d'une telle situation est représenté sur la figure VIII.4.

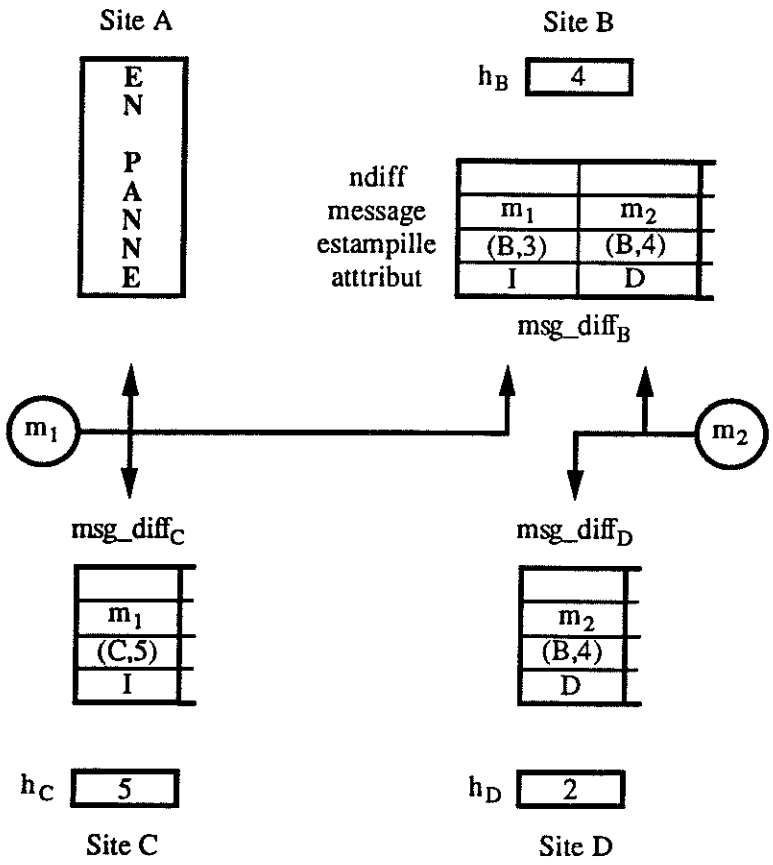


Figure VIII.4 Une situation de blocage du protocole

La délivrance du message m_2 à ses destinataires B et D est retardée par la panne du site A impliqué dans la diffusion du message m_1. Logiquement m_2 devrait être délivré avant m_1 mais, suite à la panne de A, aucune date de délivrance ne peut être attribuée à m_1. Or m_1 précède m_2 dans la liste msg_diff_B, empêchant m_2 d'être délivré. Ce type de blocage est évité grâce au délai associé à chaque transaction au moment de la création. A l'expiration du délai, si l'état de la transaction est *en_cours*, celle-ci est annulée. Le champ *état* prend alors la valeur *annulée*. L'annulation consiste à diffuser à l'ensemble des sites destinataires un message de type *annulation*. A la réception d'un tel message sur les sites destinataires, le message annulé (dont la transaction associée se trouve obligatoirement dans l'état *prête_à_valider*) est retiré de la liste *msg_diff* par ser_diff_k.

Nous terminons ce paragraphe par la description algorithmique du serveur $diff\_ser_i$ (voir figure VIII.5).

```
processus ser_diff_i =
début
  faire pour toujours
  début
  co Réception d'un message en provenance du réseau fco
    rlb_receive (p_diff_i, m, taille, infini)
    cas m.type :
      init_transaction:
        début
          co Insertion du message reçu dans la liste msg_diff_i fco
          début_maj_atomique
            h_i +:= 1 ;
            insérer_dans_ordre (msg_diff_i, (m.ndiff, m.m, m.dest, (S_i,h_i),prête_à_valider)) ;
          fin_maj_atomique
          rlb_send ((prêt_à_valider, m.ndiff, (S_i,h_i), m.émetteur) ;
        fin
      prêt_à_valider :
        début
          co Calcul de l'estampille finale de la diffusion fco
          début_maj_atomique
            chercher (desc_diff_i, desc_diff, m.ndiff) ;
            si desc_diff.estampille < m.e
            alors
              desc_diff.estampille := m.e
            fsi
            desc_diff.compteur +:= 1 ;
            si desc_diff.compteur = desc_diff.nb_destinataire
            alors
              desc_diff.état : = validée ;
              desc_diff.compteur := 0 ;
              liste_site_dest := extraire_site(desc_diff.site_dest) ;
              multi_send ((validation, m.ndiff,desc_diff.estampille), liste_site_dest) ;
            fsi
          fin_maj_atomique
        fin
```

```
        validation :
          début
            co L'estampille finale est associée au message fco
            début_maj_atomique
              h_i := maximum ( h_i, m.e) + 1 ;
              chercher (msg_diff_i, msg_diff, m.ndiff) ;
              msg_diff.attribut := validée ;
              msg_diff.estampille := m.e ;
              remplacer (msg_diff_i, m.ndiff, msg_diff) ;
            co L'acquittement est envoyé lorsque le message est
            effectivement déposé dans le port stable destinataire fco
            rlb_send ((acquittement, m.ndiff), m.ndiff.site) ;
            fin_maj_atomique
          fin
        acquittement :
          début
            début_maj_atomique
              chercher (desc_diff_i, desc_diff, m.ndiff) ;
              desc_diff.compteur +:= 1 ;
              si desc_diff.compteur = desc_diff.nb_destinataire
              alors
                ôter_élément (desc_diff_i, m.ndiff) ;
              fsi
            fin_maj_atomique
          fin
        annulation :
          début
            co Le message correspondant à la diffusion annulée est libéré fco
            début_maj_atomique
              ôter_élément (desc_diff_i, m.ndiff) ;
            fin_maj_atomique
          fin
      fcas
    fin
    fait pour toujours
fin
```

Figure VIII.5 Description algorithmique du serveur de diffusion

Le recouvrement après panne

Après la panne d'un site, le serveur gestionnaire de diffusion reprend son exécution. Aucun traitement spécial de reprise après panne n'est nécessaire. En effet, le serveur reprend son exécution à partir du dernier point de reprise sauvegardé. Il retrouve intactes ses structures de données conservées en mémoire stable. En particulier, les messages qui lui avaient été envoyés avant la panne et qu'il n'avait pas encore traités se trouvent toujours dans son port stable.

VIII.3.5 Fonctionnement du protocole dans les cas de panne

En guise de vérification, nous envisageons dans ce paragraphe plusieurs situations de panne et montrons que le protocole RBC a un fonctionnement correct.

Nous considérons la diffusion du message m initialisée sur le site S_i par le processus P_i. Soit S_j un des sites destinataires de m.

Panne du site S_i pendant l'envoi des messages de type init_transaction

Les messages de type *init_transaction* sont émis à l'aide de la primitive *multi_send*. Lors de l'appel à *multi_send*, un point de reprise de P_i est sauvegardé. Après la panne, P_i reprend son exécution à son dernier point de reprise . L'exécution de la primitive *multi_send* étant atomique, chaque message de type *init_transaction* est pris en compte une fois et une seule.

Panne du site S_i pendant l'attente des messages de type prêt_à_valider

Pendant la panne du site S_i, les messages de type *prêt_à_valider* qui sont envoyés à ser_diff_i ne peuvent lui parvenir. Or ces messages sont envoyés à l'aide de la primitive *rlb_send*. Ils sont donc réémis par le protocole RMC vers le site S_i lorsque ce dernier reprend son exécution. Tous les messages de type *prêt_à_valider* parviendront donc finalement à ser_diff_i.

Cependant la panne du site S_i peut entraîner un retard dans la délivrance à leurs destinataires des messages contenus dans la liste msg_diff_j. En effet, la transaction associée à la diffusion de m ne peut pas être validée tant que ser_diff_i n'a pas traité tous les messages de type *prêt_à_valider*. Par conséquent, tous les messages dont l'estampille est supérieure à celle de m ne peuvent pas être délivrés même si la transaction correspondante est validée.

Panne du site S_i pendant l'envoi des messages de validation

Les mêmes remarques que dans le premier cas s'appliquent ici. Les sites S_j destinataires de m auxquels le message de type *validation* a pu être envoyé avant la panne de S_i peuvent délivrer le message m. Au contraire, ceux auxquels le message de type *validation* n'a pas pu être envoyé avant la panne de S_i ne pourront délivrer m que lorsque le site S_i aura repris son exécution. L'atomicité de la diffusion est tout de même garantie du fait que les messages *validation* sont transmis à l'aide de la primitive de diffusion atomique *multi_send*.

Panne d'un site récepteur S_j pendant l'envoi du message de type init_transaction

Les sites récepteurs qui ne sont pas en panne et qui reçoivent le message de type *init_transaction* insèrent m dans leur liste *msg_diff*. L'état de la transaction passe à *prête_à_valider* et ils envoient un message de type *prêt_à_valider*. Le site S_i attend les messages *prêt_à_valider* pendant un délai T. Si un site S_j est toujours en panne au bout du délai T, S_i n'a pas reçu son message de type *prêt_à_valider*. La diffusion est annulée. Un message de type *annulation* est diffusé à tous les destinataires de m à l'aide de la primitive *multi_send*. L'atomicité est garantie puisque le message m n'a pas pu être délivré à un seul de ses destinataires puisque la transaction est dans l'état *prête_à_valider* sur tous les sites destinataires.

Panne d'un site récepteur pendant l'envoi du message validation

Les récepteurs qui ne sont pas en panne et qui reçoivent le message *validation* délivrent à la date prévue le message diffusé. Lorsqu'il reprend son exécution, le site en panne valide localement la transaction et délivre le message. En effet, le protocole RMC garantit que le message *validation* lui est délivré lors du recouvrement. Le protocole RBC assure donc bien que le message diffusé est délivré à tous ses destinataires dès que l'un d'entre-eux le reçoit.

Panne d'un site récepteur pendant l'annulation d'une diffusion

Le message d'annulation est pris en compte par les récepteurs dès sa réception. Ce message est géré par le protocole RMC ; il est donc délivré lors de la reprise après panne aux sites récepteurs qui étaient en panne au moment de son émission. L'annulation aura donc lieu à la reprise après panne. De toute façon le message diffusé n'a pas pu être délivré sur ces sites puisque la transaction n'a pas été validée.

Chapitre IX

Mise en œuvre des protocoles dans le système Gothic et analyse des performances

IX.1 Introduction

Les protocoles RMC et RBC ont été mis en œuvre dans le système GOTHIC. Les performances de ces protocoles ont ainsi pu être mesurées. Ce chapitre est consacré d'une part à la présentation de quelques aspects de la mise en œuvre du système de communication fiable de GOTHIC et d'autre part à l'analyse des performances des protocoles RMC et RBC.

Il est utile de rappeler dans cette introduction les caractéristiques matérielles du système GOTHIC. Le système GOTHIC est constitué d'un ensemble de machines multiprocesseurs Bull SPS7 reliées par un réseau local de type Ethernet de débit 10 Megabits par seconde. Les machines multiprocesseurs SPS7 possèdent un bus de type SMBUS dont la conception est ancienne et qui de ce fait n'est pas rapide au regard du matériel actuellement disponible. Chaque machine est équipée de deux processeurs de type Motorola 68020 à 16 MHz. Une mémoire stable est connectée sur le bus local de chaque processeur. La capacité utile de la mémoire stable est de 1 Mega-octet. Un mot de la mémoire stable est de taille 32 bits. D'un point de vue purement matériel, le temps d'accès à un mot de la mémoire stable est de 540 ns tandis que comparativement le temps d'accès à un mot de mémoire vive (carte mémoire Bull) est de 350 ns. Les mémoires stables sont alimentées séparément de la machine et bénéficient idéalement d'une alimentation secourue par batterie.

La suite de ce chapitre est organisée de la façon suivante. Le paragraphe IX.2 est consacré aux divers problèmes que pose l'écriture d'une application de taille significative qui utilise des données rangées en mémoire stable ; problèmes rencontrés lors de la mise en œuvre du système de communication fiable de GOTHIC. Nous abordons ensuite quelques

points épineux de la mise en œuvre du protocole RMC puis analysons ses performances. Le paragraphe IX.4 est consacré à l'analyse des performances du protocole RBC. Enfin, nous terminons par une conclusion dans laquelle nous soulignons le travail qui a été effectué pour aboutir au système de communication fiable de GOTHIC et dégageons les perspectives ouvertes par cette mise en œuvre.

IX.2 Programmation des accès à la mémoire stable

IX.2.1 Introduction

Le système de communication fiable de GOTHIC est la première application de grande taille utilisant la mémoire stable. L'expérience montre qu'il s'avère compliqué de programmer une telle application. Cette complexité est due à plusieurs facteurs que nous présentons dans ce paragraphe.

IX.2.2 La lecture et l'écriture d'objets rangés en mémoire stable

Les opérations de lecture et écriture atomiques d'un objet stable O (c'est-à-dire rangé en mémoire stable) consistent en l'accès à l'ensemble des mots de O en mémoire stable (voir paragraphe VI.3.2).

Cette caractéristique des primitives offertes par la mémoire stable a des conséquences sur la définition des structures de données rangées en mémoire stable.

Prenons l'exemple d'une liste de messages à simple chaînage. Chaque élément de la liste est une structure qui contient un champ *successeur*, un champ *numéro* et un champ *message*. Un message est une suite de n octets. La première solution qui vient à l'idée du programmeur est de ranger une telle structure dans un objet stable unique O. Cette solution est mauvaise d'un point de vue performance. En effet, un simple parcours de la liste pour rechercher un message de numéro de séquence donné entraîne la lecture de la totalité de la structure alors que seuls les champs *successeur* et *numéro* sont utiles à l'algorithme.

Pour éviter la lecture inutile du champ message, il est nécessaire de représenter une structure par deux objets stables O_1 et O_2. L'objet O_1 contient les champs *successeur* et *numéro* et l'objet O_2 contient le champ *message*. Lors d'un parcours de la liste, seul l'objet stable O_1 est lu. L'objet O_2 est lu seulement pour la structure dont le numéro de séquence correspond à celui recherché.

Le protocole RMC implique le rangement en mémoire stable de messages auxquels sont associées des informations de contrôle (par exemple le numéro de séquence du message). Ces informations sont gérées en liste (liste des messages d'un port stable, liste des messages à

envoyer à un site distant). Un élément de la liste est composé de deux objets stables : l'un contenant les informations de contrôle associées au message, l'autre contenant le message. Toutes les procédures manipulant une telle liste sont optimisées dans le sens où l'objet stable contenant le message n'est lu ou écrit que lorsque cela est absolument nécessaire.

Remarquons que les objets stables composant un descripteur de message sont créés lors de l'initialisation du système de communication pour des raisons de performances.

La même technique est appliquée dans la mise en œuvre des listes gérées par le protocole RBC [Depa90].

IX.2.3 Gestion de listes d'intentions pour la mise à jour d'un groupe d'objets stables

Pour mettre à jour de façon atomique un groupe d'objets stables, la mémoire stable offre la primitive *maj_atomique_groupe_objet*. Cette primitive fonctionne en deux phases : tous les objets sont écrits sur le banc 1 lors de la première phase puis recopiés sur le banc 2 lors de la seconde. L'automate de mise à jour d'un groupe d'objets est donné en annexe 1.

Avant d'employer la primitive *maj_atomique_groupe_objet*, il est nécessaire de connaître la liste de tous les objets stables devant être mis à jour atomiquement. Or le contenu de cette liste ne peut être calculée qu'à l'exécution, d'une part parce qu'elle peut dépendre de la valeur de certaines variables et d'autre part parce qu'il est nécessaire de connaître les adresses en mémoire vive et en mémoire stable des objets à mettre à jour.

Il faut donc construire une liste d'intentions au fur et à mesure que l'on rencontre les objets stables à mettre à jour. La liste d'intention est constituée de descripteurs qui décrivent la copie en mémoire vive et celle en mémoire stable des objets concernés par l'opération atomique. Chaque descripteur contient donc les adresses de début des copies de l'objet en mémoire vive et en mémoire stable et la taille de l'objet.

La liste d'intentions est initialement vide. Chaque fois que la copie en mémoire vive d'un objet stable est modifiée, un descripteur décrivant cet objet est inséré dans la liste d'intention. Il faut prendre garde à ne pas insérer deux fois le même objet stable dans la liste d'intention car tous les objets concernés par une mise à jour atomique groupée doivent être distincts faute de quoi la mémoire stable détecte une erreur.

IX.2.4 Séquentialité des opérations atomiques

Le prototype actuel de la mémoire stable possède la contrainte de ne permettre le déroulement que d'une seule opération atomique à un instant donné (lecture, écriture ou mise à jour d'un groupe d'objets par exemple).

En effet, la mémoire stable ne comporte qu'un seul registre d'état et ne possède pas de notion de contexte. Or, chaque opération atomique correspond à une séquence de transitions d'états de l'automate qui lui est associé. Le contrôleur de la carte mémoire stable vérifie que toutes transitions effectuées par le processeur (qui envoie les commandes de changement d'état) sont correctes ou autrement dit correspondent bien à l'automate d'une opération atomique.

Il n'est donc pas possible d'entrelacer deux actions atomiques car une erreur serait alors détectée par le contrôleur de la mémoire stable.

Il est par conséquent nécessaire d'assurer qu'une opération atomique est réalisée dans une section critique dans le cas où plusieurs processus utilisant la mémoire stable s'exécutent en parallèle sur un processeur.

La gestion de l'exclusion mutuelle à la mémoire stable est gérée explicitement en faisant appel aux deux primitives *masque_ms* et *demasque_ms* qui permettent respectivement d'obtenir la ressource mémoire stable en exclusion mutuelle et de la relâcher. Ces deux primitives sont mises en œuvre par un sémaphore.

Cette contrainte du prototype de la mémoire stable dont nous disposons entraîne une réduction sensible du parallélisme potentiel des processus qui s'exécutent sur un même processeur.

IX.2.5 Les points de reprise

Nous disposons de la primitive *sauver_point_de_reprise* pour sauvegarder l'état d'un processus en mémoire stable. La sauvegarde d'un point de reprise comporte les étapes suivantes :

- création d'un contexte en mémoire stable pour le processus appelant et sauvegarde du code du processus s'il s'agit du premier point de reprise,
- lecture du contexte stable du processus (9 mots mémoire stable),
- mise à jour, à l'aide de la primitive de mise à jour d'un groupe d'objets en mémoire stable, des données suivantes :
  - segment de données du processus (au minimum 3600 octets),
  - segment pile du processus (1024 octets par défaut) et contexte (36 octets).

Cette mise en œuvre de la primitive *sauver_point_de_reprise* est loin d'être optimale puisque toutes les données modifiables d'un processus sont systématiquement mises à jour lors de chaque point de reprise. Il serait préférable de ne mettre à jour que les données qui

	taille (octets)	lecture temps en microsecondes	écriture temps en microsecondes
code	21596	0	13497.5
pile	1024	0	640
données	10564	0	6602.5
contexte	36	16.2	28.8

Temps total pour la sauvegarde du premier point de reprise : 20785 microsecondes.
Temps total pour la sauvegarde des points de reprise suivants : 7287.5 microsecondes.

Figure IX.1 Evaluation du temps de sauvegarde d'un point de reprise

ont été modifiées depuis la sauvegarde du dernier point de reprise. A titre indicatif, nous donnons dans le tableau IX.1 la taille du point de reprise d'un des processus que nous avons utilisés pour mesurer les performances du protocole RMC. Nous y indiquons également le temps évalué nécessaire à la sauvegarde du point de reprise. L'évaluation est faite sur la base des performances des primitives de lecture et d'écriture d'objets stables données dans l'annexe 2. Il faut noter que ces chiffres représentent des valeurs minimales puisque le processus de référence ne possède que les données strictement nécessaires à la mesure des performances. Le temps de sauvegarde d'un point de reprise dépend évidemment de la taille des données du processus considéré. Pour calculer le temps de sauvegarde d'un point de reprise pour un processus application quelconque, il faut ajouter au temps indiqué dans le tableau IX.1 le temps requis pour la sauvegarde des données spécifiques à l'application (égal à 2.5 microsecondes x taille des données de l'application). Il faut également tenir compte de la taille de la pile nécessaire à l'exécution du processus application qui peut être supérieure à 1024 octets.

Le temps nécessaire pour sauvegarder un point de reprise mesuré est situé entre 8.5 et 9.2 millisecondes. Il est assez proche du temps prévu tout en étant légèrement supérieur. La différence vient du fait que nous ne considérons pas dans notre évaluation le coût de l'algorithmique mais uniquement les accès à la mémoire stable. D'autre part, l'évaluation ne tient pas compte des activités qui se déroulent concurremment dans le système. Dans la mise en œuvre du système de communication fiable de GOTHIC, notre souci permanent a par conséquent été de minimiser le nombre de points de reprise sauvegardés.

A la vue des résultats de cette évaluation, il serait particulièrement intéressant de sauvegarder des micro-points de reprise c'est-à-dire uniquement les données qui ont été modifiées depuis le dernier point de reprise sauvegardé. Les solutions à ce problème ne sont pas triviales et font l'objet de travaux de recherche.

IX.3 Le protocole RMC

IX.3.1 Présentation de quelques points de mise en œuvre délicats

Choix d'un protocole de communication intermachine de base

Les machines du système GOTHIC sont reliées par un réseau local Ethernet. Nous avons donc été amenés à choisir le protocole de communication intermachine. Notre choix s'est porté sur un protocole de type datagramme car nous ne voulions pas payer le coût d'un protocole de transport fiable étant donné que nous étions de toute façon amenés à mettre en place les mécanismes assurant la fiabilité des transferts de messages au niveau du protocole RMC.

Nous avons intégré au noyau GOTHIC le logiciel SC7, fourni par la société BULL, conçu pour assurer la gestion des communications aux niveaux 1 et 2 de la norme ISO (couche physique et couche de liaison) entre machines multiprocesseurs SPS7 reliées entre elles par réseau Ethernet [Hame89a, Hame89b].

Il aurait été possible de mettre en œuvre le protocole RMC en utilisant directement les primitives du logiciel SC7 mais nous avons préféré réaliser le système de communication fiable de GOTHIC au dessus d'un protocole standard. Notre choix s'est porté sur le protocole IP [Post81a] qui offre une communication de type datagramme et dont le portage a été effectué au dessus du logiciel SC7.

Uniformisation des communications intersites

Nous avons vu précédemment que l'architecture matérielle GOTHIC est constituée de machines multiprocesseurs interconnectées par un réseau Ethernet. Il existe donc deux sortes de réseau de communication : le bus global entre les sites d'une même machine et le réseau Ethernet entre les sites localisés dans des machines différentes. Par conséquent, des primitives différentes doivent être utilisées pour la communication intersite suivant la localisation relative des sites concernés. Dans un souci d'uniformité, nous avons développé une couche logicielle assurant la transparence de la localisation des sites pour les communications.

De plus la mise en œuvre actuelle du protocole IP dans GOTHIC distingue deux types de processeurs à l'intérieur d'une machine multiprocesseur : le processeur maître et les processeurs esclaves. Elle comporte une importante restriction : seul un processus localisé sur le processeur maître peut envoyer ou recevoir un message à l'aide du protocole IP.

Le logiciel que nous introduisons au niveau 3 de la hiérarchie du système de communication permet également de masquer cet inconvénient. Les primitives qu'il offre peuvent

être utilisées par tout processus quelle que soit sa localisation.

Exposons le principe de ce protocole d'uniformisation des communications intersites.

Sur le processeur maître de chaque machine s'exécute un processus système, *msg_ser*, qui est chargé de réaliser les envois de messages vers le réseau pour le compte des processus qui s'exécutent sur les processeurs esclaves de la machine. Tous les messages qui arrivent sur une machine en provenance du réseau sont transmis par le protocole IP à *msg_ser* qui est chargé de les délivrer à leur destinataire.

Compte tenu des remarques qui précèdent, nous pouvons mettre en évidence trois configurations possibles. Soient P_1 et P_2 deux processus, P_1 est l'émetteur d'un message à destination de P_2. P_1 et P_2 disposent chacun d'une boîte aux lettres globale (respectivement B_1 et B_2). Une boîte aux lettres globale permet à un processus qui s'exécute sur un processeur d'une machine M de recevoir les messages qui lui sont envoyés par des processus s'exécutant sur les processeurs de la même machine.

Soient M_1 et M_2 deux machines distinctes.

(1) P_1 et P_2 s'exécutent sur deux processeurs distincts de la même machine M_1.

(2) P_1 s'exécute sur le processeur maître de la machine M_1 et P_2 s'exécute sur l'un des processeurs de la machine M_2.

(3) P_1 s'exécute sur l'un des processeurs esclaves de la machine M_1 et P_2 s'exécute sur l'un des processeurs de la machine M_2.

Considérons l'envoi d'un message. La primitive d'envoi de messages offerte par le niveau 3 détermine la configuration puis traite l'envoi selon le cas.

Dans le cas (1), le message est transmis par l'intermédiaire du système de communication par boîtes aux lettres. Il est déposé dans la boîte aux lettres B_2.

Dans le cas (2), le processus émetteur se trouvant sur le processeur maître de la machine, il peut être fait appel à la primitive *envoi_ip* d'envoi de messages du protocole IP.

Enfin, dans le cas (3), le processus P_1 n'a pas accès directement au réseau. Une requête décrivant l'envoi de messages demandé par P_1 est transmise au processus *msg_ser* par l'intermédiaire du service de communication par boîtes aux lettres. Le traitement de cette requête par le processus *msg_ser* consiste simplement à réaliser l'envoi effectif du message vers le réseau à l'aide de la primitive *envoi_ip*.

Lorsque le processus *msg_ser* reçoit un message en provenance du réseau, il en détermine le destinataire (dont l'identité est contenue dans l'entête du message) et dépose le message dans la boîte aux lettres de ce dernier.

Performance du protocole IP dans le système Gothic

Il est intéressant de contrôler les performances du protocole IP puisque le système de communication fiable de GOTHIC est mis en œuvre au dessus de ce protocole. Nous avons effectué deux types de mesure de temps.

Nous avons mesuré tout d'abord le temps mis pour envoyer un message. Le programme

```
processus émetteur =
var     ent i, t_début, t_fin, temps_envoi ;
        message msg ;
        no_site récepteur ;
        début
                get_time (t_début);
                faire pour i depuis 1 jqa GRAND_NOMBRE
                        envoyer_ip(msg,taille(msg), récepteur) ;
                fait
                get_time (t_fin) ;
                temps_envoi := (t_fin - t_début) / GRAND_NOMBRE ;
        fin ;
processus récepteur =
var     ent i, taille ;
        début
                faire pour i depuis 1 jqa GRAND_NOMBRE
                        recevoir_ip(msg,taille)) ;
                fait
        fin ;
```

Figure IX.2 Premier programme de test des performances du protocole IP

de test est constitué de deux processus (cf Fig. IX.2) : le premier émet un message à destination du second. Ce test est répété un grand nombre de fois (1000 fois) de façon à obtenir une valeur moyenne du temps passé dans la primitive d'émission. Le test a été effectué pour différentes tailles de message.

Cette mesure nous permet d'évaluer le temps passé dans la primitive d'émission de message. Le graphique IX.3 montre les résultats obtenus. Nous constatons que ce temps augmente linéairement avec la taille du message émis. En effet, cette mesure comprend le temps de recopie du message émis par le protocole IP puis par le protocole SC7.

Il faut entre 24 et 30 ms pour émettre un message à l'aide de la primitive d'envoi de

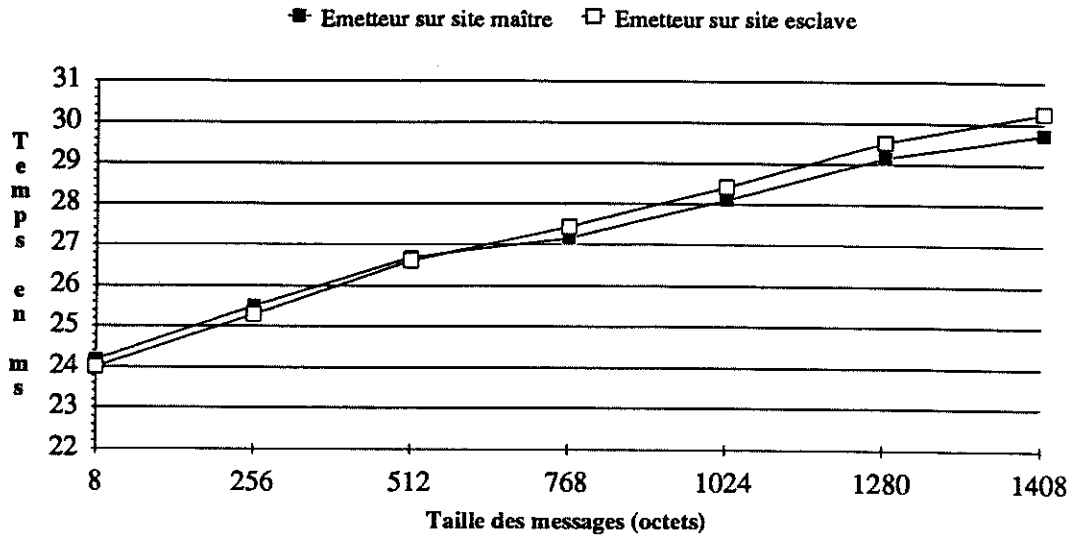


Figure IX.3 Temps passé dans la primitive d'envoi de message du protocole IP

message du protocole IP. Ces valeurs sont anormalement élevées. Elles s'expliquent par la façon dont sont mis en œuvre le logiciel SC7 et le protocole IP. Sur chaque site s'exécute un processus que nous appelons *PSC7* chargé de l'émission et de la réception de messages sur le réseau Ethernet (autrement dit de la gestion de l'unité d'échange Ethernet). L'envoi d'un message sur le réseau Ethernet par un processus se traduit par un dialogue avec le processus *PSC7*. Le protocole IP est mis en œuvre sur une machine SPS7 par deux processus, appelés *S_IP_E* et *S_IP_R* qui s'exécutent sur le processeur maître. Le processus *S_IP_E* est chargé de l'émission des messages des clients du protocole IP. Le processus *S_IP_R* est responsable de la transmission des messages reçus en provenance du réseau aux clients du protocole IP. Un envoi de message IP par un processus application *P* comporte plusieurs étapes :

(1) envoi du message à émettre au processus *S_IP_E*,

(2) traitement de l'envoi par *S_IP_E* comprenant les étapes suivantes :

   (A) allocation et initialisation de ressources,

   (B) envoi par *S_IP_E* d'une requête d'envoi de message au processus *PSC7* et attente de la réponse à cette requête,

   (C) traitement de l'envoi sur le réseau Ethernet par *PSC7* et envoi d'une réponse au processus *S_IP_E*,

   (D) réception de la réponse à la requête d'envoi par *S_IP_E*.

Ce fonctionnement entraîne des changements de contextes qui sont des opérations coûteuses. Par conséquent les valeurs mesurées comprennent un temps de latence très important.

La réception d'un message se déroule de façon similaire avec envoi d'une requête de demande de réception au processus *PSC7* par le serveur *S_IP_R*.

Nous avons également mesuré le temps nécessaire à un aller/retour de message.

Le programme de test est constitué de deux processus dont l'énoncé est donné sur la figure IX.4. Le test a été effectué pour différentes tailles de message (1000 essais à chaque

```
processus émetteur =
var     ent  i, taille, t_début, t_fin, temps_aller_retour ;
        message msg ;
        no_site récepteur ;
début
        get_time (t_début);
        faire pour  i depuis  1 jqa GRAND_NOMBRE
                envoyer_ip(msg,taille(msg), récepteur) ;
                recevoir_ip(msg,taille) ;
        fait
        get_time (t_fin) ;
        temps_aller_retour:= ( t_fin - t_début) / GRAND_NOMBRE ;
fin ;
processus récepteur =
var     ent  i, taille ;
        message msg ;
        no_site émetteur;
début
        faire pour  i depuis  1 jqa GRAND_NOMBRE
                recevoir_ip(msg,taille) ;
                envoyer_ip(msg,taille(msg), émetteur) ;
        fait
fin ;
```

Figure IX.4 Deuxième programme de test des performances du protocole IP

fois) de façon à obtenir une valeur moyenne. Les résultats obtenus sont présentés sur le graphique IX.5. Le temps mesuré correspond à deux fois la somme du temps passé dans les primitives *envoyer_ip* et *recevoir_ip* ajouté au temps de transfert d'un message.

Pour un message de 8 octets, il faut environ 80 ms pour un aller/retour de message. Cette valeur s'explique par le temps de latence très élevé, à la fois pour l'émission et la réception de messages, dû à la mise en œuvre du logiciel SC7 et du protocole IP.

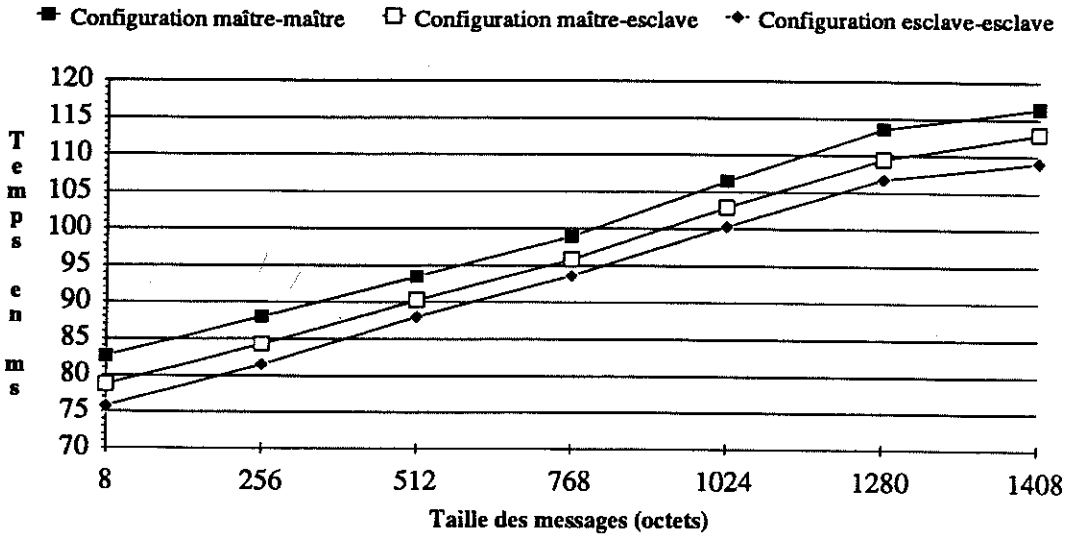


Figure IX.5 Performance du protocole IP

Nous avons effectué les tests présentés ci-dessus pour différentes configurations des processus de test (localisation sur processeur maître ou esclave de l'émetteur et du récepteur). Les courbes IX.3 et IX.5 montrent que la différence en fonction des configurations n'est pas sensible relativement aux temps mesurés.

Partage de données en mémoire stable

Le protocole RMC est mis en œuvre sur chaque site par deux processus *rec_ser* et *em_ser* qui travaillent sur des données communes (liste des messages à envoyer à un site distant par exemple). Ces données sont rangées dans un segment stable qui est projeté dans l'espace d'adressage de chacun des processus. Le segment stable est créé à l'initialisation du système par le serveur *em_ser* s'il ne s'agit pas d'un recouvrement après panne. La figure IX.6 montre l'organisation du segment stable *seg_RMC* associé au protocole RMC. Chaque champ du descripteur *topo* indique le déplacement par rapport au début du segment d'une des zones suivantes. Cette structure de données est utilisée pour faciliter les calculs d'adresses lors des accès aux données stables. Par exemple, l'adresse du premier descripteur de message est l'adresse de début du segment incrémentée de la valeur contenue dans le dernier champ du descripteur *topo*. Le catalogue $catlg\_port_i$ contient des couples (*port_id*, *identificateur*). Son utilisation est décrite dans l'annexe 3. On retrouve ensuite toutes les structures de données introduites dans le chapitre VII.

Nous avons choisi de nous limiter dans le système de communication fiable de GOTHIC à des messages de taille fixe (1024 octets). Nous avons en effet jugé qu'il n'était pas utile de prévoir une gestion de messages de taille variable pour démontrer la validité de notre

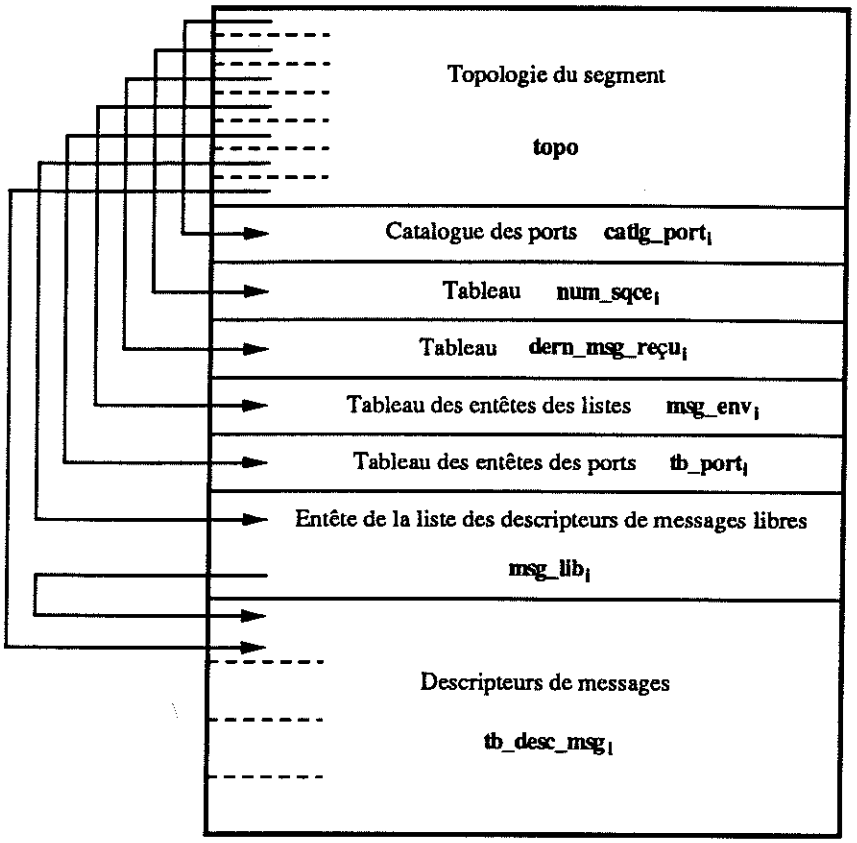


Figure IX.6 Organisation des données stables du service RMC dans le segment *seg_RMC*

approche. Les objets stables destinés à contenir les messages sont créés statiquement lors de l'initialisation du système. *free_msg$_i$* est l'entête de la liste des descripteurs de messages libres. Elle contient le nombre de descripteurs de messages libres et un pointeur sur le premier de ceux-ci. Initialement tous les descripteurs de messages (*msg_desc_tb$_i$*) sont libres et sont chaînés entre eux. Lors d'une demande d'allocation, le descripteur situé en tête de *free_msg$_i$* est alloué. Lorsqu'un descripteur de message est libéré, il est placé en tête de la liste.

Le segment stable *seg_RMC* est mis à jour non seulement par les serveurs *em_ser* et *rec_ser* mais aussi lors de chaque envoi ou réception de messages par un processus utilisateur. Dans ce cas, un point de reprise doit être sauvegardé atomiquement avec la mise à jour des données modifiées du segment *seg_RMC* pour préserver la cohérence du système de communication à la fois vis à vis des serveurs *em_ser* et *rec_ser* et des processus application émetteur et récepteur de messages fiables. Pour ce faire, nous avons défini une nouvelle primitive appelée *sauver_données_et_pt_reprise*. Cette primitive prend en paramètre une liste d'intentions décrivant les objets qui doivent être mis à jour atomiquement avec le point de reprise. Cette primitive est une simple adaptation de la primitive *sauver_point_de_reprise*. La primitive *sauver_point_de_reprise* comporte un appel à la primitive *maj_atomique_groupe_objet* avec comme

paramètre la liste des objets stables constituant le point de reprise.

Dans la primitive *sauver_données_et_pt_reprise* la liste d'intention passée en paramètre et la liste des objets stables constituant le point de reprise sont simplement regroupées en une seule liste qui est passée en paramètre de la primitive *maj_atomique_groupe_objet*. L'atomicité est ainsi assurée.

Le protocole RBC utilise également un segment stable appelé *seg_RBC*. Les clients

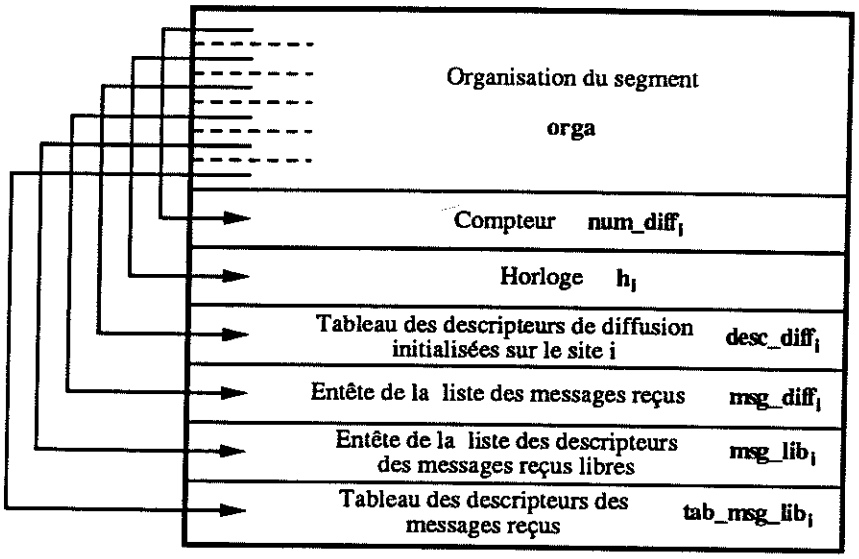


Figure IX.7 Le segment stable *seg_RBC* du service RBC

du service RBC y déposent les messages à diffuser à l'aide de la primitive *rlb_multi_cast*. Il contient en outre les données gérées sur chaque site par le serveur *ser_diff*. La structure du segment est donnée sur la figure IX.7.

Notons que le protocole RBC aurait pu être mis en œuvre sans utiliser de segment stable partagé. Il aurait été en effet possible pour les processus clients du protocole RBC de soumettre leurs requêtes de diffusion au serveur *ser_diff* en utilisant les primitives du protocole RMC. Cette solution n'a pas été retenue car elle est plus coûteuse en nombre de points de reprise sauvegardés.

IX.3.2 Evaluation du coût de la gestion des données stables dans le protocole RMC

Connaissant les performances de la mémoire stable [Mull88] (rappelées dans l'annexe 2), nous avons pu évaluer le coût des accès à la mémoire stable dans les différentes étapes du protocole RMC. Nous avons consigné les résultats dans le tableau IX.8. Dans ce tableau,

Primitive rlb_send

	Lecture	Ecriture	Point de reprise	Total
Valeur minimale (ms)	0,0313	0,6937	7,15	7,875
Valeur maximale (ms)	0,0452	0,7176	7,15	7,9128

Primitive rlb_receive

	Lecture	Ecriture	Point de reprise	Total
Valeur minimale (ms)	0,0387	0,0496	7,29	7,3758
Valeur maximale (ms)	0,0398	0,0496	7,29	7,3769

Envoi d'un message stable par le serveur em_ser

	Lecture	Ecriture	Point de reprise	Total
Valeur minimale (ms)	0,0378	0	0	0,0378
Valeur maximale (ms)	0,0378	0	0	0,0378

Réception d'un message stable par le serveur rec_ser

	Lecture	Ecriture	Point de reprise	Total
Valeur minimale (ms)	0,0342	0,6986	0	0,7328
Valeur maximale (ms)	0,0452	0,7176	0	0,7628

Réception d'un message d'acquittement par le serveur rec_ser

	Lecture	Ecriture	Point de reprise	Total
Valeur minimale (ms)	0,3897	0,0447	0	0,4344
Valeur maximale (ms)	0,3897	0,0637	0	0,4534

Figure IX.8 Evaluation du temps passé dans les accès à la mémoire stable dans le protocole RMC

nous avons considéré les cinq étapes du protocole RMC intervenant dans les échanges de messages : l'envoi d'un message (primitive *rlb_send*), la réception d'un message (primitive *rlb_receive*), l'envoi d'un message fiable sur le réseau par le serveur *em_ser*, la réception d'un message fiable en provenance du réseau par le serveur *rec_ser* et la réception d'un acquittement par le serveur *rec_ser*. Nous avons indiqué le temps évalué pour la lecture de données stables, l'écriture de données stables (sans tenir compte des données constituant un point de reprise de processus) et pour la sauvegarde d'un point de reprise.

Nous pouvons constater que les temps de lecture et d'écriture des données stables du protocole RMC dans les primitives *rlb_send* et *rlb_receive* sont négligeables comparativement au temps nécessaire pour sauvegarder un point de reprise. Il faut noter qu'un seul point de reprise est sauvegardé dans les primitives *rlb_send* et *rlb_receive*.

Les serveurs *em_ser* et *rec_ser* ne réalisent pas de point de reprise et passent toujours moins de 800 microsecondes à lire ou écrire des données stables.

IX.3.3 Analyse des performances du protocole RMC dans le système Gothic

Temps passé dans les différentes étapes du protocole RMC

Nous avons mesuré le temps passé dans les différentes étapes du protocole RMC pendant l'exécution de deux processus s'échangeant des messages. L'énoncé de ces processus est donné dans la figure IX.9. Les tests ont été effectués pour plusieurs configurations : émetteur et récepteur sur deux processeurs maîtres distincts, émetteur et récepteur sur deux processeurs esclaves distincts, émetteur sur processeur maître et récepteur sur processeur esclave de deux machines différentes. Les résultats obtenus sont indépendants de la configuration. Les graphiques présentent pour une configuration donnée des valeurs moyennes obtenues sur un grand nombre de tests.

La figure IX.10 représente le temps total t_1 passé dans la primitive *rlb_send* comparé au temps t_2 passé pour la mise à jour des données modifiées du protocole RMC ajouté au temps de sauvegarde du point de reprise du processus émetteur. t_2 est indépendant de la taille du message émis. En effet, le message émis est rangé dans un objet stable de taille 1024 octets quelle que soit sa taille (voir paragraphe IX.3.1). La quantité d'information écrite en mémoire stable est donc indépendante de la taille du message émis.

Le temps total comprend le temps nécessaire à l'obtention de la ressource mémoire stable en exclusion mutuelle. On peut constater que ce temps est variable. Il dépend de l'activité dans le système.

La sauvegarde du point de reprise atomiquement avec la mise à jour des données stables modifiées du protocole RMC représente l'essentiel du temps passé dans la primitive *rlb_send* (voir figure IX.10 pour la taille d'un message égal à 500).

La figure IX.11 représente l'analyse du temps passé dans la primitive *rlb_receive*. Le temps total comprend le temps nécessaire à l'obtention de la mémoire stable en exclusion mutuelle mais aussi le temps passé à attendre qu'un message arrive dans le port stable. Le temps sans attente est obtenu en mesurant le temps passé dans la primitive *rlb_receive* une fois que la ressource mémoire stable est obtenue en exclusion mutuelle et déduction faite du temps passé à attendre l'arrivée d'un message dans le port stable.

La comparaison de ce temps avec le temps passé pour la sauvegarde d'un point de reprise atomiquement avec l'écriture des données stables modifiées du protocole RMC montre que dans le cas de la primitive *rlb_send* l'essentiel du temps est passé à mettre jour la mémoire stable. La quantité d'informations sauvegardées en mémoire stable par la primitive *rlb_receive* est indépendante de la taille des messages. La figure IX.12 montre le temps passé par le processus *em_ser* pour l'envoi d'un message fiable sur le réseau. Le temps passé pour la lecture du message en mémoire stable est négligeable par rapport au temps passé dans la

```
processus émetteur =
var     ent i, taille, t_début, t_fin, temps_rmc, cpt_rendu ;
        uid_port ident_port, mon_port ;
        message msg ;
        début
        save_checkpoint() ;
        create_port(mon_port, cpt_rendu) ;
        port_register (mon_port,"EMETTEUR", cpt_rendu) ;
        get_port("RECEPTEUR",ident_port, cpt_rendu) ;
                get_time (t_début);
                faire pour i depuis 1 jqa GRAND_NOMBRE
                        rlb_send(ident_port,msg,taille(msg), 0, cpt_rendu) ;
                        rlb_receive(mon_port,msg,taille,INFINI, cpt_rendu) ;
                fait
                get_time (t_fin) ;
                temps_rmc:= (t_fin - t_début) / GRAND_NOMBRE ;
        fin ;
processus récepteur =
var     ent i, taille, cpt_rendu ;
        message msg ;
        uid_port ident_port, mon_port ;
        début
        save_checkpoint() ;
        create_port(mon_port, cpt_rendu) ;
        port_register (mon_port,"RECEPTEUR", cpt_rendu) ;
        get_port("EMETTEUR",ident_port, cpt_rendu) ;
                faire pour i depuis 1 jqa GRAND_NOMBRE
                        rlb_receive(mon_port,msg,taille,INFINI, cpt_rendu) ;
                        rlb_send(ident_port,msg,taille(msg),0, cpt_rendu) ;
                fait
        fin ;
```

Figure IX.9 Programme de test des performances du protocole RMC

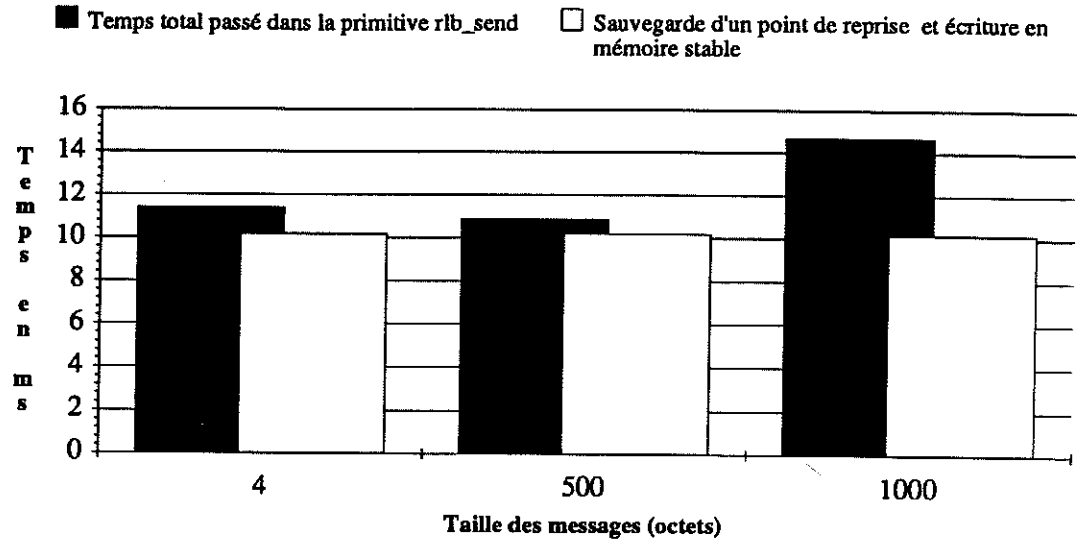


Figure IX.10 Analyse du temps passé dans la primitive *rlb_send*

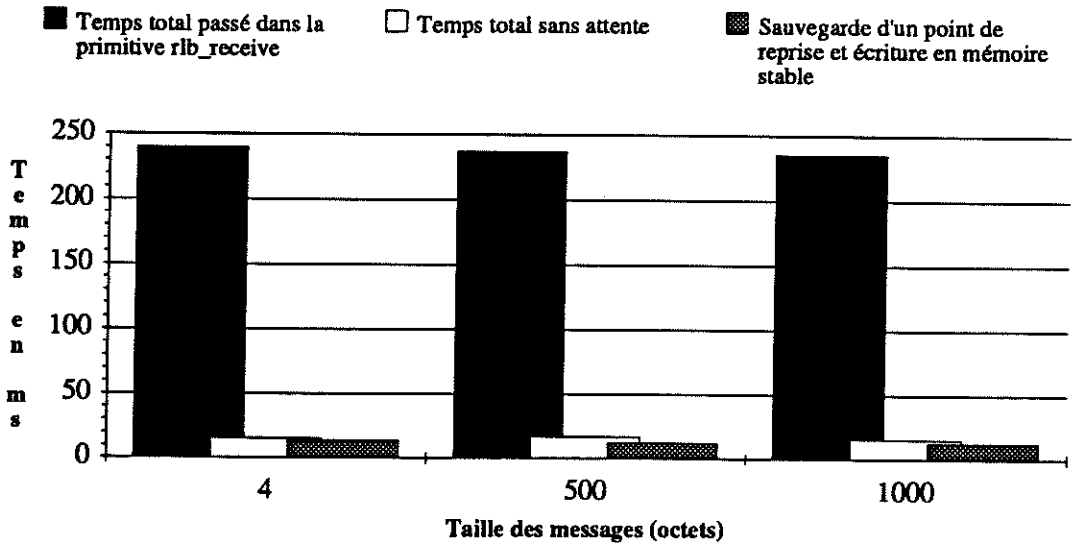


Figure IX.11 Analyse du temps passé dans la primitive *rlb_receive*

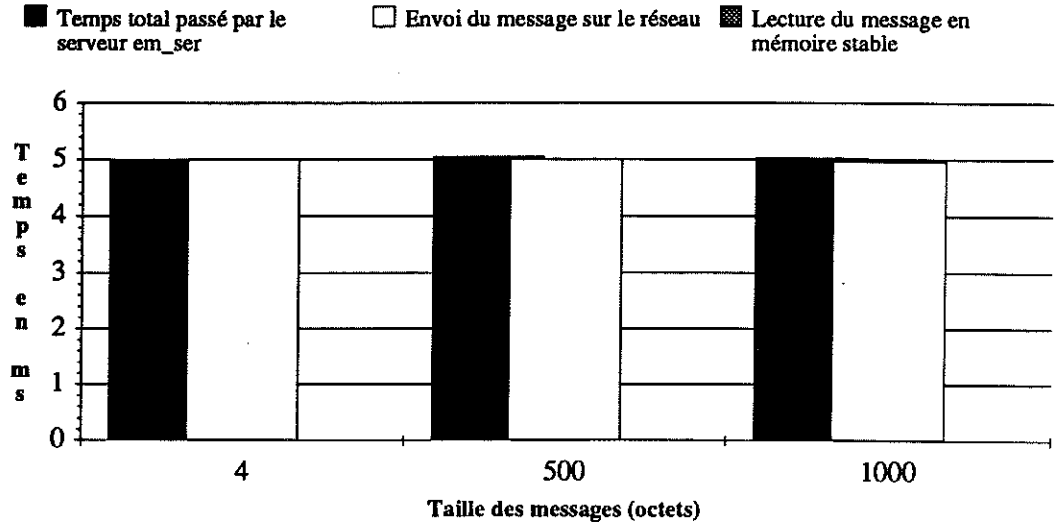


Figure IX.12 Analyse du temps passé par *em_ser* pour l'envoi d'un message fiable

primitive d'envoi de message. L'envoi de message prend seulement environ 5 ms car les temps ont été mesurés sur un site esclave. Dans cette configuration l'envoi de messages se traduit uniquement par l'envoi d'une requête d'émission au processus qui s'exécute sur le processeur maître de la machine (voir paragraphe IX.3.1). La figure IX.13 représente l'analyse du temps

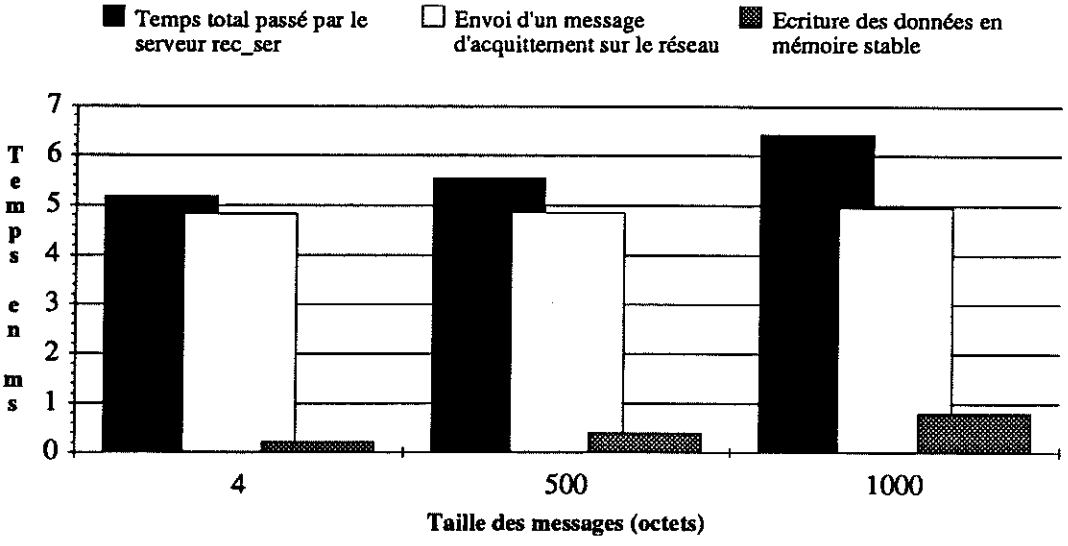


Figure IX.13 Analyse du temps passé par *rec_ser* lors de la réception d'un message fiable

passé par le processus *rec_ser* pour le traitement d'un message fiable reçu en provenance du réseau. Lorsqu'il reçoit un message fiable, le serveur *rec_ser* le recopie en mémoire stable et envoie un acquittement à l'émetteur du message. Nous pouvons constater que le temps mesuré pour l'écriture en mémoire stable est variable. Or quelle que soit la taille du message

reçu la quantité d'informations écrite en mémoire stable est la même. Le processus *rec_ser* est donc interrompu pendant son exécution. Le temps passé strictement à écrire les données modifiées en mémoire stable est donc majoré par la valeur obtenue pour un message de 4 octets.

On constate que le temps passé pour l'envoi du message d'acquittement est d'un peu moins de 5 ms et représente la part essentielle du traitement effectué par *rec_ser*. La figure

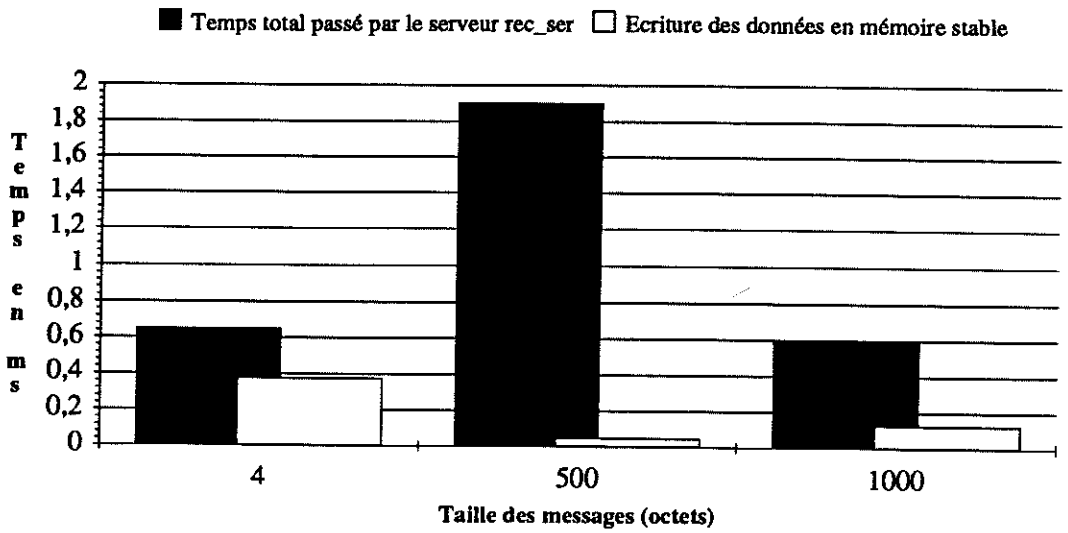


Figure IX.14 Analyse du temps passé par *rec_ser* lors de la réception d'un acquittement

IX.14 représente le temps passé par *rec_ser* lors de la réception d'un message d'acquittement. Le temps total représente le temps nécessaire à l'obtention de la ressource mémoire stable en exclusion mutuelle afin d'ôter le message acquitté de la liste des messages à envoyer. Ce temps est variable et dépend de l'activité dans le système. Le temps passé pour la mise à jour de la mémoire stable est variable bien que la quantité d'informations mises à jour soit indépendante de la taille des messages échangés entre les processus de test.

Il faut donc en conclure que l'exécution du serveur *rec_ser* est interrompu par d'autres travaux plus prioritaires dans le système (par exemple : traitement d'une interruption envoyée par le module d'échange Ethernet).

La quantité d'information à mettre à jour en mémoire stable étant la même quelle que soit la taille des messages, on peut en déduire que le temps de mise à jour de la mémoire stable est majoré par la plus petite valeur obtenue c'est-à-dire celle mesurée pour des messages échangés de longueur 500 IX.14. Les valeurs plus élevées nous indiquent simplement que l'écriture en mémoire stable est parfois interrompue par d'autres activités du système.

La principale conclusion à tirer de cette étude est que les mesures reflètent plus les performances du logiciel SC7 et celles du protocole IP que les performances du système de

communication fiable. Les temps d'accès paraissent évidemment très faible comparés aux temps de latence.

Temps pour un aller/retour de message fiable

Comme pour le protocole IP, nous avons évalué le temps nécessaire à un aller/retour de message fiable. Nous avons effectué la mesure pour différentes tailles de message. La figure IX.15 montre que le temps pour un aller/retour de message fiable est sensiblement constant et se situe autour de 240 ms. Nous avons fait une étude similaire pour le protocole TCP qui

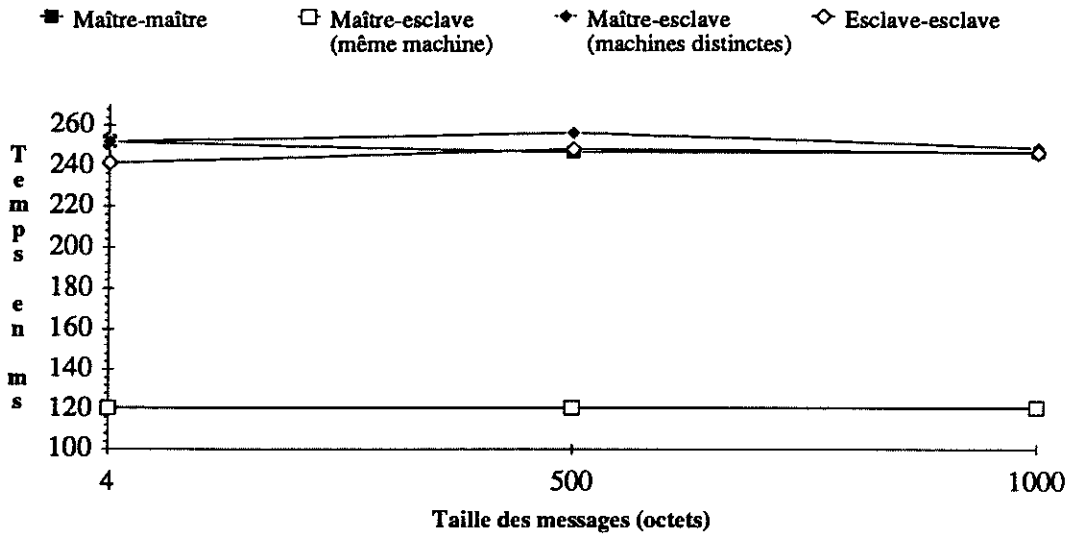


Figure IX.15 Performance du protocole RMC

assure un transfert de message fiable en l'absence de pannes de sites. Pour un message de taille 1024 octets, le temps mis pour un aller/retour de message TCP (environ 400 ms) est nettement plus élevé que la valeur correspondante pour le protocole RMC. Il faut cependant noter que le programme de test n'est pas à l'avantage de TCP qui obtient ses meilleures résultats pour des transfert de données de taille importante.

Performance de la diffusion atomique

Le protocole RMC offre la primitive *multi_send* qui permet de réaliser une diffusion atomique. Un seul point de reprise est sauvegardé quel que soit le nombre de destinataires. Nous avons mesuré le temps passé dans la primitive *multi_send* pour différentes tailles de messages et pour un nombre de destinataires variable. La figure IX.17 montre que le temps passé dans la primitive *rlb_send* est indépendant de la taille du message diffusé. Nous en avons donné la justification dans l'analyse détaillée des performances du protocole RMC. La figure

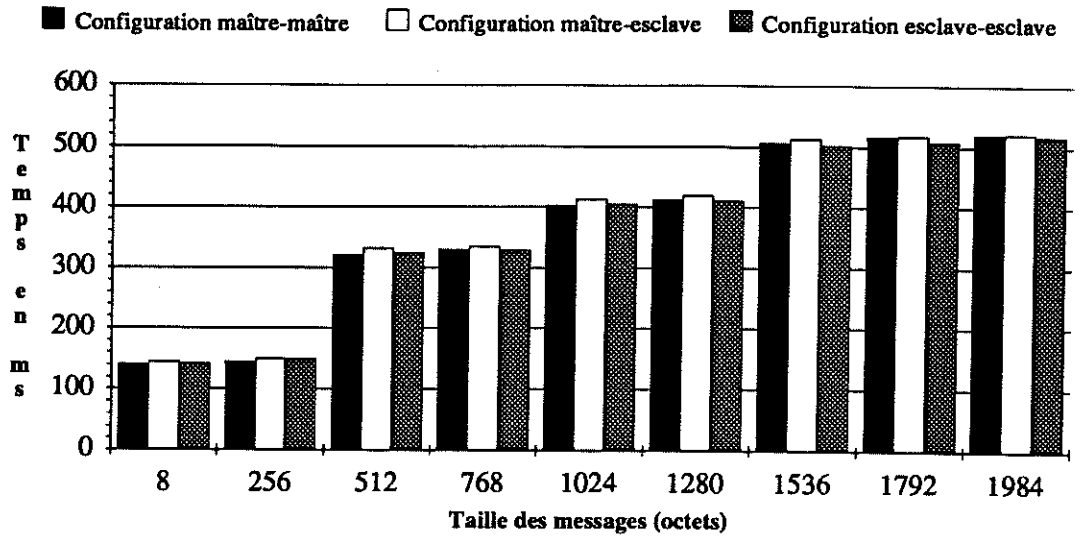


Figure IX.16 Performance du protocole TCP

IX.18 montre que le temps passé dans la primitive *multi_send* augmente quasi linéairement avec le nombre de destinataires. Cela s'explique par le fait que la primitive *multi_send* dépose le message émis dans la liste des messages à envoyer correspondant à chaque destinataire.

Une optimisation envisageable pour cette primitive, dans le cas où plusieurs destinataires sont situés sur un même site S, est de ne déposer le message qu'une seule fois dans la liste des messages à envoyer au site S.

IX.4 Analyse des performances du protocole RBC

IX.4.1 Introduction

Deux types de tests ont été effectués pour mesurer les performances du protocole RBC.

Nous avons mesuré d'une part le temps nécessaire à l'initialisation d'une diffusion atomique ordonnée en fonction de la taille du message diffusé et en fonction du nombre de destinataires.

Nous avons d'autre part mesuré le temps écoulé entre l'initialisation d'une diffusion et le moment où son initiateur est informé de sa terminaison. Ces dernières mesures ont été paramétrées selon la longueur du message diffusé, le nombre de destinataires et la localisation relative des destinataires (plusieurs destinataires sur un même site ou pas).

Nous analysons dans ce paragraphe les résultats obtenus pour les deux types de mesures.

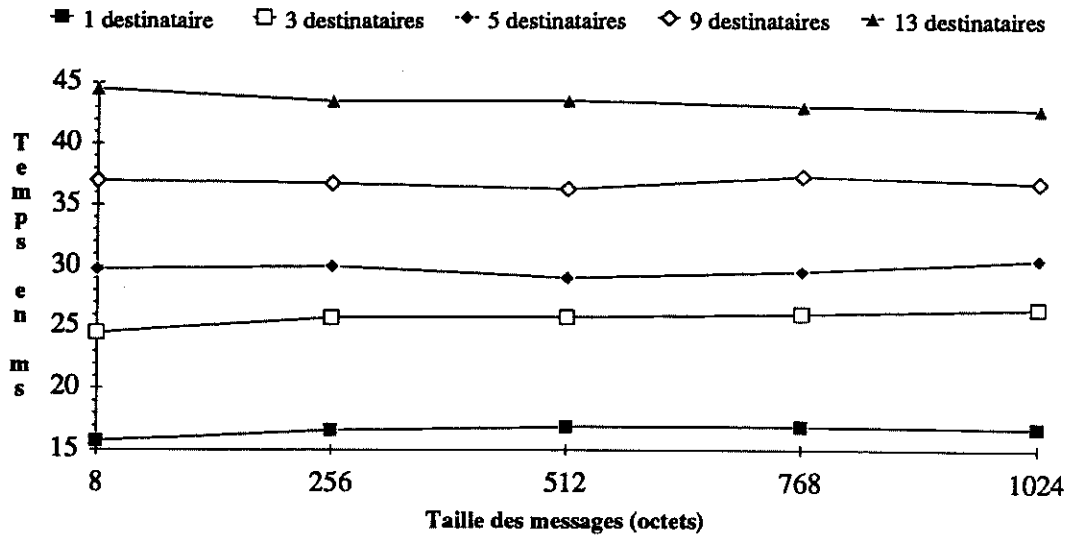


Figure IX.17 Performance de la diffusion atomique

IX.4.2 Temps nécessaire à l'initialisation d'une diffusion

Les mesures que nous avons effectuées ne permettent pas de déceler de notables variations du temps nécessaire à l'initialisation d'une diffusion en fonction du nombre de destinataires et de la taille du message diffusé lorsque tous les destinataires sont situés sur un même site. Nous avons obtenu une valeur constante de 40 millisecondes quel que soit le nombre de destinataires entre 1 et 10 et quelle que soit la longueur du message entre 8 et 728 octets. Ce temps correspond au temps nécessaire à l'allocation et à l'initialisation d'un descripteur de diffusion ajouté au temps nécessaire à l'initialisation de la diffusion atomique (*non* ordonnée) du message de type *init_transaction*.

Il faut noter qu'un seul point de reprise du processus initiateur est sauvegardé lors de l'initialisation d'une diffusion atomique ordonnée. Le temps de sauvegarde du point de reprise du processus de test est d'environ 14.5 millisecondes (pour une taille de près de 19 kilo-octets).

Lorsque plusieurs ports stables destinataires d'une diffusion atomique ordonnée sont localisés sur un même site, les messages du protocole RBC ne sont pas envoyés autant de fois qu'il y a de ports stables destinataires mais une seule fois par site sur lequel se trouve au moins un port stable destinataire. Cela explique la valeur constante du temps d'initialisation d'une diffusion atomique ordonnée lorsque tous les ports stables destinataires sont localisés sur un seul site distant. Dans ce cas, un seul message de type *init_transaction* est transmis quel que soit le nombre de destinataires. La valeur obtenue (40 millisecondes) représente donc une borne inférieure du temps d'initialisation d'une diffusion atomique ordonnée.

Lorsque les ports stables destinataires sont localisés sur des sites distincts, le temps

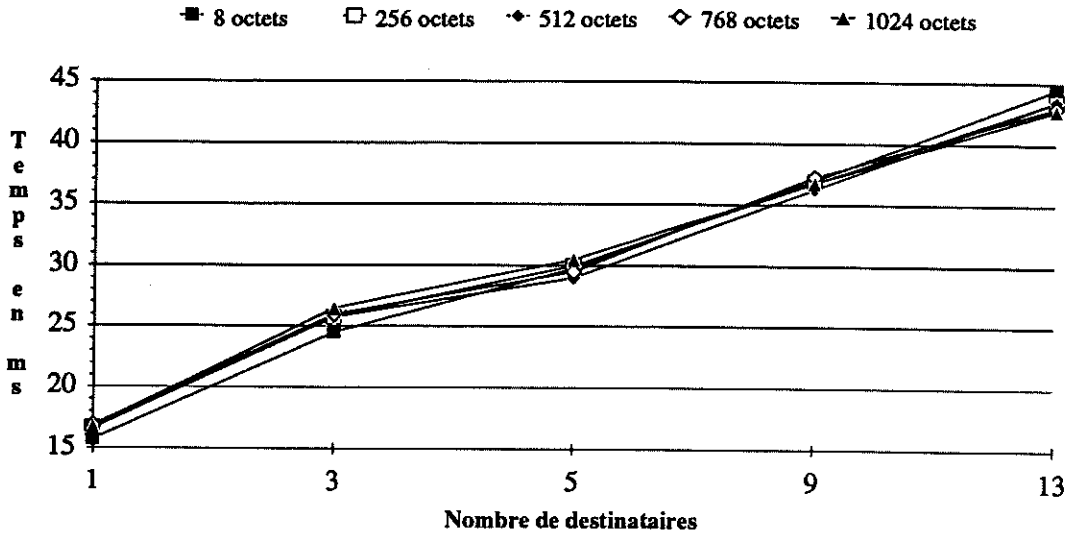


Figure IX.18 Temps pour une diffusion atomique en fonction du nombre de destinataires pour différentes tailles de messages

d'initialisation augmente linéairement avec le nombre de sites destinataires différents puisque le temps nécessaire à l'initialisation de la diffusion atomique du message de type *init_transaction* augmente linéairement en fonction du nombre de destinataires (voir figure IX.18).

IX.4.3 Temps nécessaire à la terminaison d'une diffusion atomique ordonnée

Le protocole RBC est un protocole à deux phases :

- envoi du message diffusé à ses destinataires,
- envoi par les destinataires d'une proposition d'estampille au site initiateur de la diffusion,
- envoi par le site initiateur de la diffusion de l'estampille finale aux destinataires.

En outre, de façon à être en mesure de fournir un compte rendu au processus initiateur de la diffusion, les sites destinataires envoient un acquittement au site initiateur quand le message diffusé est déposé dans ses ports stables destinataires.

Nous avons mesuré le temps passé dans la primitive *rlb_multicast* lorsqu'elle est utilisée avec un délai non nul et suffisamment élevé pour permettre à la diffusion de se terminer avec succès avant l'expiration du délai. Ce temps correspond au temps écoulé entre l'initialisation

de la diffusion et la transmission du compte rendu au processus initiateur après réception de tous les messages de type *acquittement.*

La figure IX.19 présente les résultats obtenus en fonction de la taille du message diffusé pour des ports stables destinataires localisés sur un même site distant. Le temps mesuré varie entre 1080 et 1210 millisecondes. L'écart entre ces deux valeurs extrêmes représente environ 10 % de la valeur mesurée. Compte tenu de la précision des mesures et des remarques exposées dans le paragraphe IX.3.3, nous pouvons considérer à la vue de ces résultats que le temps de terminaison d'une diffusion est sensiblement indépendant du nombre de destinataires lorsque ceux-ci sont localisés sur le même site distant. Quelque soit le nombre de destinataires, le nombre de messages échangés entre le site initiateur et le site destinataire est le même.

Bien entendu, le temps de traitement d'un message de type *validation* dépend du nombre de destinataires locaux puisque le message diffusé doit alors être déposé dans le port stable de chacun des destinataires. Il faudrait faire des mesures plus fines pour déterminer la durée des traitements effectués par le processus gestionnaire des diffusions [1] sur un site. Mais compte tenu des problèmes d'analyse des résultats déjà rencontrés au niveau du protocole RMC, nous n'avons pas entrepris de telles mesures dans la mise en œuvre actuelle du protocole RBC. Lorsque les destinataires d'une diffusion atomique ordonnée sont situés

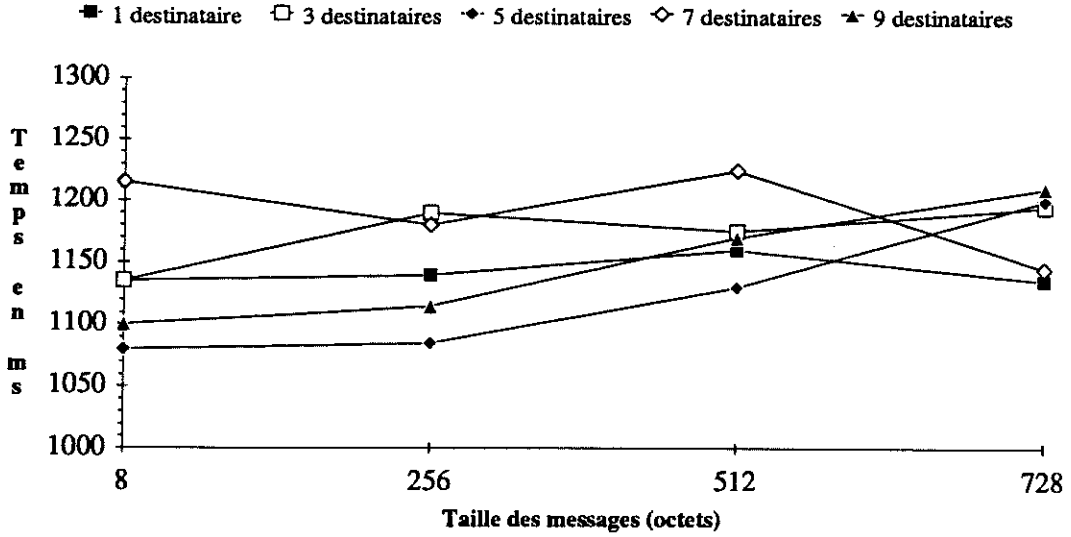


Figure IX.19 Temps pour une diffusion atomique ordonnée vers des destinataires localisés sur un même site

sur des sites différents, le temps de terminaison augmente avec le nombre de destinataires. Nous avons réalisé les mesures dont les résultats sont présentés sur les graphiques IX.20 et

[1] La taille du point de reprise du processus gestionnaire des diffusions (environ 60 kilo-octets) est nettement plus élevée que celle des processus de test des performances. Le temps de sauvegarde d'un point de reprise pour ce processus est de près de 40 millisecondes.

IX.21 avec cinq sites différents notés S_0, S_1, S_2, S_3, S_4. L'initiateur de la diffusion était situé

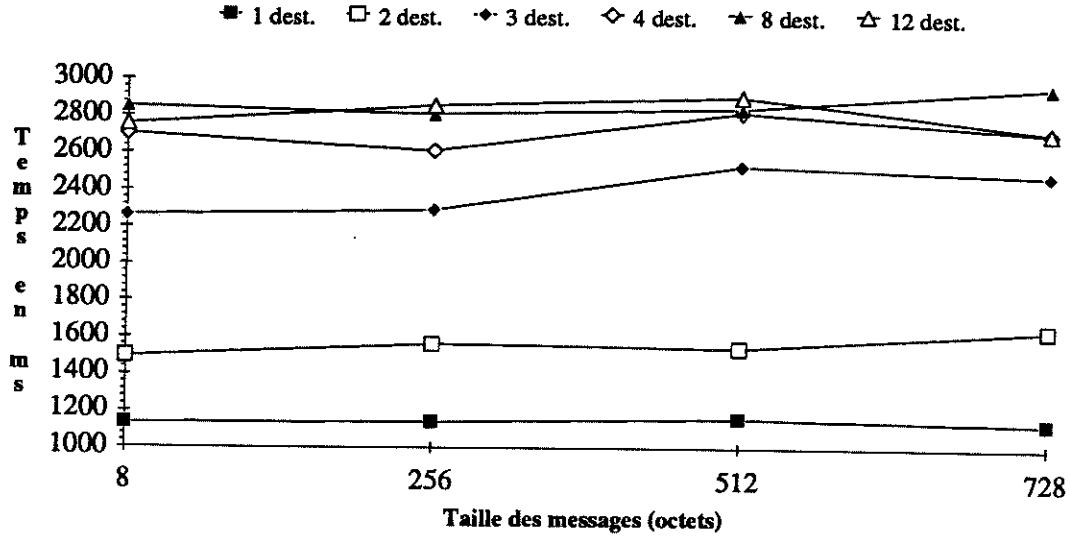


Figure IX.20 Temps pour une diffusion atomique ordonnée vers des destinataires localisés sur différents sites en fonction de la taille des messages

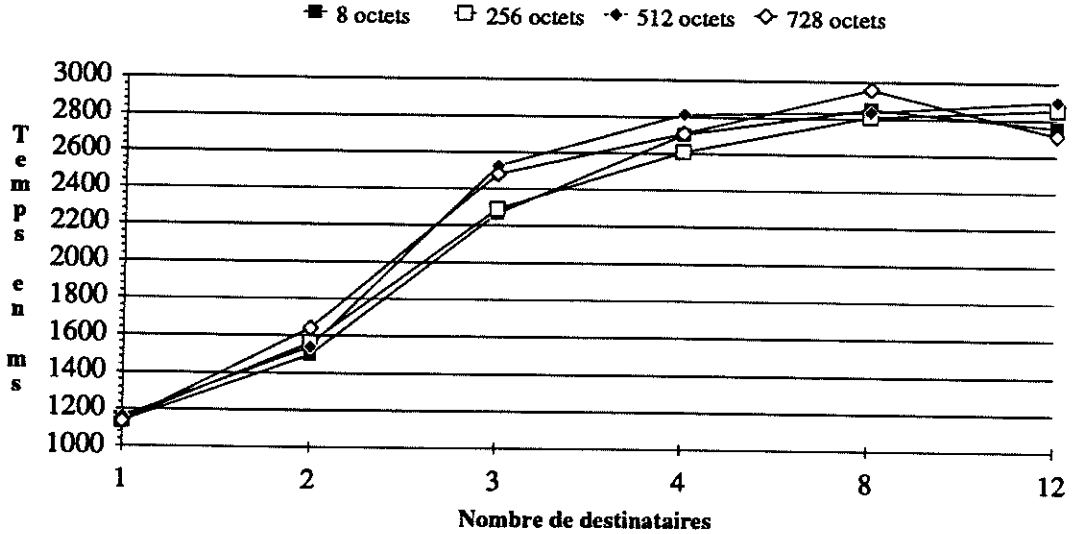


Figure IX.21 Temps pour une diffusion atomique ordonnée vers des destinataires localisés sur différents sites en fonction du nombre de destinataires

sur le site S_0 tandis que les autres sites accueillaient les destinataires des diffusions. Pour les diffusions vers au plus quatre destinataires, tous les destinataires ont été répartis sur des sites distincts. Pour les diffusions vers plus de quatre destinataires, les destinataires étaient répartis uniformément sur les sites S_1 à S_4.

Le graphique IX.21 montre que le temps de terminaison augmente linéairement (aux erreurs de mesure près) avec le nombre de destinataires lorsqu'il n'y a pas deux destinataires sur un même site (voir valeurs obtenues pour 1, 2, 3 et 4 destinataires).

Lorsque plusieurs destinataires sont sur un même site (voir valeurs obtenues pour 4, 8 et 12 destinataires), on retrouve les résultats énoncés dans le paragraphe précédent, à savoir que le temps de terminaison ne dépend pas du nombre de destinataires quand ceux-ci sont tous sur un même site. Ainsi, les temps obtenus pour 4, 8 ou 12 destinataires sont identiques (aux erreurs de mesure près). Dans les trois cas, on a en effet le même nombre de messages échangés entre le site initiateur et les sites destinataires.

La figure IX.20 montre (une fois de plus) que le temps de terminaison mesuré ne dépend pas de la taille des messages diffusés.

IX.4.4 Récapitulatif

Le coût du protocole RBC est élevé du fait que deux phases sont nécessaires à la terminaison d'une diffusion atomique ordonnée. Cependant, les mesures de performance effectuées montrent que le coût de ce protocole n'augmente avec le nombre de ports stables destinataires que si ceux-ci sont tous situés sur des sites différents. En effet, quel que soit le nombre de destinataires sur un site, le temps de terminaison du protocole RBC reste sensiblement constant. Autrement dit, le coût du protocole RBC augmente avec le nombre de sites destinataires mais pas forcément avec le nombre de ports stables destinataires car il est fonction de la localisation relative des ports stables destinataires.

Les valeurs mesurées pour le protocole RBC reflètent les problèmes de performance du protocole IP soulevés dans le paragraphe VII ; c'est pourquoi nous n'avons pas fait d'analyse plus poussée des performances du protocole RBC mis en œuvre dans le système GOTHIC.

IX.5 Conclusion

Le système de communication fiable de GOTHIC est aujourd'hui opérationnel. Les protocoles RMC et RBC sont mis en œuvre. Une véritable évaluation des performances de ces protocoles a été conduite.

Outre l'installation du logiciel de communication réseau SC7 sur le système GOTHIC et le portage du protocole IP, le système de communication fiable de GOTHIC a nécessité l'écriture et la mise au point de plus de 13 000 lignes codées en langage C réparties comme suit : approximativement 10 000 lignes pour le protocole RMC et 3 000 pour le protocole RBC.

En sus de notre travail, le développement du système de communication fiable de

GOTHIC a nécessité l'encadrement d'un mini-projet (3 étudiants en année terminale à l'INSA de Rennes), d'un stagiaire de DESS informatique de l'IFSIC et d'un stagiaire de fin d'études INSA.

Le système de communication fiable de GOTHIC est la première application majeure utilisant la mémoire stable.

Les mesures de performance révèlent des valeurs élevées pour les envois et réceptions de messages par l'intermédiaire du protocole IP. Ces valeurs, dans leur état brut, ne sont pas directement comparables aux valeurs des systèmes de communication décrits dans la littérature. Plusieurs facteurs permettent d'expliquer les valeurs relevées : l'ancienneté du matériel de test dont les performances sont évidemment loin d'égaler les performances des machines actuelles, la mise en œuvre du logiciel de communication de base SC7 qui entraîne de nombreuses commutations de processus et utilise intensivement le bus SMBUS pour l'accès à des données en mémoire globale et l'accès à l'unité d'échange Ethernet, et à un degré moindre le manque de parallélisme pour les accès à la mémoire stable.

Le logiciel de communication réseau pénalise donc considérablement les protocoles de communication fiable. Sa réécriture suffirait à augmenter les performances globales du système de communication de GOTHIC et à rendre celles-ci satisfaisantes.

L'analyse des performances montre que les accès à la mémoire stable sont peu coûteux même si la sauvegarde des points de reprise des processus prend un temps non négligeable. Il serait donc décisif de concevoir un système de micro-reprise permettant de sauvegarder des points de reprise avec une granularité très fine. Avec une telle approche, seules les variables modifiées depuis le dernier point de reprise seraient sauvegardées.

Compte tenu de l'analyse des performances des protocoles RMC et RBC, la construction d'un système de communication fiable efficace à l'aide de mémoires stables rapides est à notre portée. Un prolongement possible de cette étude serait la mise en œuvre du système de communication fiable sur du matériel actuel au dessus d'un protocole de communication réseau performant.

Mieux encore, il est possible d'envisager la conception et la réalisation d'une carte mémoire stable «intelligente» dédiée à des protocoles de communication fiable. Cela reviendrait à concevoir une unité d'échange réseau à base de mémoire stable intégrant des protocoles de communication fiable (point à point, diffusion ...). Dans cette nouvelle approche, le système de communication fiable (mis en œuvre avec du matériel adapté) serait la base de toutes les communications intersites alors que dans le système GOTHIC actuel, les protocoles RMC et RBC sont mis en œuvre au dessus de protocoles standards non fiables.

Chapitre X

Conclusion générale

X.1 Bilan

Le but de ce travail de thèse était la conception et la réalisation d'un protocole d'appel de multiprocédure à distance pour un réseau local de multiprocesseurs dotés de mémoires stables.

Dans la première partie de ce document, nous avons présenté les travaux du domaine des communications fiables dans les systèmes distribués que nous avons étudiés lors de la conception du protocole d'appel de multiprocédure à distance.

Nous avons présenté tout d'abord les protocoles d'appel de procédure à distance puisqu'ils traitent le cas particulier de l'appel 1-1 de multiprocédure. Les protocoles d'appels de procédure à distance diffèrent essentiellement par le nombre d'exécutions de la procédure appelée susceptibles de se produire en présence de pannes. La gamme des sémantiques s'étend de la sémantique *peut-être une fois* à la sémantique *au plus une fois*. La première n'offre aucune garantie quant aux nombres d'exécutions auxquelles un appel peut donner lieu. La dernière assure que la procédure appelée est exécutée exactement 0 ou 1 fois en présence de pannes. Le problème le plus délicat à résoudre est l'élimination de orphelins créés en cas de panne du site client. Un compromis entre performance et fiabilité doit être trouvé.

Nous avons ensuite décrit quelques protocoles de diffusion fiable qui permettent une communication de type 1 vers N. Un grand nombre de protocoles de diffusion fiable existe. Nous avons établi un classement suivant l'ordonnancement réalisé sur l'ensemble des messages diffusés. Nous n'avons pas bien entendu dressé une liste exhaustive des protocoles existants et avons choisi de n'en présenter que quelques uns qui nous semblent représentatifs des travaux dans ce domaine tels par exemple les protocoles ABCAST et CBCAST mis en œuvre dans le système Isis.

Enfin, nous nous sommes penchés sur quelques propositions d'extension de l'appel de procédure à distance qui permettent l'exécution d'une procédure en parallèle sur plusieurs

sites.

Dans la deuxième partie de ce document, nous avons présenté le concept de multi-procédure introduit dans le langage PolyGoth et exposé le protocole RMPC d'appel de multiprocédure à distance étudié dans le cadre de Gothic.

La mise en œuvre du protocole RMPC dans le système Gothic nous a conduit à définir deux protocoles de communication fiable : les protocoles RMC et RBC. Le protocole RMC offre des primitives de communication fiable par message ainsi qu'une primitive de diffusion atomique. Le protocole RBC offre une primitive de diffusion atomique ordonnée qui garantit que tous les message diffusés sont délivrés dans le même ordre à tous leurs destinataires communs. Ces protocoles sont décrits dans la troisième partie de ce document qui commence par une présentation des principaux aspects matériels et logiciels du système Gothic.

L'originalité des protocoles RMC et RBC repose sur l'utilisation de mémoires stables rapides. La mémoire stable de Gothic offre la possibilité de mettre à jour atomiquement un groupe d'objets stables. Nos protocoles sont fondés sur l'utilisation de cette primitive qui permet d'assurer l'atomicité des traitements effectués localement sur un site.

Les bonnes performances obtenues pour la mise à jour des informations en mémoire stable permettent d'y ranger de nombreuses informations et autorisent des mises à jour fréquentes. Par conséquent, les protocoles proposés auraient été inconcevables si nous n'avions pas disposé dans l'architecture matérielle de Gothic d'une mémoire stable *rapide*. L'analyse des performances des protocoles RMC et RBC donnée dans le chapitre IX atteste du réalisme de notre approche. Au chapitre des optimisations du système de communication, il serait judicieux de réduire le nombre de couches logicielles en songeant tout particulièrement au développement d'un système de communication fiable fondé directement sur du matériel spécialisé intégrant un processeur dédié aux communications et de la mémoire stable.

Le système de communication fiable de Gothic est la première réalisation importante qui utilise la mémoire stable. Cette réalisation nous a permis de mettre en évidence certains défauts de la mémoire stable de Gothic. Ces défauts seront corrigés dans la nouvelle mémoire stable, actuellement en cours de définition, qui est destinée à être utilisée dans deux nouvelles architectures de machines tolérantes aux fautes [Bana90b].

X.2 Le protocole de rendez-vous atomique

L'utilisation de la mémoire stable peut être étendue à d'autres protocoles. En marge du protocole RMPC, nous nous sommes intéressés au problème de la mise en œuvre du rendez-vous CSP atomique [Mori89a]. Le langage CSP a été proposé par Hoare [Hoar78]. Il est utile de donner ici une description des commandes de communication de ce langage.

Soient P_i et P_j deux processus. La commande P_i ! x dans le corps de P_j exprime une requête de P_j pour recevoir une valeur émise par P_i. La commande P_j ? m exprime une requête de P_i pour envoyer la valeur à P_j. L'exécution des commandes P_i ! x et P_j ? m est synchronisée et a pour résultat l'affectation de la valeur m à la variable x.

Le protocole de rendez-vous atomique, appelé RDV, est mis en œuvre au dessus du protocole RMC [Coll90]. L'idée à la base du protocole RDV est l'utilisation du concept de transaction atomique [Gray78] pour mettre en œuvre le rendez-vous atomique. Cette idée conduit à un protocole très simple :

- association d'une transaction à chaque rendez-vous,
- utilisation des protocoles de validation bien connus pour valider les transactions.

Le but de la transaction associée à un rendez-vous est de mettre à jour la variable x du récepteur avec la valeur m émise par le deuxième processus.

Le principal problème qui se pose est d'assurer l'unicité de la transaction associée à un rendez-vous [Bana89a]. Lorsqu'un processus exécute une commande de rendez-vous, une requête est transmise à un processus gestionnaire des rendez-vous, appelé G_RDV$_i$, qui est chargé de créer une transaction pour exécuter le rendez-vous. Compte tenu de la symétrie du rendez-vous, il faut prendre garde à ne pas créer deux transactions pour un même rendez-vous. L'unicité de la transaction associée à un rendez-vous est assurée grâce à la coopération des gestionnaires de rendez-vous des différents sites.

Plusieurs cas peuvent se présenter lorsqu'un gestionnaire de rendez-vous S_RDV_i reçoit une requête de rendez-vous entre P_i et P_j provenant de l'un des processus P_i qui s'exécute sur son site :

(1) P_j a exécuté sa commande de rendez-vous avant P_i et le gestionnaire de rendez-vous S_RDV_j a envoyé une requête de demande de rendez-vous à S_RDV_i qui l'a reçue avant que P_i n'exécute sa commande de rendez-vous.

Dans ce cas, S_RDV_i est le seul gestionnaire de rendez-vous à connaître les deux requêtes de rendez-vous. Il peut donc créer une transaction pour exécuter le rendez-vous correspondant puis en informer S_RDV_j.

(2) Au moment où P_i exécute sa commande de rendez-vous le gestionnaire de rendez-vous S_RDV_i n'a pas connaissance d'une hypothétique requête de rendez-vous émanant de P_j.

Dans ce cas, S_RDV_i ne peut pas créer de transaction car il n'a connaissance que d'une seule des deux requêtes de rendez-vous. Il envoie donc un message de requête de rendez-vous au gestionnaire de rendez-vous S_RDV_j et conserve localement les informations relatives au rendez-vous demandé par P_i.

(3) P_i et P_j exécutent simultanément leur requête de rendez-vous. Les gestionnaires de rendez-vous S_RDV_i et S_RDV_j s'envoient donc mutuellement une requête de rendez-vous. Ils connaissent donc tous les deux les deux requêtes de rendez-vous qui se correspondent. Or seul, l'un d'entre eux doit créer la transaction associée au rendez-vous. C'est le gestionnaire de rendez-vous dont l'identité est la plus petite [1] qui crée la transaction et en informe l'autre gestionnaire de rendez-vous. Ce dernier attend simplement le message indiquant la création de la transaction.

La validation de la transaction est réalisée par un protocole classique que nous ne décrivons pas.

Le protocole RDV [Mori89b] n'est qu'un exemple de protocole mis en place au dessus du protocole RMC mais il est tout à fait envisageable de concevoir d'autres protocoles utilisant les primitives de communication fournies par les protocoles RMC et RBC. Il est maintenant possible d'exécuter de véritables applications distribuées (composées de processus communicants) tolérantes aux fautes sur le système GOTHIC.

Cette étude menée sur le rendez-vous atomique ouvre quelques perspectives de recherche. Il serait par exemple intéressant d'étendre notre protocole au rendez-vous généralisé de N processus. Par ailleurs, nous n'avons pas traité l'indéterminisme du rendez-vous. Il faudrait intégrer à notre protocole un mécanisme autorisant l'utilisation des commandes de rendez-vous dans les commandes gardées. Une autre voie est l'intégration d'un mécanisme de traitement d'exception adapté au parallélisme des processus communicants. Une première étude a d'ores et déjà été menée sur ce sujet dans l'équipe LSP [Issa90].

X.3 Perspectives

Ce travail de thèse ouvre plusieurs perspectives d'une part dans le cadre du projet GOTHIC et d'autre part dans le cadre de nouvelles études qui visent à construire des machines tolérantes aux fautes : la machine *Fault Tolerant Multiprocessor* (FTM) et la machine *Fault tolerant Architecture with Stable Storage Technology* (FASST).

Dans le cadre du projet GOTHIC plusieurs études ont fait l'objet d'expérimentations dont l'ensemble constitue le prototype actuel du système GOTHIC (Gothic-v1). Ainsi, la réalisation d'une mémoire virtuelle segmentée paginée et la mise en œuvre d'un noyau de processus stables et de protocoles de communication fiable sont le fruit de deux études disjointes.

[1] Un ordre total peut être construit sur les identités de processus en considérant que les sites sont identifiés de manière unique et en prenant comme identité d'un processus un couple constitué de l'identité du site sur lequel il s'exécute et de l'identificateur local du processus.

Dans Gothic-v1, la notion de mémoire stable commune n'existe pas. En l'absence de panne, une mémoire stable ne peut être accédée que par le processeur auquel elle est associée. Une première perspective est de réaliser une mémoire stable commune à partir des mémoires stables physiques associées aux divers processeurs de façon à simplifier la programmation d'applications tolérantes aux fautes. Le principal problème à résoudre est la généralisation des opérations atomiques offertes par la mémoire stable actuelle à des groupes d'objets répartis dans des mémoires stables physiques distinctes. L'utilisation de protocoles de communication fiable (en particulier, de diffusion atomique) peut permettre de mettre en œuvre ce type d'opérations. Dans cette première optique, les deux types de mémoire (mémoire virtuelle et mémoire stable) sont toujours visibles. Même s'il dispose de l'abstraction de mémoire stable commune, le programmeur doit toujours gérer explicitement les objets rangés en mémoire stable.

Une seconde perspective est de réaliser une mémoire virtuelle segmentée stable commune de façon à rendre la tolérance aux fautes transparente aux applications. Dans Gothic-v1, l'espace mémoire partagé est l'ensemble des disques (mémoire secondaire). Les mémoires locales des processeurs constituent des caches vis à vis de la mémoire secondaire. L'idée est maintenant d'avoir comme espace mémoire partagé la mémoire stable commune et de garantir par un protocole de vidage des caches (recopie des blocs mémoire présents dans les mémoires locales) atomique que la mémoire stable commune contient à tout instant un état cohérent (point de reprise) des processus qui s'exécutent sur le multiprocesseur. Le protocole de vidage des caches est distribué compte tenu du fait que les données modifiées par un processus peuvent se trouver dans différentes mémoires locales à cause du partage. Les protocoles de diffusion fiable que nous avons étudiés pourraient servir à la mise en œuvre d'une mémoire virtuelle segmentée stable commune dans une nouvelle version de GOTHIC.

La réalisation effective d'une mémoire virtuelle segmentée stable commune dans GOTHIC n'est pas envisagée du fait des caractéristiques de la mémoire stable actuelle (accès en exclusion mutuelle, autonomie). Nous préférons poursuivre nos recherches en tolérance aux fautes dans le cadre de nouvelles études auxquelles GOTHIC a donné naissance.

Le projet FTM [Bana91b] a pour objectif de construire une architecture faiblement couplée tolérante aux fautes. La mémoire locale de chaque processeur de l'architecture est une mémoire stable. La tolérance aux fautes est transparente pour les applications mais pas pour le système. Les services fiables du système utilisent des mécanismes de construction dynamique d'actions atomiques dont la mise en œuvre fait appel à des protocoles de communication fiable tels que ceux que nous avons présentés dans ce document.

La seconde étude, dans laquelle nous sommes impliqués, a pour but de construire un multiprocesseur fortement couplé tolérant les fautes de façon transparente au système et aux utilisateurs [Bana90a]. Ces développements sont effectués dans le cadre du projet ESPRIT-II FASST. Dans la machine FASST, la mémoire commune est une mémoire stable transactionnelle (MST). La MST peut être vue comme une généralisation de la mémoire stable de

GOTHIC. Elle supporte un mécanisme de transaction atomique. Ce type d'opération était embryonnaire dans la mémoire stable de GOTHIC avec l'opération de mise à jour atomique d'un groupe d'objets. La MST n'est visible que par les caches qui envoient les commandes d'accès (gestion des transactions, lecture, écriture). La tolérance aux fautes est mise en œuvre directement par la MST qui possède un mécanisme de reprise adéquat : la tolérance aux fautes est donc transparente à tout sous-système et toute application qui s'exécute sur la machine FASST.

Bibliographie

[Baba85] O. Babaoglu et R. Drummond. Streets of Byzantium : Network Architectures for Fast Reliable Broadcast. *IEEE Transactions on Software Engineering*, SE-11(6), 1985.

[Baba88] O. Babaoglu, P. Stephenson, et R. Drummond. Reliable Broadcasts and Communication Models : Tradeoffs and Lower Bounds. *Distributed Computing*, (2):177–189, 1988.

[Baco87] J.M. Bacon et K.G. Hamilton. *Distributed Computing with RPC : The Cambridge Approach.* Technical Report 117, University of Cambridge Computer Laboratory, England, Octobre 1987.

[Baco89] J. Bacon. *Evolution of Operating System Structures.* Technical Report 166, University of Cambridge Computer Laboratory, Cambridge (England), Mars 1989.

[Ball80] Gene Ball. *Alto as Terminal.* Internal memorandum, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Avril 1980.

[Bana86] J.P. Banâtre, M. Banâtre, et Fl. Ployette. The Concept of Multi-fonction : a General Structuring Tool for Distributed Operating Systems. In *Proc. of 6th International Symposium on Principles of Distributed Computing*, pages 478–485, IEEE, Cambridge, Mai 1986.

[Bana87] J. P. Banâtre, M. Banâtre, et G. Muller. Quelques aspects du système GOTHIC. *Techniques et Sciences Informatiques*, 6(2), 1987.

[Bana88a] J. P. Banâtre, M. Banâtre, et G. Muller. Ensuring Data Security and Integrity with a Fast Stable Storage. In *Proc. of 4th International Conference on Data Engineering*, pages 285–293, IEEE, Los Angeles, Février 1988.

[Bana88b] J.P Banâtre, M. Banâtre, et G. Muller. Main Aspects of the GOTHIC Distributed System. In *European Teleinformatics Conference on Research into Networks and Distributed Applications*, pages 747–760, Vienna, Austria, Avril 1988.

[Bana89a] J. P. Banâtre, M. Banâtre, et C. Morin. Implementing Atomic Rendezvous within a Transactional Framework. In *Proc. of 8th Symposium on Reliable Distributed Systems*, pages 119–128, IEEE, Seattle, Octobre 1989.

[Bana89b] J.P Banâtre et M. Benveniste. Multiprocedures: Generalized Procedures for Concurrent Programming. In 3^{rd} *Workshop on Large Grain Parallelism*, Software Engineering Institute, CMU, Octobre 1989.

[Bana90a] M. Banâtre et P. Joubert. Cache Management in a Tightly Coupled Fault Tolerant Multiprocessor. In *Proc. of 20th International Symposium on Fault-Tolerant Computing Systems*, Newcastle, Juin 1990.

[Bana90b] M. Banâtre, P. Joubert, C. Morin, G. Muller, B. Rochat, et P. Sanchez. Stable Transactional Memories and Fault Tolerant Architectures. In *Proc. of Fourth ACM SIGOPS European Workshop, Fault Tolerance Support in Distributed Systems*, Bologne, Italy, Septembre 1990.

[Bana91a] J.P. Banâtre et M. Banâtre, éditeurs. *Les systèmes distribués : l'expérience du système Gothic.* InterEditions, Février 1991.

[Bana91b] M. Banâtre, G. Muller, B. Rochat, et P. Sanchez. *Design Decisions for the FTM : a General Purpose Fault Tolerant Machine.* Rapport de recherche, INRIA, Rennes (France), 1991.

[Bers87] B. N. Bershad et al. A Remote Procedure Call Facility for Interconnecting Heterogeneous Computer Systems. *IEEE Transactions on Software Engineering*, SE-13(8):880–894, 1987.

[Birm84] K.P. Birman, A. El Abbadi, W. Dietrich, T. Joseph, et T. Raeuchle. *An overview of the Isis Project.* Technical Report TR 84-642, Computer Science Department, Cornell University, Ithaca, New York, Octobre 1984.

[Birm85] K. Birman et T. Joseph. *Reliable Communication in an Unreliable Environment.* Technical Report TR 85-694, Computer Science Department, Cornell University, Ithaca, New York, Juillet 1985.

[Birm86] K. P. Birman. *A System for fault Tolerant Distributed Computing.* Technical Report TR 86 744, Computer Science Department, Cornell University, Avril 1986.

[Birm87] K. P. Birman et T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Février 1987.

[Birm88] K. P. Birman et T. A. Joseph. *Reliable Broadcast Protocols.* Technical Report TR 88-918, Computer Science Department, Cornell University, Ithaca, New York, Juin 1988.

[Birm90] K. Birman, A. Schiper, et P. Stephenson. *Fast Causal Multicast.* Technical Report TR-1105, Computer Science Department, Cornell University, Avril 1990.

[Birr84] A.D. Birrell et B.J. Nelson. Implementing Remote Procedure Call. *ACM Transactions on Computer Systems*, 2(1):39–59, Février 1984.

[Brin78] P. Brinch-Hansen. Distributed Processes : A Concurrent Programming Concept. *Communications of the ACM*, 21(11):934–941, Novembre 1978.

[Brow84] F. Browaeys, H. Derriennic, P. Desclaud, H. Fallour, C. Faulle, J. Febvre, J.E. Hanne, M. Kronental, J.J. Simon, et D. Vojnovic. Sceptre : proposition de noyau normalisé pour les exécutifs temps réel. *Techniques et Sciences Informatiques*, 3(1):45–62, Janvier 1984.

[Bull85] Bull. *Structure Générale de la SPS7.* Technical Report 20 893 108 158 01/FR, ref BULL-SEMS, 1985.

[Bull86] Bull. *Utilisation du module d'échange ETHERNET, Niveau 1.* Technical Report 20 893 104 108 02/FR, ref BULL-SEMS, 1986.

[Bull88] Bull. *SPS 7, SC7-X25-ETHERNET-STID, Manuel de référence.* Technical Report 72 F2 61SP REV 1, ref BULL-SEMS, Novembre 1988.

[Chan84] J. M. Chang et N. F. Maxemchuck. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Août 1984.

[Cher85] D.R. Cheriton et W. Zwaenepoel. Distributed Process Group in the V Kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, Mai 1985.

[Coll90] A. Collard, O. Ogier, et M.P. Rouillé. *Mise en œuvre du rendez-vous C.S.P. atomique.* Rapport de mini-projet INSA Rennes, IRISA, IRISA, Janvier 1990.

[Coop85] E. Cooper. *Replicated Distributed Programs.* Ph.D. dissertation, Computer Science Division, University of California, Berkeley, Mai 1985.

[Corp81] Xerox Corporation. *Courier : the remote procedure call protocol.* Technical Report XSIS-038112, Xerox System Integration Standard, Stamford, Connecticut, 1981.

[Cris86] F. Cristian, H. Aghili, R. Strong, et D. Dolev. *Atomic Broadcast : From Simple Message Diffusion to Bizantine Agreement.* Technical Report RJ 5244, Almaden Research Center San Jose (California), Juillet 1986.

[Cris89] F. Cristian. *Synchronous Atomic Broadcast for Redundant Broadcast Channels.* Technical Report RJ 7203 (67682), IBM Almaden Research center, San Jose (California), Décembre 1989.

[Dahl68] O.J. Dahl. *Simula-67 Common Base Language.* Technical Report, Norvegian Computing Center, Oslo, Norway, 1968.

[Depa90] E. Departout. *Mise en œuvre d'un protocole de diffusion atomique et ordonnée.* Rapport de stage de fin d'étude INSA Rennes, IRISA, IRISA, Juin 1990.

[Feld71] J.A. Feldman et R.F. Sproull. System Support for the Stanford Hand-Eye System. In IJCAI, éditeur, *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 183–189, London, Septembre 1971.

[Fidg88] C.J. Fidge. Timestamps in Message-Passing Systems that Preserves the Partial Ordering. In *Proc. 11th Australian Comp. Conf.*, Février 1988.

[Garc88a] H. Garcia-Molina et B. Kogan. An Implementation of Reliable Broadcast using an Unreliable Multicast Facility. In IEEE, éditeur, *Proc. of 7th Symposium on Reliable Distributed Systems*, pages 101–111, Octobre 1988.

[Garc88b] H. Garcia-Molina, B. Kogan, et N. Lynch. Reliable Broadcast in Networks with Nonprogrammable Servers. In *Proc. of 8th International Conference on Distributed Computing Systems*, Juin 1988.

[Garc88c] H. Garcia-Molina et A. Spauster. *Ordered and Reliable Multicast Communication.* Technical Report CSTR 184-88, Princeton University Department of Computer Science, Octobre 1988.

[Garc89] H. Garcia-Molina et A.M. Spauster. Message Ordering in a Multi-Cast Environment. In *Proc. of 9th International Conference on Distributed Computing Systems*, pages 354–361, Newport Beach, CA, Juin 1989.

[Gray78] J. Gray. *Notes on Database Operating Systems.* Volume 60 of *Lecture Notes in Computer Science*, Springer Verlag, 1978.

[Hame89a] J.M. Hamel. *Vers la mise en œuvre d'un protocole de communications fiables.* Rapport de stage de fin d'études DESS ISA, Université de Rennes I, Juin 1989.

[Hame89b] J.M. Hamel. *Etude du communicateur dans le logiciel SC7.* Note LSP 66, Université de Rennes I, Août 1989.

[Hedr87] C. L. Hedrick. *Introduction to the Internet Protocols.* Technical Report, Computer Science Facilities Group. RUTGERS. The State University of New Jersey, Juillet 1987.

[Hoar78] C.A.R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, Août 1978.

[Issa90] V. Issarny. Design and Implementation of an Exception Handling Mechanism for Communicating Sequential Processes. In H. Burkhart, éditeur, *CONPAR 90-VAPP IV, Joint Conference on Vector and Parallel Processing,LNCS*, pages 604,615, Springer Verlag, 1990.

[Kaas89] M.F. Kaashoek, A. Tanenbaum, S.F. Hummel, et H.E. Bal. An efficient reliable broadcast protocol. *Operating System Review*, 23(4):5–19, Octobre 1989.

[Krak85] S. Krakowiak. *Principes des systèmes d'exploitation des ordinateurs.* Dunod informatique, 1985.

[Lamp78] L. Lamport. Time, Clock and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, Juillet 1978.

[Lamp81a] B. Lampson. Atomic Transactions. In *Distributed Systems and Architecture and Implementation : an advanced course*, pages 246–265, Springer Verlag, 1981.

[Lamp81b] B.W. Lampson. Remote Procedure Call. In *Distributed Systems and Architecture and Implementation : an advanced course*, chapter 14, pages 365–370, Springer Verlag, New York, 1981.

[Lamp82] L. Lamport, R. Shostak, et M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, Juillet 1982.

[Lecl89] P. Lecler. *Une approche de la programmation des systèmes distribués fondée sur la fragmentation des données et des calculs, et sa mise en œuvre dans le système GOTHIC.* Thèse de doctorat, Université de Rennes I, Septembre 1989.

[Lisk83] B. Liskov et R. Scheifler. Guardians and actions : Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, Juillet 1983.

[Lisk84] B. Liskov. The Argus Language and System. *Lecture Notes in Computer Science*, 190:343–430, 1984.

[Lisk87] B. Liskov, D. Curtis, P. Johnson, et R. Scheifler. Implementation of Argus. In *Proc. of 11th ACM Symposium on Operating Systems Principles*, pages 111–122, 1987.

[Luan88] S. Luan et V.D. Gligor. A Fault-Tolerant Protocol for Atomic Broadcast. In IEEE Computer Society, éditeur, *Proc. of 7th Symposium on Reliable Distributed Systems*, pages 112–126, Octobre 1988.

[Mart86] B. Martin. Parallel Remote Procedure Call and Portable C Stub Compiler. In *Proc. Workshop on Design Principles for Experimental Distributed Systems*, Purdue University (Indiana), Octobre 1986.

[Matt88] F. Mattern. Virtual Time and Global States in Distributed Systems. In *Proc. Int. Conf. on Parallel and Distributed Algorithms*, pages 215–226, North-Holland Publishing, 1988.

[Metc76] R. M. Metcalfe. Ethernet: Distributed Packet Switching for Local Computer Network. *Communications of the ACM*, 7(19):395–406, Juillet 1976.

[Mich89] B. Michel. *Conception et réalisation de la mémoire virtuelle de GOTHIC.* Thèse de doctorat, Université de Rennes I, Septembre 1989.

[Mori89a] C. Morin. Fault-Tolerant Implementation of CSP Input-Output Commands. In *Proc. of The Fourth International Conference on Fault-tolerant Computing Systems*, Baden-Baden (RFA), Septembre 1989.

[Mori89b] C. Morin. An efficient implementation of the Rendezvous Atomicity property. In D.J. Evans, G.R. Joubert, et F.J. Peters, éditeurs, *Parallel Computing 89*, pages 603–608, Elsevier Science Publishers B.V (North-Holland), 1989.

[Mori90] C. Morin. Building a Reliable Communication System Using High Speed Stable Storage. In *Proc of the 5th Int. Symp. on Computer and Information Sciences,*, Cappadocia, Turquey, Novembre 1990.

[Mull88] G. Muller. *Conception et réalisation d'une machine multiprocesseur sûre de fonctionnement.* Thèse de doctorat, Université de Rennes I, Juin 1988.

[Nava88] S. Navaratnam, S. Chanson, et G. Neufeld. Reliable Group Communication in Distributed Systems. In *Proc. of 8th International Conference on Distributed Computing Systems*, pages 439–446, Juin 1988.

[Nels81] B.J. Nelson. *Remote Procedure Call.* Ph.D. dissertation, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, 1981.

[Perr86] K.J. Perry et S. Toueg. Distributed Agreement in the Presence of Processor and Communication Faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, Mars 1986.

[Post80] Jon Postel. *User Datagram Protocol.* Technical Report RFC 768, Network Information Center, SRI International, Août 1980.

[Post81a] Jon Postel. *Internet Protocol.* Technical Report RFC 791, Network Information Center, SRI International, Septembre 1981.

[Post81b] Jon Postel. *Transmission Control Protocol.* Technical Report RFC 793, Network Information Center, SRI International, Septembre 1981.

[Raja89] B. Rajagopalan et P. K. McKinley. A Token-Based Protocol for Reliable, Ordered Multicast Communication. In *Proc. of 8th Symposium on Reliable Distributed Systems*, Seattle, Washington, Octobre 1989.

[Rayn89] M. Raynal, A. Schiper, et S. Toueg. *The causal ordering abstraction and a simple way to implement it.* Rapport de recherche 1132, INRIA, Décembre 1989.

[Rayn90] M. Raynal. *Order Notions and Atomic Multicast in Distributed Systems : A Short Survey.* Rapport de recherche 524, IRISA, Mars 1990.

[Robe70] L. Roberts et B. Wessler. Computer Network Development to Achieve Resource Sharing. In AFIPS, éditeur, *AFIPS Conference Proceedings of the Spring Joint Computer Conference*, pages 543–549, Juin 1970.

[Saty86a] M. Satyanarayanan. *RPC2 User Manual.* CMU-ITC-038, 1986.

[Saty86b] M. Satyanarayanan et E.H. Siegel. *MultiRPC : A Parallel Remote Procedure Call Mechanism.* Technical Report CMU-CS-86-139, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Août 1986.

[Schi89] A. Schiper, A. Sandoz, et J. Eggli. *A new algorithm to implement causal ordering*, chapter x, pages 219–232. Volume 392 of *Lecture Notes in Computer Science*, Springer Verlag, 1989.

[Schn82] F.B. Schneider. Synchronization in Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 4(2):125–148, Avril 1982.

[Schn83] F. B. Schneider. Fail-stop processors. In *Digest of Papers Spring Compcon'83*, pages 66–70, San Fransisco, CA, 1983.

[Schn84] F. Schneider, D. Gries, et R. Schlichting. Fault tolerant Broadcast. *Science of Computer Programming*, 4(1):1–15, 1984.

[Schn86] F. B. Schneider. A paradigm for reliable clock synchronization. In *Proc. Advanced Seminar on Real-Time Local Area Network*, Bandol, France, Avril 1986.

[Sega83] A. Segall et B. Awerbuch. A Reliable Broadcast Protocol. *IEEE Transactions on Communications*, 31(7):896–901, Juillet 1983.

[Shri81] S.K. Shrivastava. Structuring Distributed Systems for Recoverability and Crash Resistance. *IEEE Transactions on Software Engineering*, SE-7(4):436–447, Juillet 1981.

[Shri82] S. K. Shrivastava et F. Panzieri. The design of a Reliable Remote Procedure Call Mechanism. *IEEE Transactions on Computers*, C-31(7), Juillet 1982.

[Shri83] S.K. Shrivastava. On the Treatment of Orphans in a Distributed System. In IEEE Computer Society, éditeur, *Proc. of the 3rd Symposium on Reliability in Distributed Software and Database Systems*, pages 155–162, Florida, Octobre 1983.

[Spec82] A.Z. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Communications of the ACM*, 25(4), Avril 1982.

[Srik87] T. K. Srikanth et S. Toueg. Optimal Clock Synchronization. *Journal of the ACM*, 34(3):626–645, Juillet 1987.

[Svob84a] L. Svobodova. Resilient Distributed Computing. *IEEE Transactions on Software Engineering*, SE-10(3):257–268, Mai 1984.

[Svob84b] L. Svobodova. File Servers for Network-Based Distributed Systems. *ACM Computing Surveys*, 16(4), Décembre 1984.

[Tane88a] A. S. Tanenbaum et R. van Renesse. A Critique of the Remote Procedure Call Paradigm. In *European Teleinformatics Conference on Research into Networks and Distributed Applications*, pages 775–783, Vienna, Austria, Avril 1988.

[Tane88b] A.S. Tanenbaum. *Computer Networks.* Prentice-Hall International, 2nd edition edition, 1988.

[Whit77] J.E. White. Elements of a Distributed Programming System. *Journal of Computer Languages*, 2(4):117–134, Avril 1977.

[Wirt84] N. Wirth. *Programmer en MODULA-2.* Presses polytechniques romandes, 1984.

[Yap86] K.S. Yap. Implementing Fault Tolerant Remote Procedure Call. M.S. Thesis, Dept. of Computer Science, University of Maryland, College Park,MD, 1986.

Annexe A

Automate des primitives de mise à jour d'objets en mémoire stable

A.1 Automates de mise à jour d'un objet en mémoire stable

L'automate correspondant à la primitive *écrire_objet_stable* est représenté sur la figure A.1.

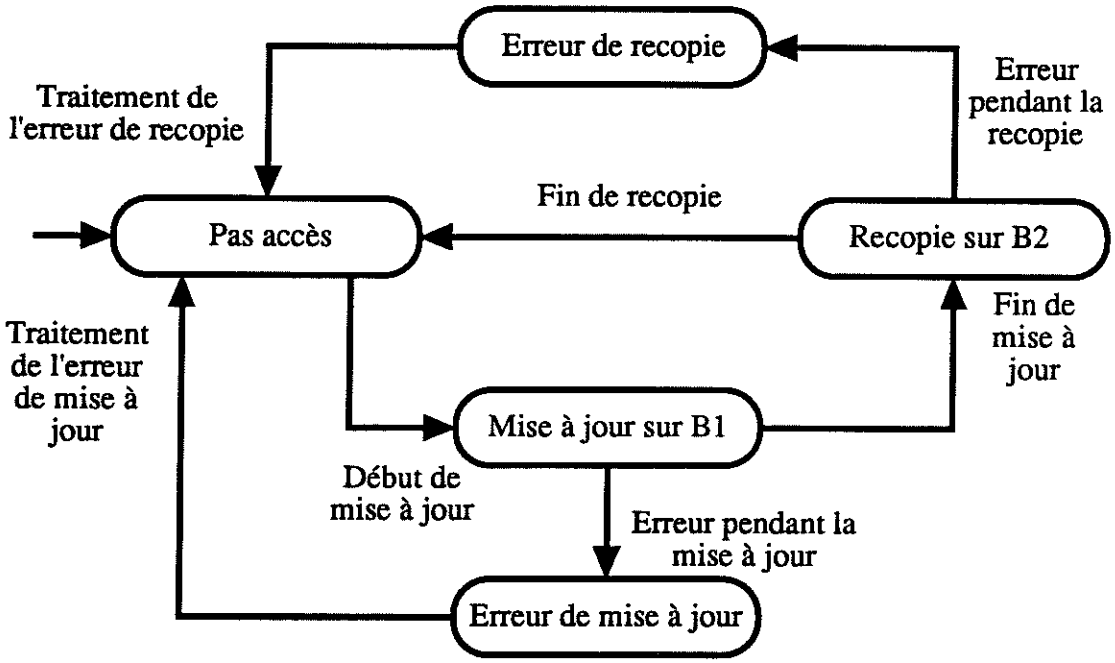


Figure A.1 Automate de l'opération d'écriture atomique d'un objet stable

La mémoire stable comporte un registre appelé *registre d'état* qui contient un état de

l'automate de l'opération en cours. Lorsqu'aucune opération atomique n'est en cours, ce registre contient un état de repos appelé (*Pas accès*).

Pour chacune des phases, le contrôleur vérifie que le processeur a accédé à tous les mots de l'objet dans l'ordre des adresses croissantes. Cette vérification est possible grâce à des bits de contrôle associés à chaque mot de la mémoire stable. Des marques de début et fin d'objets stables sont posées par la primitive *create_stable_object*. De plus, un lien est établi entre tous les mots de l'objet.

A.2 Automate de mise à jour d'un groupe d'objets en mémoire stable

L'automate correspondant à la primitive *maj_atomique_groupe_objet* est représenté sur la figure A.2.

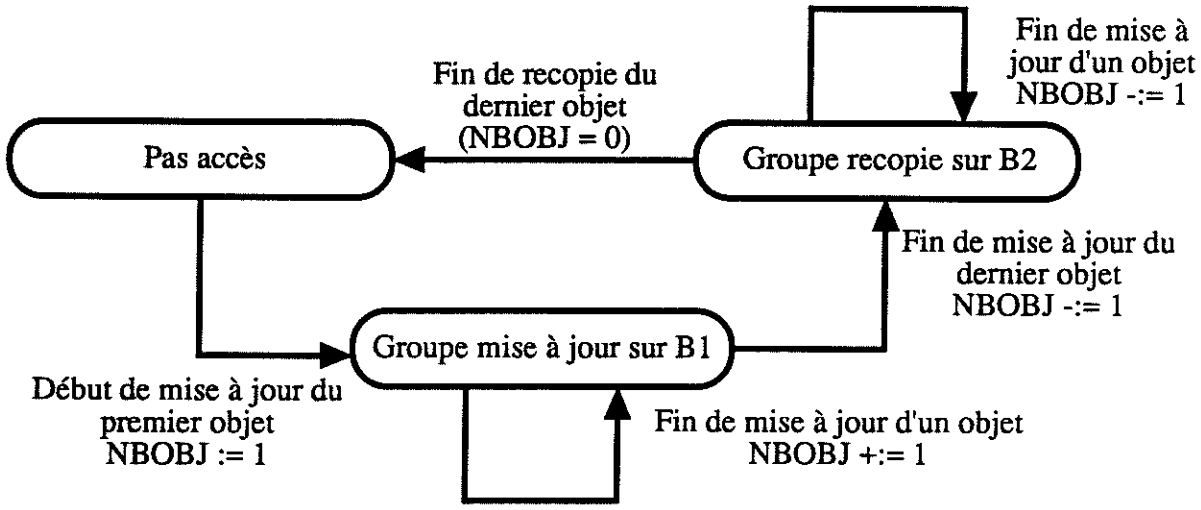


Figure A.2 Automate de l'opération *maj_atomique_groupe_objet*

La mise à jour d'un groupe d'objets est analogue à la mise à jour d'un objet. Pour vérifier que le nombre d'objets écrits sur le banc 1 est le même que sur le banc 2, un compteur *NOBJ* est incrémenté pendant la première phase puis décrémenté pendant la seconde. La taille de ce compteur entraîne une limitation sur le nombre d'objets stables pouvant être mis à jour en une seule opération atomique. La limite est de 64 objets dans la version actuelle de la mémoire stable.

Annexe B

Performance de la mémoire stable

Les résultats donnés dans le tableau ci-dessous sont extraits de [Mull88].

Nombre de mots de 32 bits	Lecture en microsecondes	Ecriture en microsecondes
1	5.4	9
2	3.4	5.8
3	2.7	4.7
5	2.2	3.8
9	1.8	3.2
100	1.4	2.5
1000	1.4	2.5

Figure B.1 Temps de lecture et d'écriture d'un mot en mémoire stable en fonction de la taille de l'objet

Annexe C

Nommage des ports stables

Les clients du servive RMC communiquent entre-eux par l'intermédiaire de ports stables. Pour communiquer avec un processus P_2, un processus P_1 doit connaître le nom interne d'un des ports stables de P_2. Un mécanisme de nommage s'avère donc nécessaire pour permettre la désignation des ports stables dans GOTHIC. L'association entre un nom symbolique et un nom interne est mémorisé dans un catalogue constitué de couple (nom interne de port stable, nom symbolique de port stable). Le catalogue est répliqué sur tous les sites. Sur chaque site, il en existe deux copies identiques, l'une en mémoire vive et l'autre en mémoire stable.

La copie en mémoire stable évite de reconstituer le catalogue en cas de panne. La copie en mémoire vive est utilisée pour toutes les consultations du catalogue.

Trois primitives permettent de consulter ou de mettre à jour le catalogue :

- *consultation*
  - **get_port ([in] port_nom, [out] port_id, [out] état)**
    La chaîne de caractères *port_nom* est recherchée dans le catalogue des ports stables. Si celle-ci est présente, le nom interne du port stable correspondant est rendu dans *port_id*. Dans le cas contraire, *état* indique que le nom ne se trouve pas dans le catalogue.
- *mise à jour*
  - **port_register ([in] port_id, [in] port_nom, [out] état)**
    Le couple (*port_nom*, *port_id*) est enregistré dans le catalogue. *état* indique une erreur si le catalogue est plein ou si la chaîne *port_nom* est déjà dans le catalogue. Dans ce dernier cas, l'ancienne correspondance est écrasée.
  - **port_unregister ([in] port_nom, [out] état)**

Si un élément du catalogue contient la chaîne *port_nom*, il est ôté. Dans le cas contraire, *état* indique l'absence de la chaîne de caractères *port_nom* dans le catalogue.

Les deux copies locales du catalogue sont mises à jour par ces deux primitives. En outre, une requête de mise à jour est envoyée à tous les sites du système. A la réception du message ceux-ci répercutent la mise à jour sur leur propre copie du catalogue.

Table des Matières

Table des Figures

Remote Multiprocedure Call Protocol in the Gothic System : Design and Implementation

The work described in this document is part of the GOTHIC INRIA/BULL project, which has been developed at IRISA and aims to build a fault tolerant distributed system for a local area network of multiprocessors. Two features of GOTHIC are the incorporation of stable storage boards in the architecture and the use of the *multiprocedure* concept which is the generalization of the procedure and is studied as a tool for structuring distributed systems.

We have defined protocols suited to the remote multiprocedure call. These protocols have been tested and implemented in the GOTHIC system.

In the first part of this document, we present the design and implementation of reliable communication protocols in distributed systems : remote procedure call (RPC) protocols, reliable broadcast protocols and protocols which generalize RPC by allowing parallel execution of the called procedure. The remote multiprocedure call protocol that we have developed as part of GOTHIC system, RMPC, is presented in the second part. The third part is devoted to the description of the GOTHIC reliable communication subsystem (including reliable message communication, atomic multicast and atomic ordered multicast protocols) on which the RMPC protocol relies. The originality of these protocols is the use of high speed stable storage to store communication subsystem data and checkpoints for communicating processes. These protocols have been tested. A description of the solutions to some implementation problems and the results on tests on the performance of the communication subsystem are presented in the concluding section of this part.

Key words :

Distributed system, stable storage, communication, reliable broadcast, remote procedure call, multiprocedure.

VU :

Le Président de la thèse

VU :

Le Directeur de thèse

VU et APPROUVE

RENNES, le 9.7.1990

Le Directeur de l'U.F.R.S.I.C.

D.T.U.R. 90/S/12 n° 179

VU pour autorisation de soutenance

RENNES, le - 7 DEC. 1990

Le Président de l'Université de RENNES I,

J.C. HARDOUIN

Résumé

Les travaux décrits dans ce document s'inscrivent dans le projet INRIA/BULL GOTHIC, développé à l'IRISA ayant pour ambition la réalisation d'un système distribué résistant aux pannes sur un réseau local de machines multiprocesseurs. L'originalité de l'architecture de GOTHIC réside dans l'association d'une mémoire particulière appelée *mémoire stable rapide* à chaque processeur. D'un point de vue langage, POLYGOTH, le langage de GOTHIC, offre le concept de *multiprocédure*, généralisation de la procédure qui est étudié comme outil de structuration des systèmes distribués.

Nos travaux ont consisté à définir les protocoles adaptés à l'appel de multiprocédure à distance, à les vérifier et à les mettre en œuvre dans le cadre de GOTHIC.

Dans la première partie de ce mémoire, nous présentons les nombreux travaux consacrés ces dernières années à l'étude et à la mise en œuvre de protocoles de communication fiable dans les systèmes distribués : protocoles d'appel de procédure à distance, protocoles de diffusion fiable et protocoles qui généralisent l'appel de procédure à distance en permettant l'exécution de la procédure appelée sur plusieurs sites en parallèle. Le protocole d'appel de multiprocédure à distance, appelé protocole RMPC, que nous avons développé dans le cadre du système GOTHIC est présenté dans la deuxième partie. La troisième partie est consacrée à la description des protocoles de communication fiable (communication fiable par message, diffusion atomique et diffusion atomique ordonnée) que nous avons mis en œuvre dans le système GOTHIC et sur lesquels s'appuie le protocole RMPC. L'originalité de ces protocoles tient à l'utilisation de mémoires stables rapides d'une part pour ranger les données du système de communication et d'autre part pour conserver des points de reprise des processus communicants. Le fonctionnement correct de ces protocoles en présence de pannes est justifié. En conclusion de cette partie, nous passons en revue les solutions apportées à quelques points délicats de mise en œuvre et donnons une analyse détaillée des performances mesurées.

Mots-clés :

Système distribué, mémoire stable, communication, diffusion fiable, appel de procédure à distance, multiprocédure.