Cours d'algorithmes distribués

Master M1 Année 2020-2021

Version 1.2

Université de Montpellier
Place Eugène Bataillon
34095 Montpellier Cedex 5

RODOLPHE GIROUDEAU
161, RUE ADA
34392 MONTPELLIER CEDEX 5
TEL : 04-67-41-85-40
MAIL : RGIROU@LIRMM.FR

Table des matières

Table des figures

Chapitre

1 Introduction et rappels

1.1 De l'algorithmique séquentielle à l'algorithmique répartie

Développer une méthode pour résoudre un problème, programmer un ordinateur pour la mettre en œuvre, tester et corriger un programme sont des activités courantes pour un informaticien. Ces méthodes, machines, tests peuvent être représentés par des *modèles séquentiels*. Un algorithme est représentable, par exemple, sous forme de machine de Turing. Chaque étape de calcul est bien définie et son résultat immédiatement accessible. Le nombre d'étapes de calculs est une mesure de la complexité en temps. Cet environnement séquentiel est plutôt bien connu maintenant (ce qui ne veut pas dire que tous les problèmes informatiques ont une solution simple, loin de là...).

L'étape naturelle qui vient juste après est de mettre en relation plusieurs machines. Cela est motivé par plusieurs facteurs :

- Scientifique : généralisation de l'algorithmique séquentielle. Qu'est ce que l'on gagne, qu'est ce que l'on perd à faire cela ?
- Pratique : dans certaines configurations de plus en plus nombreuses il est obligatoire de faire coopérer plusieurs machines distantes.

Toujours est il que l'algorithmique répartie existe et qu'il faut l'étudier pour bien maîtriser les concepts sous-jacents et éviter ses pièges. Dans ce qui suit nous allons aborder plusieurs points centraux.

1.1.1 Le temps

Lors de l'exécution d'un programme, une machine séquentielle passe par un certain nombre *d'états* successifs. Ces états peuvent être connus de manière précise en observant la machine. Cependant, lorsque plusieurs machines travaillent

en même temps, déterminer un *état global* pose des problèmes. Le premier problème est : qu'est ce qu'un état global ? est ce l'ensemble des *états locaux* ? si oui, à quel moment doit on prendre «en photo» ces états si les machines travaillent à des vitesses différentes ? (malgré le fait qu'elles peuvent coopérer pour atteindre un but global). Un autre problème, une fois qu'on a déterminé ce que devait être un état global est : comment l'observer ? Si notre système est composé de machines réagissant avec des délais variables, capturer un état global est non trivial car une information recueillie risque d'être périmée dès son arrivée, ou de ne pas être «compatible» avec celle d'une autre machine qui aura répondu plus vite par exemple. La situation idéale (de ce point de vue en tout cas) serait d'avoir un système complètement *synchrone* où chaque machine travaillerait à la même vitesse que toutes les autres. Excepté dans quelques cas bien précis, on ne peut pas avoir une telle synchronisation. Il faut donc faire avec un *asynchronisme* plus ou moins fort.

Tous ces éléments conduisent à se poser des questions sur la façon efficace de représenter un système. Cette représentation abstraite doit être suffisamment précise pour refléter la réalité mais aussi suffisamment simple pour être exploitable de manière théorique.

1.1.2 Les modèles

Pour représenter un système réparti il faut prendre en compte plusieurs éléments. L'ensemble de ces éléments constituera le *modèle* du système. On distingue :

Les sites. Ce sont toutes les machines qui constituent le système.

Les moyens de communication entre sites. Comment deux sites peuvent échanger de l'information pour coopérer ? On décrit alors le *réseau* de communication.

La synchronisation. Les sites sont ils ou non synchronisés ? Si oui jusqu'à quel point ? On peut par exemple connaître le temps de transit d'un message entre deux sites (modèle *faiblement synchrone*) ou n'avoir aucune idée de ce temps (modèle *asynchrone*).

Les pannes susceptibles de se produire dans le système. Cette connaissance est en général très difficile à avoir mais cependant nécessaire. En effet, dans un système composé de nombreuses machines, les pannes sont inévitables. Elles peuvent prendre plusieurs formes (pannes de liens et/ou de sites, définitives ou pas etc).

Dans ce cours, la plupart du temps nous adopterons un modèle du type suivant : n sites coopèrent et échangent des informations de manière asynchrone en s'envoyant

des messages à travers un réseau, plus ou moins fiable, souvent représenté par un graphe.

1.1.2.1 Les événements, le contrôle et l'écriture des algorithme répartis

Nous allons maintenant décrire les principales conventions que nous allons utiliser pour décrire nos algorithmes. De façon générale, un algorithme réparti utilisera les notations usuelles de l'algorithmique en *pseudo-code*. Nous pourrons utiliser des *ensembles*, des *tableaux*, des *variables*. Pour le contrôle nous utiliserons les tests, les boucles du type `si ... alors ... sinon`, `tant que .... faire`, `Pour tout .... faire ...` etc. Nous découperons aussi nos algorithmes en *procédures* et *fonctions*. Bref, nous utiliserons tout ce que nous pouvions déjà utiliser dans le cadre d'un cours d'algorithmique classique. A cela s'ajoute ici divers éléments qui sont propres au cadre «réparti».

— Les divers sites ne partagent *pas* de mémoire globale physique commune. Chaque site gère ses variables qui sont dites *locales aux sites*. Pour différencier les instances des variables sur tel ou tel site, nous mettrons en indice le nom du site auquel cette variable appartient. Par exemple, la variable L sur le site p sera notée L_p (ou L_i si le site est p_i).

— L'absence de mémoire commune implique des communications entre les sites pour *coopérer*. Dans ce cours nous noterons les messages entre $<>$. Chaque type de message pourra être désigné par un *nom de message* et pourra comporter (ou ne pas comporter) des *données*. Par exemple, on pourra avoir le message :

$$< JETON, 13, Tab >$$

Ici $JETON$ désigne le nom du message et $13, tab$ sont les données du message (tab pouvant être par exemple un tableau). NB : lorsque il n'y a qu'un type de message, nous n'utiliserons pas de nom.

— Lorsque l'on dispose de messages, il faut un mécanisme pour les *envoyer*. Pour décrire l'envoi d'un message, nous utiliserons la procédure primitive `Envoyer`. Pour utiliser `Envoyer` il faut préciser quel message est envoyé *et* à qui il est envoyé. Par exemple, si le site P_i doit envoyer un message de nom $JETON$ avec les données 13 et le tableau (local à P_i) tab_i au site P_j, il devra faire :

$$\texttt{Envoyer}(< JETON, 13, Tab_i >) \text{ à } P_j$$

— Il faut aussi bien sûr un mécanisme pour *recevoir* un message. La réception d'un message peut se produire, a priori, à n'importe quel moment. Nous supposerons ici que nous avons un mécanisme qui déclenche un *événement* qui

est la réception d'un message. Cet événement étant, en général, synonyme d'actions à accomplir, nous pourrons y associer une procédure de traitement du message, en fonction des données. Pour différencier les divers messages utilisés par un algorithme, nous nous baserons sur les noms des messages.

— Le mode de fonctionnement général d'un système réparti étant en général asynchrone, nous pourrons aussi prendre en compte d'autres événements. La commande `Attendre`($condition$) permet d'attendre que $condition$ soit vraie pour passer à l'instruction suivante. Ceci est particulièrement interessant à utiliser lorsque, sur un même site, plusieurs bouts de code sont exécutés en pseudo-parallélisme. Nous reviendrons sur cela dans les divers exemples et exercices.

1.1.3 Les mesures de complexité

Lorsque deux méthodes sont disponibles pour résoudre un problème, il faut un (ou des) critère(s) pour les comparer. Dans le domaine de l'algorithmique séquentielle c'est la complexité en temps et en espace mémoire qui font en général office de paramètres de comparaison. Dans le cas de l'algorithmique répartie, on peut toujours envisager ces mesures localement sur chaque site. Cependant, cela ne prend pas du tout en compte les échanges de messages entre sites. Or, il faut bien voir que pendant l'exécution d'un programme réparti, celui ci n'est en général pas le seul à tourner sur le système. D'autres tâches utilisent, en même temps les ressources du réseau. Lors du développement d'un algorithme réparti on doit donc faire en sorte que le *nombre de messages* échangés pendant son exécution soit le plus faible possible. C'est une mesure de «l'encombrement réseau» de l'algorithme. Les paramètres pouvant intervenir dans ce calcul de complexité peuvent être le nombre n de sites et le nombre de ressources mises en jeu. Le nombre de messages échangé peut être très dur à connaître de manière exacte. C'est pour cela que l'on parlera de complexité dans le pire, le meilleur des cas, de nombre moyen de messages etc.
Réduire le nombre de messages peut être possible en allongeant la *taille des messages échangés*. Cependant, il faut bien avoir à l'esprit que parfois la taille d'un message élémentaire échangeable sur un réseau ne peut pas dépasser certaines bornes. Au delà de cette taille les messages sont fragmentés ce qui conduit à augmenter le nombre de messages échangés. C'est à dire exactement le contraire de ce que l'on voulait faire ! La taille des messages sera aussi un critère à prendre en compte.
Ces deux paramètres quantitatifs seront les deux principaux considérés ici pour évaluer et comparer les algorithmes proposés. Bien sûr, dans le cas précis d'une mise en oeuvre particulière, avec un système particulier et un réseau dédié, il

conviendrait de faire encore beaucoup de travail d'affinage et de déploiement. Cette tâche spécialisée est en général le travail de l'ingénieur système et/ou réseau qui connaît très bien l'environnement dans lequel il travaille, avec toutes ses contraintes propres et ses outils dédiés.

1.1.4 Exemple : mise à jour d'une valeur partagée

Dans un système à n sites V, on veut construire un algorithme de *diffusion* (envoi d'un même message d'un site vers tous les autres). Pour cela on peut écrire la procédure suivante valable pour un site p quelconque :

```
Procédure diffuser(M)
    Pour (x_p ∈ V − {p}) Envoyer(< M >) à x_p;
```

Cette procédure est très simple. Elle consiste à envoyer le message vers tous les autres sites, dans n'importe quel ordre. Les algorithmes de diffusion sont utilisés pour de nombreuses tâches. Par exemple la mise à jour de données partagées (ou de caches). Prenons le cas où n sites d'un système doivent partager une valeur commune. Pour cela, ils maintiennent tous localement une copie de cette valeur qu'ils peuvent donc lire facilement, sans passer par le réseau. Cependant, lorsqu'un site décide de changer cette valeur, on propose de diffuser la nouvelle valeur pour que tous les autres sites la changent localement. Les procédures sont décrites pour un site p quelconque.
La procédure suivante est utilisée par un site qui veut écrire une nouvelle valeur globale. Pour cela il diffuse la nouvelle valeur.

```
Procédure diffuser_valeur(v)
    Pour (x_p ∈ V − {p}) Envoyer(< ECRITURE, v >) à x_p;
```

Ainsi, lors de la réception d'un message ayant pour nom $ECRITURE$, le site récepteur va changer sa variable locale C_p correspondant à la connaissance qu'il a de la valeur globale partagée.

```
Lors de la réception de < ECRITURE, v > depuis q
    C_p := v;
```

Ainsi, on peut dire ici que cet algorithme a une complexité de $n-1$ messages pour *chaque modification* de la valeur globale.
Inconvénients de cet algorithme : très nombreux !

— Supposons qu'un message soit perdu. Dans ce cas, $n-1$ sites auront la bonne valeur mais pas le dernier. Cette valeur globale n'est plus universelle.
— Les sites étant indépendants, on peut avoir deux mises à jour parallèles. Les mises à jour locales peuvent donc être faites dans n'importe quel ordre. Par exemple, les deux sites a et b diffusent en même temps. Le site c reçoit d'abord le message de a puis celui de b. Le contraire pour un autre site d. Ainsi c a la valeur de a mais d a la valeur de b. Qui a la bonne valeur ?

On voit ici que la première situation est liée à un problème *d'atomicité* (l'opération est terminée pour certains sites mais pas pour le dernier). La deuxième situation est plus complexe. Avoir une valeur globale partagée n'est en fin de compte pas si simple. De plus, dans l'exemple donné, quels sont les bons critères pour décider que c'est tel ou tel site qui a écrit le premier ? Nous verrons plus loin la notion de *causalité* qui permet de donner des solutions.

1.2 Des systèmes d'exploitation centralisés aux systèmes répartis

Seuls les concepteurs des tout premiers ordinateurs, étaient capables de les programmer et de les utiliser. Tout devait être géré «à la main». Depuis, les *systèmes d'exploitation* permettent de faire les choses suivantes.

— Séparer la conception des machines de leur utilisation.
— Offrir une plate forme stable au travers de laquelle on peut programmer sans avoir à ce soucier de gérer les entrées/sorties dans tous les détails.
— Partager les ressources avec d'autres utilisateurs.
— Donner à chaque utilisateur l'impression qu'il est le seul à utiliser la machine (machine virtuelle) tout en pouvant lancer plusieurs tâches en même temps : systèmes multi-tâches et multi-utilisateurs.

Ainsi, un gros système permet de faire travailler simultanément plusieurs utilisateurs qui font confiance à leur système d'exploitation pour gérer les aspects matériels de leurs programmes. Cette facilité a conduit à ce qu'il y ait de plus en plus d'utilisateurs de plus en plus exigeants connectés à ce système devant supporter de plus en plus de logiciels de plus en plus gourmands en mémoire et temps CPU. Un seul gros ordinateur ne pouvait plus supporter tout ça. Avec l'avènement des ordinateurs personnels chaque utilisateur (ou chaque petit groupe d'utilisateurs) pouvait avoir le sien, avec sa version des logiciels. Ceci conduit à multiplier les ressources disponibles et à séparer les utilisateurs. Le deuxième point important dans cette évolution fut l'arrivée des réseaux et la possibilité de connecter plusieurs machines (ordinateurs et périphériques) sur un réseau (local). Il était alors possible de faire un réseau de «petites» machines au lieu d'avoir un seul gros sys-

tème. Grâce au réseau, les utilisateurs peuvent continuer à échanger des informations et à partager des ressources trop coûteuses pour être disponibles sur chaque poste. Ce processus de «downsizing» permet de donner à chaque utilisateur une machine réelle, sur laquelle il peut utiliser ses propres logiciels, et aussi avoir accès aux ressources communes. Cette organisation pose malgré tout le problème de l'hétérogénéité du parc : si chaque utilisateur a sa propre machine et ses propres logiciels au bout d'un certain temps rien n'est plus compatible avec rien. Le parc devient donc une mosaïque de machines indépendantes reliées par un réseau. Pour tenter d'avoir une vue unifiée et rationnelle des choses les systèmes d'exploitation doivent devenir répartis. On doit rajouter les fonctions suivantes.

— Le système d'exploitation doit donner à l'utilisateur la possibilité d'utiliser toutes les ressources (autorisées) disponibles dans le système, même si celles ci ne sont pas disponibles sur la machines physique sur laquelle il est. Les traitements peuvent être effectués sur des machines distantes sans que l'utilisateur en ait forcement conscience.
— Le système doit être robuste : en cas de pannes (mineures) le système ne doit pas être bloqué.

Ainsi, les systèmes centralisés ont été amenés à évoluer pour prendre en compte et exploiter au mieux les possibilités réseaux. Cette évolution est nécessaire (les gros systèmes centralisés ne sont plus adaptés à tous les besoins) mais entraîne des difficultés de conception et de réalisation.

1.3 Quelques rappels généraux de système d'exploitation et de réseaux

Nous rappelons ici très brièvement les principales caractéristiques des systèmes d'exploitation que l'on peut utiliser pour les enrichir.

Un élément de base est le *processus*. Il appartient à un utilisateur (ou au système), a un espace mémoire réservé, et il est exécuté de temps en temps par le processeur. Un processus peut se dupliquer. Plusieurs processus de plusieurs utilisateurs peuvent coexister et s'exécuter en mode *pseudo parallèle*. Ceci pose des problèmes de *synchronisation* (par exemple pour l'accès à une ressource partagée). Il existe pour cela des mécanismes comme les *sémaphores* ou les *moniteurs*. Les processeurs peuvent communiquer grâce à des mécanismes comme les *pipes* ou des segments de *mémoire partagée*.
Tous ces mécanismes sont disponibles sur de nombreuses machines. Cependant ils permettent juste de faire communiquer des processus ce trouvant sur la même machine. Comment faire communiquer des processus qui s'exécutent sur des

machines physiques différentes ?

Pour cela on va utiliser un *réseau* qui interconnecte toutes les machines d'un parc. Il existe plusieurs types de réseau, plus ou moins étendus : des *réseaux locaux* (longueurs de moins de un Km), des réseaux *métropolitains* (à l'échelle d'un campus ou d'une ville), des *réseaux étendus* (à la taille d'un pays ou de la planète). Il y a aussi plusieurs sortes *d'interconnexions* possibles : des *réseaux à diffusions* ou des liens point à point permettant de relier deux sites (on parle alors de réseaux maillés). Pour ces types d'interconnexion plusieurs *technologies physiques* sont disponibles : câbles (réseaux à bus de type ethernet), ondes radio (réseaux locaux sans fil, satellites), liaisons optiques (sur fibres optiques). Des protocoles de communication adaptés pour chaque type existent. Ces réseaux ont des caractéristiques de *débit et de latence* propres. Les protocoles réseaux sont découpés en couches fonctionnelles, ce qui permet de les rendre (en principe) indépendants de la couche physique.

1.4 Les services

La numérisation de tous les formats de données (sons, images fixes et animées, textes) conduit à concevoir des réseaux susceptibles de transporter tous ces médias dans de «bonnes conditions». De nouveaux produits et *services* sont apparus. Parmi ceux là on peu citer :

- Vidéo conférence interactive.
- Vidéo à la demande.
- Jeux interactifs sur réseau.
- Travaux de groupe à distance (applications et données partagées).
- Applications réparties et répartition des charges de calculs sur des machines distantes (météo par exemple).

Du coup, un enjeu important est de fournir aux utilisateurs de ces services des *garanties de qualité de service* (QoS) malgré des codages de l'information différents, des sous-réseaux et des systèmes hétérogènes... Un réseau pouvant supporter du trafic de type *multimédia* doit être très «performant». A titre d'illustration, on peut citer les diverses QoS suivantes (pour simplifier nous nommerons *paquet* tout ce qui peut être trame, cellule, paquet etc.) :

- *Séquencement*. Les paquets doivent (le plus possible) arriver dans l'ordre dans lequel ils ont été émis. Ceci réduit la taille des buffers de reséquencement et permet de délivrer les paquets beaucoup plus rapidement à l'application.

— *Gigue.* Le temps d'arrivée entre paquets doit être le plus faible et le plus constant possible.
— *Contraintes temps réels*. Pour certaines applications, le temps entre la création d'un paquet et sa réception par l'application utilisatrice doit pouvoir être évalué et respecté.
— *Débit*. Certaines applications doivent pouvoir bénéficier d'un débit stable et négocié. D'autres doivent avoir des garanties sur le débit moyen, crête, etc.
— *Synchronisation inter-flux*. Plusieurs flux partant d'une même source doivent arriver à destination en même temps (par exemple synchronisation entre l'image et le son d'une même séquence vidéo).
— *Synchronisation multi-points*. Mêmes problèmes que précédemment mais avec plusieurs récepteurs. C'est-à-dire que non seulement les divers flux doivent être synchronisés entre eux en chaque récepteur mais en plus, tous les récepteurs reçoivent les mêmes flux dans la même fenêtre de temps (idéalement en même temps).
— *Acheminement d'événements (urgents) ; paquets prioritaires*. Contrairement à la situation précédente, un événement urgent prend souvent la forme d'un paquet court mais très prioritaire.
— *Temps d'établissement*. Ce point n'est pas forcement le plus pénalisant. Un utilisateur peut attendre plusieurs secondes avant d'accéder à un service (mais lorsque la connexion est faite, celle ci doit respecter les garanties du service).
— *Taux d'erreur* (par bit, paquet, etc). Ceci est lié à la pureté du support physique et des divers intermédiaires.
— ...

D'autres facteurs doivent aussi être pris en compte comme la tolérance aux pannes ou le coût d'une communication. Tous ces nouveaux services tendent à rapprocher l'informatique et les télécommunications.

1.5 Le Global Computing

On parle de plus en plus de *Global Computing* à l'echelle mondiale. La constatation de départ de ces systèmes est qu'un ordinateur passe plus de temps à ne rien faire qu'à travailler (la nuit, lors des pauses etc). Cette quantité de CPU disponible dans le monde entier est énorme. Elle pourrait être utilisée pour de gros calculs industriels ou scientifiques. L'idée qu'ont eu plusieurs organismes (privés et publics) est de récolter des adresses de volontaires qui prêtent (à distance) leur machine lorsqu'elle ne sont pas utilisées. Ces organismes (qui vendent ou qui prêtent cette

CPU) peuvent alors lancer les travaux de leurs clients) distance sur les machines disponibles.
Le cas le plus populaire est sans doute l'action *seti@home* qui a besoin de puissance de calcul pour filtrer les signaux reçus de l'espace et tenter de détecter des émissions cohérentes, signe potentiel de vie extra-terrestre. Lors du lancement de cette initiative, des milliers de volontaires ont spontanément proposé leur CPU inutilisée pour faire tourner le (un bout du) programme de filtre.
Ce type d'initiative se retrouve surtout dans les domaines comme la physique des hautes énergies, la biologie, la météo etc.
Les principaux problèmes pour utiliser toute cette CPU potentielle sont les suivants.

- Performance des communications entre le serveur principal et les divers clients. Ceci induit un certain niveau minimal de granularité du parallélisme des opérations au dessous duquel le système n'est pas du tout efficace à cause des temps de communications.
- Pour utiliser certaines machines, il faut passer des barrières de sécurité ce qui complique le système (rappelez vous qu'il y a des milliers de machines).
- Avec toutes ces machines réparties dans le monde entier, l'arrivée d'une panne n'est pas probable mais certaine. Il faut donc prendre des mesures pour que le système puisse tolérer des pannes.
- Forte hétérogénéité du parc des machines.

1.6 Les mobiles

Un autre facteur qui tend à prendre de l'importance est la *mobilité* des utilisateurs. Ceux-ci utilisent par exemple des ordinateurs portables et souhaitent être connectés de temps en temps à leur base (entreprise par exemple) pour consulter leur courrier électronique ou une base de données interne ou pour faire tout autre travail distant. Cette connexion peut être assurée via une ligne téléphonique. Les problèmes qui se posent alors sont de plusieurs types :

- Performance : la liaison est bas débit et les temps de latence sont importants.
- Sécurité : la liaison doit être sécurisée pour éviter que les données en transit ne soient lues. Il faut aussi assurer qu'un intrus ne puisse pas pénétrer le système facilement.

Les problèmes de performances sont encore plus visibles si le terminal de l'utilisateur est un téléphone portable par exemple. Dans ce cas tous les goulots d'étranglement sont présents (terminal peu puissant, limité en autonomie, écran peu confortable, débit faible). Il faut en plus prendre en compte les risques de coupures. Heu-

reusement, une nouvelle génération de «mobiles» arrive : UMTS. On devrait alors pouvoir bénéficier du haut-débit sans fil.

1.7 Le parallélisme

Les systèmes répartis sont essentiellement parallèles et concurrents. Parallèles car plusieurs sites travaillent en même temps et concurrents car ils ont besoin de partager des ressources communes. Ici le parallélisme est (plus ou moins) imposé par le système déjà mis en place, il faut faire avec. Il faut faire en sorte que tout ce passe au mieux, avec des demandes et des *événements* qui peuvent arriver à n'importe quel moment. Les systèmes répartis sont *asynchrones* (il n'y a pas d'horloge globale) et travaillent avec des informations qui sont en général locales. Ces systèmes sont dits *faiblement couplés*.

Cette situation est à comparer aux systèmes parallèles qui sont construits pour effectuer des tâches bien précises dans un environnement donné (en général fortement couplés). Le niveau de *contrôle* du système et les finalités sont donc très différents.

1.8 La communication sous UNIX

1.8.1 UNIX, IP et les sockets

Cette section a pour but de présenter de manière légère quelques éléments de communications entre machines distantes (pour plus de détails, reportez vous aux livres [8, 9]). Nous avons choisi UNIX et TCP/IP car les deux sont très répandus et encore utilisés. Il existe beaucoup d'autres possibilités de plus ou moins haut niveau.

Un élément de base d'UNIX est le *processus*. Plusieurs processus peuvent s'exécuter en même temps (pseudo-parallélisme) sur une même machine. Ces processus peuvent communiquer par des mécanismes propres (pipes, mémoire partagée etc). Cependant, comment faire lorsque les processus d'une application répartie sont sur plusieurs machines, géographiquement dispersées ? La communication sous UNIX est fortement liée (mais pas uniquement) à IP qui est un protocole de communications maintenant largement répandu. Sur IP, reposent deux protocoles de plus haut niveau : TCP et UDP. UDP est un mode *datagramme*, sans aucune garantie de service de bout en bout. On le compare traditionnellement aux services de la poste : on met une lettre dans un boite, qui arrivera peut être. TCP par

contre est orienté connexion. Pour cela il met en place tout un mécanisme pour s'assurer que chaque paquet est bien arrivé, en renvoyant éventuellement des paquets perdus. C'est un mécanisme plus fiable mais qui utilise plus de ressources que UDP.

Nous allons décrire ici quelques éléments pour faire communiquer plusieurs processus UNIX via TCP/IP (des choses similaires existent pour UDP) : les *sockets*. Ce mot veut dire «prise» en anglais et il y a effectivement des analogies à faire avec le réseau électrique. Pour faire communiquer deux processus par une socket il faut faire certaines opérations préliminaires. Le schéma de ces opérations est donné à la figure 1.1.

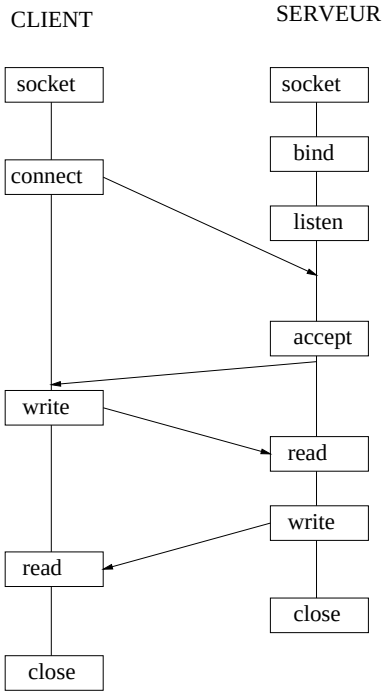

FIGURE 1.1 – Schéma de création et d'utilisation d'une socket

Les diverses opérations sont disponibles dans des librairies spécialisées. La création du «canal» de communication est faite sur le mode client/serveur. Un serveur est à l'écoute sur un de ses ports. Un client (qui connaît ce port) va faire une connexion avec le serveur par ce port. Ensuite, les deux processus peuvent com-

muniquer dans les deux sens. A la fin on ferme le canal de communication. Les paramètres de ces diverses fonctions sont les suivants.

```
int socket(domaine,type,protocole)

        int domaine;   /* AF_INET, AF_UNIX ... intra ou extra UNIX*/
        int type;      /* SOCK_DGRAM, SOCK_STREAM ... connexion ou pas*/
        int protocole; /* 0 : defaut */
```

Cela permet au client de définir une prise par laquelle il va communiquer.

```
int connect (sock, addr, lg)

        int sock;                    /* descripteur local de socket */
        struct sockaddr_in * addr;   /* pointeur sur l'adresse de la socket c
        int lg;                      /* longueur de l'adresse */
```

Cela permet au client d'associer à sa prise un «tuyau de communication» jusqu'au serveur.

```
int bind(sock,addr,lg)

        int sock;                   /* descripteur de socket */
        struct sockaddr_in * addr; /* pointeur sur l'adresse de la socket */
        int lg;                     /* longueur de l'adresse */
```

Du coté du serveur cela sert à mettre en correspondance une socket avec une adresse.

```
int listen(sock,nb)

        int sock; /* descripteur de socket */
        int nb;   /* nombre max de demandes de connexions en attente dans la
```

Le serveur se met a l'écoute des demandes de connexions.
Pour lui permettre d'extraire de la file une connexion on utilise la fonction :

```
int accept(sock,addr,lg)

        int sock;                   /* descripteur de socket */
        struct sockaddr_in * addr; /* pointeur sur l'adresse de la socket */
        int * lg;                   /* pointeur sur la taille de la zone de a
```

Le résultat renvoyé par cette fonction est le DESCRIPTEUR de la socket que le client et le serveur vont utiliser pour communiquer dans les deux sens en utilisant `write` et `read`.

```
int write(sock,message,lg)

        int sock;       /* descripteur local de socket */
        char * message; /* adresse en mémoire du message a envoyer *
        int lg;         /* longueur du message */

int read(sock,message,lg)

        int sock;       /* descripteur local de socket */
        char * message; /* adresse en mémoire de sauvegarde du messa
        int lg;         /* longueur de la zone allouée pour le messa
```

1.8.2 RPC

RPC (*remote procedure call*) est un mécanisme pour exécuter une procédure sur une machine distante. Cela a été introduit par *SUN* et a été utilisé pour écrire le système de fichiers NFS (par SUN aussi). La figure 1.2 illustre le fonctionnement général d'un RPC.

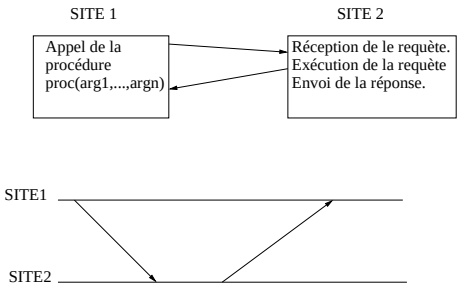

FIGURE 1.2 – Schéma d'utilisation de RPC

Les RPC sont basées sur les sockets basées elles mêmes sur UDP, basé sur IP. C'est donc déjà un mécanisme d'un certain niveau. Cependant, pour mettre en place un RPC faut il tout gérer «à la main» ? la réponse est heureusement non, il existe des outils (RPCGEN) qui permettent de générer tout le mécanisme à partir de descriptions de plus haut niveau.

La transmission des paramètres de la fonction à exécuter sur la machine distante peut être problématique. Pour faire des choses homogènes, on utilise une représentation standardisée des données : XDR (*eXchange Data Representation*). En plus de tout ce qui est décrit précédemment il faut donc rajouter le codage et le décodage XDR.

Le serveur (celui qui va exécuter la procédure) doit être en attente continuelle qu'un service lui soit demandé. Lorsqu'une requête lui arrive, il peut la traiter lui même ou créer un fils (avec un `fork`) qui va la traiter et retourner le résultat à l'appelant. Si le serveur est multi-processeurs, ce fils peut être créé sur un autre processeur (on fait alors du vrai parallélisme). Avec ce mécanisme le serveur peut se remettre en attente beaucoup plus tôt.

1.9 Les nouvelles technologies pour les systèmes répartis

Une réponse aux nouveaux problèmes induits par la répartition des traitements peut être d'enrichir l'existant pour l'adapter. On peut aussi imaginer de nouveaux systèmes dans lesquels les communications et la répartition sont prises en compte au niveau le plus bas. Plusieurs projets universitaires et industriels ont vu le jour sur ces thèmes (mach, Amoeba, Chorus ...). De nouveaux noyaux systèmes ont été développés pour créer des plate-formes robustes, efficaces et offrant une grande transparence vis-à-vis de la localisation physique des ressources (aussi bien matérielles que logicielles). Ces micros noyaux ont été écrits pour pouvoir supporter des applications UNIX. Ils incorporent quelques fois des caractéristiques temps réels.

A une plus grande échelle (middleware), l'écriture d'applications réparties pose des problèmes aux programmeurs du fait du matériel et des logiciels hétérogènes. Ces applications réparties doivent aussi prendre en compte et utiliser les applications déjà développées en interne dans de nombreux langages de programmation différents. Une bonne application répartie devra vérifier les quelques points suivants.

- Intégration des supports logiciels et matériels hétérogènes existants.
- Facilités de communication entre les divers composants de l'application répartie.
- Facilités de mises à jour et d'améliorations de l'application répartie.

De nombreuses réflexions (encore en cours) ont conduit à proposer une plate-forme pour la création et le maintien d'applications réparties. Un résultat de ces réflexions a donné *CORBA* (voir [6] pour une bonne introduction à CORBA). C'est une spé-

cification de ce que devrait être une telle plate-forme et à quoi elle devrait répondre pour résoudre les problèmes décrits plus haut. Des travaux sont encore en cours pour finir d'écrire cette spécification. Malgré tout, des produits commerciaux vérifiant cette norme sont déjà vendus.

1.10 Résumé

Les systèmes d'exploitation traditionnels (centralisés) sont adaptés pour de petites unités de travail (comme un bureau par exemple ou un atelier). Cependant on assiste à :

- Un éclatement de ces unités en sous unités fonctionnelles (éventuellement localisées sur un vaste territoire).
- Une demande croissante de puissance, de logiciels nouveaux et gourmands en mémoire et temps CPU.
- L'apparition de ressources coûteuses (jetons d'utilisation d'un logiciel, imprimantes ...), et/ou complexe à mettre en œuvre. Il faut donc les partager.

Le fédérateur dans ce cas là est un **système d'exploitation réparti** qui donne à l'utilisateur final l'impression que toutes les ressources (autorisées) sont pour lui (même celles qui sont distantes). La mise en oeuvre d'un système réparti est délicate. Quelques-uns des problèmes fondamentaux sont (ils ne sont pas nécessairement disjoints) :

- Rien n'est centralisé. Les prises de décisions peuvent être complexes. Les **sites** communiquent (en général) par **envois de messages**. La durée d'acheminement de ces messages n'est pas (en général) prévisible (**asynchronisme**).
- **Partage** (efficace) de ressources.
- Aucun **état et/ou temps global** du système n'est facilement **observable** (temps réel, **terminaison** ?). «Faire quelque chose» de **global avec des connaissances locales**.
- **Cohérence** des transactions.
- Sûreté de fonctionnement : **tolérance aux pannes** et **confidentialité**.

Du point de vue algorithmique on se pose en général les questions suivantes pour définir le **modèle** que l'on considère.

- Comment est le **réseau de communications** (bus, point-à-point, liens FIFO ...) ?
- Durée d'acheminement des messages (borne supérieure ou inconnue) ?
- Que sait chaque site (nombre de sites, topologie du réseau, **voisins**, «réveil»...) ?

— Niveau et nature des pannes à prendre en compte (crashs, pannes malicieuses, liens et/ou site...) ?

Des solutions algorithmiques (plus ou moins partielles) existent. Pour pouvoir les comparer il faut donner une notion de **complexité**. Celle ci est en général exprimée en termes de :

— Nombre de messages échangés pour faire telle ou telle opération.
— Quantité d'information dans les messages.
— Temps de réponse (parfois délicat à définir précisément).
— Nombre maximal d'opérations à réaliser sur chaque site.
— Une autre mesure de complexité pourrait être le niveau de connaissance locale exigé pour réaliser l'opération (problème de la **généralité de la solution**, modèle).

Outre la *faisabilité* effective des solutions algorithmiques proposées, d'autres problèmes plus techniques viennent se greffer.

— Hétérogénéité des machines, voire des OS et/ou des sous réseaux.
— Intégration de logiciels et de matériels nouveaux (up-grade).
— «scalabilité»...

Chapitre

2 Les communications

2.1 Introduction

Dans ce chapitre nous allons étudier quelques protocoles de communications. En particulier, nous détaillerons la *diffusion*. C'est l'opération qui consiste pour un site donné à envoyer un même message vers tous les autres sites du système. Elle est très largement utilisée comme primitive d'autres opérations comme des mises à jour de caches, envoi de résultats partiels lors de calculs répartis, envoi d'informations aux utilisateurs etc. Il est donc important qu'elle vérifie certaines propriétés que nous verrons dans les sections suivantes.

Un protocole de diffusion sert à fournir à une *application* un service fiable. D'un autre coté, ce protocole gère les réceptions et envois de messages avec le réseau qui peut être considéré comme non fiable. Ainsi, il sert de filtre entre le réseau et l'application. Lorsque le protocole est sûr qu'un message donné est bon à donner à l'application correspondante, il utilise la primitive `Délivrer` pour lui faire passer les informations. Ainsi, il peut y avoir un décalage entre le moment où un site reçoit un message et où l'application qui tourne sur ce site prend en compte ce message. Cette situation est résumée à la figure 2.1.

2.1.1 Hypothèse sur les communications

Soient P_i et P_j deux sites qui échangent des informations à une fréquence de $1\,Ghz$.

`Sur le site` P_i
$envoyer(x, P_j)$
$receive(y)$
$x = x + y$

`Sur le site` P_j

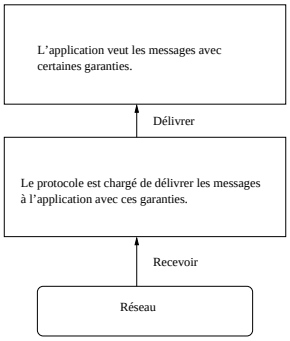

FIGURE 2.1 – Schéma des couches pour des diffusions avec garanties

$receive(x)$ $v = 2 * x + 1$
$envoyer(v, P_i)$

Sachant que $1Ghz = 10^{-9}$ s et la vitesse de la lumière $c = 3.10^8$ m/s, la distance ntre P_i et P_j doit donc être (pour ne pas interférer sur la vitesse de calcul) d'au plus : $dist(P_i, P_j) > 10^{-9}\ 3.10^8 = 0,3\ m$.
Cette distance n'est pas évidemment pas respectable dasn un vrai système distribué, ce d'autant plus qu'il s'agit ici de suppositions optimales (dans les composants électroniques, la vitesse des données n'atteind pas la vitesse de la lumièr). La plupart du temps, on considéra donc que le temps de calcul local est négligeable devant le temps de communications.

2.2 Diffusion asynchrone en cas de pannes de sites

Nous décrivons ici un algorithme de *diffusion*, à partir d'un site p donné, dans un réseau physique complet dans lequel des sites peuvent tomber en panne. Soit V l'ensemble de tous les sites et $S_0 = \{q_1, q_2, \ldots, q_t\}$ un sous ensemble de sites, $S_0 \subset V$. Ce sous-ensemble de sites S_0 est un ensemble de relais.

La procédure suivante est décrite pour p avec $p \notin S_0$.

```
Procédure diffuser(M)
    Envoyer(< M >) à q1, q2, ..., qt dans cet ordre;
    Pour tout (q ∈ V − S0 − {p}) faire
        Envoyer(< M >) à q;
```

La procédure suivante est décrite pour tout site q.

```
Lors de la réception de < M >
```

```
Si (q ∈ S0) alors /* Supposons que q = qk */
    Si (k < t) alors
        Envoyer(< M >) à qk+1,...,qt dans cet ordre;
    Pour tout (r ∈ V − S0 − {p}) faire
        Envoyer(< M >) à r;
    Accepter(M);
Sinon Accepter(M);
```

Un exemple d'exécution dans le cas où le site p de la figure 2.2 diffuse un message (dans ce cas, $S_0 = \{q_1, q_2, q_3\}$).

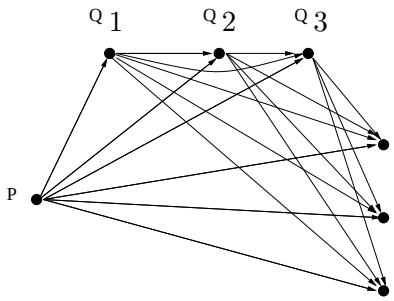

FIGURE 2.2 – Lorsque p diffuse.

On voit dans cet exemple qu'un site peut recevoir plusieurs fois le message diffusé par p. Pour simplifier on supposera ici que la fonction `Accepter` ne délivre à l'application qu'un seul exemplaire de chaque message diffusé par p (pour mettre cela en oeuvre, p peut par exemple numéroter ses messages).
Ici nous appelons panne d'un site, l'arrêt soudain de ce site. Nous supposons que lorsqu'il tombe en panne il reste en panne et ne fait plus rien.
Comment se déroule l'algorithme lorsqu'il y a une panne pendant l'algorithme ? Lorsqu'il y a s pannes ? Que peut garantir l'algorithme ?

Une panne peut arriver à n'importe quel moment pendant l'algorithme. Lorsqu'il n'y a qu'une seule panne :

— Si p tombe en panne. Si p a le temps d'envoyer au moins un message, tous les autres sites le recevront sinon personne ne le reçoit.

— Si c'est un site q qui tombe en panne tous les sites vont recevoir le message.

Dans tous les cas de figure lorsqu'un seul site tombe en panne *soit tous les sites valides acceptent le message soit aucun site ne l'accepte*. C'est le principe du tout

ou rien.
On peut généraliser cette propriété jusqu'à t sites, mais pas plus.

Lemme 2.2.1 *Le protocole de diffusion proposé ne peut pas vérifier le principe du tout ou rien avec la présence de plus de t pannes*

Preuve Pour montrer ce lemme, examinons la situation suivante.
p envoie son message à q_1 et tombe en panne juste après. q_1 envoie son message à q_2 et tombe en panne juste après. q_2 envoie son message à q_3 et tombe en panne juste après.... q_{t-1} envoie son message à q_t et tombe en panne juste après. q_t envoie son message à un sommet r de $V - S_0 - \{p\}$ et tombe en panne.
D'après l'algorithme, r accepte le message et ne fait plus rien. Ainsi, avec ces $t+1$ pannes, il y a un site valide qui a accepté le message mais pas les autres. Le principe du tout ou rien n'est pas respecté. □

Combien de messages sont échangés pendant tout l'algorithme lorsqu'il n'y a pas de panne ?
Le site p envoie $n-1$ messages. Chaque site q_i envoie $t-i+n-(t+1) = n-i-1$ messages. Les sites de $V - S_0 - \{p\}$ n'envoient pas de message. Il y a donc en tout $n-1+\sum_{i=1}^{t} n-i-1 = n-1+t(n-1)-\frac{t(t+1)}{2} = (t+1)(n-1-t/2)$ messages.

2.3 Diffusion respectant l'ordre FIFO des messages

Soit un réseau asynchrone à n sites. On supposera que les communications entre deux sites ne vérifient pas l'ordre FIFO et que le temps d'acheminement des messages entre deux sites est quelconque mais fini (le réseau n'est donc pas complètement fiable mais les sites le sont). Considérons le cas où un site particulier p fait des diffusions. On veut mettre en place un algorithme dans lequel tous les messages diffusés par p seront *délivrés* à toutes les autres applications dans le même ordre que l'ordre d'envoi à partir de p. On veut donc créer un protocole de diffusion respectant l'ordre FIFO.
Pour cela, p va numéroter chaque message créé et va diffuser ce message avec ce numéro, dit *numéro d'envoi ou numéro de séquence*. Le site p a une variable locale num_envoi_p : numéro du dernier message diffusé par p. Tous les autres sites i ont une variable d'attente dont la valeur doit correspondre au numéro d'envoi pour que le message soit délivré.

Code du site p :
Au début, $num\_envoi\_p := 0$;
`Procédure` **`diffuser(`**M**`)`**

$num\_envoi_p := num\_envoi_p + 1$;
Pour tout ($x_p \in V - \{p\}$) faire Envoyer($< M, num\_envoi_p >$) à x_p;

Code du site i :
Au début, $seq_i := 1$;
Lors de la réception de $< M, num\_envoi_M >$ de p
Stocker(M);
Attendre($num\_envoi_M = seq_i$);
Délivrer(M);
$seq_i := seq_i + 1$;
Détruire(M);

Inconvénients du protocole :

- Un site i peut avoir à stocker beaucoup de messages avant de pouvoir les délivrer et de pouvoir les détruire (par exemple si un des messages est très lent par rapport aux suivants et arrive bien après eux).
- Le numéro de séquence des messages croît au delà de toute limite raisonnable si p diffuse beaucoup de messages. C'est un problème de taille de messages.
- Ici on utilise explicitement le fait que le réseau ne perd pas de message. Dans le cas contraire, le protocole peut être bloqué et ne plus délivrer de messages.

Pour résoudre les deux premiers points on peut mettre en place un système *d'acquittements dans une fenêtre de taille t fixée*. Dans ce système, le site p fait au plus t diffusions de suite avant de recevoir des acquittements. Chaque fois qu'un site i a pu délivrer un message, il envoie un acquittement à p, comprenant le numéro du message acquitté. Lorsque p a reçu les acquittements de tous les sites, pour les derniers messages envoyés non encore acquittés, il peut continuer à diffuser. Dans ce protocole, les numéros de séquence des messages diffusés et les numéros de séquence en réception sont construits modulo t, en commençant à 0. Du coup, les numéros de séquence ne dépassent pas une certaine valeur et le deuxième problème est résolu. Chaque site n'aura à stocker qu'au plus t messages avant de les détruire et si t est suffisamment petit, le premier problème est résolu.

2.4 Diffusion respectant l'ordre causal

2.4.1 L'ordre causal

Dans les systèmes répartis sans mémoire physique commune, tous les échanges entre les sites sont faits par passages de messages. Pour pouvoir avoir un comportement global correct on aimerait pouvoir ordonner les envois et réceptions de messages. Ceci est possible sur n'importe quel site qui se comporte comme une machine séquentielle. Cependant, lorsqu'on regarde au niveau du réseau dans sa globalité il n'y a plus d'ordre total clairement défini.

En général, un **événement** sera soit **l'envoi** d'un message soit la **réception** d'un message, soit un événement interne (calcul par exemple ou l'événement délivrer). L'ordre suivant est un **ordre partiel** sur les événements qui est appelé **l'ordre causal**. Si a et b sont deux événements, on note $a \rightarrow b$ (a précède b suivant cet ordre) si et seulement si une des trois conditions est vraie :

1. a et b ont lieu sur le même site avec a avant b.
2. a=`Envoyer(`$< M >$`)` et b=`Recevoir(`$< M >$`)` du même message $< M >$.
3. Il existe un événement c tel que $a \rightarrow c$ et $c \rightarrow b$.

La condition 3 est la clôture transitive de la relation $\rightarrow$. Le graphe de la relation $\rightarrow$ n'a aucun circuit (on ne remonte pas le temps !). Certains sommets ont un (ou des) prédécesseur(s), d'autres pas. De même avec les successeurs. Le graphe donné en figure 2.3 n'est que l'application des règles 1 et 2, c'est à dire le **graphe de précédence immédiate**.

2.4.2 L'algorithme de diffusion

Le système est constitué de n sites qui font des *diffusions* à n'importe quel moment. On supposera le système sans panne mais avec des délais non bornés d'acheminement des messages et des messages qui peuvent être reçus dans un ordre différent de l'ordre d'envoi. Le but ici est d'écrire un protocole qui permet à une application (vidéo conférence par exemple) de recevoir les messages diffusés avec la garantie que les messages soient délivrés *à l'application* dans l'ordre induit par l'ordre causal. Celui-ci est construit à partir des deux événements, `Envoyer` et `Délivrer` (et aussi `Recevoir`, implicitement).

Prenons m_1 et m_2 deux messages qui sont délivrés à P_i. Si `Envoyer(`m_1`)` $\rightarrow$ `Envoyer(`m_2`)` alors on veut qu'en P_i, `Délivrer(`m_1`)` avant

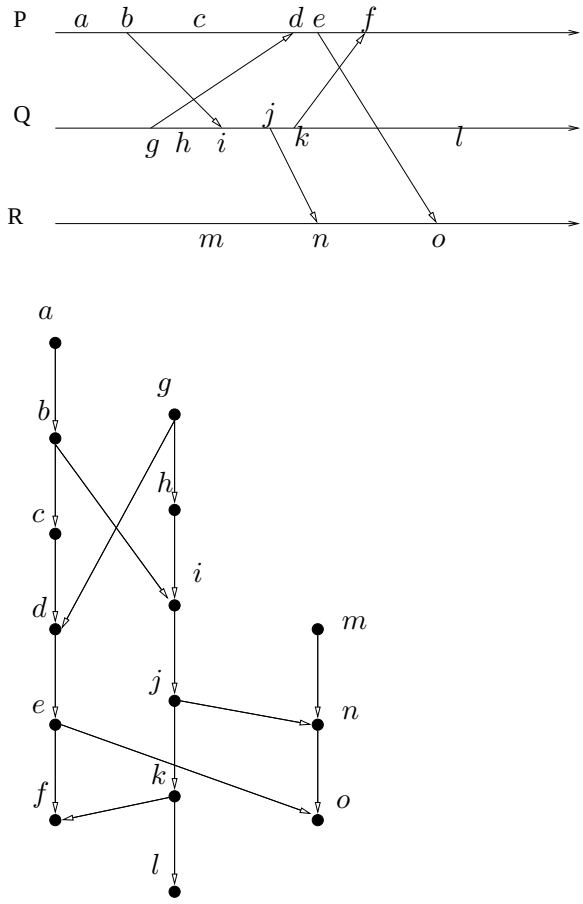

FIGURE 2.3 – Des événements + le graphe associé.

`Délivrer(`m_2`).`

PRÉCÉDENCE IMMÉDIATE DE MESSAGES.
Si m_1 et m_2 sont deux messages, on dira que m_1 *précède immédiatement* m_2, $m_1 \rightsquigarrow m_2$ si :

Envoyer(m_1) $\rightarrow$ Envoyer(m_2)
$\nexists m_3$ tel que Envoyer(m_1) $\rightarrow$ Envoyer(m_3) et Envoyer(m_3) $\rightarrow$ Envoyer(m_2)

La figure 2.5 représente le graphe de précédence immédiate des messages m_0, m_1, m_2, m_3, m_4 diffusés comme dans la figure 2.4. On suppose pour simplifier dans cet exemple que tout message qui arrive à sa destination est délivré au moment de sa réception.

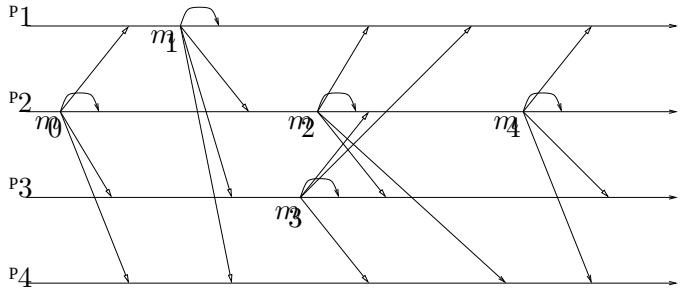

FIGURE 2.4 – 5 diffusions.

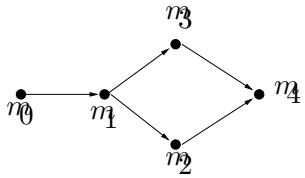

FIGURE 2.5 – Graphe de précédence immédiate des messages de la figure 2.4.

On numérotera les n sites de 1 à n. Chaque site P_i a :
- Une variable locale num_envoi_i : numéro du dernier message diffusé par P_i.
- $DEL_i[j]$ tableau de n cases :

$DEL_i[j] = d \Leftrightarrow$ le dernier message diffusé à partir de P_j et délivré à P_i avait pour numéro d.
Chaque message M sera identifié par la paire (id, num) où id est l'identité de

l'envoyeur et num le numéro de ce message lorsqu'il a été envoyé par le site id. Chaque message transportera aussi un ensemble CB_M (*barrière causale*) des identificateurs des messages qui précédent immédiatement M.

PRISE EN COMPTE DES CONTRAINTES DE PRÉCÉDENCE.
Lorsqu'un site P_i reçoit un message M avec de telles données, sous quelles conditions peut-il délivrer M en prenant en compte les contraintes de précédence immédiates ?
La condition est :

$$\forall (k,d) \in CB_M : d \leq DEL_i[k]$$

Cela exprime que tous les prédécesseurs immédiats de M ont déjà été délivrés. Le protocole de diffusion décrit pour un site i quelconque est alors :

Au départ, $CB_i := \emptyset$; $num\_envoi_i := 0$;

```
Procédure diffuser(M)
    num_envoi_i := num_envoi_i + 1;
    Pour tout (x_i ∈ V) faire Envoyer(<M, num_envoi_i, CB_i >) à x_i;
    CB_i := (i, num_envoi_i);

Lors de la réception de < M, num_M, CB_M > de P_j
    Attendre (∀(k,d) ∈ CB_M : d ≤ DEL_i[k]);
    DEL_i[j] := num_M;
    CB_i := (CB_i − CB_M) ∪ {(j, num_M)}; (*)
    Délivrer(M);
```

(*) A faire de manière atomique.
NB : le stockage (la destruction) des messages lors (après) l'attente n'est pas indiqué(e) ici mais est implicite.

Lemme 2.4.1 *Soit m_0 identifié par (k_0, d_0) et m_1 tels que $m_0 \rightsquigarrow m_1$ alors $(k_0, d_0) \in CB_{m_1}$.*

Preuve Soit P_i l'envoyeur de m_1. Comme $m_0 \rightsquigarrow m_1$ deux cas doivent être considérés. Avant d'envoyer m_1 :
— m_0 a été délivré à P_i.
— m_0 a été envoyé par P_i.

Dans les deux cas, P_i met à jour CB_i (voir algorithme) en prenant en compte m_0. La délivrance d'un autre message m' entre m_0 et l'envoi de m_1 soit ne change rien soit est impossible (car $m_0 \rightsquigarrow m_1$). □

Théorème 2.4.1 *Les délivrances de messages respectent l'ordre causal.*

Preuve
Considérons deux messages m_0 et m_x tels que :
— `Envoyer(`m_0`)` $\rightarrow$ `Envoyer(`m_x`)` et
— m_0 et m_x sont délivrés à P_i.
Il faut montrer que m_0 est délivré avant m_x en P_i.
On montre cela par récurrence sur la longueur l du chemin entre m_0 et m_x dans le graphe de précédence immédiate ($\rightsquigarrow$) des messages.

Si $l = 1$, $m_0 \rightsquigarrow m_x$ et d'après le résultat du lemme 2.4.1, $m_0 \in CB_{m_x}$. Or, P_i va attendre d'avoir délivré m_0 avant de délivrer m_x (c'est l'attente de l'algorithme lors de la réception de m_x).

Hypothèse de récurrence : lorsque le chemin de causalité est de longueur strictement supérieure à $l \geq 1$, les contraintes de précédence sont respectées.

Considérons maintenant un chemin de longueur l de m_0 à m_x :

$$m_0 \rightsquigarrow m_1 \rightsquigarrow \dots m_{l-1} \rightsquigarrow m_x$$

Ainsi, par hypothèse : tous les messages $m_0, m_1, \dots, m_{l-1}$ délivrés à P_i le sont suivant l'ordre causal.
m_{l-1} est délivré à P_i. Comme $m_{l-1} \rightsquigarrow m_x$, m_{l-1} est délivré avant m_x (c'est le cas de base). De plus, comme $m_0 \rightsquigarrow m_1 \rightsquigarrow \dots m_{l-1}$, par hypothèse de récurrence, m_0 est délivré avant m_{l-1}. Ainsi, m_0 est délivré avant m_x en P_i.

□

Chapitre

3 Allocation de ressources

Dans ce chapitre nous allons aborder un des premiers problèmes qui s'est posé lors de l'élaboration des systèmes répartis. Plusieurs sites distants coopèrent et partagent des ressources communes. Ces ressources peuvent être de plusieurs types : matériels (imprimantes, scanners...) ou logiciels (accès au jeton donnant droit à l'utilisation d'un logiciel, accès à une zone mémoire commune, à une BD etc...). Ces *ressources partagées* ont des propriétés et doivent être gérées en respectant certaines règles d'accès.
Dans ce chapitre (mais aussi en TD) nous allons nous intéresser à une ressource qui aura les propriétés suivantes.

— Elle est complètement partagée (tous les sites peuvent y accéder via le réseau).
— Au plus M sites peuvent y accéder *simultanément*.

Si à un moment donné, plus de M sites y accèdent simultanément nous sommes dans une situation d'**erreur**. Il faut donc construire un *protocole d'accès à cette ressource* qui respecte la condition qu'il n'y ait pas d'erreur. De manière plus précise, notre protocole devra vérifier les propriétés suivantes pour une ressource à M entrées.

Sûreté. A chaque instant au plus M sites utilisent la ressource.

Vivacité. Chaque site demandant un accès à la ressource l'obtiendra au bout d'un temps fini (en l'absence de pannes). On parle aussi *d'absence de famine*.

Les deux propriétés sont indispensables pour avoir un système sûr et équitable. Les demandes d'accès à une ressource sont totalement asynchrones (elle peuvent avoir lieu n'importe quand). Pour encadrer la bonne utilisation d'une ressource, le protocole est exécuté schématiquement de la manière suivante en chaque site désirant utiliser la ressource.

```
< Acquisition >
   < Section critique >
```

`< Libération >`

La *phase* d'acquisition consiste pour le site à demander la ressource (nous verrons comment) jusqu'à l'obtenir finalement. Une fois que le site a obtenu la ressource il peut l'utiliser en *section critique*. L'utilisation d'une ressource peut être totalement quelconque. Nous ne décrirons pas du tout cette phase qui est propre à chaque type d'application et à chaque ressource. Nous supposerons malgré tout que la *phase d'utilisation a une durée finie*. Après l'utilisation de la ressource, le site qui l'avait la libère pour que d'autres sites puissent l'utiliser à leur tour. Les protocoles que nous aurons à décrire sont donc constitués des deux phases d'acquisition (ou réservation) et de libération.

Nous supposerons en général (sauf précision contraire) que :
- Chaque site qui est dans sa phase d'acquisition ne rentre pas dans une nouvelle phase d'acquisition avant d'avoir exécuté la phase de libération (un site ne demande qu'une ressource à la fois).
- Il n'y a pas de panne (certains des protocoles exposés plus loin ont été enrichis pour tolérer des pannes).

Comme tous les autres protocoles répartis, le nombre de messages échangés pendant l'acquisition d'une ressource et/ou pendant la libération de la ressource devra être le plus petit possible.

3.1 Protocoles à base de permissions

Pour les protocoles suivants nous aurons besoin d'estampiller les messages avec des marques d'horloge. Hélas aucune horloge globale n'est disponible. Dans ce qui suit nous décrivons un mécanisme d'estampillage basé sur des *horloges logiques* locales. Le mécanisme d'estampillage lui même peut être vu comme un sous-protocole associé et rendant un service à un autre. Pour ce qui nous concerne dans ce chapitre, le protocole utilisateur sera le protocole d'allocation de ressource.
Chaque site i maintient une variable entière $horloge_i$ initialisée à 0.

```
Lors de la réception d'un message < M, h >
    horloge_i := max{horloge_i, h} + 1;
    Délivrer(M);

Lors de l'envoi des données M
    horloge_i := horloge_i + 1;
```

`Envoyer(`$< M, horloge_i >$`);`

Les messages sont donc transportés avec l'heure logique à laquelle ils ont été émis. Cependant, à un moment donné on peut très bien avoir deux messages possédant la même estampille. Il faut pouvoir les départager. Le moyen usuel de faire cela est de prendre en compte l'identité (unique) de l'émetteur du message. Ainsi, au lieu de ne considérer que l'horloge logique h associée au message on peut prendre le couple (h, i) où i est l'identité du site qui a envoyé le message. L'ordre sur ces couples est le suivant.

$$(h_i, i) < (h_j, j) \Leftrightarrow (h_i < h_j \text{ ou } (h_i = h_j \text{ et } i < j))$$

Ce qui peut se traduire par : i est plus prioritaire que j car :
Soit la demande de i a été faite avant celle de j (en prenant en compte les horloges locales de i et j). Soit les horloges sont les mêmes mais l'identité de i est plus petite que celle de j.
NB : cet ordre est un *ordre total* (toutes les paires sont comparables).

3.1.1 L'algorithme de Lamport (1978)

La version présentée ici est légèrement modifiées par rapport à celle de Lamport ; l'algorithme initial de Lamport exigeait que les messages ne soient pas déséquencés, cette hypothèse n'est pas utile dans la version ci-dessous.
Cet algorithme utilise um maillage complet et deux techniques : les estampilles pour établir un ordre total entre les demandes d'accès à la ressource, et la distribution d'une file d'attente des processus en attente de la ressource. On fait l'hypothèse que les lignes sont fiables ; les messages ne sont pas perdus.
Nous notons h la valeur de l'horloge. Un message envoyé par un site i lorsque son horloge vaut h est estampillé par le couple (h, i).
Il y a trois types de messages :

- le type requête : un message de ce type est diffusé par un site qui désire obtenir la ressource vers tous les autres gestionnaire.
- le type libération : un message de ce type est diffusé par un site qui vient de recevoir du processus auquel il est associé un message $Restitutiondelaressource$ vers tous les autres sites.
- le type acquittement : lorsque site i reçoit un message de type requête, il répond à j par un message de type $acquittement$, autrement dit un accusé d réception, sauf si le site i est dans le protocole d'acquisition ; dans ce dernier cas, le site i ne répond rien.

Chaque site i possède un tableau nommé T, indicé par $0, 1, \ldots, n-1$ où i inscrit, à l'indice j :

— (requête, h) lorsqu'il reçoit un message de type requête estampillé par (h, j)

— (libération,h) lorsqu'il reçoit un message de type libération estampillé par (h, j) (acquittement,h) lorsqu'il reçoit un message de type acquittement estampillé par (h, j)

On nomme $T[i].type$ le premier champ de $T[i]$ et $T[i].date$ le second champ. Le site i range également en $T[i]$ les informations correspondant aux messages qu'il envoie, dans les mêmes conditions que ci-dessus.

Un site i qui désire obtenir la section critique pour le processus dont il gère le contrôle ne peut le faire que si $T[i]$ est le message le plus ancien noté dans T selon l'ordre défini par les estampilles ; on rappelle que $T[i]$ est plus ancien que $T[j]$ si, ou bien l'heure portée par $T[i]$ est strictement inférieure à l'heure portée par $T[j]$, ou bien ces deux heures sont égales et $i < j$; on dit alors aussi que l'estampille de $T[i]$ est inférieure à l'estampille de $T[j]$.

Lors d'une demande accéder à la ressource pour un site i
    $Diffuser(requte, h, i)$
    $T[i] \leftarrow (requte, h, i)$;

Lors d'une reception d'un message du type $(type, k, j)$
    $h \leftarrow \max(h, k) + 1$;
    $T[j] \leftarrow (type, k, j)$
    Si $(T[i].type \neq requte)$
        Si $(type = requte)$ envoyer $(acquittement, h, i)$ à j
    Sinon Si $(\forall j \neq i, (T[i].date, i) < (T[j].date, j)$ Accéder à la section critique

Lors de la libération de la ressource pour un site i
    $Diffuser(liberation, h, i)$
    $T[i] \leftarrow (liberation, h, i)$;

Théorème 3.1.1 *L'algorithme de Lamport assure l'exclusion mutuelle et la complexité est de* $3(n-1)$ *pour l'utilisation de la ressource.*

Preuve

Montrons que cet algorithme assure l'exclusion mutuelle. Supposons que i et j aient fait une requête pour obtenir la ressource. Il est clair que si i et j reçoivent les

autorisations de j et de i. Sachant que les estampilles induisent un ordre total sur les requêtes.
Seul le site qui a effectué la demande la plus ancienne (selon l'ordre total défini par les estampilles) peut obtenir la ressource ; cette situation n'évolue qu'au moment où le site envoie un message de libération. D'où l'exclusion mutuelle et plus précisément seul le site qui a fait la requête de plus petite estampille peut obtenir la ressource.
Inversement, grâce aux messages d'acquittement, le site qui a fait la requête non encore satisfaite de plus petite estampille ne peut pas être bloqué pour l'obtention de la ressource : il n'y a pas d'interblocage. Une requête aura au bout d'un temps fini la petite estampille ; il n'y a pas de famine.
On constate qu'une utilisation de la ressource par un processus nécessite l'échange entre les gestionnaire de $3(n-1) : (n-1)$ pour la requête, $n-1$ pour les acquittements, $n-1$ pour la libération.

□

Remarque : L'algorithme peut être amélioré pour résister aux pannes des sites ; lorsqu'un gestionnaire tombe en panne, un message est diffusé pour signaler cette panne et le test sur l'ancienneté de la requête n'est alors fait qu'entre les stes non en panne. Il faut prévoir aussi les rmises à jour lors d'une réinsertion de sites.

3.1.2 L'algorithme de Chandy et Misra 1984 : permissions sans estampillages

La différence essentielle entre cet algorithme et l'algorithme précédent st que les règles de priorité sont définies sans utiliser d'estampillage de messages.Lorsqu'un site i reçoit de j la permission de i, il est prioritaire sur j pour une utilisation de la ressource : lorsque cette utilisation aura été effectuée, la priorité sera à j ; si le site j demande la permission à i, i lui enverra cette permission.
L'absence d'estampillage des messages évite aux sites de gérer une heure logique ; elle évite aussi la nécessité que chaque site ait un nom. Il suffit que chaque site puisse identifier les canaux de transmission qui le relient aux $n-1$ autres sites.
Nous notons $c_1, c_2, \ldots, c_n$. Nous utiliserons les variables suivantes :

- $etat$: peut prendre les valeurs demandeur, utilisateur, ailleurs, initialisée à ailleurs ;
- $utilise$: tableau indicé par $c_1, c_2, \ldots, c_n$ de bolléens ; $utilise_i[c_j]$ est vrai si le site i possède la permission venant du canal c_j et si le site i a utilisé au moins une fois depuis qu'il possède cette permission ; sinon cette variable vaut faux ; l'initialisation est donnée ci-dessous.

— $permission\_manquantes$: ensemble d'identificateurs de canaux ; l'initialisation est donné ci-dessous ;
— $en\_attente$: ensemble d'identificateurs de canaux initialisé à $\varnothing$

IL faut être vigilant pour l'initialisation des ensembles $permissions\_manquantes$ et des tableaux appelés « utilisés ». Précisions les hypothèses qu'on veut avoir à l'initialisation et conserver au cours de l'exécution.

Il faut tout d'abord que , pour chaque paire de sites, il y ait une permission virtuelle qui soit « chez » l'un , ou « chez » l'autre, ou en transition de l'un à l'autre, et en aucun cas « chez » les deux sites. Cela se traduit par le fait que, appelant c le canal joignant le site i au site j :

1. soit la permission est « chez » i : $c \notin permissions\_manquantes$ pour i et $c \in permission\_manquante$ pour j.
2. soit la permission est « chez » j ; $c \notin permissions\_manquantes$ pour j et $c \in permissions\_manquantes$ pour i
3. soit une permission est en transit sur c de i vers j
4. soit une permission est en transit sur c de j vers i

Les conditions initiales sont les suivantes : i est prioritaire sur j si et seulement si :

— on est le cas 1 et, pour le site i nous avons $utilise_i[c] := faux$
— on est dans le cas 2 et, pour j nous avons $utilise_j[c] := vrai$
— on est dans le cas 4

L'algorithme est le suivant :

```
Lors d'une demande de la ressource pour un site i
    Si permissions_manquantes_i = ∅
    etat_i := utilisateur
    Accès à la section critique
    Sinon etat_i := demandeur
    ∀c_j de permissions_manquantes_i
        envoyer Demande_de_permissions par c_j;

Lors d'une reception d'un message du type
Demande_de_permissions venant de j
Si (etat_i = utilisateur ou utilise_i[c_j] = faux
    en_attente := en_attente ∪ {c_j}
Sinon     envoyer Permission par c_j;
    permission_manquantes_i := permissions_manquantes_i ∪ {c_j};
    utilise_i[c_j] := faux;
    Si (etat_i = demandeur) envoyer Demande_de_permissions
par c_j
Lors de la réception de permissions par le canal c_j
```

$permission\_manquantes_i := permissions\_manquantes_i - \{c_j\}$;
Si $permission\_manquantes_i := \varnothing$;
Accès à la section critique
$etat_i := utilisateur$

Lors d'émission d'un message envoie restitution des permissions
$etat_i := ailleurs$
Pour tout c_j, $utilise_i[c_j] := vrai$
$permission\_manquantes_i := en\_attente$;
Pour tout c_j dans $en\_attente$;
envoyer $Permission$ par c_j;
$permission\_manquantes_i := permissions\_manquantes_i \cup \{c_j\}$;
$utilise_i[c_j] := faux$;
$en\_attente := \varnothing$;

Considérons le graphe des priorités ; il s'agit du graphe dont les sommets sont les sites et où il existe un arc de i vers j si j est prioritaire sur i. Il est facile de vérifier que l'absence d'interblocage est assuré par l'absence de circuit dans le graphe des priorités.
L'initialisation doit assurer cette absence de circuits. Pour cela, il suffit de considérer au départ un ordre total sur les sites ; si i se trouve avant j dans cet ordre, on met la permission chez i ($c \notin permissionmanquante$ pour i, et $c \in permissionmanquante$ pour j) ; par ailleurs, on donne la valeur vrai à toutes les composantes de tous les tableaux « utilisé ». Dans le graphe des priorités, l'arc entre i et j va de i vers j is et seulement si i est avant j dans l'ordre total considéré. Les arcs du graphe de priorités ne s'inversent que lorsqu'un site k libère la ressource : le site k a alors un demi-degré intérieur nul dans ce graphe. Il ne peut appartenir à aucun circuit : aucune inversion d'arcs au cours de l'algorithme ne crée donc de circuit. L'initialisation proposée assure à l'algorithme l'absence d'interblocage.
On peut également vérifier que cet algorithme est vivace.

3.2 Protocoles avec un jeton

Dans cette section nous allons étudier des protocoles pour gérer une ressource à une entrée ($M = 1$). Le principe de base est le suivant. La ressource est représentée par un *jeton*. La condition nécessaire pour utiliser la ressource est d'avoir le jeton.

Le fait qu'il n'y ait qu'un seul jeton qui circule permet de garantir la propriété de sûreté (il faut être sûr que le jeton ne peut pas être dupliqué dans le protocole).

3.2.0.1 Dans un arbre

Dans la situation suivante nous supposons que les sites sont interconnectés par un graphe quelconque. Tout site peut communiquer directement avec n'importe quel autre. Nous allons maintenir un *arbre logique*. Chaque site p maintient les variables locales suivantes. $Avoir\_jeton_p, En\_SC_p \in \{vrai, faux\}$; $Racine_p$ identificateur de site qui indique à p vers quel voisin dans l'arbre il doit demander la ressource. $Request_p$ est une file d'identificateurs de sites, initialisée vide. A l'initialisation, chaque variable En_SC_p est à $faux$ et les sites sont initialement organisés sous forme d'arbre d-aire de telle manière que chaque site p a sa variable $Racine_p$ qui a pour valeur l'identificateur de son voisin qui est sur (l'unique) chemin de p vers le sommet q possédant le jeton (q est le seul site ayant initialement $Avoir\_jeton_q = vrai$). L'organisation logique des sites est du type de celle décrite à la figure 3.1.

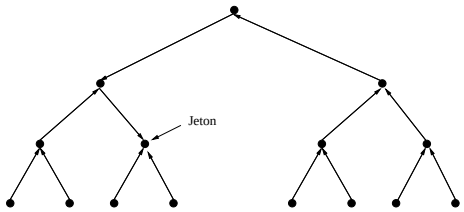

FIGURE 3.1 – Exemple d'organisation logique des sites (via la variable $Racine$)

Nous utiliserons les messages $REQUEST$ pour demander à utiliser la ressource et le message $JETON$ qui représente la ressource (ou l'autorisation de l'utiliser). L'idée de ce protocole est que chaque site p a toujours un «pointeur» ($Racine_p$) qui est dirigé vers le jeton dans l'arbre.

```
Procédure acquisition
    Si (non Avoir_jeton_p) alors
        Si (File_vide(Request_p)) alors Envoyer(<
REQUEST >) à Racine_p;
        Ajouter(Request_p, p);
        Attendre(Avoir_jeton_p);
    En_SC_p := vrai;
```

```
Procédure libération
    En_SC_p := faux;
    Si (non File_vide(Request_p)) alors
        Racine_p := Défiler(Request_p);
        Envoyer(< JETON >) à Racine_p;
        Avoir_jeton_p := faux;
        Si (non File_vide(Request_p)) alors
            Envoyer(< REQUEST >) à Racine_p;

Lors de la réception de < REQUEST > de q
    Si (Avoir_jeton_p) alors
        Si (En_SC_p) alors Ajouter(Request_p, q);
        Sinon
            Racine_p := q;
            Envoyer(< JETON >) à Racine_p;
            Avoir_jeton_p := faux;
    Sinon
        Si (File_vide(Request_p)) alors Envoyer(<
REQUEST >) à Racine_p;
        Ajouter(Request_p, q);

Lors de la réception de < JETON > de q
    Racine_p := Défiler(Request_p);
    Si (Racine_p = p) alors Avoir_jeton_p := vrai;
    Sinon
        Envoyer(< JETON >) à Racine_p;
        Si (non File_vide(Request_p)) alors Envoyer(<
REQUEST >) à Racine_p;
```

La file $Request$ sert à ordonner les demandes en chaque sous-arbre. Il est important de noter les points suivants dans l'algorithme.

- Si un site reçoit un message $< REQUEST >$ il le transmet dans son sous-arbre dans lequel il y a le jeton si sa file est vide (c'est à dire si personne n'a demandé le jeton avant). Si sa file n'est pas vide il bloque la demande à son niveau et ne la servira que lorsque les autres demandes qui sont avant dans la file auront été servies.
- Lors de la phase de libération, le site possédant le jeton le transmet au premier de sa file et si sa file n'est pas vide, il transmet une nouvelle demande (envoi du message $< REQUEST >$) pour que le suivant dans sa file puisse être servi.

Les files d'attente permettent d'assurer la vivacité du mécanisme. Dans cet algorithme, le jeton ne circule dans l'arbre que si on a besoin de lui sur un site distant. Si l'arbre est un arbre d-aire, son diamètre est en $O(\log n)$ où n est le nombre total de sites, ce qui limite le mouvement des messages.

Chapitre

4 Problèmes de l'élection

4.1 Introduction

Ce chapitre est consacrée au problème de l'élection. A un moment donné, lors d'une exécution ou après une panne (réparée) les sites ont tous une valeur (deux à deux distinctes). Le but de l'élection est de déterminer le site qui a la plus grande valeur. Ce site est le *gagnant*. A la fin de l'élection, chaque site doit connaître ce gagnant. Nous supposerons dans un premier temps que le réseau est un cycle ou un circuit. Nous supposerons que les *identificateurs* (adresse) des sites transitent avec les messages (champ source du message) et que les liens sont FIFO. Au début de l'algorithme, chaque site i a une valeur (locale) val_i qui va servir de base à l'élection (sans perte de généralité on peut confondre cette valeur avec l'identificateur du site). Chaque valeur est unique. Le but d'un algorithme d'élection est de faire en sorte qu'au bout d'un temps fini, tous les sites soient d'accord sur le site élu.

4.2 Algorithme sur l'anneau

La solution la plus simple au problème de l'élection dans un cycle unidirectionnel est décrite de manière informelle et incomplète par :

— Au début, chaque site i maintient une variable max_i initialisée à val_i puis il envoie sa valeur val_i vers son voisin.

— Chaque fois qu'un site i reçoit une valeur V il fait $max_i := \max\{max_i, V\}$ et fait suivre V vers son autre voisin.

Dans la suite, nous allons répondre aux questions suivantes :

1. Si un seul (ou plusieurs) site initie l'élection, comment tous les autres sites peuvent participer à l'élection ? (problème du réveil).
2. Quelle est la condition à rajouter pour que l'algorithme se termine ? (problème de la terminaison). Qui sait qui a gagné l'élection, comment proclamer le résultat ? (problème de la prise de décision).

3. Combien de messages transitent durant cet algorithme ?
4. Ecrire l'algorithme.
5. Donner une exécution sur le graphe donné par la figure 4.1.
6. Est-ce que cet algorithme reste valide en mode asynchrone ?
7. Comment peux-t'on améliorer l'algorithme ?

Dans un premier temps, donnons l'algorithme :
Voici l'algorithme :

```
En cas de réveil spontané
     participant_i := Vrai;
     max_i := val_i
     Envoyer(< ELECTION, val_i >);

 Lors de la réception d'un message < ELECTION, V >
     max_i := max{max_i, V}
     Si  val_i = V  alors
     Arret(succès)          sinon Envoyer(<
ELECTION, V >);
```

Nous allons illustrer cet algorithme

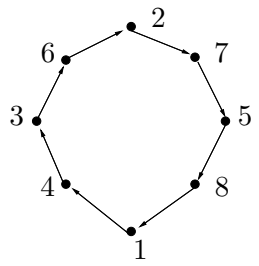

FIGURE 4.1 – Une configuration dans C_8.

1. Pour le réveil des sites on combine un réveil spontané et un réveil qui a lieu lors de la réception d'un message lié au protocole de l'élection. Dans ce dernier cas, les sites initialisent max_i et font tourner l'algorithme.
2. On suppose que les valeurs sont deux à deux différentes. Lorsqu'un site i reçoit sa propre valeur de son voisin, c'est le signe qu'il a reçu toutes les valeurs. Il peut alors arrêter de retransmettre. Le site i connaît alors le site ayant la valeur maximale (gardée dans max_i). Tous les autres sites vont finir par être aussi dans cet état. Il est alors inutile de proclamer le résultat.
3. Supposons que nous ayons n sites. Chaque site envoie son message qui fait

le tour du cycle et parcours donc n arcs. Cela fait donc n^2 messages échangés en tout.

4. Non l'algorithme ne fonctionne pas en mode asynchrone. Pour qu'il marche il faut mettre un compteur sur chaque site qui correspond au nombre de site dans l'anneau. Et tant que je n'ai pas reçu n messages et le processus n'est pas terminé.

4.3 Le protocole YO-YO

Dans cette section, nous proposons un algorithme pour le problème de l'élection dans un graphe quelconque non orienté. L'algorithme se compose en deux parties : une phase de pré-processing et une séquence d'itérations.

La phase de pré-processing Dans la phase de pré-processing chaque site x échange sa valeur $id(x)$ avec ses voisins. Tous les sites connaissent le maximum local ainsi. Une orientation est crée entre x et y si $id(x) < id(y)$. Soit $\vec{G}$ le graphe ainsi créé.

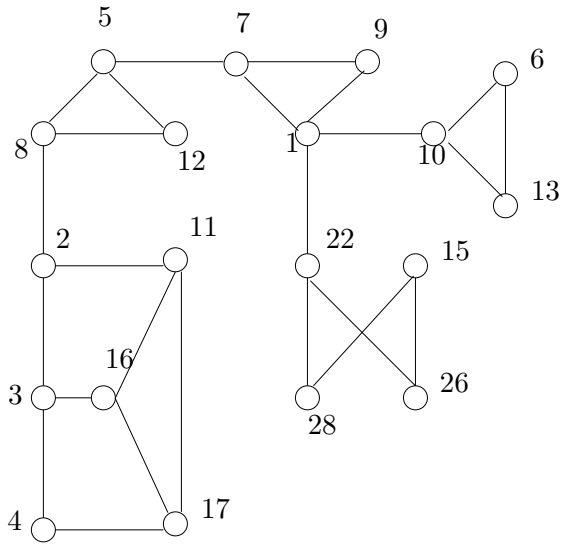

FIGURE 4.2 – *Un graphe non orienté.*

Lemme 4.3.1 *Le graphe $\vec{G}$ est acyclique.*

Preuve

Pour montrer ceci, supposons qu'il existe un cycle $x_0, x_1, \ldots, x_k$ dans $\vec{G}$. Par construction nous avons $id(x_0) < id(x_1) < \ldots < id(x_k) < id(x_0)$ ce qui est impossible.

□

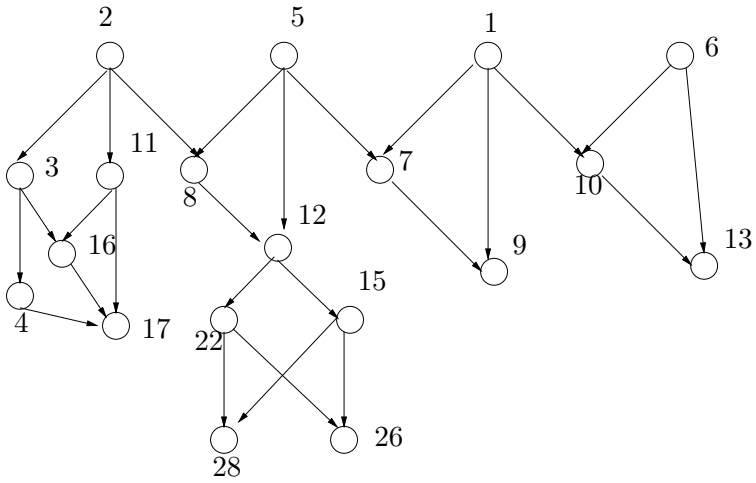

FIGURE 4.3 – *Orientation suite à la phase de pré-processing du graphe donné par la figure 4.2, ce qui conduit à un graphe acyclique.*

Sachant que $\vec{G}$ est un graphe orienté nous allons considérer trois types de sommets :

- — Source, des sommets ayant que des arcs successeurs. Ce sont des minimums locaux.
- — Puits, des sommets ayant que des arcs prédécesseurs, ce sont des maximums locaux.
- — Sommet interne, les autres.

A la fin de phase de préprocessing tous les sommets connaissent leur status. La seconde phase commence.

La phase YO-YO La partie centrale de l'algorithme est une séquence d'itérations appelé YO- et -YO.

La phase YO- Cette phase est initiée par les sources. Dans cette phase, les valeurs des sites sources seront transmises aux puits

1. Une source envoie sa valeur à tous ces voisins.
2. Les sommets internes attendent de recevoir les valeurs de voisins via les arcs

prédécesseurs, et calcule minimum et envoie cette valeur aux autres voisins via les arcs successeurs.

3. Les sommets puits attendent de recevoir toutes les valeurs de ces voisins et calcule le minimum, et commence la seconde phase -YO.

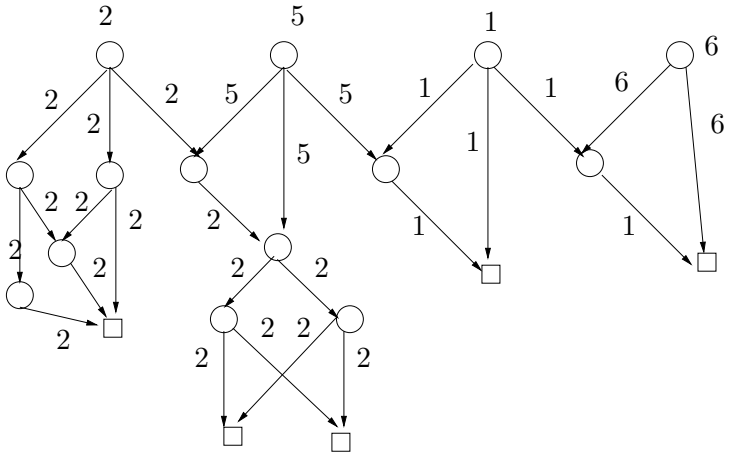

FIGURE 4.4 – *Dans une itération, seules les sources sont candidates.*

La phase -YO Cette phase est initiée par les puits, et le but est éliminé des candidats transformant des sources en puits ou en sommets internes. Les puits avertissent les sources si leurs valeurs sont minimales.

4. Un puits envoie le message Yes à tous les voisins qui a transmis la valeur minimale reçu.. Sinon il envoie le message No.
5. Un sommet interne attend de recevoir les valeurs de tous ses voisins successeurs . Si toutes les votes Yes, il envoie la valeur Yes aux sommets voisins prédécesseurs pour lesquels la plus petite valeur a été envoyé, et No pour les restes. Si il existe un vote No, il envoie No à tous ces voisins.
6. Une source attend de recevoir les valeurs de tous sommets successeurs. Si toutes les votes sont Yes, il survit à cette itération et peut initier la suivante. Si une vote est No, le sommet n'est plus candidat.

Avant que la prochaine itération peut commencer, les orientations dans le graphe $\vec{G}$ sont modifiées telles que les sources qui sont encore candidats (ceux qui ont reçu que des Yes) restent des sources, clairement cette modification ne créera pas de cycle. Le graphe $\vec{G}$ reste acyclique, mais le nombre de sources diminuent à chaque itération.

1. Quand un sommet x envoie No à un voisin prédécesseur y, l'arc est inversé.

2. Quand un sommet y reçoit un message du type No venant d'un voisin successeurs, l'arc est inversé.

Une source recevant un avis No, ne reste pas une source, et devient un puits. Certains puits peut devenir un sommet interne et certains sommet interne peut devenir puits. Par contre aucun sommet interne, ni sommet puits ne peuvent devenir une source.

Lemme 4.3.2 *L'application d'une phase de l'algorithme sur $\vec{G}$ avec z sommets sources conduit à un nouveau graphe $\vec{G}$ avec moins de z sommets. La source avec la plus petite valeur reste une source.*

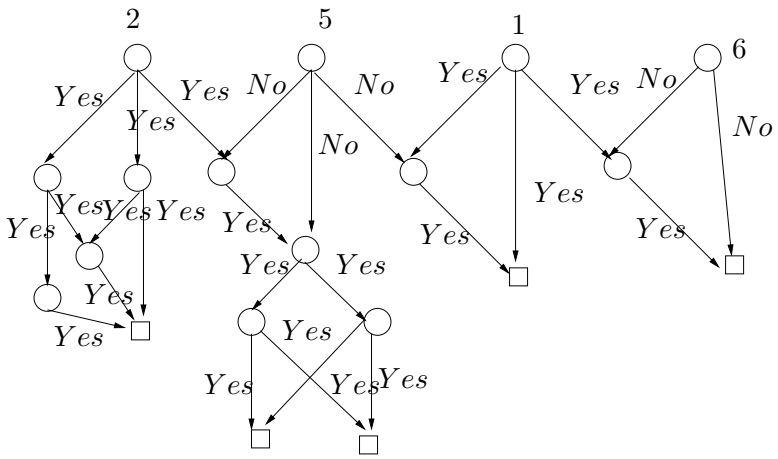

FIGURE 4.5 – *Graphe valué par la décision Yes/No jusqu'au source.*

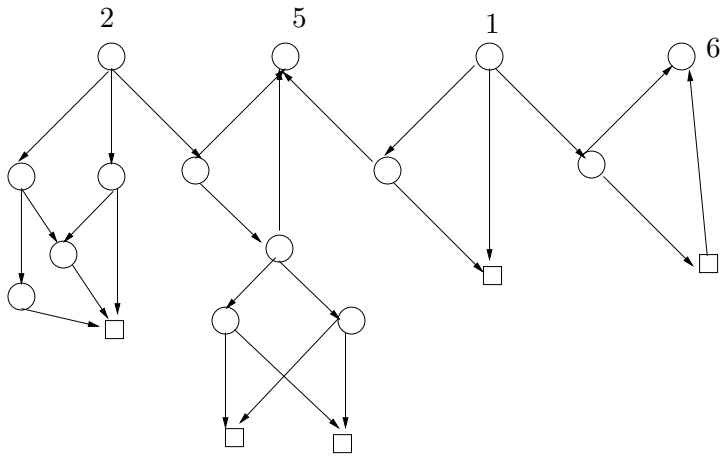

FIGURE 4.6 – *Dans la phase $-YO$, l'orientation des arcs changent en fonction de la réception d'un No*

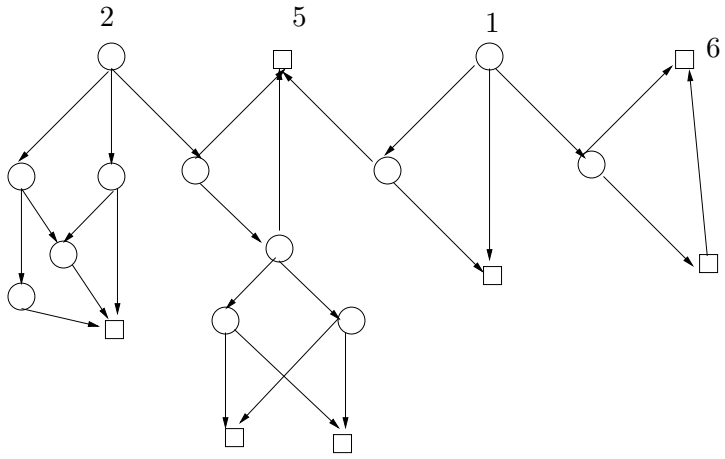

FIGURE 4.7 – *Création d'un nouveau graphe acyclique, où seulement sources qui restent candidats.*

La source ayant la plus petite valeur restera en vie après les itérations. Comment détecter la terminaison de cet algorithme ? c'est à dire le cas où il reste une seule source.
Lors de la dernière itération (celle où il reste une unique source) dans la phase YO-seule la valeur c circulera et dans la phase -YO seulement des Yes vont circuler. La source recevra donc que des Yes, mais dans les itérations précédentes la source

recevra également que des Yes. Comment distinguer les deux cas de figures ? Pour cela il est nécessaire d'ajouter un mécanisme additionnel.

Elagage Le principe de l'élagage est de retirer des sommets dans certaines itérations, qui sont considérés comme inutiles. Ainsi nous avons deux grands principes qui vont être utilisés. La première meta-règle est une règle structurelle. Pour l'expliquer, rappelons que la fonction des puits consiste à réduire le nombre de sources par le vote des valeurs reçues. Considérons un puits qui est une feuille (*i.e.* un seul voisin prédécesseur) ; tel que le sommet reçoit seulement une valeur ; ainsi il ne peut être qu'un YES. En d'autres termes, le puits-feuille ne peut-être que d'accord qu'avec la décision du prédécesseur. Alors le puits est inutile.

1. Un puits qui est une feuille, alors il devient inutile, et demande à ses prédécesseurs d'être élaguées. Si un sommet demande à être élaguées, il sera en se déclarant inutile (à enlever dans le prochaine itérations),
2. Si dans la phase $YO-$, un sommet reçoit la même valeur à partir d'au moins un voisin prédécesseur, il demande à tous sauf un d'élaguer le lien les connectant, et déclare ces liens inutiles. Les sommets recevant une telle demande, ils se déclarent inutiles et retire les liens our la prochaine itérations.

Les règles d'élagages requièrent de la communication : dans la règle (7), un puits-feuille nécessite de communiquer avec son unique voisin que le lien est inutile ; dans la règle (8), un sommet reçoit une information d'élaguer les voisins et les liens associés.
L'élagage se fait lors de la prise de décision.
Nous essayons de réduire les informations redondantes. Durant la phase $YO-$, un sommet interne ou un puits peut recevoir plusieurs fois la même valeur venant de la même source, cette information est clairement redondante, la réception unique de cette valeur est suffisante. Soit x le sommet qui reçoit la même valeur de la source s à partir des ces sommets prédécesseurs $x_1, \ldots, x_k, k > 1$. Ceci indique que dans le graphe acyclique, il existe des chemins directs entre x et les k voisins distincts. Si nous supprimons un voisin prédécesseur, l'information arrive quand même à x. Il suffit donc de garder qu'un seul voisin prédécesseur, les autres envoient de l'information redondante, et donc on peut supprimer ces voisins.
Revenons au problème de la terminaison. Comme dit lors des opérations d'élagages intégrées dans la phase $-YO$: pour comprendre comment et pourquoi, considérons les effets d'une exécution d'une itération complète (avec élagage) sur le graphe acyclique avec une seule source.
Notons qu'après l'élagage un lien du fait de la règle (10), un puits peut devenir une feuille et alors inutile (par la règle (9)).

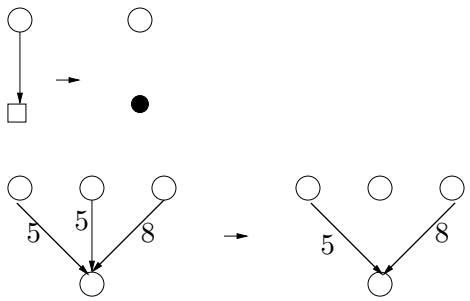

FIGURE 4.8 – *Règle d'élagage.*

Lemme 4.3.3 *Si le graphe admet une seule source, alors, après une itération, le nouveau graphe est composé par un seul sommet, la source.*

Autrement dit, quand il existe qu'une seule source c, tous les autres sommets ont été retirés et c est le dernier sommet dans le graphe. Cette situation sera découverte par c, quand du fait de l'élagage qui n'aura plus de voisin.

Evaluation des coûts La formule générale qui exprime le coût du protocole $YO - YO$ est facile à établir ; cependant la détermination exacte du coût reste une question d'actualité. Nous proposons la formule générale.
Dans la phase d'amorçage, chaque sommet envoie sa valeur à son voisinage ; alors sur chaque lien deux messages sont échangées, donc un total de $2m$ messages.
Considérons maintenant une itération. Dans la phase $YO-$, chaque sommet utile (à l'exception des puits) envoie un message aux voisins successeurs ; alors sur chaque lien considéré un seul message exactement un seul message. De manière similaire dans la phase $-YO$, chaque sommet utile (à l'exception les sources) envoie un message à ces voisins prédécesseurs. Ainsi dans chaque itération i il y a exactement $2m_i$ messages, avec m_i est le nombre de liens dans le graphe à l'étape i.
La notification de la terminaison à partir du leader peut-être calculé par la diffusion sur l'arbre couvrant construit avec seulement $n - 1$ messages.
Alors, le coût total est

$$2\sum_{i=0}^{k(G)} m_i + n - 1$$

avec $m_0 = m$ et $k(G)$ est le nombre total d'itérations sur le réseau G. Il est nécessaire de déterminer le nombre d'itérations $k(G)$. Soit $D(1) = \vec{G}$ le graphe initial obtenu à partir de G lors de la phase d'amorçage. Soit $G(1)$ le graphe non-orienté défini de la manière suivante : il existe un sommet pour chaque source dans $D(1)$

et il existe entre deux sommets si et seulement si les sources associées admettent un puits en commun. Considérons maintenant le diamètre $d(G(1))$ de ce graphe.

Lemme 4.3.4 *Le nombre d'itérations est au plus* $\lceil \log\ diam(G(1)) \rceil + 1$.

Preuve
Pour cela, considérons deux voisins a et b dans $G(1)$. Par définition, les sources de $D(1)$ admettent un puits commun, tel que au moins une des deux sources sera défait (car le puits envoie un YES à seulement à l'un des deux). Ceci implique si nous prenons un chemin dans $G(1)$, au moins la moitié des sommets sur ce chemin seront des sources qui cesseront d'être des sources à la fin de l'itération.
Par conséquent, si la source a survit, ceci signifie qu'il existe un puits en commun avec chaque sommet non-éliminés (sources associées à) aux voisins de b. Ceci implique que si nous considérons le nouveau graphe acyclique $D(2)$, le graphe correspondant $G(2)$ est exactement le graphe obtenu par la suppression des sommets associés aux sources éliminées, et liant ensemble les sommets précédent à distance deux. En d'autres mots, $d(G(2)) \leq \lceil d(G(1))/2 \rceil$.
De manière similaire, la relation entre les graphes $G(i-1)$ et $G(i)$ correspondant au graphe acyclique $D(i-1)$ à l'itération $i-1$ et le nouveau graphe acyclique $D(i)$, respectivement,. En d'autres mots $d(G(i)) \leq \lceil d(G(i-1))/2 \rceil$. On observe que $diam(G(i)) = 1$ correspond à la situation où toutes les sources , à l'exception une, sont défaits dans cette itération, et $d(G(i)) = 0$ correspondant à la situation où il y a une unique source qui est éliminée (que nous ne savons pas encore). Comme $d(G(i)) \leq 1$ après au plus $\lceil \log diam(G(1)) \rceil$ itérations, la propriété suit :
Comme le diamètre du graphe ne pas être plus grand que le nombre de sommets, comme les sommets de $G(1)$ correspond aux sources de $\vec{G}$, nous avons

$$k(G) \leq \lceil \log s(\vec{G}) \rceil \leq \lceil \log n \rceil$$

Nous pouvons établir sans élagage, avec $m_i = m$, donc la complexité est en $0(m \log n) : M[YO - YO$ (without pruning) $] \leq 2m \log n$
Quelle est la complexité lorsque nous utilisons la procédure d'élagage. □

Chapitre

5 Terminaison

5.1 Introduction

Détecter qu'une application répartie (appelée calcul sous-jacent) est terminée consiste à détecter un état dans lequel d'une part tous les processus ou sites qui la constituent sont dans l'état passif et d'autre part tous les canaux de communication sont vides de messages relatifs à cette application.
Dans certains cas cela peut être simple, par exemple si l'application est pilotée par un site qui contrôle l'activité des autres sites. Mais ce que l'on recherche ici c'est une solution au cas général, c'est à dire sans faire aucune hypothèse sur la structure ou l'organisation du calcul sou-jacent. La détection va être réalisée par un algorithme de contrôle qui va observer le calcul sous-jacent. La correction de cette détection s'exprime par les deux propriétés classiques suivantes :

— propriété de sûreté : si l'observation indique que le calcul sous-jacent est terminé, alors ce calcul est effectivement terminé.
— propriété de vivacité : si le calcul sous-jacent se termine, alors l'algorithme de détection annoncera sa terminaison en un temps fini après qu'elle se soit produite.

5.1.1 Hypothèses sur le calcul sous-jacent

Le calcul sous-jacent est composé d'un nombre fini n de processus (un par site) et d'un certain nombre de canaux de communication ; les canaux de ce calcul sont supposés uni-directionnels et fiables, ils peuvent être ou non fifos. Les délais de transfert des messages sont quelconques. On appellera $voisins_i$ auquel i peut envoyer directement des messages.
Chaque processus est doté d'une variable locale $etat_i$ qui peut prendre la valeur $actif$ ou $passif$. Les règles de comportement associées aux processus sont les suivantes :

1. un processus doit être dans l'état $actif$ pour envoyer des messages.

2. un processus dans l'état $actif$ peut passer dans l"état $passif$.
3. lorsqu'un processus dans l'état $passif$ reçoit un message il repasse dans l'état $actif$.

5.1.2 Principes des algorithmes de détection de la terminaison

Détecter la terminaison requiert de visiter tous les sites et tous les canaux et de les observer respectivement passifs et vides de messages, et cela de façon cohérente. Pour cela l'algorithme de détection est décomposé en n contrôleurs, un par site. Ces contrôleurs doivent alors observer l'état des sites auxquels ils sont associés et l'état des canaux (vides ou non) qui connectent les sites.
Lrs différents algorithmes de détection se distinguent ainsi par :

— le type de coopération que réalisent les contrôleurs pour détecter que les canaux sont vides de messages du calcul sous-jacent et pour réaliser une observation globalement cohérente,
— les hypothèses particulières faites sur le calcul sous-jacent (c'est à dire les règles que l'on peut ajouter à 1, 2 et 3 pour particulariser un type de calcul).

5.2 Cas d'un anneau uni-directionnel

Le système est composé de n sites qui sont physiquement reliés par un anneau. Les communications ne peuvent se faire que dans un sens sur cet anneau. Chaque site connaît le nombre total n de sites. On suppose le réseau fiable et les liens FIFO (les messages sont reçus dans l'ordre envoyé).

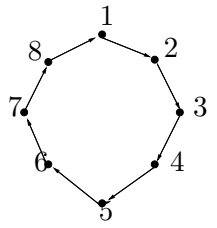

FIGURE 5.1 – Un anneau à 8 sites.

Une application répartie s'exécute sur les sites. Sur chaque site un *travail* consiste en du calcul, des envois et des réceptions de messages. Les messages échangés dans le cadre de l'application répartie sont dits *messages de travail*. A la fin d'un travail, un site s'arrête et ne redémarre que s'il reçoit un nouveau message de travail qui le ré-active. Tout le problème ici est de trouver un moyen pour détecter que

l'application est terminée ; c'est-à-dire détecter le moment à partir duquel tous les sites n'ont plus de travail et n'en auront plus à faire (plus aucun message de travail ne circule). un site détecte de manière sûre la terminaison. **Remarque :** il est important de noter qu'un site peut répartir que lorsqu'il reçoit un message de type travail. IL n'y a pas de redémarrage spontanémént. Nous allons donner le pseudo-code de l'algorithme de détection et dire quelques mots sur son fonctionnement. Ces quelques explications ne constituent pas une preuve formelle de bon fonctionnement. Pour réaliser la détection nous allons rajouter à l'application répartie, dont il faut détecter la terminaison, des variables et un message de *contrôle* qui seront indépendants de l'application. Les messages de travail ainsi que les mécanismes propres à l'application elle-même ne sont pas décrits. Chaque site p maintient les variables suivantes : $etat_p \in \{actif, passif\}$, $couleur_p \in \{noir, blanc\}$, $jeton\_present_p \in \{vrai, faux\}$. Le message de contrôle est $JETON$. Au début, un site est désigné pour envoyer le jeton vers le suivant.

```
Lors de la réception d'un message de travail
     etat_p := actif;
     couleur_p := noir;
     faire le travail...
A la fin d'un travail faire
     etat_p := passif;
     Si (Jeton_present_p) alors
          couleur_p := blanc;
          Envoyer(JETON, 1);
          Jeton_present_p := faux;
Lors de la réception d'un message < JETON, v >
     Si (etat_p = actif) alors
          Jeton_present_p := vrai;
     Sinon
          Si (v = n et couleur_p = blanc) alors
               Terminaison détectée;
          Sinon
               Si (couleur_p = blanc) alors
                    Envoyer(< JETON, v + 1 >);
               Sinon
                    couleur_p := blanc;
                    Envoyer(< JETON, 1 >);
          jeton_present_p := faux;
```

Les valeurs des variables $etat_p$ et $couleur_p$ peuvent être interprétées informellement de la manière suivante. $etat_p = actif$ lorsque p travaille et $etat_p = passif$ lorsque p ne travaille pas. $couleur = noir$ si p travaille ou si p a reçu un message

de travail pendant que le jeton n'était pas là. Cette variable sert donc d'indicateur de travail accompli pendant l'absence du jeton. Le jeton doit passer sur tous les sites pour constater si ceux ci travaillent ou pas. Il est une sorte d'inspecteur des travaux finis. Lorsque le jeton est sur un site en cours de travail ($actif$), il ne part avec la valeur 1 que lorsque ce site a terminé son travail. Chaque fois que le jeton passe par un site qui travaille, ou qui a travaillé pendant son absence ($noir$), sa valeur est remise à 1 à son départ. Ainsi lorsque le jeton arrive sur un site $blanc$ avec la valeur n la terminaison est effective car pendant un tour complet le jeton n'a pas rencontré de site noir, ce qui implique que tous les sites ont arrêté de travailler. Comme les envois de messages sont FIFO il n'y a plus de message en transit pouvant précéder le jeton.

5.2.1 Critiques de l'algorithme

Cet algorithme de détection est intéressant par sa simplicité : introduction du jeton, puis de sa couleur, puis de la couleur des processus permettant au processus P_0 d'avoir une image fiable de cet état global, jusqu'à la terminaison.
S'il possède ainsi un intérêt didactique certain, cet algorithme possède cependant trois inconvénients majeurs :

1. tout d'abord il impose une topologie de contrôle (l'anneau) qui vient se rajouter à la topologie des communications de l'algorithme distribué qui réalise le calcul : cela introduit des nouvelles voies de communication.
2. Ensuite un processus joue un rôle privilégié : c'est P_0 qui initialise et détecte la terminaison : cela introduit une dissymétrie sur les processus.
3. le dernier inconvénient enfin est lié à l'hypothèse d'instantanéité faite sur les transferts de messages : dans une implémentation ceci peut conduire à une restriction trop forte du parallélisme (pour réaliser cette instantanéité).

5.3 Cas général

Misra propose un algorithme de détection de la terminaison qui ne fait aucune hypothèse sur la topologie des voies de communication ni sur le délai de transfert des messages ; les seules hypothèses faites concernent la non-perte des messages et leur non-déséquencement : entre deux processus communicants, les messages sont reçus dans leur ordre d'émission. Avec ces hypothèses, la terminaison est réalisée lorsque tous les processus sont passifs et qu'il n'y a plus de messages en transit. Pour détecter la terminaison, Misra propose, comme dans les algorithmes précédents, de faire visiter les processus par un jeton ; comme on l'a vu précédemment, la constatation par le jeton que tous les processus étaient passifs lorsqu'ils ont été

visités, ne permet pas de conclure que la terminaison est réalisée : des processus ont pu être réactivés et des messages peuvent être en transit. Examinons le cas particulier de l'anneau : dans ce cas le jeton peut affirmer que le calcul est terminé, si, après un tour de visites sur l'anneau, il constate que chacun des processus est en permanence resté passif depuis la dernière visite qu'il lui a effectué. En effet, les messages ne pouvant se doubler, entre deux visites du jeton un processus a nécessairement reçu les messages (en provenance de son prédécesseur) qui étaient en transit lors de la première visite du jeton. Le fait que le jeton ait effectué deux tours sur l'anneau et qu'il ait sans cesse observé des processus restés en permanence passifs permet donc de conclure à la passivité des processus et à l'absence des messages en transit, c'est-à-dire à la terminaison. Pour mettre en place ce principe dans un algorithme deux problèmes sont à résoudre :

— Tout d'abord capter le fait qu'un processus est resté passif en permanence depuis la dernière visite de l'anneau,
— Ensuite éliminer la contrainte de l'anneau.

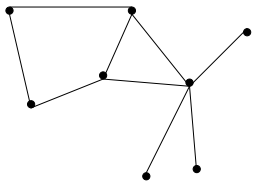

FIGURE 5.2 –

Comme précédemment des couleurs associées aux processus, vont permettre de résoudre le premier problème. Lorsqu'il devient actif un processus devient noir, c'est le jeton qui peint en blanc lorsqu'il quitte le processus. Ainsi le jeton retrouve blanc un processus c'est ce que ce dernier est resté passif en permanence depuis, la dernière visite. Lorsque le jeton a visité les n processus et qu'il les a tous trouvé blancs il peut en conclure la terminaison (initialement les processus sont noirs). Comme on le voit, l'accent est mis sur le comportement du jeton, et tous les processus ont des comportements identiques : il n'y a pas de processus P_0 jouant un rôle particulier. Tout processus peut initialiser une détection avec un jeton (auquel il peut donner son identité afin de ne pas confondre avec d'autres jetons). Le second problème à régler est d'admettre une topologie quelconque pour les communications et de ne pas se limiter à un anneau. La solution est simple : le jeton doit d'une part visiter tous les processus et d'autre part s'assurer qu'il n'y a plus de messages en transit. Pour cela il doit donc parcourir chaque arc du réseau que forme les processus et leurs voies de communications Supposons que le réseau

est fortement connexe. Dans une telle structure il existe un circuit C qui comprend chaque arc du réseau (au moins une fois). Il suffit alors de remplacer l'anneau par un tel circuit.
Les états initiaux sont :

- $couleur \in \{blanc, noir\}$ initailisé à $noir$,
- $etat \in \{actif, passif\}$ initialisé à $actif$
- $jetonpresent \in \{true, false\}$ initialisé à $faux$
- nb sert à mémoriser la valeur associée au jeton enter sa réception et sa réémission initialisé à 0 .
- une fonction $taille(C : circuit), c \to N$ donne la taille du circuit,
- une fonction $successeur(C : circuit, i = 1 \ldots n)$ donne pour le processeur p_i le nom du successeur sur le circuit.

5.3.1 L'algorithme

Voici le code pour un sommet P_i :

```
Lors de la réception d'un message du type (message, m)
     etat ← actif;

    couleur ← noir;
Lors de l'attente d'un message du type (message, m)
     etat ← passif

Lors de la RECEPTION d'un message du type (JETON, j)
     nb ← j

    jetonprsent ← true
    Si nb = taille(C) et couleur = blanc
        Alors terminaison détectée
Lors de l'émission d'un message du type (JETON, j)
(*** Possible seulement si jetonpresent et etat = passif :
***)
     Si couleur = noir      Alors nb ← 0

         Sinon nb ← nb + 1
    ENVOYER (jeton, nb) à successeur(C, i)
    couleur ← blanc
    jetonpresent ← false
```

On peut étendre facilement cet algorithme n'est pas fortement connexe : le réseau

est décomposé en ses composantes maximales fortement connexes : le jeton visite alors successivement ses composantes selon un ordre topologique.
Un inconvénient de l'algorithme provient de la nécessité d'avoir préalablement défini un circuit qui inclut tous les arcs du graphe des communications. Celui-ci peut-être défini à la compilation de l'algorithme distribué car on dispose alors de toutes les informations nécessaires. L'établissement de manière distribué d'un tel circuit reste un problème d'actualité ; deux choix évidents y sont possibles : en profondeur d'abord ou en largeur d'abord.

5.4 L'algorithme de Mattern

On se place dans le cadre d'un modèle atomique pour le comportement des processus c'est à dire dans lequel leurs temps de calcul sont considérés comme étant de durée nulle. Seules les communications prennent du temps. La réception d'un message par un site et l'envoi des messages qu'entraîne le traitement associé sont donc considérés comme seule action qui ne prend pas de temps. L'exécution d'un calcul peut être représentée par un diagramme de temps comme indiqué sur la figure 5.3 (Un point y représente un événement du type réception + émission). Dans ce modèle de comportement le problème de la terminaison revient donc à vérifier si tous les canaux sont simultanément vides.

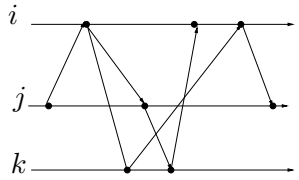

FIGURE 5.3 – Une exécution dans le modèle atomique.

L'algorithme de Matterné Le principe consiste à compter le nombre message total E de messages émis, le nombre total R de messages reçus pour conclure à la terminaison lorsque ces deux nombres sont égaux.
Un observateur envoie sur le chacun des sites P_i une requête pour que ceux-ci indiquent les nombres E_i et R_i des messages respectivement émis et reçus depuis le début et attend les résultats. On dit ainsi qu'il a effectué une vague d'observation.

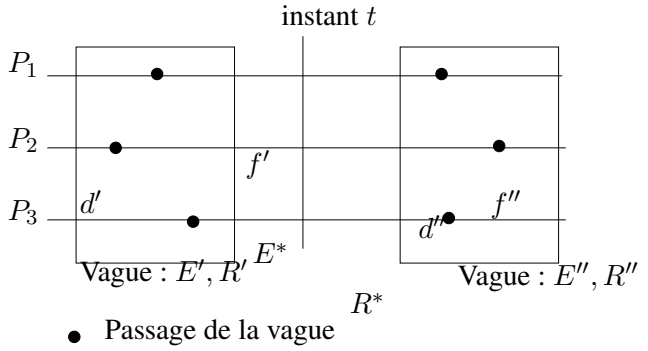

FIGURE 5.4 – *Illustration d'une vague*

Soient $E = \sum_{i=1}^{n} E_i$ et $R = \sum_{i=1}^{n} R_i$ le nombre de message emis et reçus au total.
Pour cela le contrôleur C_i placé sur chaque site i compte les nombres de messages qui ont été respectivement émis et reçus par le processus placé sur ce site. Un contrôleur particulier est chargé de collecter ces nombres, appelons α le site sur lequel est placé ce contrôleur. C_α diffuse donc une demande à chacun des contrôleurs. On appelle t_i l'instant réel auquel le contrôleur C_i reçoit la demande; il y répond en renvoyant les valeurs $e_i(t_i)$ et $r_i(t_i)$ qui représentent les nombres de messages respectivement émis et reçus par i jusqu'à la date t_i. Lorsque C_α a reçu toutes les réponses il calcule : $E1 = \sum_i e_i(t_i)$ et $R1 = \sum_i r_i(t_i)$. Il est facile de voir que si $E1 \neq R1$ on ne peut conclure à la terminaison.

Théorème 5.4.1 *Si nous avons $E' = R''$ alors nous pouvons conclure à la terminaison.*

Preuve
Dans un premier temps, nous allons montrer que dans le cas où $E = R$, nous ne pouvons pas conclure sur le problème terminaison.
Malheureusement non, comme le montre le contre exemple suivant, figure 5.5
On a effet $E1 = R1$ avec : $e_1(t_1) = 0, e_2(t_2) = 1, e_3(t_3) = 0$ et $r_1(t_1) = 1, r_2(t_2) = 0, r_3(t_3) = 0$. La droite V sur le diagramme sépare le calcul observé en deux parties : pour chaque site il y a avant l'observation et après l'observation. Si les 3 sites étaient observés simultanément, i.e. $t_1 = t_2 = t_3$, on pourrait conclure à la terminaison à partir de l'égalité $E1 = R1$. Tout le problème de l'observation dans un contexte réparti provient du fait que l'on ne peut garantir que les sites sont observés simultanément : il n'y a pas de référentiel de temps commun à l'ensemble des sites et les messages nécessaires pour réaliser l'observation ont des temps de transit arbitraires et a priori indépendants les uns des autres. Aucun

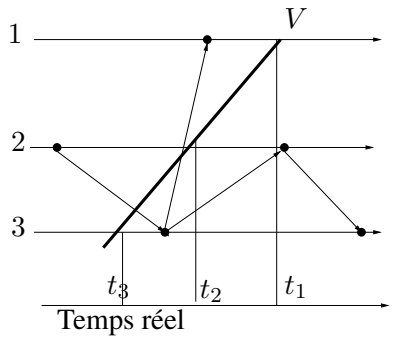

FIGURE 5.5 – Une observation repartie

site n'a connaissance de tous les instants réels $t_1,\ t_2$ et t_3, l'hypothèse de ce temps global ne peut donc servir qu'à l'analyse du problème et pas à l'algorithme qui le résoud !

Un observateur fait deux vagues d'observation, ne lançant la seconde vague que lorsqu'il a obtenu tous les résultats de la première. Nous notons E' et R' les résultats de la première vague, E'' et R'' les résultats de la seconde vague. Nous notons par t un instant entre la fin de la première vague et le début de la seconde. Soient E^* et R^* le nombre exacts de messages qui ont été émis et reçu avant t. Donner une relation entre R', R^*, E^* et E''.

Nous allons montrer que si l'observateur constate que $E' = R''$ alors on peut conclure qu'aucun message n'était en transit à l'instant t.

Est-ce que C_α peut conclure à la terminaison ?

Pour résoudre ce problème nous allons utiliser une vague V de contrôle : il y a un initiateur, ici C_α, qui lance la vague par l'envoi des questions ; la vague est terminée lorsque celui-ci a reçu toutes les réponses. Combien de vagues sont nécessaires ? Donner un critère de terminaison en fonction des variables utilisées.

La solution proposée par Mattern est simple et élégante, elle consiste à utiliser deux vagues successives. Appelons $(E1, R1)$ et $(E2, R2)$ les deux couples de valeurs collectées par les deux vagues successives V_1 et V_2. Nous allons voir que $R1 = E2 \Rightarrow terminaison$. En d'autre termes le test $R1 = E2$ garantit la propriété de sûreté. La vivacité est facile à montrer : lorsque l'application est terminée il n'y a plus de messages de calcul en transit et les deux premières vagues lancées après la terminaison seront donc telles que $R1 = E2$. On appelle $d1$ (resp ; $d2$) l'instant réel auquel est lancée la première vague (resp ; la seconde) et $f1$ (resp. $f2$) celui auquel la promière vague (resp. la seconde) se termine. Comme les deux vagues sont séquentielles on a $d1 < f1 < d2 < f2$

Les compteurs utilisés sont monotones. On a donc pour un observateur omniscient

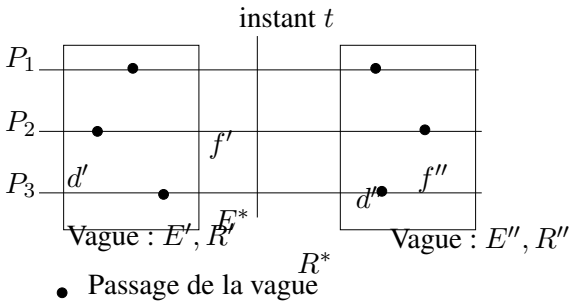

FIGURE 5.6 – Deux vagues consécutives

qui serait sur tous les sites simultanément :

$$(P1) e_i(t) \leq e_i(t') \text{ et } r_i(t) \leq r_i(t'), \forall t \leq t', \forall i$$

Si on apelle $E(t)$ et $R(t)$ les deux compteurs abstraits, inévaluables par l'algorithme, représentant les nombres totaux de messages émis et reçus à l'instant réel t on a :

$$(P2) E(t) \leq E(t') \text{ et } R(t) \leq R(t'), \forall t \leq t'$$

On a de plus : le nombre de messages en transit à un instant donné est positif ou nul c'est à dire

$$(P3) R(t) \leq E(t), \forall t$$

. Les valeurs collectées par la première vague sont relatives à des instants inférieurs ou égaux à $f1$; cette remarque jointe à $P1$ permet de conclure

$$(P4) R1 \leq R1(f1)$$

. De manière analogue les valeurs collectées par la seconde vague sont postérieures à la date $d2$ d'où l'on conclut que

$$(P4) E(d_2) \leq E2$$

. Ces cinq propriétés sur les compteurs vont permettre d'établir la sureté. En effet,

$$R1 = E2 \Rightarrow E(d2) \leq R(f1)$$

par $P4$ te $P5$ et

$$E(d2) \leq R(f1) \Rightarrow E(f1) \leq R(f1)$$

par $P2$ et $f1 < d2$: $E(f1) \leq E(d2)$, or par $P3$ on a

$$E(f1) \geq R(f1)$$

;
d'où l'on conclut que $E(f1) = R(f1)$ c'est à dire qu'à dire la fin la première vague (instant global $t1$) il n'y avait pas de message en transit.
Nous avons $R' \leq R^* \leq E^* \leq E''$.
D'autre part, nous supposons qu'une réponse à une demande d'information ne quitte un site que lorsque ce site est au repos, et que la réponse envoyée tient compte de tous les envois et réception de message jusqu'à ce moment de repos. Nous supposons qu'après la deuxième vague, on ait $E'' = R'$ (nous sommes dans le cas où il n'y a aucun message reçu après t), on a alors aussi : $E'' = E^*$ (grâce à l'inégalité précédente) ; cela signifie qu'aucun site n'a envoyé de message entre l'instant t et le moment de sa réponse à la deuxième vague ; notons S le premier site réactivé après le passage de la deuxième vague ; S a été activé par la réception d'un message envoyé par un site S' ; l'envoie de ce message ne peut avoir lieu entre l'instant t et le passage de la seconde vague sur S' ni non plus avant l'instant t car alors le message aurait été en transit à l'instant t. S' (car nous sommes dans le cas de l'hypothèse, où il n'y a aucun message reçu après t) a donc envoyé son message après le passage de la seconde vague ; il y a donc eu réactivation de S' après le passage de la seconde vague et avant la réactivation de S : d'où une contradiction sur le choix de S. En conséquence, aucun site ne sera réactivé après le passage de la seconde vague. De plus, s'il y a terminaison, deux successives donneront bien $R'' = E'$ lors de la seconde vague.
Remarque 1 : Ce type de raisonnement est caractéristique du réparti. On cherche à conclure sur une propriété du système à un instant global t, la même sur tous les sites. Comme on ne peut garantir que tous les sites seront observés simultanément on calcule des approximations des valeurs recherchées. La preuve consiste alors à montrer avec ces valeurs approchées qu'il a existé un instant t auquel la propriété recherchée était vérifiée.
Remarque 2 : Si la terminaison n'est pas détectée par ls vagues $V1$ et $V2$, il faut en lancer une autre $V3$ et on compare les valeurs collectées par $V2$ et $V3$ et ainsi de s suite. Dans tous les cas, i faut toujours deux vagues après la terminaison pour pouvoir la détecter. De plus l'algorithme ne donne aucune information, dans ce cas où il ne conclut pas à la terminaison, sur l'instant auquel il faudrait démarrer la prochaine vague afin d'avoir des chances de détecter la terminaison : le contrôleur C_α ne dispose d'aucune information sur les instants auxquels il doit lancer une vague.

□

Chapitre

6 Consensus

Dans ce chapitre nous allons aborder des problèmes de prise de décisions dans un système réparti. Dans de nombreuses situations, les sites doivent se mettre d'accord sur une valeur commune. Ces situations se rencontrent souvent suite à des pannes, après lesquelles on veut s'accorder sur une valeur commune pour reprendre automatiquement une exécution par exemple. Le contexte de panne est très important et a beaucoup été étudié. Des solutions (algorithmes) ont été proposées pour résoudre le problème du consensus malgré des pannes.
Cependant, ces algorithmes font souvent appel à des mécanismes de *temporisation*. On considère donc que le système n'est pas complètement asynchrone et qu'il existe un temps maximum de réponse de la part des sites impliqués. Cette supposition permet en fait de savoir si un site est en panne ou non.

6.1 Gestion de pannes : l'algorithme des généraux byzantins

6.1.1 Introduction

Les systèmes informatiques fiables doivent être capables de gérer des composants défectueux qui donnent des informations conflictuelles aux différentes parties du système. Ces composants peuvent être soit des ordinateurs reliés entre eux par un réseau (c'est alors celui-ci que l'on considère comme système) ou bien des processeurs dont la redondance permet de garantir une certaine sécurité dans les systèmes critiques. La gestion de ces composants défectueux est aussi appelée gestion de pannes.
Le problème de composants défectueux dans un système informatique (ou ailleurs) peut être exprimé de façon abstraite en terme de généraux de l'armée Byzantine qui campent autour d'une cité ennemie. Ne communicant qu'à l'aide de messagers, ceux-ci doivent se mettre d'accord sur un plan de bataille commun, sinon la

défaite est assurée. Mais il se peut que l'un ou plusieurs de ces généraux soient des traîtres, qui essayent de semer la confusion parmi les autres. Le problème est donc de trouver un algorithme pour s'assurer que les généraux loyaux arrivent tout de même a se mettre d'accord sur un plan de bataille.
Nous allons donc montrer que, en utilisant uniquement des messages oraux, ce problème peut être résolu, si, et seulement si plus des deux tiers des généraux sont loyaux ; ainsi un seul traître peut confondre deux généraux loyaux. Avec des messages écrits non modifiables, le problème peut être résolu pour nombre quelconque de traître. Tout au long de cette étude nous tenterons de voir les analogies avec les systèmes informatiques.

6.1.2 Le problème des Généraux Byzantins

Imaginons que plusieurs divisions de l'armée Byzantine campent autour de la cité ennemie, chacune d'entre elle étant dirigée par son propre général. La seule façon de communiquer dont ils disposent est l'utilisation de messagers. Après avoir observer l'ennemi, ils doivent se mettre d'accord sur un plan d'action commun. Le problème est que certains de ces généraux peuvent être des traîtres, qui tentent d'empêcher les généraux loyaux de se mettre d'accord.
Les généraux doivent donc disposer d'un algorithme pour garantir que :

1. Tous les généraux loyaux se mettent d'accord sur le même plan d'action. Les généraux loyaux feront tous ce que l'algorithme leur a dit de faire, mais les traîtres peuvent faire ce qu'ils veulent. L'algorithme doit donc garantir la condition A, et ce sans se préoccuper de ce que les traîtres choisissent de faire. Les généraux loyaux ne doivent pas seulement trouver un accord, mais aussi trouver un plan raisonnable.
2. Un petit nombre de traître ne peut pas faire que les généraux loyaux choisissent un mauvais plan.

La dernière condition est difficile à formaliser étant donné qu'elle nécessite de définir ce qu'est un mauvais plan. Nous nous contenterons donc d'étudier la façon dont ils arrivent à se mettre d'accord.
Chaque général observe l'ennemi et communique ces observations aux autres. Soit $v(i)$ l'information communiqué par le ième général. Chaque général doit donc utiliser une méthode pour combiner les valeurs $v(1), \ldots, v(n)$ en un plan d'action unique, avec n le nombre de généraux. La première condition peut être obtenue si tous les généraux utilisent la même méthode pour combiner les informations, et la seconde condition peut être obtenue en utilisant une méthode "robuste ".
Par exemple, si la décision qui doit être prise est soit Attaque ou Retraite, alors v(i) peut être la décision du général i et la décision finale peut être basée sur la majorité

des de ces décisions. Des traîtres peuvent modifier cette décision seulement si les généraux loyaux étaient divisés de manière égale quant à la décision à prendre, auquel cas aucune des décisions ne peut être considérée comme mauvaise.
Bien que cette approche puissent ne pas être la seule façon de satisfaire les conditions 1 et 2, c'est la seule connue. Elle suppose qu'il existe une méthode qui permette aux généraux de communiquer leurs valeurs $v(i)$ aux autres. Une méthode évidente est, pour le ième général, d'envoyer $v(i)$ par messager aux autres généraux. Malheureusement, cela ne fonctionne pas car pour que la condition 1 soit satisfaite, il faut que tous les généraux obtiennent le même ensemble de messages $v(1), \ldots, v(n)$, et un traître peut très bien envoyer des messages différents à chacun des autres généraux. Pour que la condition 1 soit satisfaite, il faut que la condition suivante soit vraie :
Tous les généraux loyaux doivent obtenir les mêmes informations $v(1), \ldots, v(n)$. Cette condition implique qu'un général loyal, ne va pas forcément utiliser la valeur $v(i)$ reçue de i, puisque si le ième général est un traître, il a très bien pu envoyer des valeurs différentes à chacun. Cela signifie donc que, si l'on souhaite vérifier la condition 1 et qu'on ne fait pas attention, il est possible que les généraux utilisent une valeur de $v(i)$ différente de celle envoyée par i,et ce même si le général i est loyal. Il ne faut pas permettre cela si nous souhaitons vérifier la condition 1. Nous avons alors de plus la nécessité suivante :
Si le ième général est loyal, alors la valeur qu'il a envoyé doit être utilisée par tous les généraux loyaux comme étant $v(i)$. Nous pouvons alors réécrire la condition 1 comme ceci (que le ième général soit loyal ou pas) :

1. Deux généraux loyaux quelconques utilisent la même valeur pour v(i).

Résultat important :
Les conditions $1'$ et 2 portent toutes deux sur une même valeur envoyée par le ième général. On peut alors restreindre notre étude du problème à : comment un seul des généraux envoie-t-il sa valeur aux autres ?
On formulera cela en terme de commandant qui envoie un ordre à ces lieutenants, ce qui nous amène au problème suivant.
Définition du Problème des Généraux Byzantins
Un Général commandant doit envoyer un ordre à ses n-1 lieutenants, de manière à ce que :

- $IC1$: Tous les lieutenants loyaux obéissent au même ordre.
- $IC2$: Si le général est loyal, alors chaque lieutenant doit obéir à l'ordre qu'il a envoyé.

$IC1$ et $IC2$ sont connues comme les conditions de consistance interactive (Interactive Consistency conditions). Il faut remarquer que si le commandant est loyal, alors $IC2$ implique $IC1$. Mais bien sûr, le commandant peut très bien être un traître.

Ce problème, du point de vue informatique, peut être représenté de la façon suivante :
Soit un réseau de n processeurs qui peuvent communiquer les uns avec les autres seulement par le biais de messages, à travers des canaux de communications bidirectionnel, il faut s'assurer qu'un processeur envoie des données aux $n-1$ autres processus, de telle façon que :

- $IC1$: Les processeurs fiables reçoivent les mêmes données.
- $IC2$: Si le processeur émetteur est fiable, alors la valeur reçue est celle qui a été envoyée.

On voit bien que le problème de non-fiabilité que l'on cherche à résoudre peut alors provenir soit du processeur lui-même, soit de la liaison de donnée. Les deux cas n'ont donc pas à être différenciés. Si une liaison n'est pas fiable, alors le processeur sera considéré comme non fiable. Il faut de plus remarquer que dans le cadre de systèmes informatiques, il est plus probable que des données ne soient pas envoyées, plutôt que fausses.
Il s'agit donc plutôt d'erreurs par omissions, dues au crash d'un processeur par exemple.
Note : ici le mot processeur est employé dans sa définition globale, c'est à dire qu'il s'agit d'une entité effectuant un traitement sur des données, il peut très bien désigner un processeur, un processus, ou encore un ordinateur.

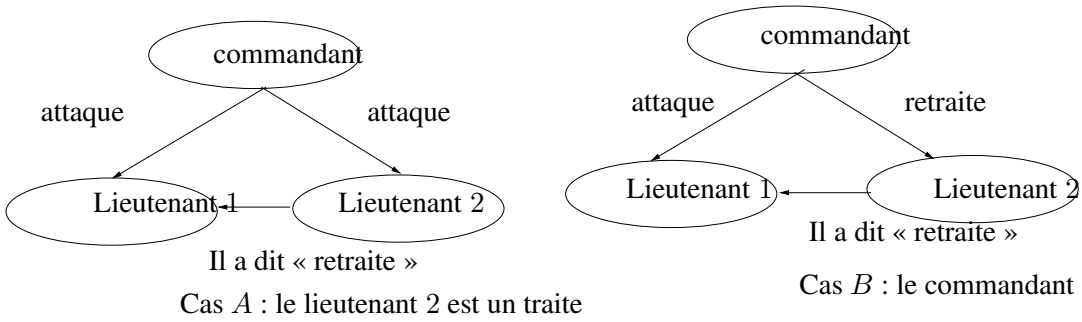

FIGURE 6.1 – *Généraux byzantins : impossibilité*

Il est impossible pour le Lieutenant 1 de savoir qui est le traître, car dans les deux cas il reçoit les mêmes informations.
Dans le cas A, pour satisfaire $IC1$ le Lieutenant 1 doit attaquer.
Dans le cas B, si le Lieutenant 1 attaque, il viole $IC2$.
IL N'EXISTE PAS DE SOLUTIONS POUR TROIS GENERAUX EN LA PRESENCE D'UN TRAÎTRE.

La preuve est en dehors du cadre de ce travail, mais les lecteurs intéressés pas celle ci peuvent se reporter à [1].
Observons maintenant des algorithmes permettant de résoudre ce problème. Dans la suite de ce document, n représentera le nombre total de généraux impliqués dans la communication, et m le nombre de traîtres parmi eux.

6.1.3 Solution avec des messages oraux $(n > 3m)$

Il nous faut tout d'abord définir les suppositions suivantes sur les messages (les remarques entre parenthèses correspondent aux conséquences sur un système informatique) :

- $S1$: Chaque message envoyé est délivré correctement (pas de pertes de messages).
- $S2$: Le destinataire d'un message sait qui lui a envoyé (réseau complètement connecté avec liaisons fiables (du fait de S1)).
- $S3$: L'absence d'un message peut être détectée (les systèmes doivent être synchrones, notion de Time-Out).

Les suppositions S1 et S2 permettent d'empêcher un traître d'interférer dans une communication entre deux autres généraux. S3 permet d'éviter qu'un traître ne sème la confusion en n'émettant pas de messages.
L'algorithme
Cet algorithme est appelé $OM(m)$ (Oral Message), mais on le trouve aussi sous la forme $UM(n, m)$ (Unsigned Message).
On suppose qu'une valeur par défaut vdef est définie pour le cas ou aucun message n'est envoyé par un traître. On suppose aussi qu'une fonction majorité à été définie telle que : majorité$(v_1, v_2, ..., v_{n-1}) = v$ si la majorité des valeurs $v_i = v$.

Algorithme $OM(0)$ (cas où il n'y a pas de traître)
    Le commandant envoie v à chacun des $n - 1$ lieutenants
    Chaque lieutenant utilise la valeur reçue du commandant ou vdef si il n'a rien reçu

Algorithme $OM(m)$ (cas où il y a m traîtres)
    Le commandant envoie v à chacun des $n - 1$ lieutenants
    Pour Chaque Lieutenant i,
        Soit v_i = valeur reçue du commandant ou vdef si aucune valeur n'a été reçue
        Envoyer v_i aux $n - 2$ lieutenants en utilisant

$OM(m-1)$
`Pour chaque` i `et chaque` $j \neq i$`,`
`Soit` v_j `= valeur que Lieutenant` i `a reçue du Lieutenant` j `à l'étape précédente ou vdef si il n'a rien reçu`
`Lieutenant` i `utilise la valeur majorité`$(v1, ..., vn-1)$
.
Exemple (tiré de [2]) :
Prenons le cas où $n = 4$ et $m = 1(OM(1))$.

— Premier cas : le Lieutenant 3 est un traître.

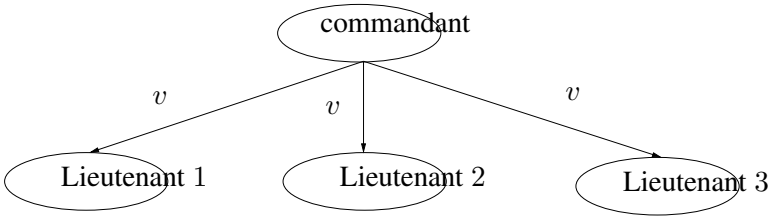

FIGURE 6.2 – *Après la première étape*

A la fin de l'étape 1, nous avons
— lieutenant 1 $v_1 = v$
— lieutenant 2 $v_2 = v$
— lieutenant 3 $v_3 = v$

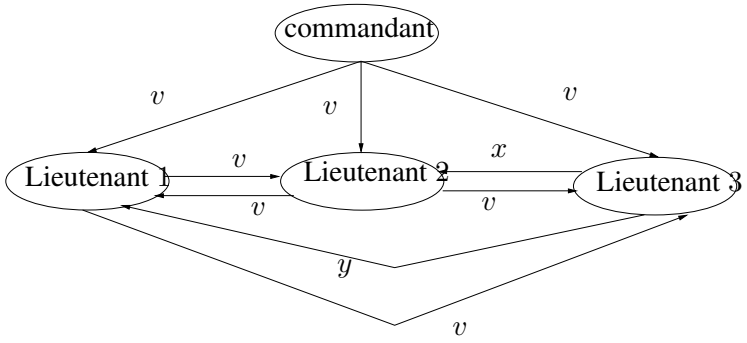

FIGURE 6.3 – *Après la dernière étape*

— lieutenant 1 $v_1 = v, v_2 = v, v_3 = y$
— lieutenant 2 $v_1, = v, v_2 = v, v_3 = x$

— lieutenant 3 $v_1 = v, v_2 = v, v_3 = v$

Il est possible que $x = y$, de même qu'il peut s'agir d'une absence de message. A la fin de la dernière étape chacun des lieutenants a reçu un ensemble de valeurs et arrive à la même décision $IC1$. LA valeur envoyé par la commandant est la bien la valeur majoritaire $IC2$.

— Second cas : le commandant est un traître

A la fin de l'étape 1, nous avons

— lieutenant 1 $v_1 = x$
— lieutenant 2 $v_2 = y$
— lieutenant 3 $v_3 = z$

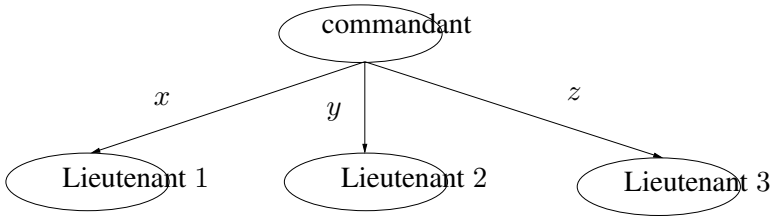

FIGURE 6.4 – *Après la dernière étape*

— lieutenant 1 $v_1 = x, v_2 = y, v_3 = z$
— lieutenant 2 $v_1, = x, v_2 = y, v_3 = z$
— lieutenant 3 $v_1 = x, v_2 = y, v_3 = z$

Dans la dernière étape les trois lieutenants ont reçu la même valeur majorité(x, y, z). et les contraintes $IC1$ et $IC2$ sont donc respectées.

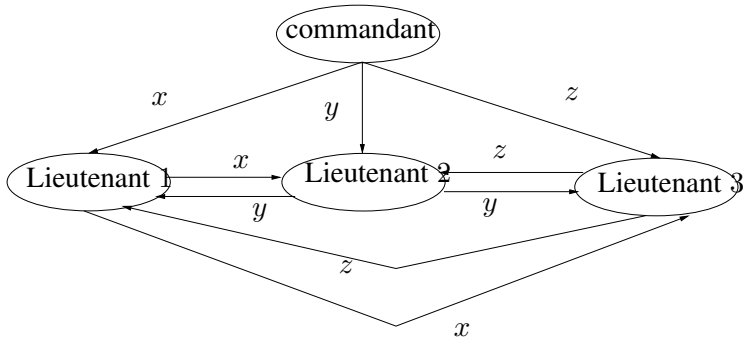

FIGURE 6.5 – *Après la dernière étape*

Théorème 6.1.1 *Pour tout m et k, l'algorithme $OM(m)$ satisfait $IC2$ si il y a plus de $2k + m$ généraux et au plus k traîtres.*

Preuve

— $m = 0$, $OM(0)$ fonctionne uniquement si : le général est fiable, et si les messages sont acheminés correctement.

— $m > 0$: Supposons par hypothèse de récurrence que $OM(m-1)$ satisfait la condition IC2. Prouvons que $OM(m)$ satisfait la condition IC2.
Regardons le comportement de l'algorithme :

— A l'étape 1 : l_0 (le général) envoie $< v >$ à tous ses lieutenants.

— A l'étape 2 : chaque lieutenant loyal l_j exécute $OM(m-1)$. De plus, on peut noter que $n - 1 > 2k + (m-1)$ car $n > 2k + m$ et $m > 0$.
donc on peut appliquer l'hypothèse de récurrence. Chaque lieutenant loyal l_j obtient $v_j = v$ de chaque lieutenant loyal l_j. Sa liste $(v_1, \ldots, v_{n-1})$ contient $1 + n - 1 - k = (n - k)$ éléments ayant la. valeur v.
Donc pour que le lieutenant loyal l_i choisit v, il faut que la valeur v soit majoritaire dans $(v_1, \ldots, v_{n-1})$ il faut $n - k > n/2$. Comme $n > 2k+m$, on a $n-1 > 2k+(m-1)$ et $m > 0$, donc on a $k > (n+1)/2$; alors $n - k = n - n/2 + 1/2 > n/2$. Donc v est majoritaire dans $(v_1, \ldots, v_{n-1})$ et chaque lieutenant loyal l_i choisit la valeur v. Ce qui vérifie la condition IC2.

□

Théorème 6.1.2 *Si il y a n généraux avec m byzantins ($n > 3m$), l'algorithme $OM(m)$ satisfait les deux conditions suivantes :*

— *IC1 : tous les lieutenants loyaux prennent la même décision processus normaux doivent connaître/prendre la décision de général ;*

— *IC2 : « si le général l_0 n'est pas byzantin, alors les lieutenants loyaux aboutissent à la même décision que celle du général »*

Preuve Encore une fois la preuve se fait en raisonnant par récurrence sur m.
Si il n'y a pas de traîtres, il est alors facile de démontrer que $OM(0)$ satisfait $IC1$ et $IC2$.
Nous supposons alors que le théorème est vrai pour $OM(m-1)$ et prouvons qu'il est vérifié pour $OM(m), m > 0$.
Considérons tout d'abord le cas où le commandant est loyal.
En prenant $k = m$ dans le Lemme précédent, nous pouvons voir que $OM(m)$ satisfait $IC2$. $IC1$ est impliqué par $IC2$ puisque le lieutenant est loyal, donc nous n'avons en fait qu'à considérer le cas où le commandant est un traître. Il y a au

plus m traîtres et le commandant est l'un d'entre eux, donc au plus $m-1$ lieutenants sont des traîtres. Puisqu'il y a plus de $3m$ généraux, il y a plus de $3m-1$ lieutenants, et $3m-1 > 3(m-1)$. Nous pouvons alors appliquer l'hypothèse de récurrence pour déduire que $OM(m-1)$ satisfait $IC1$ et $IC2$.
Ainsi pour chaque j, chaque groupe de deux lieutenants obtient la même valeur pour v_j à l'étape (3) (cela se déduit par $IC2$ si l'un des deux lieutenants est le lieutenant j, et par $IC1$ dans les autres cas). Ainsi, chaque groupe de deux lieutenants obtient le même vecteur de valeurs $v_1, \ldots, v_{n-1}$ et obtient donc la même valeur majorité $(v_1, \ldots, v_{n-1})$ à l'étape (3), ce qui prouve $IC1$.
Une façon de voir la preuve :

— Si j est un byzantin, alors pour toutes les valeurs v_j des lieutenants loyaux sont identiques (par IC1).
— Si j n'est pas byzantin, alors pour toutes les valeurs v_j des lieutenants loyaux sont identiques (par IC2) ? Donc tous les lieutenants loyaux ont la même liste $(v_1, \ldots, v_{n-1})$.

Comme ils appliquent la même fonction majorité, ils obtiennent le même résultat. Ce qui vérifie la condition IC2.

□

Lemme 6.1.1 *La complexité est en $O(n^{m+1})$.*

Preuve La mise en oeuvre de l'algorithme $OM(m)$ cause tout d'abord l'émission de $n-1$ messages. Chacun de ces messages invoque $OM(m-1)$ causant l'émission de $n-2$ nouveau messages etc.

$$\begin{array}{cc} (1) & (n-1) \\ (2) & (n-1)(n-2) \\ \ldots & \\ (m+1) & (n-1)(n-2)\ldots(n-(m+1)) \end{array}$$

et donc le total est $O(n^{m+1})$.

□

Le résultat le plus important à retenir ici, en dehors de l'algorithme bien sûr, est que le Problème des Généraux Byzantins est solvable si $n > 3m$ (dans le cas de l'utilisation de messages oraux).

6.1.4 Solution avec des messages écrits et signés (m quelconque)

La difficulté à résoudre le problème des 3 généraux se trouvent dans la capacité d'un lieutenant traître de mentir à propos de l'ordre reçu du commandant, ainsi, si nous pouvons restreindre cette capacité en ajoutant les suppositions suivantes aux trois précédentes (S1, S2 et S3), le problème des trois généraux est alors solvable pour un nombre quelconque de traîtres.

— La signature d'un général loyal ne peut pas être imitée, et toute modification du contenu d'un message peut être détectée (un message peut être supprimer, mais pas modifié).
— N'importe qui peut vérifier l'authenticité d'un message (personne ne peut tromper un général).

Cela s'applique facilement à des systèmes informatiques par l'utilisation de signatures numériques (souvent associées aux algorithmes de cryptographie moderne). Ce type de solution va donc nous intéresser tout particulièrement.
Maintenant que nous avons introduit les messages signés, l'argument précédent qui faisait que nous avions besoin de quatre généraux pour pouvoir tolérer la présence d'un traître n'a plus de raison d'être. Nous donnons maintenant un algorithme capable ce gérer m traîtres avec un nombre quelconques de généraux (le problème étant cela dit sans intérêts si il y a moins de $m + 2$ généraux).
Dans cet algorithme, le commandant envoie un message signé à chacun de ces lieutenants. Chaque lieutenant appose ensuite sa signature sur cet ordre et l'envoi aux autres, qui font alors de même, etc. Cela signifie qu'un lieutenant doit recevoir un message, en faire des copies, les signer, puis les envoyer. Peu importe comment les copies sont effectuées.
L'algorithme suivant suppose que l'on définisse une fonction choix qui s'applique à un ensemble d'ordre afin d'en obtenir un seul. Les seuls conditions pour cette fonction sont :

— Si l'ensemble V contient un unique élément v, alors $choix(V) = v$.
— $choix() = vdef$.

Notation : $v : j : i$ représente la valeur v signé par j, puis la valeur $v : j$ signé par i.
On considère que le général0 est le commandant.
Chaque lieutenant conserve un ensemble V_i d'ordres signés corrects qu'il a reçu jusqu'alors (avec un commandant loyal, cet ensemble ne doit pas contenir plus d'un seul élément).

Algorithme $SM(m)$
Initialement $V_i = \{\}$
. (1) Le commandant envoie sa valeur signée à chacun des lieutenants

(2) Pour chaque i :
Si le lieutenant i reçoit un message v : 0 et qu'il n'a pas encore reçu d'autres ordres, alors :
il fixe V_i à $\{v\}$
il envoie $v:0:i$ à chacun des autres lieutenants
Si le lieutenant i reçoit un message de la forme v : 0 : j_1 : ... : j_k et v n'est pas dans l'ensemble V_i alors
$V_i = V_i + \{v\}$
Si $k < m$ alors envoyer le message $v:0:j_1:...:j_k:i$ à chacun des lieutenant autres que $j_1, ..., j_k$
(3) Pour chaque i :
Quand il n'y a plus de messages, le lieutenant i obéit à l'ordre choix(V_i)

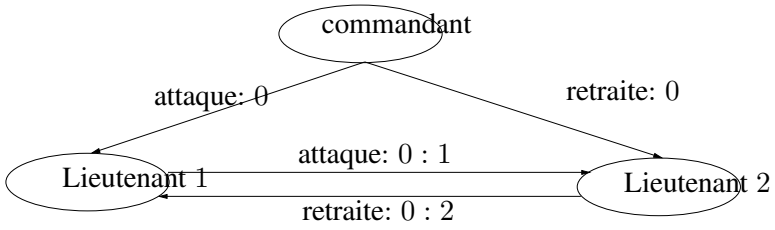

FIGURE 6.6 – *Exemple*

$V1 = V2 = choix(attaque, retraite)$
Note : avec des messages signés, les lieutenants peuvent détecter que le commandant est un traître du fait que sa signature apparaît sur deux ordres différents, et d'après S4 il est le seul à avoir pu les signer.

Théorème 6.1.3 *Pour chaque m, l'algorithme $SM(m)$ résout le Problème des Généraux Byzantins si il y a au plus m traîtres.*

Preuve Prouvons tout d'abord $IC2$. Si le commandant est loyal, alors il envoie son ordre signé $v : 0$ à chacun des lieutenants à l'étape (1). Chaque lieutenant va alors recevoir l'ordre v à l'étape (2)(A). De plus, puisqu'aucun traître ne peut imiter un ordre de la forme $v' : 0$, un lieutenant loyal ne peut pas recevoir d'autre ordre à l'étape (2)(B). Ainsi, pour chaque lieutenant loyal i, l'ensemble V_i obtenu à l'étape (2) consiste en un unique ordre v, auquel il obéira à l'étape (3), du fait de la 1ère propriété de la fonction choix. Cela prouve $IC2$.

Puisque $IC1$ découle de $IC2$ lorsque le commandant est loyal, nous n'avons plus qu'à considérer le cas où le commandant est un traître. Deux lieutenants loyaux i et j obéissent au même ordre à l'étape (3) si les ensembles V_i et V_j obtenus à l'étape (2) sont les même. Donc, pour prouver $IC1$ il suffit de prouver que, si i place un ordre v dans V_i à l'étape (2), alors j doit placer le même ordre v dans V_j à l'étape (2). Pour cela, il nous faut montrer que j reçoit un message correctement signé contenant cet ordre. Si i reçoit l'ordre v à l'étape (2)(A), alors il l'envoie à j à l'étape (2)(A)(ii) ; donc j le reçoit (du fait de S1). Si i ajoute l'ordre v à V_i dans l'étape (2)(B), alors il doit recevoir un premier message $v : 0 : j_1 : ... : j_k$. Si j est l'un des j_r, alors, du fait de S4, il a déjà dû recevoir l'ordre v. Si ce n'est pas le cas, il faut considérer 2 cas :

— $k < m$. Dans ce cas, i envoie le message $v : 0 : j_1 : ... : j_k : i$ à j ; donc j doit recevoir l'ordre v.

— $k = m$. Puisque le commandant est un traître, au plus $m - 1$ des lieutenants sont des traîtres. Ainsi, au moins l'un des lieutenants $j_1, ..., j_m$ est loyal. Ce lieutenant loyal doit avoir envoyer à j la valeur v quand il l'a reçue pour la première fois, donc j doit avoir reçu cette valeur.

Cela complète la preuve.

□

Lemme 6.1.2 *La complexité est en* $O(n^{m+1})$.

Preuve

La mise en oeuvre de l'algorithme $SM(m)$ cause tout d'abord l'émission de $n-1$ messages. Chacun de ces messages invoque $SM(m-1)$ causant l'émission de $n-2$ nouveau messages etc.

$$\begin{array}{cc} (1) & (n-1) \\ (2) & (n-1)(n-2) \\ \dots & \\ (m+1) & (n-1)(n-2)\dots(n-(m+1)) \end{array}$$

et donc le total est $O(n^{m+1})$.

□

Le problème du nombre de chemins nécessaires à la communication

Le lecteur attentif aura très certainement remarqué que les algorithmes précédemment énoncés considéraient qu'il existe un chemin direct entre chaque général (ou

processeur). Or, dans la pratique, il est peu probable que de telles conditions soient présentes. Nous allons donc voir comment ces algorithmes peuvent être modifiés afin de pouvoir être appliqués à des graphes de connectivité plus réduite.
Nous considérons donc les généraux comme les noeuds d'un graphe G simple (c'est à dire qu'il n'y a au plus un arc entre deux noeuds), fini et non orienté, où un arc entre deux noeuds indique que ceux-ci peuvent s'envoyer des messages directement. Nous allons maintenant étendre les algorithmes $OM(m)$ et $SM(m)$, qui supposaient que G était complètement connecté, afin de les appliquer à des graphes plus généraux.
Afin d'étendre l'algorithme utilisant les messages oraux $OM(m)$, nous avons besoin de la définition suivante, où deux généraux sont dits voisins s'ils sont reliés par un arc.

Définition 6.1.1 *Un ensemble de noeuds $\{i_1, ..., i_p\}$ est appelé ensemble régulier de voisins d'un noeud i si :*

— *chaque i_j est un voisin de i, et pour tout général k différent de i, il existe des chemins $g_{j,k}$ de i_j à k qui ne passent pas par i tel que quelques soient deux chemins différents $g_{j,k}$ ils n'aient pas d'autre noeud en commun que k.*

Le graphe G est dit p-régulier si tous les noeuds ont un ensemble régulier de voisins constitué de p noeuds distincts.
Nous étendons maintenant $OM(m)$ en un algorithme qui résout le Problème des Généraux Byzantins en la présence de m traîtres si le graphe G de généraux est 3m-régulier. (Il faut noter qu'un graphe $3m$-régulier contient au moins $3m + 1$ noeuds) Pour tout entier positif m et p, nous définissons l'algorithme $OM(m, p)$ comme suit, quand le graphe G de généraux est p-régulier. La définition utilise une récursivité sur m.

Algorithme $OM(m, p)$
Choisir un ensemble régulier N de voisins du commandant, constitué de p lieutenants
Le commandant envoie sa valeur à chacun des lieutenants contenus dans N
Pour chaque i dans N, soit v_i la valeur reçue par le lieutenant i du commandant, ou vdef si il n'a rien reçu. Le lieutenant i envoie v_i à tous les autres lieutenants comme suit :
    Si $m = 1$, en envoyant la valeur le long du chemin $g_{i,k}$ dont l'existence est garantie par la définition précédente $(a)(ii)$
    Si $m > 1$, en agissant comme le commandant dans

l'algorithme $OM(m-1, p-1)$, avec le graphe de généraux obtenu en supprimant le commandant original de G.

Pour chaque k, et chaque i dans N avec $i \neq k$, soit v_i la valeur reçu par le lieutenant k du lieutenant i à l'étape (2), ou vdef si il n'a rien reçu. Le lieutenant k utilise la valeur majorité$(v_{i_1}, ..., v_{i_p})$, où $N = \{i_1, ..., i_p\}$

Il faut remarquer que si l'on supprime un seul noeud d'un graphe p-régulier, il devient un graphe $(p-1)$-régulier. Ainsi on peut appliquer l'algorithme $OM(m-1, p-1)$ à l'étape $(2)(B)$.

Lemme 6.1.3 *Pour tout $m > 0$ et pour tout $p > 2k + m$, l'algorithme $OM(m, p)$ satisfait $IC2$ et il y a au plus k traîtres.*

Théorème 6.1.4 *Pour tout tout $m > 0$ et tout $p > 3m$, l'algorithme $OM(m, p)$ résout le Problème des Généraux Byzantins si il y a au plus m traîtres.*

Le lecteur intéressé par les preuves de ces deux énoncés se reportera à [1].

Cette extension de l'algorithme $OM(m)$ nécessite que le graphe G soit $3m$-régulier, ce qui est une hypothèse de forte connectivité. En fait si il y a seulement $3m+1$ généraux (c'est à dire le minimum requis), alors la $3m$-régularité du graphe correspond à un connectivité complète, et $OM(m, 3m)$ se réduit à $OM(m)$.

Par contre, l'algorithme $SM(m)$ est facilement étendu afin de permettre la plus faible connectivité possible. Il nous faut tout d'abord établir quelle connectivité est nécessaire pour que le Problème des Généraux Byzantins soit solvable. $IC2$ requiert qu'un lieutenant loyal obéisse à un commandant loyal. Cela est clairement impossible si le commandant ne peut pas communiquer avec le lieutenant. En particulier si le message du commandant au lieutenant doit être relayé par des traîtres, alors il n'est pas possible de garantir que le lieutenant obtiendra le message du commandant. De même, $IC2$ ne peut être garantie si deux lieutenants ne peuvent communiquer que par l'intermédiaire d'un traître.

L'hypothèse de connectivité la plus faible pour laquelle le problème est solvable est que le sous-graphe formé par les généraux loyaux soit connecté. Nous pouvons montrer que sous cette hypothèse, l'algorithme $SM(n-2)$ est une solution, où n est le nombre de généraux, indépendamment du nombre de traîtres. Bien sûr, il est nécessaire de modifier l'algorithme afin de n ?envoyer des messages que là où ils peuvent l'être. Plus précisément, à l'étape (1), le commandant n'envoie on ordre signé qu'à ces voisins ; et à l'étape $(2)(B)$, le lieutenant i n'envoie le message qu'à ses voisins qui n'apparaissent pas dans la signature.

Définition 6.1.2 *Le diamètre d d'un graphe est le plus petit nombre tel que deux noeuds quelconques sont reliés par un chemin d'au plus d arcs.*

Théorème 6.1.5 *Pour tout m et d, si il y a au plus m traîtres et que le sous-graphe de généraux loyaux a un diamètre égal à d, alors $SM(m+d-1)$ (avec les modifications évoquées plus haut) résout le Problème des Généraux Byzantins.*

Corollaire 6.1.1 *Si le graphe des généraux loyaux est connecté, alors $SM(n-2)$ résout le problème des Généraux Byzantins pour n généraux.*

6.1.5 Conclusion

Problèmes, critiques
Ces algorithmes proposent bien une solution au Problème des Généraux Byzantins, néanmoins, quelques problèmes restent posés, principalement dans le cadre de leur mise en oeuvre dans les systèmes informatiques existants.
La première remarque à faire est le nombre important de messages générés par de tels algorithmes. Mais cela est dû à la complexité du problème et ne peut malheureusement pas être réduit.
Il est bien sûr possible de faire quelques suppositions supplémentaire telles que, en informatique, il est plus probable qu'un composant tombe en panne, plutôt que se mette à envoyer des données corrompues. Mais si l'on cherche à mettre en oeuvre des systèmes réellement fiable, alors de telles suppositions n'ont pas lieu d'être.
Une autre remarque est que, si l'on souhaite mettre en oeuvre ce système sur un réseau d'ordinateurs, il est rare que ceux-ci ait une topologie p-régulière [3].
En conclusion, ces solutions ne sont applicables que dans des systèmes spécifiquement prévus à cet effet (du fait de la topologie), mais aussi où la performance a moins d'importance que la fiabilité. En effet ces algorithmes sont excessivement consommateurs de ressources (processeur et réseau).

Bibliographie

[1] R. Balter, J.-P. Banâtre, S. Krakowiak, *Construction des systèmes d'exploitation répartis*, INRIA, collection didactique, 1991.

[2] R. Chow, T. Johnson, *Distributed operating systems and algorithms*, Addison Wesley, 1997.

[3] D. Conan, G. Bernard, *La reprise sur erreur par recouvrement arrière automatique dans les systèmes répartis*, TSI.

[4] J.-P. Delahaye, *Merveilleux nombres premiers*, Belin, pour la science, 2000.

[5] S. Garfinkel, G. Spafford, *Pratical UNIX and Internet security*, O'Reilly and Associates, 2nd edition, 1996.

[6] J.-M. Geib, C. Gransart, P. Merle, *CORBA des concepts à la pratique*, Inter-Edition, 1997.

[7] A. Menezes, P. Van Oorschot, S. Vanstone, *Handbook of applied cryptography*, CRC press series on discrete mathematics and its applications, 1997.

[8] J.-M. Rifflet, *La programmation sous UNIX*, Ediscience, 3ème édition, 1997.

[9] J.-M. Rifflet, *La communication sous UNIX*, Ediscience, 2ème édition, 1996.

[10] B. Schneier, *Cryptographie appliquée*, Thomson publishing, 2ème édition (traduction L. Viennot), 1997.

[11] S. Singh, *Histoire des codes secrets*, Jean-Claude Lattes, 1999.

[12] G. Tell, *Introduction to distributed algorithms*, Cambridge University Press, 1994.

Ce sont quelques références bibliographiques utilisées pour la construction de ce cours. Des articles de recherche ainsi que des rapports de recherche ont aussi été utilisés. Ceux-ci étant plus difficiles à trouver, les références exactes ne sont pas données ici. Quelques auteurs de ces articles sont :
Lamport, Maekawa, B. Netzer, M. Raynal (et tous les chercheurs de son équipe à l'INRIA), Ricart, Toueg, J. Xu... Je suis à la disposition du lecteur pour lui donner des compléments bibliographiques si nécessaire.