BADJI MOKHTAR-ANNABA UNIVERSITY
UNIVERSITE BADJI MOKHTAR-ANNABA
جامعة باجي مختار – عنابـــــــــــة

FACULTE DES SCIENCES DE L'INGENIORAT
DEPARTEMENT D'INFORMATIQUE
كلية علـــــــــوم الهندســـــــة
قسم الإعـــــــلام الألـــــــــي

FORMALISME ET ALGORITHMES DE BASE DES SYSTEMES DISTRIBUES

Programme de Première Année Master en Informatique

OPTION INGENIERIE DES LOGICIELS COMPLEXES (ILC)

SUPPORT DE COURS REALISE PAR
PR DJAMEL MESLATI

VERSION 2017

PDF-XChange Viewer Click to buy NOW! www.docu-track.com

INTRODUCTION

PROGRAMME OFFICIEL

DESCRIPTION

Intitulé de la matière : Formalisme et algorithmes de base des systèmes distribués
Code : FORSYD
Semestre : 2
Unité d'Enseignement : ILC 8 **Code :** ILC 8
Enseignant responsable de l'UE : Dr Toufik BENOUHIBA
Enseignant responsable de la matière: Pr Djamel MESLATI
Nombre d'heures d'enseignement : 48h Cours (3h/semaine)
Nombre d'heures de travail personnel pour l'étudiant : 2h
Nombre de crédits : 4
Coefficient de la matière : 4

Mode d'évaluation : *Examen écrit + Travail pratique soutenu*

Objectifs de l'enseignement (*Décrire ce que l'étudiant est censé avoir acquis comme compétences après le succès à cette matière).*

Ce cours vise à permettre aux étudiants de cerner l'aspect implémentation des systèmes distribués en proposant un formalisme basé sur les événements et en donnant une étude détaillée de nombreux algorithmes de base avec calcul de complexité. Outre les algorithmes d'exclusion mutuelle, la synchronisation répartie et la détermination/utilisation de l'état global, il est recommandé d'utiliser Java comme langage support pour implémenter divers algorithmes distribués sur un réseau local.

Connaissances préalables recommandées (*descriptif succinct des connaissances requises pour pouvoir suivre cet enseignement).*

Langage orienté objets, systèmes d'exploitation, logique mathématique, compilation, réseaux et communication

CONTENU

Chapitre 1 : Formalisme à base d'événements (20%)

1. Les entités en exécution
2. La communication
3. Axiomes et restrictions
4. Coût et complexité
5. Exemple (la Diffusion)
6. Etats et Evénements
7. Considérations techniques

Chapitre 2 : Les protocoles de base (30%)

1. La diffusion
2. Le réveil
3. Parcours de réseaux et évaluation des fonctions
4. Construction d'une arborescence couvrante

Chapitre 3 Problèmes d'exclusion mutuelle, allocation des ressources et synchronisation distribuée (30%)

1. Introduction aux classes d'algorithmes
2. Allocation répartie des ressources

PDF-XChange Viewer Click to buy NOW! www.docu-track.com

Chapitre 4 Observation et état global (20%)

1. L'observation répartie
2. Détection répartie de la terminaison
3. Détection répartie de l'interblocage
4. Calcul d'états globaux

REFERENCES BIBLIOGRAPHIQUES

1- Nicola Santoro, "Design and Analysis of Distributed Algorithms", John Wiley & Sons, 2007.

2- Jia Weijia, Zhou Wanlei, "Distributed Network Systems, From Concepts to Implementations", Springer Science, 2005.

3- George Coulouris, Jean Dollimore & Tim Kindberg, "Distributed Systems, Concepts and Design, Addison-Wesley", 2001.

4- Andrew Tannenbaum, "Distributed Operating Systems", Prentice Hall International, 1995.

5- Michel Raynal, « Synchronisation et état global dans les systèmes répartis », Editions Eyrolles, 1992.

PDF-XChange Viewer Click to buy NOW! www.docu-track.com

CHAPITRE 1

FORMALISME

CONTENU

1.1 RESUME

Ce chapitre est dédié à la présentation d'un formalisme, tiré de [1], qui permet la description des systèmes distribués, leur analyse et leur conception.

Dans ce formalisme, on considère qu'un environnement de calcul distribué est une collection finie ε d'entité en exécution (processus ou autre) qui communiquent par des messages. Cette communication vise à atteindre un objectif commun qui peut être la réalisation d'une tâche donnée, l'exécution d'une solution d'un problème, la réponse à une requête venant de l'utilisateur (i.e. de l'extérieur de l'environnement) ou d'une autre entité, ...

1.2 LES ENTITES EN EXECUTION

L'unité de calcul d'un environnement de calcul distribué est appelée une *entité*. En fonction du système qui est modélisé par l'environnement, une entité peut correspondre à un processus, un processeur, un switch, un agent, etc.

Aptitudes. Chaque entité $x \in \varepsilon$ est dotée d'une mémoire locale *Mx* qui est privée et non partagée. L'aptitude de *x* comprend l'accès à la mémoire locale (écriture et lecture), le traitement local et la communication qui consiste en la préparation, la transmission et la réception des messages. La mémoire locale comprend un ensemble de *registres définis* dont les valeurs sont toujours initialement définies. Parmi ces registres, on trouve le registre *statut* (noté *status*(*x*)) et le registre *input value* (noté *value*(*x*)). Le registre *status*(*x*) prend ses valeurs dans un ensemble fini d'états du système S; les exemples de telles valeurs sont "Idle", "Processing", "Waiting", etc.

De plus, chaque entité $x \in \varepsilon$ dispose d'une horloge alarme locale c_x qui peut être armée ou désarmer (set et reset). Une entité peut réaliser quatre types d'opérations uniquement :

- Stockage et traitement locaux
- Transmission des messages
- Armer et désarmer l'horloge alarme
- Changer la valeur du registre *status*

Notons que « Armer et désarmer l'horloge alarme » peut-être considéré comme faisant partie du traitement local, cependant, vu le rôle spécial que ces opérations jouent, nous les considérons comme un type distinct d'opérations.

Les événements externes. Le comportement d'une entité $x \in \varepsilon$ est *réactif* : *x* répond uniquement aux stimuli externes que nous appelons événements externes (ou événement); en absence de stimuli, *x* est inerte et n'exécute aucune action. Il existe trois types d'événement externes :

- Arrivée d'un message
- Déclenchement de l'alarme de l'horloge
- Impulsion spontanée

L'arrivée d'un message et le déclenchement de l'alarme sont des événements externes à l'entité mais sont issus du système lui-même : le message est envoyé par une autre entité et l'horloge est armée par l'entité elle-même.

Contrairement à ces deux événements, l'impulsion spontanée est déclenchée par des forces externes au système et donc externes à l'univers perçu par l'entité. Comme exemple, considérons un système automatique de transactions bancaires : ses entités sont les serveurs de la banque où les données sont stockées et les distributeurs

automatiques de billets (automated teller machine (ATM)) ; la demande de retrait d'argent par un client (i.e. mise à jour des données stockées dans le système) est une impulsion spontanée pour le distributeur (l'entité) où la demande est faite. Comme autre exemple, considérons un sous-système de communication dans le modèle de référence d'interconnexion de systèmes ouverts (OSI) : la demande de service qui émane de la couche réseau pour la couche liaison de données est une impulsion spontanée pour l'entité « couche liaison de données ». Les impulsions spontanées sont des événements qui lancent le calcul et la communication, ils apparaissent aux entités comme des actions « divines ».

Les actions. A l'occurrence d'un événement externe e, une entité $x \in \mathcal{E}$ réagira à e en exécutant une séquence d'opération finie, indivisible et terminable appelée *action*.

Une action est indivisible (ou atomique) dans le sens ou ses opérations sont exécutées sans interruption. En d'autres termes, une fois l'action lancée elle ne s'arrêtera que lorsqu'elle est finie.

Une action est terminable dans le sens où, une fois lancée, son exécution s'achèvera au bout d'un temps fini (les programmes qui ne se terminent pas ne sont pas considérés comme des actions).

L'entité peut entreprendre une action spéciale, notée *Nil,* qui est une action nulle. Dans ce cas on dit que l'entité ne réagit pas à l'événement.

Le Comportement. La nature de l'action exécutée par l'entité dépend de la nature de l'événement e, ainsi que du statut dans lequel se trouve l'entité (i.e., la valeur de $status(x)$) au moment de l'occurrence de l'événement. Nous notons cela par la spécification de la forme :

$$Status \times Event \rightarrow Action$$

que nous appelons *règle* (ou *method*, ou *production*). Dans une règle $s \times e \rightarrow A$, nous disons que la règle est activée par (s, e).

La spécification du comportement, ou simplement le comportement d'une entité x est l'ensemble $B(x)$ de toutes les règles auxquelles obéit x. Cet ensemble doit être *complet* et *non ambiguë* : pour chaque événement e et une valeur de statut s, il existe une et une seule règle dans $B(x)$ qui est activée par (s,e). En d'autres termes, x doit toujours savoir quoi faire lorsqu'un événement a lieu.

L'ensemble des règles $B(x)$ est aussi appelé *Protocole* ou *Algorithme distribué* de x.

La spécification du comportement de tout l'environnement de calcul distribué n'est autre que la collection des comportements individuels des entités. Plus précisément, le comportement collectif $B(\varepsilon)$ d'une collection ε d'entités est l'ensemble $B(\varepsilon)$ tel que :

$$B(\varepsilon) = \{B(x): x \in \varepsilon\}$$

Ainsi, dans un environnement ayant le comportement collectif $B(\varepsilon)$, chaque entité x agira (se comportera) selon son algorithme distribué ou son protocole $B(x)$.

Comportement homogène. Un comportement collectif est *homogène* si toutes les entités du système ont le même comportement, càd :

$$\forall x, y \in \varepsilon, B(x) = B(y)$$

Ceci signifie que pour spécifier un comportement collectif homogène il suffit de spécifier le comportement d'une seule entité. Dans ce cas, nous indiquerons le comportement, simplement, par B. Un fait intéressant et important est le suivant :

Propriété 1.1.1 *Tout comportement collectif peut devenir homogène.*

Ceci implique qu'on présence d'un système où différentes entités ont différents comportements, nous pouvons écrire un nouvel ensemble de règles, qui est le même pour toutes les entités et qui permet d'avoir le même comportement initial.

Exemple. Considérons un système composé d'un réseau de plusieurs stations de travail identiques et un serveur ; il est claire que l'ensemble des règles auxquelles obéissent le serveur et les stations de travail n'est pas le même vu que leur fonctionnalités diffèrent. Néanmoins, un programme unique peut être écrit et s'exécutera sur les deux entités sans modifier leurs fonctionnalités. Pour cela, nous avons besoin d'ajouter un registre d'entrée (*input register*) nommé *my role*, qui est initialisé à soit "workstation" soit "server," selon l'entité. Pour chaque paire statut–événement (s, e) nous créons une nouvelle règle avec l'action :

$$s \times e \rightarrow \{ \textbf{\textit{if}}\ my\ role = workstation\ \textbf{\textit{then}}\ Aworkstation\ \textbf{\textit{else}}\ Aserver\ \textbf{\textit{endif}}\ \},$$

Où *Aworkstation* (respectivement, *Aserver*) est l'action originale associée à (s, e) dans l'ensemble des règles de la station de travail (respectivement, le serveur). Si (s, e) n'active aucune règle de la station de travail (e.g., s était un statut défini uniquement pour le serveur), alors *Aworkstation* = ***Nil*** dans la nouvelle règle, et de même pour le serveur.

Il est important d'insister sur le fait que dans un système homogène, bien que toutes ses entités aient la même description comportementale (logiciel), elles ne se comportent pas de la même façon ; leurs différences dépendront seulement des valeurs

initiales de leurs registres d'entrée. Comme analogie, l'ensemble des lois d'un system juridique est le même pour tous les citoyens, néanmoins, un policier exerçant ses fonctions peut entreprendre certaines actions qui sont illégales pour le reste des autres citoyens.

Une conséquence importante de la propriété des comportements homogènes et qu'il est possible de se concentrer uniquement sur les environnements où les entités ont les mêmes comportements (c'est le cas dans le reste de ce document).

1.3 LA COMMUNICATION

Dans un environnement de calcul distribué, les entités communiquent par transmission et réception de *messages*. Le *message* est l'unité de communication d'un environnement distribué.

Selon une définition plus générale, un message est juste une *séquence finie de bits*. Une entité communique par transmission/réception de messages vers ou des autres entités. L'ensemble des entités avec lesquelles une entité peut communiquer directement n'est pas nécessairement ε; En d'autres termes, il est possible qu'une entité ne puisse communiquer directement qu'avec un sous ensemble des autres entités. Nous notons par $N_{out}(x) \subseteq \varepsilon$ l'ensemble des entités vers lesquelles x peut transmettre un message directement. Nous les appelons les voisins de sortie de x (*out-neighbors* of x) . De façon similaire, $N_{in}(x) \subseteq \varepsilon$ est l'ensemble des entités à partir desquelles x peut recevoir directement un message. Nous les appelons les voisins d'entrée (*in-neighbors* of x).

La relation de voisinage définit un graphe orienté $\vec{G} = (V, \vec{E})$, où V est l'ensemble des sommets et $\vec{E} \subseteq V \times V$ est l'ensemble des arrêtes. Les sommets correspondent aux entités, et $(x, y) \in \vec{E}$ si et seulement si l'entité (correspondant à) y est un voisin de sortie de l'entité (correspondant à) x. Le graphe orienté $\vec{G} = (V, \vec{E})$ décrit la topologie de communication de l'environnement. On note par $n(\vec{G})$, $m(\vec{G})$ et $d(\vec{G})$ le nombre de sommets, d'arcs et le diamètre de $\vec{G}$, respectivement. Si aucune ambiguïté n'a lieu, nous omettons la référence à $\vec{G}$, et nous utilisons simplement n, m, et d.

Dans le reste du document et en absence d'ambiguïté, les termes sommet, nœud, site et entité sont utilisés dans le même sens. De même, arrête, arc, et lien seront utilisés de façon interchangeable.

En résumé, une entité peut recevoir des messages des ses voisins d'entrée uniquement et peut envoyer des messages à ses voisins de sortie uniquement. Les messages reçus par une entité sont traités au niveau de celle-ci dans l'ordre de leur arrivée. Si plus d'un

message arrivent en même temps, ils seront traités selon un ordre arbitraire. Les entités et les communications peuvent échouer.

1.4 AXIOMES ET RESTRICTIONS

La définition d'un environnement de calcul distribué avec des communications point à point a deux *axiomes* de base. L'un concerne les retards dans la communication et l'autre concerne l'orientation locale des entités dans le système. Toute supposition supplémentaire (e.g., une propriété du réseau, connaissances a priori qu'ont les entités) est appelée *restriction*.

1.4.1 Axiomes

Retards des communications. La communication par message implique plusieurs activités : préparation, transmission, réception, et traitement. Dans les systèmes réels décrit par notre modèle, le temps que nécessitent ces activités et imprédictible. Par exemple, dans un réseau de communication, un message peut faire l'objet de retards (attente dans une file, retard dans le traitement) qui dépendent du trafic dans le réseau à ce moment. Par exemple, considérons les délais d'accès (i.e., envoi d'un message à et réception de réponse de) à un site populaire.

La totalité des retards accumulés par un message sera appelée *délai de communication* de ce message.

Axiome 1.3.1 Délais de communication finis

En absence de pannes, les délais de communication sont finis.

En d'autres termes, en absence de pannes, un message envoyé à un voisin de sortie y arrivera éventuellement en préservant son intégrité et y sera traité. Notons que l'axiome « Délai de communication finis » n'implique pas l'existence d'une limite sur les délais de transmission, de mise en file et de traitement. Il spécifie seulement qu'on absence de pannes, le message arrivera après un temps fini sans corruption.

Orientation Locale. Une entité peut communiquer directement avec un sous ensemble d'autres entités : ces voisins. Le seul autre axiome du modèle est qu'une entité peut distinguer ces voisins.

Axiome 1.3.2 Orientation Locale

Une entité peut faire la distinction entre ses voisins d'entrée.

Une entité peut faire la distinction entre ses voisins de sortie.

En particulier, une entité est capable d'envoyer un message uniquement vers un voisin de sortie spécifique (sans l'envoyer à tous les autres voisins de sortie). De même, lors du traitement d'un message (i.e., exécution de la règle activée par la réception de ce message), l'entité peut identifier lequel des ces voisins a envoyé le message.

En d'autres termes, chaque entité x dispose d'une fonction locale λ_x associant des labels (étiquettes), dits aussi *numéros de ports*, à ses liaisons incidentes (ou *ports*), et cette fonction est injective. Nous notons les numéros de ports par $\lambda_x(x, y)$, le label associé par x à la liaison (x, y). Nous insistons sur le fait que ce label est local à x et n'a en général aucune relation avec les labels que y pourrait utiliser pour nommer cette liaison (ou nommer x ou se nommer lui même). Notons que pour chaque arc $(x,y) \in \vec{E}$, *il existe deux labels :* $\lambda_x(x, y)$ local à x *et* $\lambda_y(x, y)$ local à y (voir Figure 1.1).

A cause de cet axiome, nous aurons toujours affaire à des *graphes aux arcs étiquetés* $(\vec{G}, \lambda)$, où $\lambda = \{ \lambda_x : x \in V \}$ est l'ensemble des fonctions injectives d'étiquetage.

FIGURE 1.1: Chaque arc a deux labels

1.4.2 Les Restrictions

En général, un système de calcul distribué peut avoir des propriétés et des aptitudes supplémentaires qui peuvent être exploités pour résoudre un problème, accomplir une tâche, et fournir un service. Ceci peut être fait en utilisant ces propriétés et aptitudes dans l'ensemble des règles.

Cependant, toute propriété utilisée dans le protocole limite l'applicabilité de ce dernier. En d'autres termes, toute propriété ou aptitude supplémentaire du système représente, en réalité, une *restriction* (ou sous modèle) du modèle général.

Remarque. Lorsqu'on a affaire à un système de calcul distribué (e.g., conception, développement, test, utilisation) ou seulement à son protocole, il est crucial et impératif que toutes les *restrictions soient rendues explicites*. Si tel n'est pas le cas, le logiciel ne sera pas valide.

Les restrictions peuvent varier en nature et en type : Elles peuvent être en relation avec les propriétés de communication, la fiabilité, la synchronisation, etc. Nous discuterons quelques unes dans les sections suivantes.

Restrictions de Communication. La première catégorie de restrictions comprend celles relatives à la communication entre les entités.

Politique de mise en file. Une liaison (x, y) peut être vue comme un canal ou une file (voir section 1.10) : x envoi un message à y est équivalent à x insert le message dans le canal.

En général, toute sorte de situation est possible ; par exemple, les messages peuvent se dépasser dans le canal de telle sorte que le dernier message est reçu en premier. Différents restrictions sur ce modèle, dériveront différentes disciplines employées pour gérer le canal ; par exemple les files FIFO (first-in-first-out) sont caractérisées par la restriction suivante :

> *Les messages sont ordonnés :* En absence de pannes, les messages transmis par une entité, à un même voisin de sortie, arriveront dans le même ordre dans lequel ils sont envoyés.

Notons que cette restriction ne signifie pas qu'il existe un respect de l'ordre pour les messages transmis à une même entité à partir d'arcs différents, ni les messages envoyés par une même entité sur différents arcs.

Propriété de la liaison. Les entités d'un système de communication sont reliées par des liens physiques qui peuvent avoir des aptitudes très différentes. Par exemple, les liaisons simplexes et full duplexes. Avec une liaison full duplexe il est possible de transmettre dans les deux directions. Les liaisons simplexes sont déjà définies dans le cadre du modèle général. Evidement, une liaison duplexe peut être décrite comme deux liaisons simplexes, une dans chaque direction ; ainsi, un système ou toutes les liaisons sont full duplexes peut être décrit par la restriction suivante :

> *Communication Réciproque :* $\forall x \in \mathcal{E}$, $N_{in}(x) = N_{out}(x)$. En d'autres termes, si $(x,y) \in E$ alors $(y,x) \in E$ aussi.

Notons cependant que, $(x, y) \neq (y, x)$, et en général $\lambda_x (x, y) \neq \lambda_x (y, x)$; de plus, x ne devrait par savoir que les deux liaisons sont des connexions de et vers la même entité.

> *Liaisons Bidirectionnelles :* $\forall x \in \mathcal{E}$, $N_{in}(x) = N_{out}(x)$ *et* $\lambda_x (x, y) = \lambda_x (y, x)$.

IMPORTANT. Le cas des liaisons bidirectionnelles est spécial. En sa présence, on utilise une terminologie simplifiée. Le réseau est vu comme un graphe *non-orienté* $G = (V, E)$ (i.e., $\forall x,y \in \mathcal{E}$, $(x,y) = (y, x)$), et l'ensemble $N(x) = N_{in}(x) = N_{out}(x)$ sera appelé l'ensemble des voisins de x. Notons que dans ce cas, $m(\vec{G}) = |\vec{E}| = 2|E| = 2m(G)$.

Par exemple, la figure 1.2 montre un graphe $\vec{G}$ où la restriction « Liaisons Bidirectionnelles » et son graphe non-orienté correspondant G sont vérifiés.

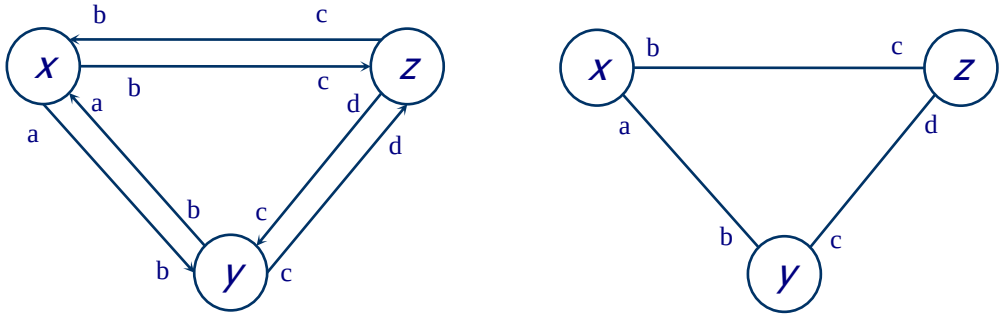

FIGURE 1.2: Réseau avec liaisons bidirectionnelles et graphe non-orienté correspondant

Restrictions de fiabilité. D'autres types de restrictions sont celles relatives à la fiabilité, les fautes et leur détection.

Détection des fautes. Certains systèmes pourraient fournir un mécanisme de détection de fautes fiable. Dans ce qui suit, on donne deux restrictions qui décrivent les systèmes qui offrent de tels mécanismes face aux pannes des composants :

> *Détection des pannes des arcs :* $\forall (x, y) \in \vec{E}$, *x* et y détecteront si (*x*, *y*) a échoué et, suite à cet échec, s'il a été réactivé.
>
> *Détection des pannes des entités :* $\forall x \in V$, tous les voisins d'entrée et de sortie de *x* peuvent détecter si *x* a échoué et, suite à cet échec, s'il a repris.

Restriction sur les Types de Fautes. Dans certains systèmes, seuls quelques types de pannes peuvent avoir lieu : par exemple, les messages peuvent être perdus mais pas corrompus. Chaque situation donnera naissance à une restriction correspondante. Des restrictions plus générales dériveront des systèmes ou des situations où il n'y aura pas de pannes.

> *Garantie d'acheminement :* Tout message qui est envoyé sera reçu avec un contenu non corrompu.

Sous cette restriction, les protocoles n'ont pas à tenir compte des omissions ou des corruptions des messages durant les transmissions. Une restriction plus générale est la suivante :

> *Fiabilité partielle :* Aucune panne n'aura lieu.

Sous cette restriction, les protocoles n'ont pas à tenir compte des pannes. Notons que sous la fiabilité partielle, les pannes peuvent avoir eu lieu *avant* l'exécution d'un calcul. Un système totalement exempt de toute fautes est défini par la restriction suivante :

Fiabilité totale : Aucune panne n'a eu lieu ni n'aura lieu. Il est clair que les protocoles développés sous cette restriction ne sont pas garantis de fonctionner correctement en présence de fautes.

Restrictions Topologiques. En général, une entité n'est pas directement connectée à toutes les autres entités ; mais pourrait être capable, malgré cela, de communiquer des informations à une entité distante, en utilisant d'autres entités comme relais. Un système qui fournit cette aptitude à toutes les entités est caractérisé par la restriction suivante :

Connectivité : La topologie de communication $\vec{G}$ est fortement connectée.

Càd, partant de n'importe quel sommet de $\vec{G}$ il est possible d'atteindre n'importe quel autre sommet. Si de plus la restriction « Liaisons bidirectionnelles » est vérifiée, la connectivité spécifiera que « G est connecté ».

Restrictions temporelles. Un type de restrictions intéressant est celui relatif au temps. En effet, le modèle général ne fait aucune supposition sur les retards (excepté qu'ils sont finis).

Délais de communication limités : Il existe une constante Δ telle que, en absence de pannes, le délai de communication de tout message sur n'importe quelle liaison est au plus égal à Δ. Un cas spécial de délais limités est le suivant :

Délais de communications unitaires : En absence de pannes, le délai de communication de tout message sur toute liaison vaut une unité de temps.

Le modèle général, ne fait pas, aussi, des suppositions sur les horloges locales.

Les horloges synchronisées : Toutes les horloges locales sont incrémentées d'une unité simultanément et l'intervalle de temps de temps entre deux incrémentations successives est constant.

1.5 COÛT ET COMPLEXITE

L'environnement de calcul que nous considérons est défini à un niveau abstrait. Il modélise plutôt des systèmes différents (e.g., réseaux de communication, systèmes distribués, réseaux de données, etc.), dont les performances sont déterminées par des facteurs et des coûts très distinctifs. L'efficacité d'un protocole dans le modèle doit d'une façon ou d'une autre refléter les coûts réals qu'on peut avoir lors des exécutions de ce protocole dans ces systèmes différents. En d'autres termes, nous avons besoins de mesures de coût abstraites qui soient assez générales tout en restant significatifs.

Nous verrons deux types de mesures : le *total des activités de communication* et le *temps* nécessité par l'exécution d'un calcul. On peut les voir comme des mesures de coût du point de vue du système (combien de trafic le calcul générera-t-il et à quel point le système sera-t-il occupé ?) et d'un point de vue utilisateur (combien faut-il de temps avant d'avoir le résultat du calcul ?).

1.5.1 Total de l'activité de communication

La transmission d'un message sur un port de sortie (i.e., vers un voisin de sortie) est *l'activité de communication* de base dans le système ; notons la transmission d'un message qui ne sera pas reçu suite à une panne constitue aussi une activité de communication. Ainsi, pour mesurer le total des activités de communication, la fonction commune la plus utilisée est le nombre de transmissions de messages **M**, dit aussi *coût en message*. Par conséquent, en général, étant donné un protocole, nous mesurons ses coûts de communication en termes de nombre de messages transmis.

D'autres fonctions intéressantes sont la *charge par entité* $\boldsymbol{L}_{node} = \boldsymbol{M}/|V|$, càd le nombre de messages par entité, et *charge de transmission* $\boldsymbol{L}_{link} = \boldsymbol{M}/|E|$, càd, le nombre de messages par liaison. Les messages sont des séquences de bits ; certains protocole utiliseraient des messages cours (e.g., O(1) signaux par bit), d'autres, des messages très longs (e.g., fichiers .gif). Ainsi, pour une évaluation précise de protocole, ou pour comparer différentes solutions d'un même problème qui utilise des messages de tailles différentes, il serait nécessaire d'utiliser comme mesure de coût le nombre de bits **B** appelée aussi la *complexité en bit*. Dans ce cas, on considère parfois les fonctions de charges définies en bit : *charge par entité en bit* $\boldsymbol{Lb}_{node} = \boldsymbol{B}/|V|$, qui est le nombre de bits par entité et la *charge de transmission en bit* $\boldsymbol{Lb}_{link} = \boldsymbol{B}/|E|$, qui est le nombre de bits par liaison.

1.5.2 Le Temps

Une mesure importante de l'efficacité et de la complexité est le total du délai d'exécution, càd, le délai entre le temps où la première entité débute l'exécution d'un calcul et le temps où la dernière entité termine son exécution. Notons que « temps » est supposé être celui mesuré par un observateur externe au système et sera appelé aussi temps réel ou temps physique.

Dans le modèle général, il n'y a aucune supposition concernant le temps, excepté que les délais de communication d'un message sont finis en absence de panne (Axiome 1.3.1).

En d'autres termes, les délais de communication sont en général imprédictibles. Ainsi, même en absence de pannes, le total du délai d'exécution d'un calcul est totalement imprédictible ; de plus, deux exécutions distinctes du même protocole peuvent

connaître des délais radicalement différents. En d'autres termes, on ne peut mesurer le temps avec précision.

Cependant, nous pouvons mesurer le temps en supposant certaines conditions. La mesure utilisée usuellement est le *délai d'exécution idéal* ou la *complexité en temps idéal*, **T :** qui est le délai d'exécution obtenu sous les restrictions "Délais de Transmission Unitaire" et "Horloges Synchronisées" ; càd, lorsque le système est synchrone et, en absence de pannes, le message arrive et est traité en une unité de temps. Une mesure de coût très différente est la *complexité en temps causal*, $\boldsymbol{T}_{causal}$. Elle est définie comme étant la longueur de la plus longue chaîne de transmissions de messages causalement reliés, en considérant toutes les exécutions possibles. Le temps causal est rarement utilisé et est très difficile à mesurer avec exactitude ; on l'utilisera uniquement lorsqu'on aura affaire à des calculs synchrones.

1.6 EXEMPLE : LA DIFFUSION (BROADCASTING)

Clarifions les concepts vus jusqu'à présent par un exemple. Considérons un système de calcul distribué où une entité possède une information importante inconnue des autres entités et voudrait la partager avec toutes les autres.

Ce problème est connu sous le nom de *diffusion* (*broadcasting*) et fait partie d'une classe générale de problèmes appelée *diffusion d'information* (*information diffusion)*. Résoudre ce problème signifie concevoir un ensemble de règles qui, lors de l'exécution par les entités, conduira (au bout d'un temps fini) à la situation ou toutes les entités connaissent l'information ; la solution doit fonctionner quelque soit l'entité qui détient l'information au départ.

Soit ε la collection des entités et $\vec{G}$ la topologie de communication.

Pour simplifier la discussion, nous ferons des suppositions additionnelles (i.e., restrictions) sur le système :

1. Liaisons bidirectionnelle ; càd, nous considérons le graphe non-orienté *G*. (voir section 1.4.2).

2. Fiabilité totale, càd, nous nous ne soucierons pas des pannes.

Notons que si *G* est déconnecté, certaines entités ne recevront jamais l'information et le problème de diffusion est insoluble. Par conséquent, une restriction, différentes des précédentes, doit être faite, il s'agit de :

3. Connectivité ; càd, *G* est connecté.

Notons encore que dans la définition même du problème, il y a une supposition que seule l'entité possédant l'information lance l'exécution du protocole. Ainsi, la restriction découlant de la définition s'écrit

4. Initiateur Unique, càd, une seule entité lance l'exécution.

Une stratégie simple pour résoudre le problème de diffusion est la suivante :

"Si une entité connaît l'information, elle la partage avec ses voisines"

Pour construire l'ensemble des règles implémentant cette stratégie, nous avons besoin de définir l'ensemble S des valeurs des statuts ; de l'énoncé du problème il apparaît clairement qu'on a besoin de distinguer entre l'entité qui initialement possède l'information est les autres : $\{initiator, idle\} \subseteq S$. Le processus ne peut être lancé que par l'initiatrice. Soit I l'information à diffuser. Dans la suite, nous donnons l'ensemble des règles $B(x)$ (qui sont les mêmes pour toutes les entités) :

1. INITIATOR × I → {SEND(I) TO N(X)}
2. IDLE × RECEIVING(I) → {PROCESS(I); SEND(I) TO N(X)}
3. INITIATOR × RECEIVING(I) → NIL
4. IDLE × I → NIL

Où *i* représente l'événement d'impulsion spontanée et **Nil** l'action nulle.

Grâce aux restrictions de connectivité et de fiabilité totale, chaque entité recevra éventuellement l'information. Donc, le protocole réalise son objectif et résout le problème de diffusion.

Cependant, il y a un problème sérieux avec ces règles :

L'activité générée par le protocole ne se termine jamais.

Considérons, par exemple, un système simple de trois entités *x*, *y*, *z* connectées les unes aux autres (voir figure 1.3). Supposons que *x* a le statut *initiator*, *y* et *z* ont le statut *idle*, et tous les messages circulent à la même vitesse ; alors *y* et *z* s'enverront mutuellement des messages (vers *x* aussi).

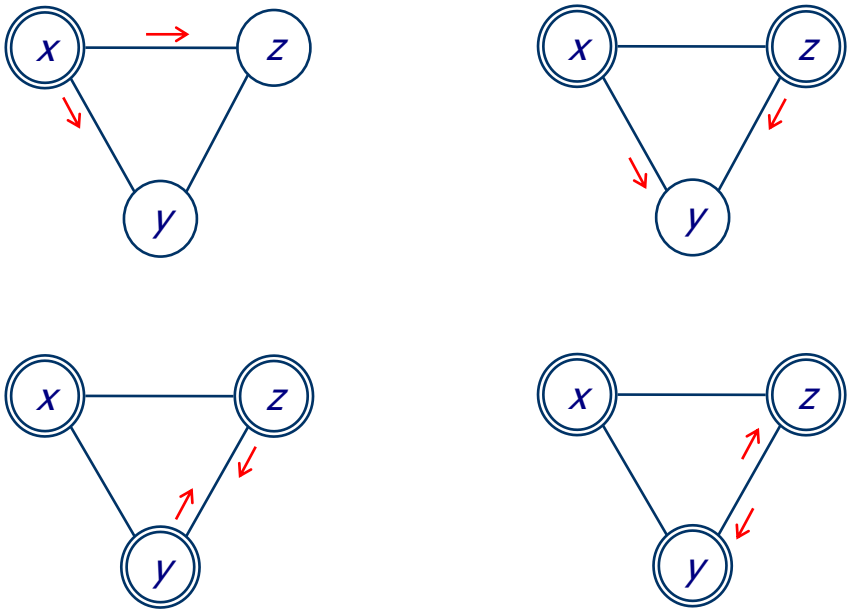

FIGURE 1.3: Exécution de la diffusion

Pour éviter cet effet indésirable, une entité ne doit envoyer l'information à ses voisines qu'une seule fois : lorsqu'elle acquière l'information pour la première fois. Ceci peut être atteint en introduisant un nouveau statut *done* ; càd, *S = {initiator, idle, done}.*

1. INITIATOR × I → {SEND(I) TO N(X); BECOME DONE }
2. IDLE × RECEIVING(I) → {PROCESS(I); BECOME DONE; SEND(I) TO N(X)}
3. INITIATOR × RECEIVING(I) → NIL
4. IDLE × I → NIL
5. DONE × RECEIVING(I) → NIL
6. DONE × I → NIL

Où **become** dénote l'opération de changement de statut.

Cette fois l'activité de communication du protocole se termine : au bout d'un temps fini, toutes les entités passent à l'état *done* ; vu qu'une entité dans cet état connaît l'information, le protocole est correct. Notons qu'on fonction des délais de transmission, différentes exécutions sont possibles. Une exécution possible serait celle donnée en figure 1.3 ou *x* est l'initiatrice du protocole.

IMPORTANT. Notons que les entités terminent leur exécution du protocole (i.e., passent à l'état *done*) à des moments différents ; il est réellement possible qu'une entité termine alors que d'autres n'ont pas encore commencé. C'est une situation typique des calculs distribués : il y a une différence entre la *terminaison locale* et la *terminaison globale*.

IMPORTANT. Notons aussi que dans ce protocole, personne ne peut savoir quand le processus entier est fini. La *détection de la terminaison* sera traitée dans les chapitres suivants. L'ensemble des règles précédentes résout le problème de la diffusion.

Calculons maintenant le coût des communications de l'algorithme. Avant tout, déterminons le nombre de transmissions de messages. Chaque entité, initiatrice ou non, envoie l'information à toutes ses voisines. Donc le nombre total de messages transmis est exactement :

$$\sum_{x \in \mathcal{E}} N(x) = 2|E| = 2m$$

On peut réellement réduire ce coût. Actuellement, lorsqu'une entité en état *idle* reçoit le message, elle le diffusera à toutes ses voisines, y compris l'entité à partir de laquelle elle a reçu l'information, ce qui est clairement non nécessaire. Rappelons que selon axiome de l'orientation locale, une entité peut distinguer entre ses voisines ; en particulier, lors du traitement du message, elle peut identifier de quel port il est reçu et évite d'envoyer un message sur ce dernier. Le protocole final reste comme précédemment avec seulement une petite modification.

PROTOCOLE D'INONDATION

1. INITIATOR × I → {SEND(I) TO N(X); BECOME DONE}
2. IDLE × RECEIVING(I) → {PROCESS(I); BECOME DONE; SEND(I) TO N(X)-SENDER}
3. INITIATOR × RECEIVING(I) → NIL
4. IDLE × I → NIL
5. DONE × RECEIVING(I) → NIL
6. DONE × I → NIL

Où **sender** est l'entité voisine ayant envoyé le message en cours de traitement. Cet algorithme est dit *algorithme d'Inondation* vu que le système entier est *inondé* par le message durant son exécution, et est un outil algorithmique de base pour les calculs distribués. Concernant le nombre de messages transmis, nécessaires à l'algorithme d'inondation, et vu que nous évitons de transmettre certains messages, il est inférieur à $2m$:

$$M[Inondation] = 2m - n + 1 \qquad (1.1)$$

Examinons maintenant la complexité en temps idéal de l'algorithme d'inondation. Soit $d(x, y)$ la distance (i.e., la longueur du plus court chemin) entre x et y dans G. Clairement, le message envoyé par l'entité initiatrice doit parvenir à chaque entité du système, y compris celle qui est la plus éloignée de l'initiatrice. Donc, si x est l'initiatrice, la complexité en temps idéal sera $r(x) = Max\ \{d(x, y) : y \in \mathcal{E}\}$, que nous appelons excentricité (ou rayon) x. En d'autres termes, le temps total dépend de l'entité initiatrice et ne peut être connu préalablement avec précision. Cependant, nous pouvons déterminer exactement la complexité en temps idéal dans le cas le plus mauvais.

Vu que n'importe quelle entité peut être initiatrice, la complexité en temps idéal dans le cas le plus mauvais sera $d(G) = Max \{r(x) : x \in \mathcal{E}\}$, qui est le diamètre de G. En d'autres termes, la complexité en temps idéal sera au plus égale au diamètre de G :

$$\boldsymbol{T}[Inondation] \leq d(G) \qquad (1.2)$$

1.7 ETATS ET EVENEMENTS

Une fois que nous avons défini le comportement des entités, leur topologie de communication, et l'ensemble des restrictions sous lesquelles elles opèrent, nous devons décrire les conditions initiales de notre environnement. Ceci est fait en premier par la spécification des conditions initiales de toutes les entités. Le contenu initial de tous les registres de l'entité x et la valeur initiale de son horloge alarme c_x au temps t constitue l'état interne initial $\sigma(x, 0)$ de x. Notons par $\Sigma(0) = \{\sigma(x, 0) : x \in \mathcal{E}\}$ l'ensemble de tous les états internes initiaux.

Une fois $\Sigma(0)$ défini, nous complétons la spécification *statique* de notre environnement : la description du système avant l'occurrence de tout événement et avant que toute activité ne soit lancée.

Nous nous intéressons aussi à la description du système *durant* les activités de calcul aussi bien qu'*après* celles-ci. Pour ce faire, nous avons besoin de pouvoir décrire les changements que le système subi dans le temps. Comme mentionné précédemment, les entités (et, donc l'environnement) sont *réactives*. Càd, que toute activité est déterminée entièrement par les événements externes. Nous examinant ces faits dans ce qui suit.

1.7.1 Temps et événements

Dans un environnement de calcul distribué, il n'y a que trois types d'événements externes : Impulsion spontanée (*Spontaneously*), Réception d'un message (*Receiving*) et déclenchement de l'alarme de l'horloge (*When*). Lorsqu'un événement externe a lieu dans une entité, il déclenche l'exécution d'une action (la nature de l'action dépend du statut de l'entité à l'occurrence de l'événement). L'exécution d'une action peut générer de nouveaux événements : l'opération **send** génèrera un événement *Receiving*, et l'opération **set alarm** génèrera un événement *when*.

Notons avant tout que les événements générés ainsi peuvent ne pas avoir lieu totalement. Par exemple, une panne de liaison pourrait détruire le message en acheminement, détruisant de même l'événement *Receiving* correspondant. Dans une action ultérieure, une entité peut désarmer l'horloge alarme précédemment armée détruisant du coup l'événement *when*.

Notons aussi que si ces événements ont lieu, ils auront lieu ultérieurement dans le temps (i.e., à l'arrivée d'un message, au déclenchement de l'alarme). Ce délai pourrait être connu avec précision dans le cas de l'horloge alarme (car armée par l'entité); il est, cependant, imprédictible dans le cas de la transmission de messages (car il dépend de conditions externes à l'entité). Des délais différents donnent naissance à des exécutions différentes des mêmes protocoles avec éventuellement des sorties différentes.

En résumé, chaque événement *e* est généré à un temps *t(e)* et, en cas d'occurrence, il aura lieu plus tard dans le temps.

Par définition, les impulsions spontanées sont déjà générées avant le début de l'exécution ; leur ensemble sera nommé *ensemble des événements initiaux*. L'exécution, du protocole commence lorsque la première impulsion spontanée a lieu réellement ; par convention ceci correspondra au temps *t=0*.

IMPORTANT. Remarquons que le "temps" est considéré ici comme celui d'un observateur externe et est vu comme le temps réel. Chaque instant du temps réel *t* sépare l'axe du temps en trois parties : *passé* (i.e., $\{t' < t\}$), *présent* (i.e., *t*), et *futur* (i.e., $\{t' > t\}$). Tous les événements générés avant *t* et qui auront lieu après *t* sont appelé *futur* à l'instant *t* et sont notés par *Future(t)*; c'est l'ensemble des événements futurs déterminé par l'exécution jusqu'à présent.

Une exécution est totalement décrite par la séquence des événements qui ont lieu. Pour les petits systèmes, une exécution peut être visualisée par ce qui est appelé un diagramme *Time* × *Event* (TED). Un tel diagramme se compose de lignes temporelles, un pour chaque entité dans le système. Chaque événement est représenté dans un tel diagramme comme suit :

- Un événement *Receiving r* est représenté par une flèche allant du point $t_x(r)$ dans la ligne temporelle de l'entité *x* générant *e* (i.e., envoyant le message) au point $t_y(r)$ dans la ligne temporelle de l'entité *y* où l'événement a lieu (i.e., réception du message).

- Un événement *When w* est représenté par une flèche allant du point $t_x'(w)$ au point $t_x''(w)$ dans la ligne temporelle de l'entité ayant armé l'horloge alarme.

- Un événement *Spontaneously ı* est représenté comme une courte flèche indiquant le point $t_x(ı)$ dans la ligne temporelle de l'entité *x* où l'événement a lieu.

Par exemple, la figure 1.4 montre un diagramme TED correspondant à l'exécution du protocole d'inondation de la figure 1.3.

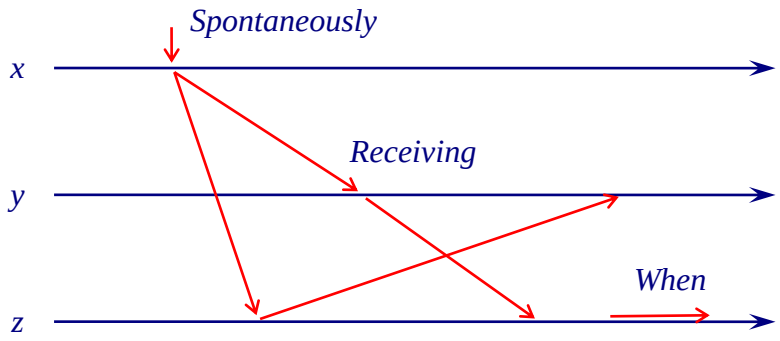

FIGURE 1.4: Diagramme TED

1.7.2 Etats et Configurations

La mémoire privée de chaque entité comprend, en plus du comportement, un ensemble de registres dont certains sont déjà initialisés et d'autres le seront durant l'exécution. Le contenu de tous les registres de l'entité x et la valeur de son horloge alarme c_x à l'instant t constitue ce qu'on appelle *état interne* de x à l'instant t, il est noté $\sigma(x, t)$. Nous notons par $\Sigma(t)$, l'ensemble des états internes à l'instant t de toutes les entités. Les états internes changent avec l'occurrence des événements.

Il existe un fait important relatif aux états internes. Considérons deux environnements différents ε_1 et ε_2, où, par accident, l'état interne de x à l'instant t est le même. Alors x *ne peut pas distinguer entre les deux environnements, càd,* x n'est pas capable de déterminer s'il est dans l'environnement ε_1 ou ε_2.

De là découle une conséquence importante. Considérons la situation décrite précédemment : A l'instant t, l'état interne de x est le même dans les deux environnements ε_1 et ε_2. Supposons, maintenant, que par accident aussi, exactement le même événement a lieu en x (e.g., déclenchement de l'horloge alarme ou le même message est reçu du même voisin). Alors, x exécutera exactement la même action dans les deux cas, et son état interne continuera à être le même dans les deux situations.

Propriété 1.6.1 *Supposons l'occurrence du même événement à l'instant t dans x, dans deux exécutions différentes, et soient σ_1 et σ_2 les états internes au moment de l'occurrence. Si $\sigma_1 = \sigma_2$, alors les deux états internes de x seront les mêmes dans les deux exécutions.*

De manière similaire, si deux entités ont le même état interne, elles ne peuvent se distinguer l'une de l'autre. De plus, si par accident, le même événement a lieu aux niveaux des deux entités (e.g., déclenchement de l'horloge alarme ou réception du même message du même voisin), alors, elles effectueront exactement la même action dans les deux cas, et leur état interne restera le même dans les deux situations.

Propriété 1.6.2 *Supposons l'occurrence du même événement en x et y à l'instant t, et soient σ_1 et σ_2 leurs états internes respectifs à cet instant. Si $\sigma_1 = \sigma_2$, alors le nouvel état interne de x et y sera le même.*

Rappel : Les états internes sont locaux et l'entité n'est pas capable d'inférer de ces états une information qui concerne le statut du reste du système.

Pour décrire l'état *global* d'un environnement à l'instant *t*, nous avons besoin, évidemment de spécifier l'état interne de toutes les entités à cet instant ; càd, l'ensemble $\Sigma(t)$. Cependant, cela n'est pas suffisant. En effet, l'exécution jusqu'à présent pourrait avoir déjà généré des événements qui auront lieu après l'instant t ; ces événements, représentés par l'ensemble *Future*(*t*), sont une partie intégrale de l'exécution et doivent être spécifiés aussi. Explicitement, l'état global, appelé *configuration*, d'un système durant son exécution est spécifié par le couple :

$$C(t) = (\Sigma(t), Future(t))$$

La configuration *initiale C(0)* contient non seulement l'ensemble initial des états $\Sigma(0)$ mais aussi l'ensemble *Future(0)* des impulsions spontanées. Les environnements qui diffèrent uniquement dans leur configuration initiale seront nommés *instances* du même système. La configuration *C(t)* ressemble à une photographie instantanée du système à l'instant *t*.

1.8 PROBLEMES ET SOLUTIONS

L'objectif de ce cours est comment concevoir des algorithmes distribués et analyser leur complexité. Un algorithme distribué n'est autre qu'un ensemble de règles qui régissent les *comportements* des entités. Nous avons besoin de concevoir les comportements pour permettre aux entités de résoudre un problème donné, effectuer une certaine tâche ou fournir un service demandé.

En général, nous disposerons d'un problème donné et notre tâche est de concevoir un ensemble de règles qui résolvent toujours le problème en un temps fini. Nous discutons ces concepts dans ce qui suit.

Problèmes. Enoncer un problème (ou une tâche, ou un service) P signifie donner une description de ce que les entités doivent accomplir (c'est le *what*). Ceci est fait en spécifiant ce que sont les conditions initiales des entités (donc du système), et ce que doivent êtres les conditions finales ; il faut aussi spécifier toutes les restrictions données. En d'autres termes,

$$P = <P_{INIT}, P_{FINAL}, R>$$

Où P_{INIT} et P_{FINAL} sont des prédicats sur les valeurs des registres des entités, et R est un ensemble de restrictions.

Soit $w_t(x)$ la valeur d'un registre d'entrée $w(x)$ à l'instant t et $\{w_t\} = \{w_t(x) : x \in \mathcal{E}\}$ les valeurs de ce registre dans toutes les entités au même instant. Ainsi, par exemple, $\{status_0\}$ représente la valeur initiale du registre de statut de toutes les entités.

Par exemple, dans le problème de la diffusion de l'information I (*Broadcasting (I)*) décrit en section 1.6, les conditions initiales et finales sont données par les prédicats :

$P_{INIT}(t)$ ≡ " *Seule une entité possède l'information à l'instant t"*

≡ $\exists x \in \mathcal{E}\ (value_t(x) = I \ \wedge \ \forall y \neq x\ (value_t(y) = \emptyset))$

$P_{FINAL}(t)$ ≡ " *chaque entité possède l'information à l'instant t"*

≡ $\forall x \in \mathcal{E}\ (value_t(x) = I)$

Les restrictions que nous avons imposées sur notre solution sont BL (Liaisons Bidirectionnelles), TR (Fiabilité Totale), et CN (Connectivité). Il existe une condition implicite dans le problème que seule l'entité possédant l'information lancera l'exécution du protocole. Appelons UI (*Unique Initiator*) le prédicat décrivant cette restriction. En résumé, pour la diffusion, l'ensemble des restrictions faites est {BL, TR, CN, UI}.

Statut. Un protocole solution B pour $P = <P_{INIT}, P_{FINAL}, R>$ spécifiera *comment* les entités accomplirons la tâche requise. Une partie de la conception de l'ensemble des règles $B(x)$ concerne la définition de l'ensemble des valeurs des statuts S, càd, les valeurs maintenues par le registre de statut $status(x)$.

Nous appelons valeurs *initiales* du statut les valeurs de S contenues dans les registres *status* au lancement de l'exécution de $B(x)$ et que nous notons par S_{INIT}. A l'opposé, les valeurs *terminales* du statut sont les valeurs qui une fois atteintes, ne peuvent plus être changées par le protocole. Leur ensemble sera nommé S_{TERM}. Toutes les autres valeurs de S seront appelées valeurs *intermédiaires du statut*.

Par exemple, dans le protocole Inondation décrit dans la section 1.6, $S_{INIT}=\{initiator, idle\}$ et $S_{TERM}=\{done\}$.

En fonction des restrictions du problème, seules les entités ayant des valeurs initiales de statut spécifiques peuvent lancer l'exécution du protocole ; nous notons par $S_{START} \subseteq S_{INIT}$ l'ensemble de ces valeurs de statut. Typiquement, S_{START} consiste en un seul statut ; par exemple, dans le protocole d'Inondation, $S_{START}=\{initiator\}$. Il est possible de réécrire le protocole de telle sorte à ce que ceci soit toujours le cas.

Parmi, les valeurs terminales du statut, on devra distinguer celles à partir desquelles aucune activité supplémentaire ne peut avoir lieu ; càd, celles où la seule action est **Nil**

Nous appellerons de telles valeurs *finales* et nous notons $S_{FINAL} \subseteq S_{TERM}$ l'ensemble de ces valeurs de statut. Par exemple, dans l'algorithme d'inondation $S_{FINAL}=\{done\}$.

Terminaison. Le protocole *B* se termine si, pour toutes les configurations initiales *C(0)* satisfaisant P_{INIT}, et pour toutes les exécutions lancées à partir de ces configurations, le prédicat

$$Terminate\ (t) \equiv (\{status_t\} \subseteq S_{TERM}) \wedge (Future(t)= \varnothing)$$

est vérifié pour une certaine valeur de $t > 0$, càd, toutes les entités entrent dans un statut terminal après un temps fini et tous les événements générés ont eu lieu.

Nous avons déjà soulevé la remarque sur le fait que les entités pourraient ne pas se rendre compte que la terminaison a eu lieu. En général, nous souhaiterons que chaque entité se rende compte au moins de sa terminaison. Cette situation, appelée *terminaison explicite*, a lieu si le prédicat

$$Explicit\text{-}Terminate\ (t) \equiv (\{status_t\} \subseteq S_{FINAL})$$

est vérifié pour une certaine valeur de $t > 0$, càd, toutes les entités entrent dans un statut final après un temps fini.

Correctness (Correction). Le protocole *B* est correct si, pour toutes exécutions commençant à partir de configurations initiales satisfaisant P_{INIT}, $\exists t > 0$: $Correct(t)$ est vérifié. Avec $Correct(t) \equiv (\forall t' \geq t, P_{FINAL}(t'))$; càd, le prédicat final sera éventuellement vérifié et ne changera pas.

Protocole Solution. L'ensemble des règles *B* résout le problème *P* s'il se termine toujours correctement sous les restrictions *R* du problème. Comme il y a deux types de terminaison (simple et explicite), nous aurons deux types de solutions :

Simple Solution[B,P] où le prédicat

$$\exists t > 0\ (Correct(t) \wedge Terminate(t))$$

est vérifié sous les restrictions *R* du problème pour toutes les exécutions commençant à partir des configurations initiales satisfaisant P_{INIT}; et

Explicit Solution[B,P] où le prédicat

$$\exists t > 0\ (Correct(t) \wedge Explicit\text{-}Terminate(t))$$

est vérifié sous les restrictions *R* du problème pour toutes les exécutions commençant à partir des configurations initiales satisfaisant P_{INIT};

1.9 ACQUISITION DE LA CONNAISSANCE

La notion de l'information et la connaissance est fondamentale dans le calcul distribué. De manière informelle, tout calcul distribué peut être vu comme le processus d'acquisition d'information à travers une activité de communication ; à l'inverse, la réception d'un message peut être vue comme le processus de transformation de l'état de connaissance du processeur (entité) recevant le message.

1.9.1 Niveaux des Connaissances

Le contenu de la mémoire locale d'une entité et l'information qui peut en être dérivée constitue la connaissance locale de l'entité (*local knowledge*). Nous notons par

$$p \in LK_t[x]$$

le fait que p est une connaissance locale au niveau de x à l'instant du temps global t. Par définition, $\lambda_x \in LK_t[x]$ pour tout t, càd, les arcs (labels) entrant vers/ sortant de x sont des connaissances locales de x qui sont temporellement invariantes.

Parfois, il est nécessaire de décrire la connaissance supportée par plus d'une entité à un moment donné. L'information p est dite connaissance implicite (*implicit knowledge*) dans $W \subseteq \varepsilon$ à l'instant t, et notée $p \in IK_t[W]$, si au moins une entité dans W connaît p *à l'instant* t, càd,

$$p \in IK_t[W] \text{ ssi } \exists x \in W\ (p \in LK_t[x])$$

Un niveau plus fort de connaissance dans un groupe W d'entités est vérifié quand, à un instant donné t, p est connu à chaque entité du groupe. Nous notons cela par $p \in EK_t[W]$, càd,

$$p \in EK_t[W] \text{ ssi } \forall x \in W\ (p \in LK_t[x])$$

Dans ce cas, p est dite connaissance explicite dans $W \subseteq \varepsilon$ (*explicit knowledge*) à l'instant t.

Considérons par exemple la diffusion discutée dans la section précédente. Initialement, à l'instant $t = 0$, seule l'entité initiatrice s connaît l'information I; en d'autres termes, $I \in LK_0[s]$. Ainsi, à cet instant, I est implicitement connue par toutes les entités ; càd $I \in IK_0[\varepsilon]$. A la fin de la diffusion, à l'instant t', toutes les entités connaîtront l'information, en d'autres termes, $I \in EK_{t'}[\varepsilon]$.

Notons que, en absence de pannes, la connaissance ne peut pas être perdue mais seulement acquise, càd, pour tout $t' > t$ et tout $W \subseteq \varepsilon$, si aucune panne n'a lieu, $IK_t[W] \subseteq IK_{t'}[W]$ et $EK_t[W] \subseteq EK_{t'}[W]$.

Supposons qu'un fait p soit une connaissance explicite dans W à l'instant t. Il est possible que certaines entités (éventuellement toutes) ne soient pas au courant de cette situation. Par exemple, supposons qu'à l'instant t, les entités x et y connaissent la valeur d'une variable de z, disant son ID; alors ID de z est une connaissance explicite dans $W=\{x, y, z\}$; cependant, z pourrait ne pas savoir que x et y connaissent son ID. En d'autres termes, lorsque $p \in EK_t[W]$, le fait que "$p \in EK_t[W]$" pourrait même ne pas être connu localement à toutes les entités de W.

Ceci donne naissance au niveau le plus élevé de connaissance dans le groupe : Connaissance commune (common knowledge). L'information p est dite connaissance commune dans $W \subseteq \varepsilon$ à l'instant t , et notée $p \in CK_t[W]$, si et seulement si à l'instant t toute entité de W connaît p, et sait que toute entité de W connaît p, et sait que toute entité de W sait que toute entité de W connaît p, et … etc. Càd,

$$p \in CK_t[W] \text{ ssi } \wedge_{1 \le i \le \infty} P_i$$

Où les Pi sont les prédicats définis par

$$P_1 = [p \in EK_t[W]] \text{ et } Pi+1 = [P_i \in EK_t[W]].$$

Dans la majorité des problèmes distribués ; il sera nécessaire aux entités d'atteindre la *connaissance commune*. Cependant, nous n'avons pas à aller jusqu'à ∞ pour atteindre la connaissance commune, un nombre fini d'étapes pourra le faire, comme indiqué par l'exemple suivant.

Exemple (Front boueux (muddy forehead)): Imaginons n élèves intelligents et perceptifs qui jouent ensemble durant la récréation. Il leur est interdit de jouer dans les flaques de boues et la maîtresse leur a dit que s'ils le faisaient, cela entraînerait des conséquences sévères. Chaque élève veut rester propre, mais la tentation de jouer avec la boue est irrésistible. Par conséquent, k élèves se sont salis le front avec la boue. Lorsque la maîtresse arrive, elle leur dit : « Je vois que certains d'entre vous ont été jouer dans la flaque de boue : la boue sur vos fronts vous dénonce ! » et elle continue : « les coupables qui se présentent spontanément recevront une punition légère ; ceux qui ne le font pas recevrons une punition qu'ils n'oublieront pas facilement ». Après, elle ajoute, « je vais quitter la classe, et je reviendrai périodiquement ; si vous décidez d'avouer, vous devez vous présenter tous ensemble quand je suis dans la classe. En attendant, chacun doit rester absolument calme sans parler ».

Chaque élève dans la classe comprend clairement que ceux qui ont la boue sur le front sont « cuits », et seront punis peu importe comment. Évidemment, les élèves ne veulent pas avouer si leurs fronts sont propres, et si leurs fronts sont salis, ils se présenteront ensemble pour éviter la punition terrible de ceux qui n'avouent pas. Comme tous les élèves sont concernés, leur objectif commun est que les élèves ayant

un front propre n'avouent pas et ceux ayant un front sali avouent simultanément et cela sans communication.

Examinons cet objectif. La première question est comme suit : un élève x peut-il savoir si son front est sali ou non ? Il peut voir combien d'autres sont salis, par exemple un nombre *fx* parmi les autres élèves. Ainsi, la question devient : *x* peut-il déterminer si *k* = *fx* ou *k* = *fx* + 1 ?

La seconde question, plus complexe, est la suivante : les élèves avec la boue sur leurs fronts peuvent-ils le déduirent *tous* en même temps de telle sorte à pouvoir se présenter ensemble ? En d'autres termes, la valeur *k* peut elle devenir une connaissance commune ?

Les élèves, étant intelligents et perceptifs, déterminent que la réponse aux deux questions est positive et trouvent le moyen d'atteindre le but et donc la connaissance commune sans communiquer.

En effet, dans le cas ou *k = 1*, un seul élève, par exemple *z*, a le front sali. *z* verra que tous les autres ont les fronts propres, comme la maîtresse a dit qu'au moins un élève a le front sali, *z* sait que ça ne peut être que lui. Ainsi, lorsque la maîtresse arrive, il se présentera. Notons qu'un élève propre voit que *z* est sali mais déduit que son front est propre lorsque *z* se présente.

Considérons maintenant le cas où *k = 2* : Il y a deux élèves, *x* et *y*; avec le front sali. *x voit que y a le front sali et que tous les autres sont propres. Evidemment, il ignore son statut, s'il est propre, y est le seul qui a le front sali et se présentera lorsque la maîtresse arrivera.* Donc si la maîtresse arrive et y ne se présente pas, x comprendra qu'il a le front sali (le même raisonnement est fait par y). Ainsi, au retour de la maîtresse, x et y se présenteront tous les deux.

IMPORTANT. Lorsqu'on travail avec un sous-modèle, toutes les restrictions définissant le sous-modèle sont une connaissance commune à toutes les entités, sauf spécification contraire.

1.9.2 Types de Connaissances

On peut avoir divers types de connaissances, telle que la connaissance sur la topologie de communication, l'étiquetage du graphe de communication, données d'entrée des entités communicantes. En général, si nous avons une certaine connaissance du système, nous pouvons l'exploiter pour réduire le coût du protocole, bien que cela aura pour conséquence de rendre plus limitée l'applicabilité du protocole.

Un type de connaissance particulièrement intéressant est celui concernant la topologie de communication (i.e., graphe *G*). En fait, comme il sera vu ultérieurement, la

complexité d'un calcul peut varier grandement en fonction de ce que les entités connaissent sur *G*. Dans ce qui suit, nous donnons certains éléments qui s'ils deviennent des connaissances communes aux entités peuvent affecter la complexité.

1. *Information Métrique* : informations numériques sur le réseau ; par exemple, le nombre de nœuds $n = |V|$, le nombre de liaisons $m = |E|$, le diamètre, circonférence, etcetera. Cette information peut être *exacte* ou *approximative*.

2. *Propriétés Topologique* : Connaissance de certaines propriétés de la topologie ; par exemple, " $\vec{G}$ est un anneau", " $\vec{G}$ n'a pas de cycle", " $\vec{G}$ est un graphe de Cayley", etcetera.

3. *Carte topologique* : Une carte du voisinage de l'entité jusqu'à une distance *d*, une carte complète de $\vec{G}$ (e.g., matrice d'adjacence de $\vec{G}$); une carte complète de $(\vec{G},\lambda)$ (i.e., contenant les labels aussi), etcetera.

Notons que certains types de connaissances impliquent d'autres connaissances. Par exemple, si une entité avec *k* voisins sait que le réseau est un graphe non orienté complet, alors elle sait que $n = k+1$.

Comme une carte topologique fournie toutes les informations métriques et structurelles, ce type de connaissance est puissant et important. La forme la plus forte de ce type est la *connaissance topologique totale* : disponibilité au niveau de chaque entité d'un graphe étiqueté isomorphique à $(\vec{G},\lambda)$, l'isomorphisme, et sa propre image. Càd, chaque entité a une carte complète de (v,λ) avec une indication, "Vous êtes ici". Un autre type de connaissance concerne l'étiquetage λ. Ce qui est très important c'est de savoir si l'étiquetage a une propriété globale de consistance.

On peut distinguer deux autres types, en fonction du fait que la connaissance est sur les données (d'entrée) ou le statut des entités et celui du système. Nous les appellerons type-D et type-S, respectivement.

Exemples de connaissance de type-D :

Identificateur unique : toutes les valeurs d'entrée sont distinctes
Multiset: les valeurs d'entrée ne sont pas nécessairement identiques
Taille: nombre de valeurs distinctes.

Exemples de connaissance de type-S :

Système avec leader: Il existe une seule entité avec le statut de "leader"
Reset: tous les nœuds sont dans le même statut
Initiator Unique: une seule entité a le statut "initiator." par exemple, dans le problème de diffusion, nous avons supposé que cette information était une partie de la définition du problème.

1.10 CONSIDERATIONS TECHNIQUES

1.10.1 Messages

Le contenu d'un message dépend évidemment de l'application. Dans tous les cas, il consiste en une séquence finie (usuellement limitée) de bits. Le message est typiquement divisé en sous-séquences, appelées *champs*, avec un sens prédéfini ("type") dans le protocole.

Comme types de champs :

> *Identificateur de message* ou *entête* utilisée pour distinguer entre différents types de messages ;
> *Champs émetteur et destination* utilisés pour spécifier l'entité (son identité) émettrice du message et l'entité à laquelle le message est destiné.
> *Champs de données* utilisés pour contenir les informations nécessaires au calcul (la nature de l'information dépend de l'application considérée).

Ainsi, en général, un message M sera vu comme un tuple $M = <f_1, f_2, \ldots f_k>$ où k est une constante prédéfinie (petite), et chaque f_i $(1 \leq i \leq k)$ est un champ d'en le type est spécifié, et chaque type a une longueur fixe.

Ainsi, par exemple, dans le protocole d'inondation, il y a uniquement un type de message composé de deux champs $M = <f_1, f_2>$ où f_1 est un identificateur de message (contenant l'information : "ceci est un message de diffusion"), et f_2 est un champ de données contenant l'information I en diffusion.

Si la taille d'un message (sa limite) est un paramètre du système (i.e., elle ne dépend pas d'une application particulière), on dira que le système à des *messages limités* (*bounded messages*). Un tel cas est celui, par exemple, de la limite imposée sur la taille du message dans les réseaux à commutation de paquets, de même pour la taille des messages de contrôle dans les réseaux à commutation de circuits (e.g., réseau téléphonique) et les réseaux à commutation de messages.

Les messages limités sont aussi appelés *paquets* et contiennent au plus $\mu(G)$ bits, où $\mu(G)$ est la limite qui dépend du système appelée *taille de paquets* (*packet size*). Notons que la transmission d'un message de K bits sur G requière la transmission d'au moins $\lceil K/\mu(G) \rceil$ paquets.

1.10.2 Protocoles

Notation. Un protocole *B*(*x*) est un ensemble de règles. Nous avons déjà introduit la notation décrivant ces règles. Nous complétons cette notation par les conventions suivantes :

1. Les règles seront groupées par statut.

2. Si l'action pour une paire (*statut*, *événement*) est **Nil**, alors, pour la simplicité, la règle correspondante sera omise dans la description. Par conséquent, si aucune règle n'est décrite pour une paire (*statut*, *événement*), alors par défaut la paire n'active que l'action nulle. Bien que convenable (simplifie l'écriture), l'utilisation de cette convention doit se faire avec attention : si on oublie d'écrire une règle pour un événement qui a lieu dans un état donné, c'est l'action nulle qui sera considérée au moment de l'exécution.

3. Si l'action contient un changement de statut, cette opération sera la dernière opération avant de terminer l'action.

4. L'ensemble des valeurs de statut du protocole et l'ensemble des restrictions sous lesquelles opère le protocole seront explicites.

En utilisant ces conventions, la figure 1.5 montre comme le protocole d'inondation défini en section 1.6 est réécrit.

```
PROTOCOL Flooding.
• Valeurs de statut: S = {INITIATOR, IDLE, DONE};
      SINIT = {INITIATOR, IDLE};
      STERM = {DONE}.
• Restrictions: Liaisons bidirectionnelles, Fiabilité Totale,
  Connectivité, et Initiateur Unique.
INITIATOR
      Spontaneously
      begin
         send(M) to N(x);
         become DONE;
      end
IDLE
      Receiving(I)
      begin
         Process(M);
         send(M) to N(x) − {sender};
         become DONE;
      end
```

FIGURE 1.5: Protocole d'inondation

Précédence. Les événements externes (*Spontaneously*, *Receiving*, *When*) peuvent avoir lieu simultanément. Par exemple, l'alarme de l'horloge peut se déclencher au même moment que l'arrivée d'un message. Les événements simultanés seront traités séquentiellement en utilisant la précédence suivante :

Autrement dit, l'impulsion spontanée sera prioritaire par rapport à l'horloge alarme qui est prioritaire sur l'arrivée d'un message.

Au plus une impulsion spontanée peut toujours avoir lieu dans une entité à un moment donné. Comme il n'y a qu'une seule horloge alarme localement, seul un événement de déclenchement d'alarme (*When*) peut avoir lieu à un moment donné. A l'opposé, il est possible que plusieurs messages arrivent au même moment à une entité venant de voisins différents. Si c'est le cas, ces événements (*Receiving*) simultanés seront traités séquentiellement dans un ordre arbitraire.

1.10.3 Mécanismes de Communication

Les mécanismes de communication des environnements de calcul distribués doivent manipuler les transmissions et les arrivées de messages. Les mécanismes au niveau d'une entité peuvent être vus comme un système de files.

Chaque liaison $(x, y) \in \vec{E}$ correspond à une file avec l'accès en *x* et la sortie en *y*. L'accès est dit port de sortie (*out-port*) et la sortie est dite port d'entrée (*in-port*). Chaque entité a alors deux types de ports : les ports de sortie, un pour chaque voisin de sortie (ou liaison) et des ports d'entrée, un pour chaque voisin d'entrée (ou liaison). Au niveau d'une entité, chaque port de sortie a un label distinct appelé numéro de port. Le port de sortie correspondant à (x, y) a le label $\lambda_x(x, y)$; et de manière similaire pour les ports d'entrée. Les ensembles *N*in et *N*out consisteront, dans la pratique, en les numéros de ports associés à ces voisins. Ceci est du au fait que l'entité n'a pas d'autres informations concernant les voisins (à moins qu'en ajoute des restrictions).

La commande "**send** *M* **to** *W*" permet d'envoyer une copie du message *M* envoyée sur chaque port de sortie spécifié par *W*.

Lorsqu'un message *M* est envoyé à travers un port de sortie *l*, il est inséré dans la file correspondante. En absence de pannes, le mécanisme de communication le prendra éventuellement de la file et le livre à l'autre entité par le port d'entrée correspondant, générant l'événement *Receiving*(*M*); à ce moment, *l* sera assignée à la variable **sender**.