

Précis de répartition
Aspects algorithmiques -suite
3ième Année Informatique et Mathématiques Appliquées

Gérard Padiou, Philippe Quéinnec

11 octobre 2004

Table des matières

1	Datation	3
1.1	La datation Temps réel	3
1.2	Datation Temps logique	3
1.2.1	Horloges de Lamport	4
1.2.2	Horloges de Mattern	5
2	Protocoles d'ordre causal	8
2.1	Implantation du protocole dans le cas point à point	8
2.1.1	Approche par matrice de précedence (Raynal)	8
2.1.2	Approche par gestion d'histoires	10
2.2	Implantation du protocole dans la cas de la diffusion	11
3	Tolérance aux fautes : modèle d'exécution ISIS, HORUS	13
3.1	Groupe de processus	13
3.2	Les solutions traditionnelles	14
3.2.1	Transfert de messages	14
3.2.2	Gestion des groupes et ordonnancement	14
3.2.3	Tolérance aux fautes	15
3.3	Le synchronisme virtuel	16
3.3.1	Synchronisme fort	16
3.3.2	Ordre total et ordre causal	16
3.4	La boîte à outils Isis	17
3.4.1	Hypothèses sur l'environnement	17
3.4.2	Styles de groupes	17
3.5	Programmer avec Isis	17
3.5.1	Structure de l'application	17
3.5.2	Initialisation et connexion à Isis	18
3.5.3	Déclaration des tâches et point d'entrée	18
3.5.4	Initialisation de l'application et entrée dans Isis	18
3.5.5	Groupes et clients	19
3.5.6	Diffusions et messages	20
3.5.7	Réception d'un message	21
3.5.8	Les tâches	21
3.5.9	Synchronisation par jeton	22
3.6	Horus	22

3.7 Bibliographie 23

1 Datation

Un problème de base des systèmes répartis est la datation des événements significatifs (émission et réception des messages en particulier) durant l'activité du système. Ceci permet de reconstituer l'exécution à des fins de mise au point par exemple ou de contraindre la prise en compte de requêtes dans un certain ordre.

En général, un mécanisme de datation doit respecter une règle fondamentale : être compatible avec la relation de causalité qui peut exister entre toute paire d'événements. Autrement dit,

$$\forall e, e' : e \prec e' \Rightarrow date(e) < date(e')$$

On peut distinguer deux approches :

- une approche "temps réel" consistant à dater les événements avec une horloge la plus précise possible. La difficulté est alors de disposer de cette horloge globale ;
- une approche "temps logique" consistant à dater les événements en respectant la causalité selon la règle énoncée.

1.1 La datation Temps réel

Cette approche pose le problème de la disponibilité d'une horloge globale. En effet, comme nous l'avons souligné, un système réparti ne dispose justement pas d'un référentiel global de temps. Chaque nœud possède une horloge locale plus ou moins précise et surtout non synchronisée a priori avec les horloges des autres nœuds. Une datation directe à l'aide de telles horloges ne convient donc pas car des anomalies causales peuvent être engendrées. Il suffit que l'horloge du nœud émetteur soit en avance sur celle du nœud récepteur pour obtenir un message dont la date de réception est antérieure à la date d'émission.

Pour résoudre ce problème, des algorithmes de synchronisation d'horloges ont été conçus et implantés. Ils permettent de recaler les horloges des nœuds de façon à ce que leur différence reste dans un intervalle borné connu. L'algorithme doit maintenir un invariant du type :

$$\text{invariant} \quad \text{Max}(h_i : i = 1, N) - \text{Min}(h_i : i = 1, N) < \epsilon$$

On obtient ainsi une précision ϵ qui garantira une datation correcte si tous les événements causalement liés sont séparés par un délai supérieur à la précision de l'horloge globale ainsi implantée.

La datation temps réel nécessite donc un protocole de synchronisation d'horloge complexe et relativement coûteux. La disponibilité d'un émetteur unique (diffusion de tops par une horloge atomique par exemple) peut apporter une simplification dans la mise en œuvre et plus de précision. Cependant, la solution est alors centralisée par nature et donc moins tolérante aux défaillances : défaillance de l'émetteur, mais aussi défaillance locale des récepteurs.

Enfin, pour de nombreuses applications, seul le respect de la causalité est important. Il est même parfois souhaitable de savoir si deux événements sont causalement liés ou non. Une datation temps réelle ordonne totalement tous les événements et ne permet donc pas de distinguer ceux qui sont indépendants (sans causalité) malgré leur précedence temporelle.

Face à ces inconvénients, des solutions fondées sur un temps logique ont été étudiées.

1.2 Datation Temps logique

Deux solutions ont été découvertes :

- la première, due à L. Lamport, permet de dater les événements selon un ordre total. L'inconvénient éventuel de cette approche est donc de même nature que celui d'une datation réelle : l'introduction d'un ordre arbitraire entre des événements indépendants.
- la seconde, due à F. Mattern, permet de dater les événements selon un ordre partiel isomorphe à la relation de causalité. Ce mécanisme est plus précis mais plus coûteux à implanter et permet de distinguer (détecter) les événements indépendants. Toutefois, un tel mécanisme ne permet pas de décider de l'existence d'un événement entre deux événements causalement liés.

1.2.1 Horloges de Lamport

Une date est un couple (s, cpt) , où s est un numéro de site et cpt est un entier, la numérotation des sites étant supposée totalement ordonnée. L'entier permet de comparer deux dates et en cas d'égalité, le numéro de site permet de distinguer les deux dates. Si deux dates, ayant le même champ site, n'ont jamais la même valeur entière, toutes les dates sont différentes et comparables.

Une horloge locale à chaque site permet de dater les événements ayant lieu sur le site correspondant. Cette horloge mémorise une date courante (s, cpt) où s est donc le site local de l'horloge. Pour que les dates obtenues à partir de ces horloges soient toutes différentes mais comparables, l'algorithme de gestion d'horloge doit assurer l'invariant :

$$\text{invariant } \forall d, d' : d.s = d'.s \Rightarrow (d.cpt \neq d'.cpt)$$

Pour cela, il suffit que toute consultation de l'horloge pour dater un événement, entraîne l'incrémement de l'entier compteur.

Les classes *Date* et *Horloge* suivante donnent une implantation possible du mécanisme de datation.

```
class Date implements Cloneable { // définition et comparaison de date

    protected int s ; protected int cpt ;

    static boolean Prec ( Date d1, Date d2 ) {
        return (d1.cpt < d2.cpt) ||( (d1.cpt == d2.cpt) && (d1.s < d2.s)) ;
    }
}

class Horloge extends Date {
    // Création de l'horloge
    Horloge( int où ) { s = où ; cpt = 0 ; }

    // Lecture-Incrémement de l'horloge
    Date Top()
        { Date dc = (Date) super.clone(); this.cpt++; return dc ; }

    // Recalage de l'horloge
    void Recaler( Date d ) { if (Prec(this,d)) this.cpt = d.cpt + 1 ; }
    // soit encore this.cpt = Max(this.cpt,d.cpt) + 1
}

```

Les actions suivantes seront exécutées sur occurrence de chaque type d'événement :

Type d'événement sur un site s	Action mettant en jeu l'horloge du site s
Événement interne sur s	$H_s.Top()$
Émission sur s de m	$dm = H_s.Top()$; envoi de $\langle dm, m \rangle$
Réception sur s de $\langle dm, m \rangle$	$H_s.Recaler(dm)$

Le chronogramme de la figure (1) montre l'évolution des horloges de chaque site et la surcharge des messages par la date d'émission de chaque message. On remarquera que les événements de réception ne sont pas datés. Ils ne sont l'occasion que d'un recalage de l'horloge du site de réception mais n'entraîne pas d'opération $Top()$.

Les horloges de Lamport permettent un ordonnancement total des événements d'un calcul réparti en respectant la causalité qui peut exister entre ces événements. Néanmoins, l'ordre introduit entre des événements causalement indépendants (\equiv logiquement simultanés) est arbitraire. À titre d'exemple, la figure (1) montre que l'événement a_3 a pour date $(A, 2)$ et l'événement c_2 a pour date $(C, 1)$: on a donc c_2 qui précède a_3 avec cette datation alors que dans le temps absolu c'est l'inverse qui s'est produit. Ceci n'est pas erroné puisque ces deux événements ne sont pas causalement liés.

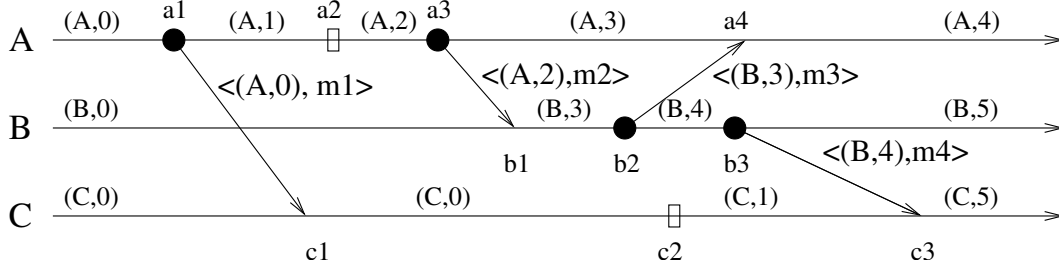


FIG. 1 – Horloges de Lamport

1.2.2 Horloges de Mattern

Nous venons de voir que le mécanisme de datation par les horloges de Lamport respecte l'ordre causal mais ne permet pas d'avoir l'implication réciproque : $\forall e, e' : d_e < d_{e'} \Rightarrow e < e'$. C'est ce que les horloges de Mattern vont assurer.

Pour ce faire, il faut passer à une datation vectorielle. Une date (globale) est un vecteur D de dimension égale au nombre de sites. La composante $D[i]$ d'un tel vecteur indique le nombre d'événements ayant eu lieu sur le site i et qui précède causalement cette date. La relation d'ordre entre 2 dates devient alors :

$$\forall D, D' : D \leq D' \equiv \forall i : D[i] \leq D'[i]$$

et

$$\forall D, D' : D < D' \equiv D \leq D' \wedge \exists k : D[k] < D'[k]$$

On peut alors constater que deux dates peuvent évidemment être non comparables puisqu'il peut exister des dates telles que :

$$D \parallel D' \equiv \neg(D < D') \wedge \neg(D' < D)$$

En fait, la relation d'ordre définie peut correctement traduire la relation de causalité par un mécanisme adéquat de gestion des horloges correspondantes.

Chaque site s gère une horloge vectorielle H_s . La datation d'un événement se produisant sur un site s entraînera l'incrémement de la composante correspondante de l'horloge locale au site. Deux événements distincts quelconques n'auront donc jamais la même date et par conséquent seule la relation $<$ nous intéresse.

Comme précédemment, chaque message sera surchargé par la date de l'événement d'émission.

La classe décrivant une date vectorielle devient :

```
class DateV { // définition et comparaison de date

    protected int cpt[] ;

    // Opérateur de précédence
    public boolean Prec ( DateV d1, DateV d2 ) {
        for ( int i = 0 ; i < cpt.length ; i++)
            if (d1.cpt[i] > d2.cpt[i]) return false ;
        return true;
    }

    // Constructeur
    DateV(int dim) {
        cpt = new int[dim] ; for ( int i=0 ; i < cpt.length ; i++) cpt[i] = 0 ;
    }
}
```

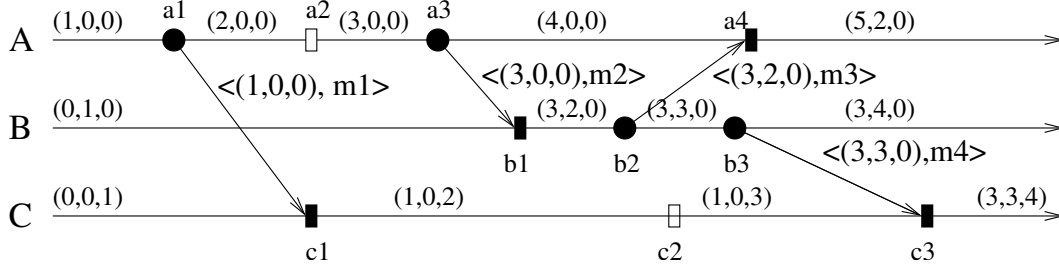


FIG. 2 – Horloges de Mattern

On peut remarquer que la notion de site n'intervient plus en composante d'une date puisque l'on a une valeur propre à chaque site. Par contre, un objet horloge reste présent sur chaque site.

Les actions associées aux événements internes, d'émission et de réception restent les mêmes que pour les horloges de Lamport, seul le type date ayant changé (voir 1.2.2).

```
class Horloge extends DateV {
    protected int s ; // localisation de l'horloge
    // Création de l'horloge
    Horloge( int où , int dim ) { super(dim) ; this.s = où ; cpt[où] = 1 ; }

    // Lecture-Incrémentation de l'horloge
    DateV Top() throws Exception
    { DateV dc = (DateV)super.clone() ; this.cpt[s]++ ; return dc ; }

    // Recalage de l'horloge
    void Recaler( DateV D ) {
        for (int i=0 ; i < cpt.length ; i++)
            this.cpt[i] = Math.max(this.cpt[i],D.cpt[i]) ;
        this.cpt[s]++ ; // Top implicite
    }
}
```

Le tableau suivant précise les actions exécutées lors sur les événements internes ainsi que lors des émissions et réceptions.

Type d'événement sur un site s	Action mettant en jeu l'horloge du site s
Événement interne sur s	$H_s.Top()$
Émission sur s de m	$dvm = H_s.Top()$; envoi de $\langle dvm, m \rangle$
Réception sur s de $\langle dvm, m \rangle$	$H_s.Recaler(dvm)$

La figure (2) illustre l'évolution des horloges vectorielles sur 3 sites. On remarquera que la composante "locale" du vecteur $(H_s[s])$ comptabilise exactement le nombre d'événements ayant lieu sur le site. Par ailleurs, toute date "vecteur" d_e associée à un événement e mémorise le nombre d'événements causalement liés à e . À titre d'exemple, l'événement c_3 est daté $(3, 3, 3)$ et l'on constate qu'il y a effectivement 3 événements sur les sites A (a_1, a_2, a_3) et B (b_1, b_2, b_3) et 2 événements sur C (c_1, c_2) qui précèdent c_3 .

Enfin, on remarquera que les événements de réceptions sont datés explicitement et comptabilisés.

Exercices

1. Dans le mécanisme d'horloge de Lamport, pourrait-on ne pas dater explicitement les événements d'émission ?

2. On essaie de définir un mécanisme de datation avec seulement un compteur local par site (on élimine la composante site des horloges de Lamport). Donner les règles de gestion de telles horloges pour assurer le respect de la relation : $\forall e, e' : e \prec e' \rightarrow d_e < d_{e'}$

3. Montrer sur un exemple, que le mécanisme précédent n'assure pas la réciproque :

$$\forall e, e' : d_e < d_{e'} \rightarrow e \prec e'$$

4. Montrer que les horloges de Mattern assurent :

$$\forall e, e' : e \prec e' \equiv d_e < d_{e'}$$

5. Pourrait-on ne pas dater explicitement les événements de réception avec des horloges de Mattern ? (Tout en conservant leur propriété précédente).

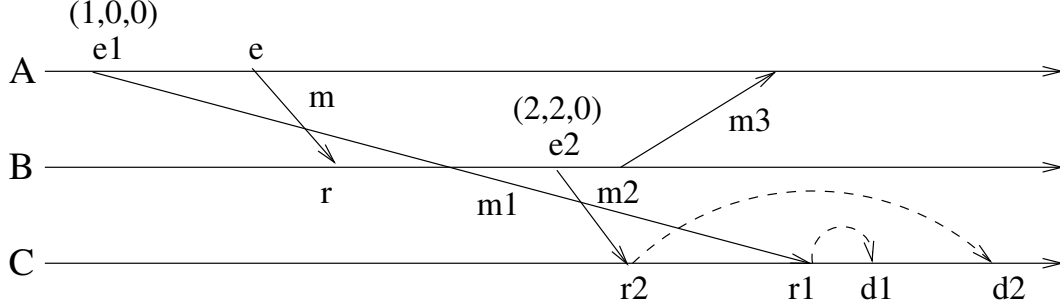


FIG. 3 – Protocole point à point d'ordre causal

2 Protocoles d'ordre causal

Le transfert des messages dans un réseau ne respecte pas forcément des règles d'ordonnancement strictes. Par exemple, dans un protocole point à point, l'ordre de réception des messages issus d'un **même** site par un site fixé n'est pas forcément identique à leur ordre d'envoi.

Plus généralement, il peut être intéressant d'ordonner la délivrance des messages reçus par un site de façon à respecter la causalité qui peut exister entre les événements d'émission de ces messages. Pour ce faire, on doit donc distinguer d'une part, l'événement de réception d'un message par un site et d'autre part, l'événement de délivrance de ce message au niveau applicatif. En effet, des messages reçus dans l'ordre inverse de leur causalité d'émission devront être réordonnés sur le site de réception pour être délivrés dans le bon ordre.

Cette contrainte peut être formalisée de la façon suivante :

$$\forall s : (\forall m, m' : r_s \wedge r'_s :: e \prec e' \Rightarrow d \prec d')$$

La figure (3) montre un exemple qui peut très bien se produire lorsqu'on communique par e-mail. Un usager A pose une question à deux usagers B et C en leur envoyant un message à chacun. L'utilisateur B reçoit en premier le message m question m . Il répond alors à A et à C (constatant que la question a été aussi envoyée à C grâce au champ destinataires du message). L'utilisateur C reçoit alors via le message m_2 une réponse dont il ne connaît pas la question. Celle-ci arrivera plus tard sous la forme du message m_1 . On remarquera que cette anomalie tient au fait que les deux messages m_1 et m_2 qui sont reçus par le site C dans l'ordre inverse de leur précédence causale d'émission puisque $e_1 \prec e_2$. Dans ce cas un protocole d'ordre causal délivrera les messages dans l'ordre inverse. Si l'on note d_1 et d_2 les événements de délivrance des deux messages, alors on aura $d_1 \prec d_2$.

2.1 Implantation du protocole dans le cas point à point

Si l'on utilise le mécanisme d'horloge de Mattern pour dater les événements, les messages m_1 et m_2 emportent avec eux la date de leur émission. Sur l'exemple de la figure (3), on constate que l'on a bien $d_{e_1} \prec d_{e_2}$ (puisque $(1, 0, 0) < (2, 2, 0)$). Mais, lors de la réception du message m_2 , la date d_{e_2} ne permet pas de conclure qu'un message m_1 existe et aurait dû être déjà reçu et délivré. Ce n'est que lorsque le message m_1 arrivera que l'on détectera l'anomalie de causalité dans la réception des deux messages. Par conséquent, il faut mettre en œuvre un protocole spécifique.

2.1.1 Approche par matrice de précédence (Raynal)

Pour décider si un message m peut être délivré à l'applicatif d'un site s , il faut savoir si tous les messages qui devaient être reçus par s et dont l'émission précède causalement ce message m sont arrivés et ont été délivrés.

Pour ce faire, un site s doit gérer :

- d’une part, un vecteur $Dernier[N]$ indiquant le numéro d’ordre du dernier message reçu par ce site et provenant de chaque site origine possible.
- d’autre part, une matrice carrée de précédence $MP[N, N]$ dont chaque élément $MP[i, j]$ indique le nombre d’émissions du site i vers le site j connu du site s .

Les actions suivantes seront exécutées sur occurrence des événements d’émission et de réception :

Type d’événement sur un site s	Action mettant en jeu l’horloge du site s
émission sur s de m vers s'	$MP_s[s, s'] ++$; envoi de $\langle MP_s, m \rangle$
Réception sur s' de $\langle MP_s, m \rangle$	Réordonner(MP_s, m)

Une description en Java de ce protocole comporte deux classes :

- d’une part, la classe *Précédence* qui décrit le type de matrice carrée $N \times N$ captant la causalité des messages;

```
import java.util.* ;

class Précédence {
    protected int[] [] cpt ; // matrice de précédence
    protected int orig; // identité du créateur
    protected int N ; // nombre de sites

    // Création de la matrice de précédence
    Précédence ( int loc, int nbSite ) {
        cpt = new int[nbSite][nbSite] ;
        for (int i=0 ; i < nbSite; i++)
            for (int j=0 ; j < nbSite ; j++) cpt[i][j] = 0 ;
        orig = loc ; N = nbSite ;
    }

    // Recaler la matrice sur une autre
    void Max(Précédence MX) {
        for (int i=0 ; i < N ; i++)
            for (int j=0 ; j < N ; j++)
                cpt[i][j]=Math.max(cpt[i][j],MX.cpt[i][j]);
    }
}
```

- d’autre part, la classe *Protocole* qui décrit l’algorithme de traitement des émissions (*TopEnvoi*) et des réceptions (*Réordonner*).

Le contrôle de l’ordonnancement des messages reçus pour assurer leur délivrance dans un ordre causalment correct s’appuie sur la précondition suivante pour un message $\langle MP, m \rangle$ reçu par s :

- d’une part, ce message issu du site $MP.orig$ doit bien être le prochain message à délivrer (caractère FIFO de la communication entre le site émetteur et le site récepteur). Autrement dit le prédicat suivant doit être vérifié :

$$with\ Protocole@s :: MP.cpt[MP.orig][s] = Dernier[MP.orig] + 1$$

- d’autre part, ce message ne doit pas précéder des messages dont l’émission le précède causalement, c’est-à-dire :

$$with\ Protocole@s :: \forall i \neq s :: MP.cpt[i, s] \leq Dernier[i]$$

C’est cette conjonction qui est testée par la fonction *délivrable*.

Toute réception de message se traduira par un appel à la méthode *Réordonner* qui retardera éventuellement la délivrance du message correspondant. Les messages applicatifs seront consommés dans l’ordre chronologique de terminaison des opérations *Réordonner*.

```

class Protocole extends Précédence {

    protected int s ;           // localisation
    protected int[] Dernier ;   // vecteur de comptage des reçus
    protected List AttDel      ; // liste d'attente de délivrance

    // Top sur envoi de message vers le site $dest$
    Précédence TopEnvoi(int dest) throws Exception {
        this.cpt[s,dest]++;
        return (Précédence)super.clone();
    }

    // test si le message associé à la matrice de précédence est délivrable
    boolean délivrable(Précédence MP) {
        for (int i=0 ; i < N ;i++)
            if ((i == s && MP.cpt[MP.orig][s] != Dernier[MP.orig] + 1)
                || (i != s && MP.cpt[i][s] > Dernier[i])) return false ;
        return true ;
    }

    // Ordonnancement de la délivrance des messages
    synchronized void Réordonner( Précédence MP, String Message ) throws Exception {
        if (!délivrable) { AttDel.add(MP) ; MP.wait() ; }
        // délivrable(MP) => mise-à-jour de this
        this.Dernier[MP.orig]++ ; this.Max(MP) ;
        // Réveil de messages en attente
        ListIterator curseur = AttDel.listIterator(0) ;
        while (curseur.hasNext()) {
            Précédence mp = (Précédence) curseur.next() ;
            if (délivrable(mp)) { mp.notify() ; curseur.remove() ; }
        }
    }

    // Constructeur
    Protocole(int où , int dim) {
        super(où,dim) ; s = où ; Dernier = new int[dim] ;
        for (int i=0 ; i < N ; i++) Dernier[i]=0 ;
        AttDel = Collections.synchronizedList(new LinkedList());
    }
}

```

Le transport d'une matrice carrée de dimension N égale au nombre de sites pose deux problèmes : d'une part, il faut connaître N , d'autre part, pour N grand, le coût en communication devient important. C'est pourquoi une autre approche a été utilisée.

2.1.2 Approche par gestion d'histoires

Une autre approche consiste à concaténer à tout message la suite des messages qui le précède causalement. On appelle cette technique "piggybacking". La figure (4) montre que la réception du message m_2 sur le site C s'accompagne en fait de la réception d'une copie du message m_1 qui était lui aussi destiné à C et placé dans l'histoire causale de m_2 . Dans un tel cas, c'est le message m_1 qui sera délivré, puis le message m_2 . La réception ultérieure du message original m_1 issu du site A se traduira par l'oubli pur et simple de ce message (dont une copie a déjà été délivrée à l'applicatif).

Un avantage de cette approche est qu'elle apporte une certaine redondance par les copies de messages

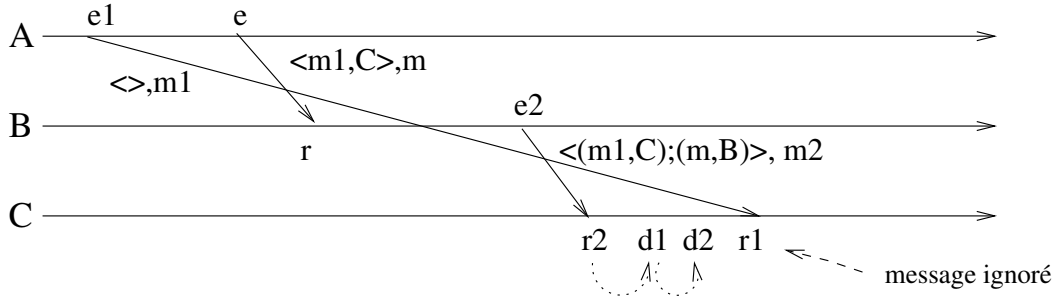


FIG. 4 – Approche par piggybacking

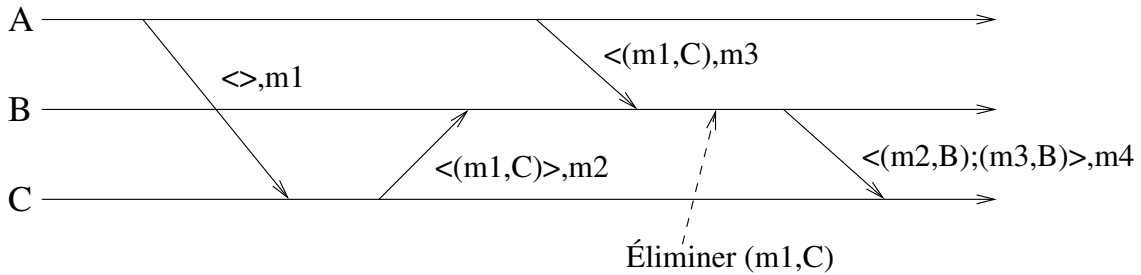


FIG. 5 – Optimisation des histoires

qui sont engendrées apportant ainsi un degré de tolérance vis-à-vis des pertes de messages. Par contre, la surcharge des messages est importante. L’histoire causale des nouveaux messages s’alourdit au fur et à mesure du calcul. Il faut pouvoir optimiser ce coût qui peut devenir prohibitif en réduisant la taille de l’histoire emportée par un nouveau message. La figure (5) montre un exemple d’élimination d’un message dans une histoire. Après la réception des messages m_2 et m_3 , le site B “sait” que le message m_1 a été envoyé ET délivré. Par conséquent, il peut être désormais ignoré par B .

2.2 Implantation du protocole dans la cas de la diffusion

Pour un protocole de diffusion se pose les mêmes problèmes de causalité. La figure (6) illustre un début de calcul réparti utilisant la diffusion. Cependant, l’implantation d’un protocole de diffusion à ordre causal devient plus simple lorsqu’on utilise l’approche par compteurs. En effet, la matrice de précédence se simplifie : un site envoie un message vers tous les autres sites systématiquement. Par conséquent, dans cette matrice, on a $\forall j : MP[i, j] = k$. Il suffit donc d’un vecteur $VP[N]$ tel que $VP[i]$ indique le nombre de messages envoyé par i vers tout site.

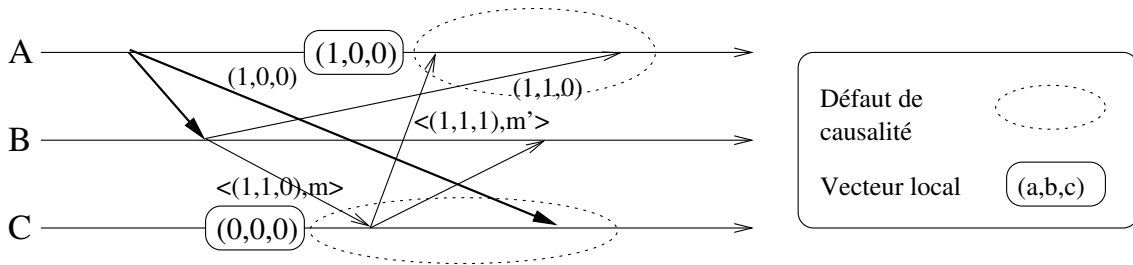


FIG. 6 – Diffusion et causalité

Le contrôle de l'ordonnancement des messages reçus s'appuie cette fois-ci sur la précondition suivante pour un message $\langle VP, m \rangle$ reçu par le site s :

- d'une part, ce message issu du site $VP.orig$ doit bien être le prochain message à délivrer :

$$\text{with Protocole@s} :: \text{cpt}[VP.orig] = \text{Dernier}[MP.orig] + 1$$

- d'autre part, ce message ne doit pas précéder des messages dont la diffusion le précède causalement, c'est-à-dire :

$$\text{with Protocole@s} :: \forall i \neq s :: VP.cpt[i] \leq \text{Dernier}[i]$$

La figure (6) montre que les message m et m' seront retardés puisque les vecteurs associés à ces messages ne satisfont pas la précondition attendue.

3 Tolérance aux fautes : modèle d'exécution ISIS, HORUS

Remarque préliminaire : Ce cours est inspiré de différents articles de Kenneth Birman, principalement les deux articles dans les Communications de l'ACM, respectivement sur Isis [Bir93] et Horus [RBM96], le livre réunissant divers articles sur Isis [BR94] et bien évidemment, le manuel d'Isis [BCJ+90].

La boîte à outils Isis est issue de travaux de l'équipe de Kenneth P. Birman à Cornell University, de 1983 à 1990. Ces mêmes travaux ont défini la notion de synchronisme virtuel, sur lequel repose Isis. Isis a ensuite été commercialisé par une compagnie ultérieurement rachetée par Stratus Computer, Inc. Après avoir poursuivi le développement de la boîte à outils Isis, Stratus a intégré Isis dans Orbix pour définir un ORB Corba tolérant aux fautes. Actuellement, Isis semble avoir disparu à l'intérieur du produit Radio pour l'administration de serveurs tolérants aux fautes sous Windows NT. Parallèlement, la recherche à Cornell s'est poursuivie au travers du nouveau projet Horus. Ce projet, initialement une refonte d'Isis, a notamment étudié l'utilisation modulaire de micro-protocoles pour construire des protocoles de communication évolués.

Isis est une boîte à outils fournissant principalement des primitives de communication pour construire des applications distribuées tolérantes aux fautes, basées sur des groupes de processus coopérants. à ce titre, Isis fournit des primitives pour gérer les groupes, détecter et traiter les fautes, synchroniser des processus et, bien sûr, échanger des messages.

3.1 Groupe de processus

Isis est basé sur la notion de groupe de processus avec un modèle d'exécution bien défini : le synchronisme virtuel. On rencontre principalement deux types de groupes :

- Les groupes anonymes : ils apparaissent quand une application publie des données selon des "thèmes", et que d'autres processus souhaitent s'abonner à ce thème. Pour qu'une telle application soit tolérante aux fautes, les groupes anonymes doivent fournir plusieurs propriétés :
 - Il doit être possible d'envoyer un message à un groupe via une adresse, sans que le programmeur ait besoin d'expanser lui-même les membres du groupe.
 - Si l'émetteur et les souscripteurs fonctionnent correctement, un message doit être délivré exactement une fois. Si l'émetteur s'arrête, un message doit être délivré à tous les souscripteurs ou à aucun d'entre eux. Le programmeur de l'application n'a pas à se préoccuper des messages perdus ou dupliqués.
 - Les messages doivent être délivrés aux abonnés selon un ordre bien défini. Par exemple, une application pourra nécessiter un ordonnancement causal.
 - Il doit être possible à un abonné d'obtenir un historique du groupe : l'ensemble des événements importants (entrée et sortie de membres du groupe) et des messages échangés, ainsi que l'ordre de ces événements.
- Les groupes explicites : un groupe est explicite quand ses membres coopèrent directement. Ils savent qu'ils sont membres du groupe et utilisent explicitement la liste des membres. Un changement dans la liste des membres est une information visible et utilisée dans un groupe explicite. Par exemple, un service tolérant aux fautes peut être réalisé avec un serveur primaire qui réalise le service et un ensemble ordonné de serveurs secondaires qui prendront successivement en charge le service si le primaire s'arrête. Si tous les événements ne sont pas vus dans le même ordre par tous les sites, il peut arriver qu'aucun primaire n'existe, ou, à l'inverse, que plusieurs sites pensent être le serveur primaire. On utilise aussi un groupe explicite lorsqu'on souhaite paralléliser un traitement sur n sites, en assurant un partage correct du travail.

Lorsque l'on souhaite utiliser des groupes de processus, trois problèmes doivent être résolus :

- support pour la communication de groupe : il faut pouvoir adresser un groupe, échanger des messages en respectant certaines propriétés d'ordonnancement (fifo, causal, total), et gérer atomiquement les défaillances ;
- connaissance de la relation d'appartenance (membership) : il faut pouvoir connaître à quel(s) groupe(s) appartient un site, de quels sites est composé un groupe donné, quelle a été l'évolution d'un groupe. . .

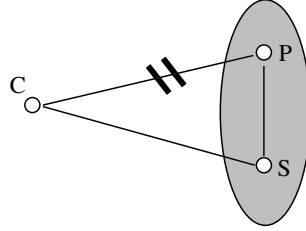


FIG. 7 – Etat incohérent lors d’une faute

- synchronisation : pour obtenir un comportement correct d’une application distribuée, il est nécessaire de synchroniser l’ordre dans lequel les actions des différents membres d’un groupe sont effectuées.

L’intégration de ces trois problèmes interdépendants forme la base du modèle d’exécution nommé synchronisme virtuel. Avant de présenter ce modèle, commençons par un rapide survol des solutions traditionnelles.

3.2 Les solutions traditionnelles

3.2.1 Transfert de messages

Les systèmes d’exploitation fournissent généralement les mécanismes suivants pour communiquer :

- datagrammes : le seul service assuré est la détection de corruption. Hors cela, les messages peuvent être perdus, dupliqués ou réordonnés.
- appel de procédure à distance : un RPC est relativement fiable mais en cas d’échec, l’émetteur ne peut généralement pas savoir si le réseau a perdu la requête ou perdu la réponse ou si le destinataire s’est arrêté avant ou après avoir traité la requête.
- canaux fiables : il s’agit du moyen de communication le plus courant. Un canal fiable assure la délivrance fiable des messages dans leur ordre d’émission. Cependant, des difficultés similaires à celles du RPC existent aussi. Considérons l’exemple de la figure 7 : le client C est connecté à un serveur primaire P et à son secours S. La connexion entre C et P est rompue. Le client C et le primaire P ne peuvent connaître l’état de l’autre site (bien souvent, ils sont même incapables de savoir si la rupture de la connexion provient d’une défaillance du réseau ou d’un arrêt de l’autre site). Pire, le secours S n’est pas au courant du problème, et, constatant que le primaire est toujours opérationnel, ignorera les requêtes de C.

3.2.2 Gestion des groupes et ordonnancement

La gestion des groupes se fait au moyen d’un service d’appartenance (membership service). Le rôle de ce service est d’assurer la liaison entre l’adresse d’un groupe et la liste de ses membres. Ce service doit évidemment être tolérant aux fautes. Lorsqu’un site diffuse un message aux membres d’un groupe, il est nécessaire que cette liste de membres soit valide, et ne corresponde pas un état intermédiaire du service d’appartenance (atomicité des mises à jour).

La notion de temps logique compatible avec la causalité, introduite par Lamport en 1978, est fondamentale à la bonne intégration entre évolution des groupes et délivrance des messages. L’idée est qu’un message diffusé à un groupe doit être délivré à tous les membres au même instant. En l’absence d’horloges synchronisées, il est impossible d’assurer cette délivrance au même instant physique. Par contre, il est possible d’assurer que, sur tous les sites, l’ordre de délivrance est “raisonnable”, c’est-à-dire compatible avec les liens de causalité. Un observateur est alors incapable de distinguer entre une délivrance au même instant et une délivrance “au bon moment”.

La figure 8 illustre plusieurs des problèmes qui peuvent survenir si le service d’appartenance et le système de diffusion n’assurent pas un bon ordonnancement. Nous considérons un groupe composé de trois sites S_1 , S_2 , S_3 . Les messages m_1 et m_2 sont émis sans lien causal par deux clients. Avec une diffusion causale, ils peuvent être délivrés dans des ordres différents aux membres du groupe (par exemple sur S_1 et S_2) ; une telle

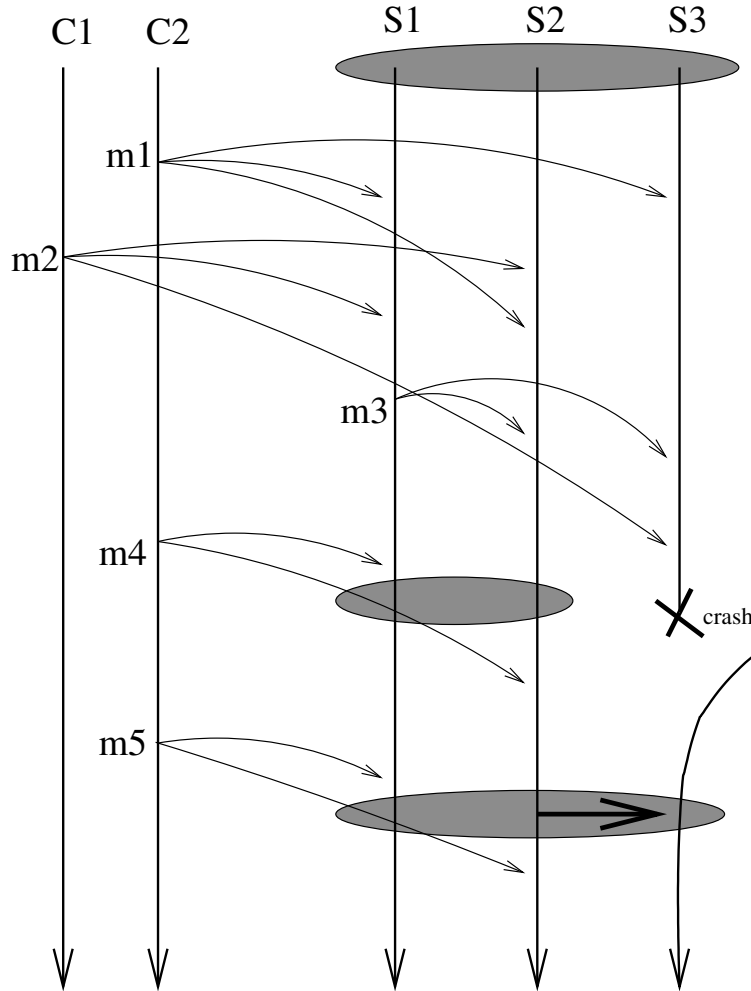


FIG. 8 – Mauvais ordonnancements de messages

situation est par contre interdite avec une diffusion atomique¹. Suite à m_2 , le site S_1 diffuse un message m_3 à l'intérieur du groupe. Le schéma présente une délivrance non causale sur S_3 : ce site reçoit le message m_3 avant le message responsable m_2 . Enfin, nous supposons que le site S_3 s'arrête suite à une défaillance. Une nouvelle liste d'appartenance est constituée avec seulement S_1 et S_2 . Le message m_4 est délivré à S_1 avant qu'il n'ait connaissance de l'arrêt de S_3 , mais il est délivré à S_2 après la nouvelle constitution. De manière symétrique, lorsque S_3 rejoint le groupe, le message m_5 est délivré et traité par S_1 avant la jonction, mais S_2 n'en a pas connaissance lorsqu'il transmet son état à S_3 .

3.2.3 Tolérance aux fautes

En se basant sur les mécanismes de communication précédemment évoqués, il est possible de construire un protocole de diffusion qui tolère les arrêts de site, en particulier de l'émetteur. Cependant de tels protocoles sont relativement complexes et coûteux (la méthode naïve conduit à $5n$ messages, où n est la taille du groupe). En outre, il est difficile d'intégrer de manière cohérente la délivrance d'événements comme l'apparition ou la disparition d'un membre.

¹Dans Isis, une diffusion est dite atomique si elle impose un ordre total compatible avec la causalité vis-à-vis des autres diffusions atomiques.

3.3 Le synchronisme virtuel

Pour résumer, l'utilisation des systèmes d'exploitation traditionnels soulève plusieurs problèmes : le support pour les communications fiables (notamment les diffusions) est faible et conduit à des incohérences lors des ruptures de canaux ; l'expansion de l'adresse d'un groupe en sa liste de membres doit être faite explicitement par le programmeur ; l'ordonnancement des messages concurrents ou causalement liés doit être fait au sein même de l'application ; la gestion cohérente des fautes est complexe et coûteuse. C'est dans ce cadre qu'Isis propose un ensemble de primitives de diffusion et de gestion des groupes associées à un modèle d'exécution simplifiant l'écriture d'applications distribuées.

3.3.1 Synchronisme fort

Idéalement, on voudrait pouvoir développer des applications dans le modèle de synchronisme fort (close synchrony), dans lequel les propriétés suivantes sont garanties :

- les communications sont fiables ;
- une diffusion est considérée comme un événement atomique, i.e. de temps logique nul ;
- l'expansion d'une adresse de groupe en sa liste de membres est un événement atomique qui se produit à l'instant logique de la diffusion ;
- les messages concurrents sont délivrés à tous les destinataires dans le même ordre ;
- les messages causalement liés sont délivrés selon un ordre compatible avec cette causalité ;
- le transfert de l'état d'un site lors de la jonction d'un nouveau site est effectué atomiquement vis-à-vis de tous les autres événements (notamment les diffusions) ;
- les fautes sont atomiques vis-à-vis des diffusions, et sont ordonnées identiquement sur tous les sites.

Malheureusement, un tel synchronisme fort est, d'une part, impossible à réaliser en présence de fautes, et d'autre part, extrêmement coûteux à obtenir. Il impose généralement une exécution pas à pas de tous les sites, et désavantage donc les interactions asynchrones, dans lesquelles un site continue son exécution immédiatement après avoir initié une communication sans attendre la délivrance du message. Le modèle de synchronisme virtuel est un affaiblissement du modèle de synchronisme fort, où l'atomicité des communications est relâchée de manière à privilégier les communications asynchrones mais où le respect d'un ordre bien défini de délivrance est conservé.

3.3.2 Ordre total et ordre causal

Le synchronisme fort impose un ordre total entre tous les événements. Cet ordre total peut être obtenu au moyen de la *diffusion atomique*, nommée ABCAST dans Isis. Cependant, dans nombre de cas, l'ordre causal suffit. Selon l'ordre causal, des événements causalement liés doivent être délivrés selon ce lien causal. Par exemple, sur la figure 8, le message m_3 est causalement ultérieur à m_2 , il doit donc être délivré après. Par contre, il n'y a pas de lien causal entre m_1 et m_2 qui peuvent donc être délivrés dans des ordres différents selon les sites. La diffusion causale, ou CBCAST, assure un tel ordre causal. L'intérêt principal de la diffusion causale est son faible coût, tant d'un point de vue retard de délivrance que d'un point de vue surcoût de communication : un message d'une diffusion atomique ne peut être délivré que quand le site est sûr que tous les ABCAST précédents (vis-à-vis de l'ordre total) sont terminés ; pour une diffusion causale, il suffit de s'assurer que les messages causalement précédents ont été délivrés. Cela signifie en outre que, si l'émetteur est membre du groupe auquel il diffuse, il peut se délivrer immédiatement le message. L'implantation performante du CBCAST d'isis se base sur des vecteurs d'horloges, ce qui entraîne un faible surcoût dans les messages.

Les changements dans les groupes (dus à un retrait volontaire ou à une défaillance pour les réductions) doivent par contre être ordonnés totalement vis-à-vis des délivrances, pour éviter des incohérences similaires à celles illustrées par les messages m_4 et m_5 de la figure 8.

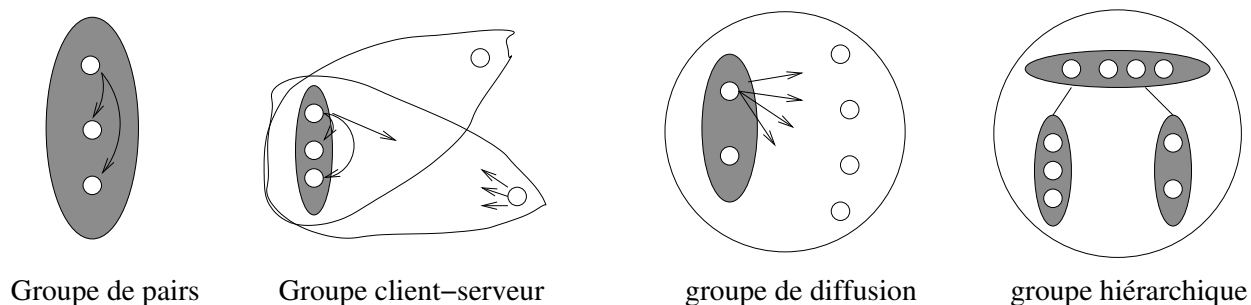


FIG. 9 – Styles de groupes

3.4 La boîte à outils Isis

3.4.1 Hypothèses sur l'environnement

La boîte à outil Isis fait les hypothèses suivantes sur son environnement : le réseau est constitué de réseaux locaux (LAN) interconnectés par des réseaux à grande distance (WAN), beaucoup plus lents. Les horloges des différents sites ne sont pas synchronisées. Sur un LAN, les messages peuvent être perdus, dupliqués ou ré-ordonnés ; si une partition se produit, une partition majoritaire est déterminée et sera la seule à pouvoir continuer à travailler. Les seules fautes de sites sont des arrêts, sans action ou émission illégale (crash failure). Du fait de la règle de la majorité pour les partitions, il est déconseillé d'utiliser des groupes répartis sur plusieurs réseaux locaux, mais plutôt d'utiliser un paquetage spécial de communication pour les réseaux à grande distance.

3.4.2 Styles de groupes

Isis est optimisé pour traiter de manière efficace quatre styles d'utilisation des groupes (figure 9), sans qu'il soit pour autant nécessaire de se conformer à l'une de ces structures :

- groupe de pairs (peer group) : tous les processus coopèrent étroitement, par exemple pour répliquer des données. L'application utilise explicitement l'appartenance et un ordre parmi les membres.
- groupe client-serveur : un client communique avec un groupe qui remplit un service. Le client n'est pas membre du groupe, mais, pour des raisons de performance, il peut s'enregistrer en tant que *client*.
- groupe de diffusion : c'est un groupe client-serveur où les clients sont enregistrés et où les serveurs diffusent à tous les clients sans que ceux-ci ne déposent de requêtes.
- groupes hiérarchiques : pour éviter qu'un groupe ne soit composé de trop de membres, il peut être utile de construire une structure hiérarchique arborescente pour structurer ces membres. Une application contacte initialement le groupe racine puis ne communique plus qu'avec un des sous-groupes. La plupart des communications se limitent à l'intérieur d'un sous-groupe, et ne concernent que rarement l'ensemble des sous-groupes.

3.5 Programmer avec Isis

La programmation avec Isis est une programmation de type événementiel, où le développeur déclare des points d'entrée pour gérer la réception d'un message. Quand un message est reçu, Isis crée une nouvelle tâche pour exécuter le code associé au point d'entrée.

3.5.1 Structure de l'application

Une application suit les étapes suivantes :

1. initialise les bibliothèques Isis et se connecte à Isis ;
2. déclare les tâches et les points d'entrées ;

3. déclare les gestionnaires de signaux et d'entrées-sorties ;
4. initialise l'état de l'application ;
5. se joint à des groupes et/ou devient client ;
6. indique à Isis qu'elle est prête à recevoir les messages.

3.5.2 Initialisation et connexion à Isis

Trois méthodes d'initialisation sont disponibles :

- connexion à l'Isis local :

```
int isis_init (int client_port) ;
```

Se connecte à Isis via la port tcp `client_port`. Il s'agit du deuxième port listé dans le fichier *sites* (généralement 1602).

Une version longue permet de spécifier de nombreux drapeaux, notamment `ISIS_AUTORESTART` (la bibliothèque tente de démarrer Isis s'il ne tourne pas actuellement sur la machine) et `ISIS_TRACE` (Isis affichera tous les messages reçus ou envoyés).

- connexion à un Isis distant :

```
int isis_remote (char *mother_host, int flags) ;
```

Tente de se connecter à l'Isis tournant sur `mother_host`

- connexion libre : les deux précédentes formes d'initialisation sont remplacées par une unique routine :

```
int isis_remote_init (char *hostlist, int lport, int rport, int flags) ;
```

`hostlist` est une chaîne de caractères contenant la liste des machines, séparées par des virgules, où tenter de se connecter. Utiliser `(char *)0` pour ne tenter que la machine locale. `lport` est le port local de connexion à une machine locale (port tcp, 1602). `rport` est le port à utiliser pour trouver Isis sur une machine distante (port bcst, 1603).

3.5.3 Déclaration des tâches et point d'entrée

Déclaration des tâches Cette partie n'est pas obligatoire, mais est très utile pour que les enregistrements ou les traces contiennent des noms symboliques, plutôt que des numéros arbitraires alloués par Isis.

```
int isis_task (void (*routine) (void *arg), char *taskname) ;
```

Une tâche est une procédure prenant un paramètre de type pointeur.

Déclaration des points d'entrées Cette partie définit les points d'entrées où seront reçus les messages. Elle est donc obligatoire, si jamais l'application espère recevoir des messages.

```
int isis_entry (int entry, void (*handler) (message *mp), char *name) ;
```

`entry` est le numéro de l'entrée. Il doit être strictement positif, et identifie l'entrée (voir 3.5.6).

`handler` est la routine à appeler quand un message est reçu. La routine sera appelée avec le message reçu en paramètre. Pour exécuter cette routine, Isis créera une nouvelle tâche qui sera détruite à la fin du traitement.

`name` permet à Isis d'être plus bavard dans ses explications.

3.5.4 Initialisation de l'application et entrée dans Isis

Ceci se fait par :

```
void isis_mainloop (void (*task) (void *arg), void *arg) ;
```

Isis crée alors une tâche pour exécuter `task` (avec l'argument passé en paramètre). Cette tâche a pour but d'initialiser complètement l'application, de se joindre à des groupes ou de devenir client d'autres groupes.

Quand l'application a fini de s'initialiser (ie s'est connectée aux groupes souhaités, ...), il est nécessaire d'indiquer à Isis que l'application est maintenant prête à recevoir des messages. Ceci est fait au moyen de :

```
void isis_start_done (void) ;
```

Tant que l'application n'a pas appelée cette routine, elle ne peut pas émettre ou recevoir de messages, et aucune tâche ne peut être créée implicitement. Si la tâche principale créée par `isis_mainloop` se termine sans avoir appelée `isis_start_done`, Isis le fait automatiquement.

3.5.5 Groupes et clients

Jonction à des groupes La routine de jonction à un groupe est :

```
address *pg_join (char *gname, [int key, [arg,...]..., 0);
```

`gname` est le nom du groupe auquel l'application essaie de se connecter. Un nom de groupe a une syntaxe de nom de fichier Unix, ie `"/sys/databases/CCP"`. Par défaut, le "working group" est `/`. Contrairement à Unix, les groupes intermédiaires n'ont pas besoin d'exister. En cas de succès, `pg_join` renvoie une adresse qui sera utilisée pour envoyer des messages.

Pour tester l'échec, il convient d'utiliser la routine :

```
int addr_isnull (address *adr);
```

qui renvoie vrai si l'adresse est nulle (invalide).

Enfin, `pg_join` accepte de nombreux paramètres optionnels. Les principaux sont :

- `PG_DONTCREATE` (pas de paramètre) : ne crée pas le groupe s'il est inexistant (par défaut le groupe est créé).
- `PG_LOGGED` (4 paramètres non détaillés) : pour enregistrer tout ce qui se passe dans le groupe.
- `PG_INIT`, `void (*routine) (address *)` : la routine est appelée quand un groupe est créé et qu'aucun enregistrement ne permet de reconstruire un état.
- `PG_MONITOR`, `void (*routine) (groupview *, void *arg)`, `void *arg` : la routine est appelée à chaque changement dans la composition du groupe (jonction ou retrait de membres). Noter que la routine sera aussi appelée pour *cette* jonction.
- `PG_XFER`, `int domain`, `void (*send) (int position, address *group, address *whojoined)`, `void (*receive) (int position, message *msg)` : permet l'initialisation de l'état d'un nouveau membre par copie de l'état d'un autre membre. Dans notre cas, `domain` est généralement spécifié à 0. Voir 3.5.5 pour les détails.

Client Un client obtient l'adresse d'un groupe avec :

```
address *pg_lookup (char *groupname);
```

Comme précédemment, on utilise `addr_isnull` pour vérifier si le groupe existe. Un client peut utiliser l'adresse qu'il a obtenue par `pg_lookup` pour envoyer une requête et obtenir la réponse.

Par ailleurs, un client régulier d'un groupe peut avoir intérêt à se déclarer comme tel. Outre le fait qu'il existe des moyens pour obtenir la liste des clients déclarés d'un groupe, cela permettra aussi au client de détecter des événements survenant dans le groupe. Un processus se déclare client avec :

```
int pg_client (address *groupaddr, char *credentials);
```

Le paramètre `credentials` sert à l'authentification, et peut être NULL si le groupe ne fait pas d'authentification.

Surveiller l'état d'un groupe Il est possible de surveiller l'état d'un groupe, de façon encore plus riche que le `PG_MONITOR` de `pg_join`. En outre, il n'est pas nécessaire d'être membre du groupe. Nous ne présentons ici que la routine qui détecte la disparition d'un groupe. Il s'agit de :

```
int pg_detect_failure (address *group, int (*routine) (address *, int what, void *arg), void *arg);
```

Quand le groupe spécifié disparaît, une tâche est créée pour exécuter le paramètre procédure de `pg_detect_failure` avec `routine(group, WFAIL, arg)`. Une telle surveillance est très coûteuse sauf dans deux cas particuliers : soit le processus est membre du groupe qu'il surveille, soit il est client *et* il s'est déclaré comme tel avec `pg_client`.

Il existe deux autres routines pour surveiller un groupe ou un processus, qui sont `pg_monitor` et `pg_watch`. Ces routines ne marchent que pour un membre du groupe ou un client déclaré. Pour un membre du groupe, `pg_monitor` est identique à l'argument `PG_MONITOR` de `pg_join`.

Transfert de l'état Lorsqu'un nouveau membre est introduit dans un groupe, il peut être nécessaire de lui donner l'état courant du groupe. Pour cela, `PG_XFER` est fourni. Isis sélectionne un membre actif du groupe, et appelle la routine `send` de ce membre. Si tout va bien, ce membre enverra (au moyen de `xfer_out`, voir 3.5.6) l'état au nouveau membre. La routine `receive` du nouveau membre est appelée pour chacun des messages envoyés, et il récupérera l'état au moyen de `msg_get` (voir 3.5.7).

Comme le transfert d'un état peut nécessiter plusieurs messages, le paramètre `position` de `send` et `receive` permet de déterminer où en est le transfert. Noter enfin que Isis s'occupe de tout vis-à-vis des éventuelles défaillances du membre retenu pour transférer l'état (un autre membre sera choisi et Isis lui fournira, au moyen du paramètre `position`, l'endroit où le précédent s'était arrêté).

La routine `void xfer_refuse` peut être utilisée par un membre du groupe pour signifier à Isis que ce membre n'est pas apte à transférer l'état (et Isis en trouvera alors un autre).

Autres routines sur les groupes Quelques routines utiles :

`int pg_leave (address *)` ; pour quitter un groupe (que le processus soit membre ou client déclaré).

`int pg_delete (address *)` ; pour détruire un groupe.

`int pg_rank (address *group, address *paddr)` ; renvoie le rang du processus `paddr` dans le groupe spécifié. Tout processus possède un rang unique dans le groupe (commençant à 0), qui est fonction du moment où `pg_join` a été appelé. `pg_rank` retourne `-1` si `paddr` n'est pas membre du groupe. Ce rang est souvent utilisé lorsqu'il faut élire un coordinateur ou un maître.

`address my_address` ; est l'adresse du processus courant. Elle peut être utilisée avec `pg_rank` pour obtenir son propre rang dans un groupe.

3.5.6 Diffusions et messages

Nous ne présentons ici que les formes les plus simples de l'envoi de messages. Il en existe de nombreuses variantes pour les situations les plus particulières. Toutes les routines d'émission et de réception de messages utilisent des formats à la `printf/scanf`, plus de nombreuses extensions pour manipuler les tableaux ou les entités d'Isis (`address, message, ...`).

émission de message

Diffusion sans réponse : Pour envoyer un message sans réponse, utiliser :

```
int bcast (address *addr, int entry, char *outformat, [arg,...], 0);
```

où `addr` est l'adresse du groupe ou du processus concerné, et `entry` est l'entrée sur laquelle délivrer le message. `bcast` peut être `fbcast`, `cbcast`, `abcast` ou `gbcast`, selon que l'on souhaite une diffusion FIFO, causale, atomique ou de groupe (la diffusion de groupe est une diffusion imposant un ordre total vis-à-vis de *toutes* les diffusions. En ce sens, elle est similaire à un événement de changement dans la composition d'un groupe).

Dans le cas où aucune réponse n'est attendue, il s'agit d'un envoi asynchrone (l'application continue immédiatement après le `bcast`).

Diffusion avec réponse : Pour envoyer un message avec réponses attendues, utiliser :

```
int bcast (address *addr, int entry, char *outformat, [arg,...], int nwant, char *replyformat [, replyarg] ...);
```

`nwant` indique le nombre de réponses attendues. Ce peut être un entier (notamment 1) ou les constantes `MAJORITY` ou `ALL`. Si `nwant` n'est pas nul, il s'agit d'un appel bloquant, jusqu'à réception des `nwant` messages. Si l'application est membre du groupe vers lequel elle diffuse, elle recevra aussi ce message.

Transfert de l'état : Le transfert de l'état se fait au moyen de :

```
int xfer_out (int position, char *format [, arg]...);
```

3.5.7 Réception d'un message

Un message récupéré comme paramètre d'un point d'entrée (ou de la routine de réception de l'état) est analysé au moyen de :

```
int msg_get (message *msg, char *format [, arg]...);
```

Réponse à un message

Pour répondre à un message, utiliser :

```
int reply (message *m, char *outformat [, arg]...);
```

`m` est le message auquel le processus répond.

Pour éviter de répondre (alors qu'une réponse était attendue), on utilise :

```
int nullreply (message *m);
```

 Ceci est différent de `reply (m, "")` ; qui indique une réponse vide.

Enfin, il est possible de répondre avec

```
int abortreply (message *m);
```

Dans ce cas, la diffusion du client se termine immédiatement en retournant `-1`, et `isis_errno` contient `IE.ABORT`.

Vérification de la cohérence questions/réponses

Quand un processus diffuse un message avec ALL réponses, comment sait-il qu'il a bien eu autant de réponses que de messages envoyés ? Deux variables peuvent alors servir :

```
int isis_nsent ;
int isis_nreplies ;
```

La variable `isis_nsent` contient le nombre de messages envoyés par le dernier `bcst`. La variable `isis_nreplies` contient le nombre de réponses récupérées. Les non-réponses générées par `nullreply` ne sont pas comptabilisées dans `isis_nreplies`.

Pour être précis, une diffusion se termine quand l'une des trois conditions suivantes devient vraie :

- l'initiateur de la diffusion a reçu le nombre de messages désiré ;
- chacun des processus qui a reçu le message a renvoyé une réponse (par `reply`), ou a renvoyé une non-réponse (par `nullreply`), ou s'est arrêté ;
- un des processus a renvoyé un abort (par `abortreply`).

3.5.8 Les tâches

Les tâches dans Isis sont non-préemptives sauf si Isis a été construit avec une librairie de tâches préemptives (Sun lwp ou Mach), et que l'application le demande explicitement. Chaque tâche possède sa propre pile.

Des tâches sont créées implicitement pour appeler les points d'entrée de messages ou les handlers de signaux et d'entrées-sorties.

Il est par ailleurs possible de créer explicitement une tâche avec :

```
int t_fork (void (*routine)(void *arg), void *arg);
int t_fork_msg (void (*routine) (message *arg), message *arg);
```

La routine est alors appelée avec le deuxième paramètre de `t_fork`. Quand la routine se termine, la tâche est terminée.

Si une tâche bloque, une autre tâche active peut être exécutée, et la tâche bloquée sera réactivée quand la condition de blocage disparaîtra. Si aucune tâche n'est active, Isis attend qu'une tâche devienne active. Une tâche bloque sur certaines opérations telles l'envoi de messages avec réponses, ou `t_wait`.

3.5.9 Synchronisation par jeton

à partir de la diffusion atomique, il est aisé de construire un jeton d'exclusion mutuelle dans un groupe. La gestion de l'arrêt du processus possédant le jeton est par contre un peu plus délicate. Isis fournit directement des routines de manipulations de jeton.

Obtention du jeton `int t_request (address *gaddr, char *tokenname, int pass_on_fail);`

Un jeton est identifié par son nom (une chaîne de caractères). Lors de `t_request`, si le jeton n'existe pas, il est créé. Le paramètre `pass_on_fail` est `TRUE` si le jeton doit être transmis quand le processus le possédant s'arrête sans l'avoir transmis. S'il vaut `FALSE`, le jeton n'est pas transmis automatiquement en cas de panne.

Passage du jeton `int t_pass (address *gaddr, char *name);`

Normalement, seul le processus possédant le jeton peut le passer. Cependant, si `t_request` avait été appelé avec `pass_on_fail` à `FALSE`, n'importe quel processus peut utiliser `t_pass` en cas de défaillance du processus possédant. Il obtient alors le jeton.

Possesseur du jeton `address *t_holder (address *gaddr, char *name);`

permet d'obtenir l'adresse du possesseur du jeton.

3.6 Horus

Horus présente un modèle de communication de groupes extrêmement flexible. Pour cela, les protocoles évolués sont obtenus en empilant des micro-protocoles rendant des services bien spécifiques. Basé sur l'expérience d'Isis, Horus permet d'obtenir une implantation très performante du modèle de synchronisme virtuel, mais il n'interdit pas d'autres modèles plus adaptés à une application donnée, par exemple dans le domaine du multimédia ou du temps réel. Horus fournit une architecture où le protocole supportant un groupe peut être modifié dynamiquement à l'exécution, en fonction des besoins de l'application et des évolutions de l'environnement.

Voici quelques-uns des modules fournis par Horus :

COM : émission de message vers une liste de destinataires et réception d'un message, en utilisant des protocoles de plus bas niveau tel UDP ou ATM;

NAK : assure l'ordre FIFO et la fiabilité, au moyen d'accusés négatifs de réception;

FRAG : fragmentation/réassemblage de messages;

MBRSHIP : assure, au moyen d'un algorithme de consensus, la correspondance entre nom de groupe et liste des membres accessibles;

TOTAL : assure la délivrance selon un ordre total;

CRYPT : chiffrement/déchiffrement;

MERGE : transfert d'état lors de l'intégration d'un nouveau membre ou lors de la résolution d'une partition.

Par exemple, le protocole `TOTAL :MBRSHIP :FRAG :NAK :COM :ATM` correspond à une implantation de la diffusion atomique sur ATM. `MERGE :MBRSHIP :CRYPT :FRAG :NAK :COM :UDP` correspond à la diffusion causale avec déchiffrement des messages et transfert de l'état.

Pour pouvoir empiler flexiblement les modules, les interfaces (montantes et descendantes) sont standardisées. Par exemple, il existe un appel descendant pour émettre un message, et un appel montant pour recevoir un message. Ces interfaces sont définies par le Horus Common Protocol Interface. Ces interfaces se décomposent en deux catégories : celles qui traitent l'échange de messages, et celles qui concernent la gestion des groupes.

Ensemble est une nouvelle étape dans le projet Horus. Ensemble est une implantation des protocoles d'Horus en Caml qui s'attache spécifiquement à la manipulation des micro-protocoles pour reconfigurer

dynamiquement des macro-protocoles. La structure en pile permet de simplifier le problème de la reconfiguration : la décision est à la charge du niveau supérieur (souvent l'application) et la réalisation est à la charge de la couche inférieure. Ces travaux ont aussi permis de montrer qu'il est possible de compiler une pile de protocoles pour le cas général de manière à obtenir une performance accrue en évitant le surcoût dû à un nombre trop élevé de couches. Par exemple, sur un réseau local, les messages sont rarement perdus ou délivrés dans le désordre et les changements dans la composition des groupes sont relativement peu fréquents. On souhaite donc pouvoir court-circuiter les couches responsables de ces traitements.

3.7 Bibliographie

Compte tenu de la durée du projet et de son ampleur, les publications sur Isis et Horus sont très nombreuses. Pour débiter, l'article [Bir93] est le premier article à lire, qui, plus qu'Isis lui-même, présente les concepts de groupe et de synchronisme virtuel tel qu'ils ont été utilisés dans Isis. Pour continuer, le livre [BR94] réunit de nombreux papiers sur Isis, sur les concepts (reprenant l'article précédent et détaillant le synchronisme virtuel ainsi que les notions de causalité), l'implantation des différents protocoles et de la gestion des groupes dans Isis, et différentes applications développées au-dessus d'Isis. Le problème des réseaux à grande distance (WAN) est abordé dans [MB90], ce problème difficile étant malheureusement trop souvent ignoré. Sur la tolérance aux fautes dans Isis, [BG93], [RSB93] et [BL94] présentent clairement le problème et les choix effectués.

La littérature sur Horus est heureusement encore assez limitée. On trouve un premier papier dans [BR94], mais l'article [RBM96] est la meilleure introduction à Horus, à compléter ensuite par [RBF⁺95]. Ensemble est présenté dans [RBH⁺97]. Enfin, de manière plus générale, je ne peux que conseiller la lecture enrichissante du livre [Bir96].

Références

- [BCJ⁺90] Kenneth P. Birman, Robert Cooper, T. Joseph, Keith Marzullo, , Messac Makpangou, K. Kane, Frank Schmuck, and Mark Wood. *The ISIS System Manual, Version 2.1*. Cornell University, 2.1 edition, September 1990.
- [BG93] Kenneth P. Birman and Bradford Glade. Consistent failure reporting in reliable communications systems. Technical Report 93-1349, Cornell University, May 1993.
- [Bir93] Kenneth P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12) :37–53, December 1993.
- [Bir96] Kenneth P. Birman. *Building secure and reliable network applications*. Manning, 1996.
- [BL94] Michel Banâtre and Peter A. Lee, editors. *Hardware and Software Architectures for Fault Tolerance*, volume 774 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
- [BR94] Kenneth P. Birman and Robbert van Renesse, editors. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1994.
- [MB90] Mesaac Makpangou and Kenneth P. Birman. Designing application software in wide area network settings. Technical Report 90-1165, Cornell University, October 1990.
- [RBF⁺95] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in Horus. In *14th Symposium on Principles of Distributed Computing*, pages 80–89. ACM, August 1995.
- [RBH⁺97] Robbert van Renesse, Kenneth P. Birman, Mark Hayden, Alexey Vaysburd, and David Karr. Building adaptative systems using Ensemble. Technical Report 97-1638, Cornell University, 1997.
- [RBM96] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffei. Horus : A flexible group communications system. *Communications of the ACM*, 39(4) :76–83, April 1996.
- [RSB93] Aleta Ricciardi, André Schiper, and Kenneth P. Birman. Understanding partitions and the “no partition” assumption. In *4th Workshop on Future Trends Of Distributed Computing Systems*, pages 354–360, Lisboa, Portugal, September 1993. IEEE.