Etude du collecticiel hétérogène

Christophe Tronche

11 septembre 1992

Table des matières

1 Introduction **3**

2 Le collecticiel **3**
2.1 Variabilité du collecticiel . . . 3
2.2 Exemples . . . 5
2.2.1 Jeux . . . 7
2.2.2 Edition partagée . . . 7
2.2.3 Rencontres informatiques . . . 9
2.3 Analyse du collecticiel . . . 10
2.3.1 Différence entre le collecticiel et d'autres applications multi-utilisateurs. . . . 10
2.3.2 Taxonomie du collecticiel . . . 10
2.3.3 Le problème des performances . . . 11
2.3.4 Architecture répliquée contre architecture centralisée . . . 12
2.3.5 Problèmes spécifiques issus d'une interface répartie . . . 13
2.3.6 Facteurs humains . . . 14
2.4 Les problèmes techniques : état de l'art . . . 15
2.4.1 Les conflits . . . 15
2.4.2 Couplage . . . 17
2.4.3 Sécurité et confidentialité . . . 17

3 Hétérogénéité **18**
3.1 Pourquoi écrire du collecticiel hétérogène . . . 18
3.2 Identifier l'hétérogénéité . . . 18
3.3 Problèmes spécifiques du collecticiel hétérogène . . . 19
3.3.1 Inadaptation des environnements actuels . . . 19
3.3.2 Architecture répliquée . . . 19
3.3.3 Nommage . . . 19
3.3.4 Formats . . . 20

4 Ecrire du collecticiel hétérogène **20**
4.1 Retour sur IP et AppleTalk . . . 21
4.2 Le ShareBoard et son implémentation . . . 22
4.2.1 Le ShareBoard Macintosh . . . 22
4.2.2 Le ShareBoard Unix. . . . 22
4.2.3 Un ShareBoard pour stocker des documents sous Unix . . . 26
4.3 Le moteur réactif . . . 26
4.3.1 Mécanismes de communication . . . 26
4.3.2 Réaction à des méthodes venues de l'extérieur . . . 26
4.3.3 Acteurs distants . . . 28
4.4 Un exemple simple . . . 29
4.4.1 Le serveur . . . 29

4.4.2 Le client . . . 30
4.5 Edition de dessins structurés . . . 30
4.5.1 Un editeur de dessin structuré sous Unix . . . 32
4.5.2 Passer à l'hétérogène . . . 32
4.5.3 Transmission . . . 33
4.6 Aller plus loin : des entités autonomes . . . 34
4.6.1 bugGame . . . 34
4.6.2 Simulateur de réseau de Pétri . . . 34

5 Comment écrire facilement une application collecticielle hétérogène 35
5.1 Une petite expérience de programme réparti en environnement hétérogène . . . 35
5.2 Quelques solutions possibles . . . 37
5.2.1 Améliorer le moteur réactif . . . 37
5.2.2 Un langage avec une architecture dataflow . . . 37
5.2.3 Ada9x . . . 40

6 conclusion 41

1 Introduction

L'objectif de ce stage est l'étude du collecticiel hétérogène. Le *collecticiel* désigne les systèmes informatiques qui permettent à plusieurs utilisateurs de travailler à une tâche commune dans un environnement partagé. Le terme "*hétérogène*" indique que les applications s'exécuteront simultanément sur des matériels différents (par exemple des Macintosh et des stations Unix).

Nous nous intéressons notamment, du point de vue de l'interaction homme-machine, aux problèmes de performances : que se passe-t'il si la session comporte un grand nombre d'utilisateurs, si le réseau est très étendu, si les réseaux ou les machines sont lentes ? Nous aborderons ces questions sous l'angle expérimental : en réalisant des applications mettant en œuvre les concepts correspondants.

Le rapport se compose de cinq parties. La première partie présente le collecticiel et fait un tour des travaux déjà effectués dans ce domaine. La deuxième partie présente l'hétérogénéité et les problèmes spécifiques qu'elle implique. La troisième partie décrit le travail d'implémentation réalisé dans le cadre de ce stage, les problèmes rencontrés et les solutions apportées. Ecrire des applications collecticielles s'est révélé être difficile. La dernière section indique quelques voies futures qui permettront, peut-être de faciliter l'implémentation de telles applications.

2 Le collecticiel

Un programme muni d'une interface utilisateur peut être symbolisé par la figure 1a. Il se compose d'un *noyau fonctionnel* qui délivre les services de l'application (base de donnée, communication, etc.). L'utilisateur communique avec l'application au travers de son *interface*. L'interface traite le flot de données allant du noyau fonctionnel vers l'utilisateur et crée pour lui une représentation des données abstraite, supposément plus facile à comprendre que les données brutes, de l'état de l'application (figure 2). Par exemple, une valeur numérique venant de l'application peut être traduite par un compteur ou un thermomètre. L'utilisateur communique des informations à l'ordinateur sous une forme qui est humainement compréhensible : par exemple sur le Macintosh, on peut cliquer sur l'icône représentant un fichier et la déplacer vers la poubelle. L'interface traduit ceci pour le noyau fonctionnel (qui est dans ce cas le système de gestion de fichier de l'ordinateur), sous la forme d'une commande qu'on pourrait exprimer par "effacer le fichier ayant tel nom". L'interface doit dialoguer avec le noyau fonctionnel pour obtenir les informations nécessaires pour faire ce type de traduction (dans cet exemple, comment associer un nom de fichier à un icône).

Dans le cadre d'une application *collecticielle* (ou "*groupware*"), le noyau fonctionnel (qui peut être une application répartie ou non), communique avec *plusieurs* utilisateurs au travers de *plusieurs* interfaces homme-machine, que ces interfaces soient plusieurs exemplaires d'une même interface ou des interfaces distinctes. Dans ce dernier cas, on parle de *collecticiel hétérogène*. Un collecticiel est cependant plus qu'un ensemble d'interfaces qui travaillent de concert : un collecticiel doit fournir aux utilisateurs les moyens de coordonner leurs activités.

Cette première définition peut être qualifiée de structurelle : elle définit une application collecticielle comme une entité logiciel+matériel dans une certaine configuration.

Voici une deuxième définition possible, que nous qualifierons cette fois de fonctionnelle :

> (Une application collecticielle est) un système informatique qui assiste des groupes de gens engagés dans une tâche (ou un but) commun et qui fournit une interface vers un environnement partagé. [EGR91]

Le collecticiel possède donc des liens étroits avec le *travail coopératif assisté par ordinateur* (*Computer Supported Cooperative Work* ou CSCW).

2.1 Variabilité du collecticiel

Comme le note Grudin [Gru91], le collecticiel se caractérise actuellement par un manque de définition claire et largement acceptée. Une seule certitude : le collecticiel existe. Le terme "collecticiel" indique que plusieurs personnes seront impliquées dans l'emploi d'une même application pour la réalisation d'une tâche commune. Les différences entre plusieurs applications collecticielles peuvent porter sur :

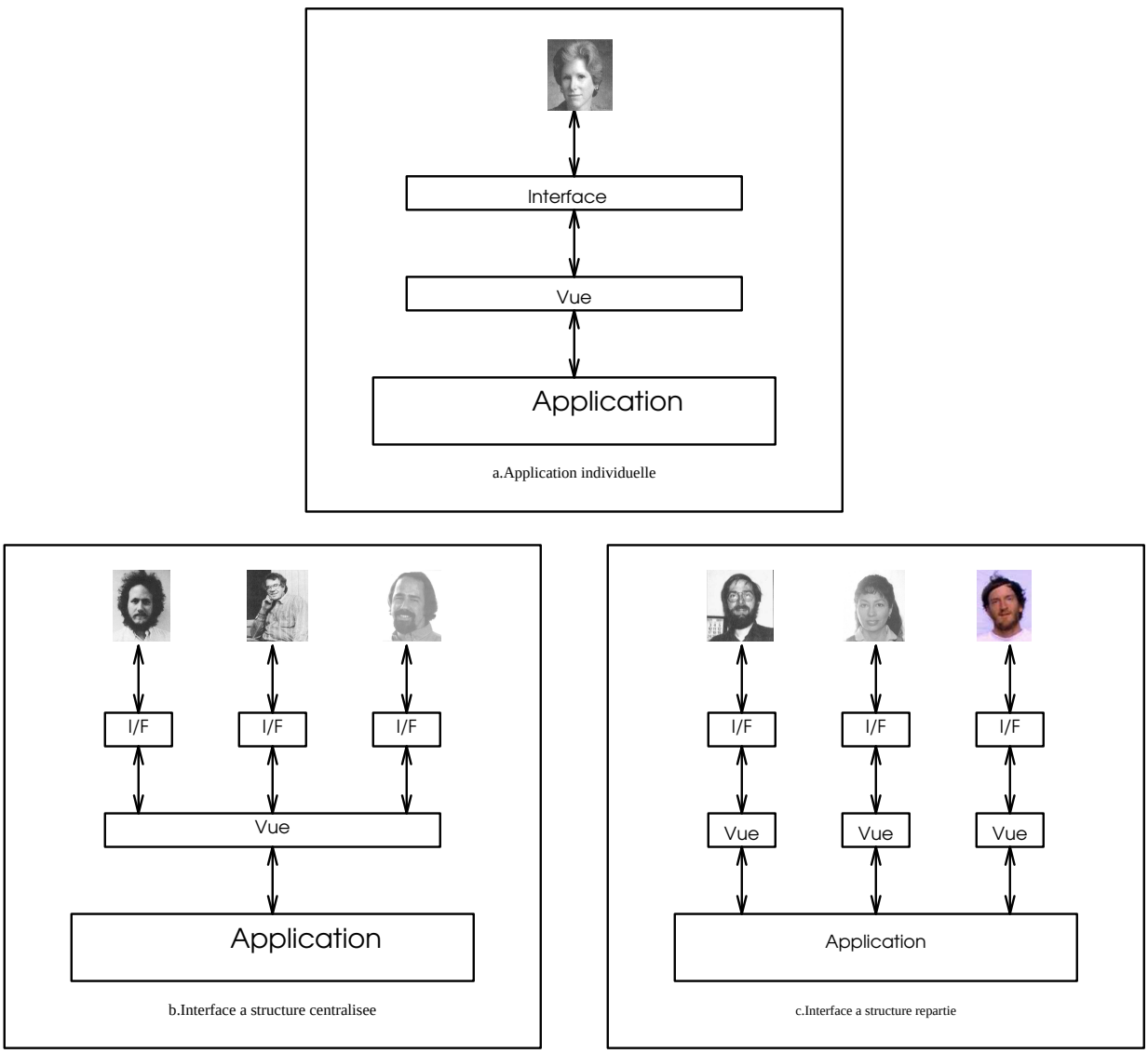

Figure 1: Architectures d'interface

```
Received 12345 bytes of 1165350
```

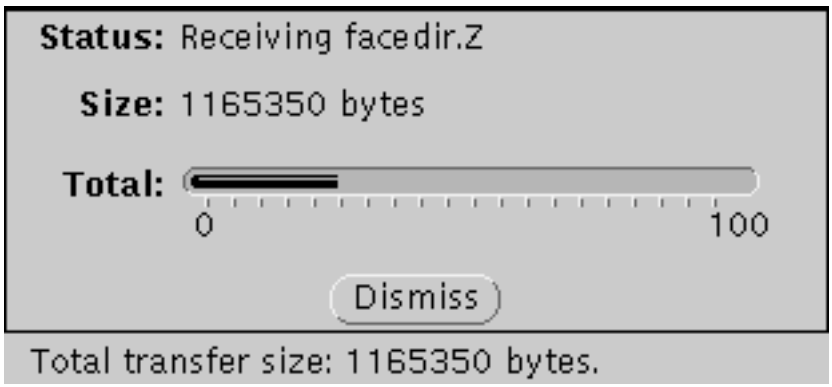

Figure 2: Information interne et information abstraite

- La taille du groupe mis en jeu. Un groupe peut être aussi petit que les deux co-auteurs d'un article, ou aussi grand que les centaines d'abonnés d'un serveur informatique.
- La dispersion géographique. Le groupe peut être dispersé dans une aire géographique aussi faible qu'une salle de réunion, ou aussi vaste que le monde entier, comme c'est le cas pour le courrier électronique.
- Les temps de réponse nécessaires. Le temps de réponse demandé peut être d'une fraction de seconde dans un simulateur de vol, quelques instants dans une réunion, ou plusieurs jours pour du courrier électronique. Ceci nous amènera à distinguer le collecticiel *synchrone* (à temps de réponse bref) du collecticiel *asynchrone* (à temps de réponse plus long).

2.2 Exemples

Quelques exemples vont permettre d'appréhender les différentes catégories de collecticiel.

Jusqu'ici, peu de véritables applications collecticielles ont été réalisées. Les programmes qui ont été développés sont avant tout des maquettes qui permettent de tester des idées. Les principaux développement concernent :

- Les jeux : jeu de cartes ([Pat91]), de Black Box ([DKR90]),MazeWar, Xtrek...
- Les outils d'édition partagée, sans doute les plus nombreux : InterNote [CBY89],VNS [SCG89], quilt [LFK88], Suite [DC91a]... Parmi eux, des outils d'édition de dessins, que l'on peut considérer comme l'exemple type d'interface à manipulation directe; GroupDesign et Wscrawl (figure 3, notez le télépointeur).
- Les systèmes d'agenda multi-utilisateurs, de conférence, de vote, etc., également très nombreux ([Gib89], [SG85] et [GS87]; on peut trouver une revue dans [KK88]).
- Le ShareBoard écrit par Alain Karsenty au LRI.
- Téléconférence Multimédia Mondiale et mondes virtuels multi-participants.

Ces catégories se recouvrent partiellement; par exemple les systèmes de co-authoring peuvent contenir un agenda multi-utilisateur, etc. Il y a des limites à ces recouvrements sur lesquelles nous allons revenir après avoir décrit quelques applications plus en détail.

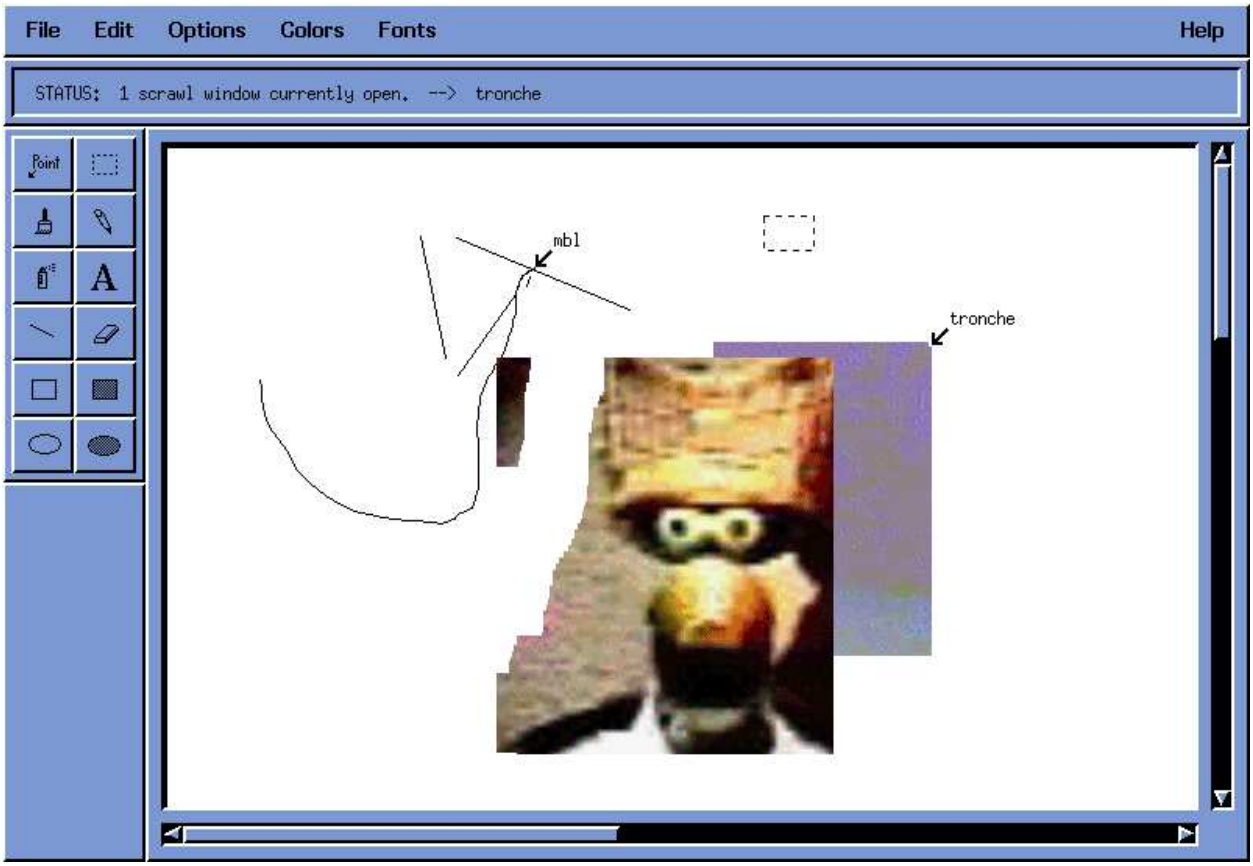

Figure 3: Wscrawl

2.2.1 Jeux

RENDEZVOUS [Pat91] est un jeu de cartes à plusieurs utilisateurs, chacun ayant sa propre vue du jeu. RENDEZVOUS est une application centralisée (les notions d'application centralisée et répartie seront définies précisément à la section 2.3.2, page 11).

DBB (Distributed Black Box) décrit dans [DKR90] est un jeu dans lequel plusieurs joueurs doivent deviner la position de planètes dans un espace bidimensionnel en lançant dans cet espace des rayons qui y subissent la gravitation, et en observant la déviation. Chaque utilisateur ne voit qu'une partie du jeu et doit donc communiquer avec les autres (verbalement) pour obtenir les renseignements permettant au groupe d'aboutir.

MazeWar (figure 4) est un jeu créé à MIT au début des années 70. La version Unix est due à Christopher Kent de DEC (ce qui est pour le moins paradoxal !). Dans ce jeu en temps réel, tous les joueurs se déplacent dans un labyrinthe où ils doivent trouver et abattre les autres joueurs. Le programme a une architecture *répartie*. Une instance particulière du programme gère l'entrée en session des nouveaux joueurs. Un programme permet de diffuser un message permettant de trouver des sessions déjà en cours. La légende veut que le programme ait été supprimé dans les années 70 car, au cours d'un certain mois, la moitié des paquets circulant sur ArpaNet étaient des paquets MazeWar circulant entre Stanford et MIT !

Xtrek est un jeu temps-réel créé par Chris Guthrie à Berkeley, et modifié par beaucoup d'autres depuis. Dans cette application, centralisée et construite au-dessus de X, chaque joueur dispose d'un vaisseau et doit aller abattre d'autres vaisseaux. Le processus qui détient l'état de l'application doit être lancé à la main avant le début du jeu.

2.2.2 Edition partagée

Ce sont les systèmes les plus répandus et les plus anciens. Ils correspondent à des programmes qui permettent à plusieurs utilisateurs d'éditer simultanément un même document.

InterNote [CBY89] est un programme d'annotation de documents hypermédia qui peut être considéré comme le niveau zéro du collecticiel. InterNote tourne sur le Macintosh. Les utilisateurs doivent se succéder devant la machine ou utiliser le programme à distance via AppleShare. Dans ce dernier cas, l'application ne pourra toujours être utilisée que par une seule personne. Néanmoins l'application prend explicitement en compte la présence de plusieurs utilisateurs, par exemple en gardant la trace de l'auteur de chaque annotation. InterNote est une application centralisée.

VNS (Virtual Notebook System) [SCG89] est le niveau juste au-dessus. Il s'agit également d'un système de création de document hypermédia. Bien qu'il ne prenne pas plus en compte un accès multi-utilisateur qu'InterNote, VNS a été écrit sous Unix et utilise le système graphique X. Il bénéficie donc "gratuitement" des possibilités d'accès réseau et multi-utilisateurs de Unix. Le système s'appuie sur une base de données multi-utilisateurs, une base étant stockée dans chacun des *serveur de groupes de travail*. Les serveurs communiquent à travers le réseau via un ordinateur particulier. L'application est donc centralisée.

Le ShareBoard écrit par Alain Karsenty du LRI, est une extension de l'album du Macintosh (un bloc-note dans lequel un utilisateur peut stocker du texte, des graphiques, ou n'importe quel type de données reconnu par le Macintosh). Chaque machine possède un ShareBoard, et un utilisateur peut stocker et récupérer des documents dans un ShareBoard *autre que le sien*. Le ShareBoard garde pour chaque document la trace de l'utilisateur qui l'a créé.

Quilt [LFK88] est un système de création et d'annotation de documents (asynchrone) qui peut être considéré comme un véritable collecticiel : il prend en compte explicitement l'exploitation par plusieurs utilisateurs et dispose de mécanismes de notification entre les utilisateurs à plusieurs niveaux.

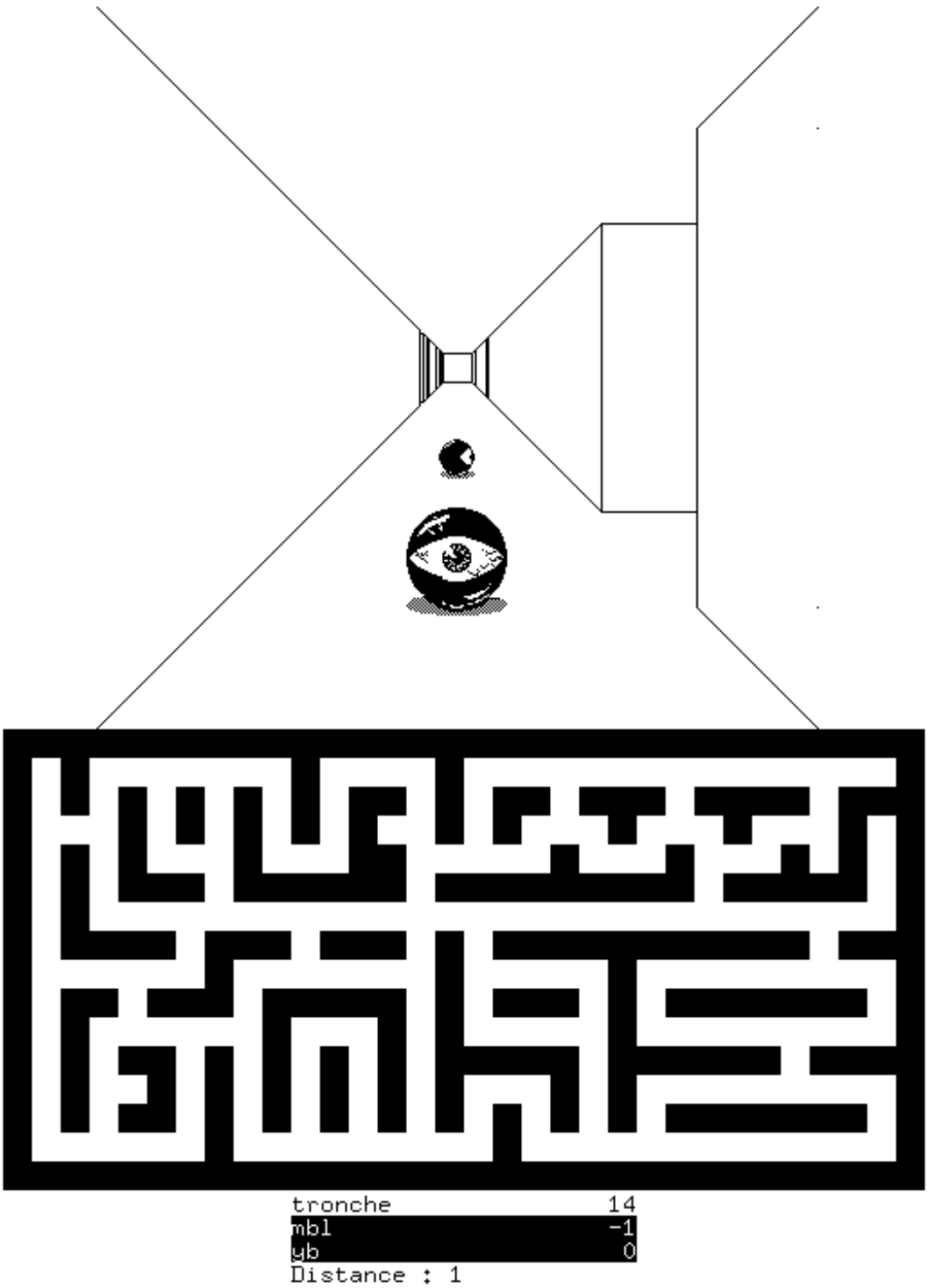

Figure 4: mazewar

Suite [DC91a] est également un véritable système collecticiel qui permet à plusieurs utilisateurs de remplir simultanément des formulaires. Suite est une application centralisée sous X.

CES (Collaborative Editing System) [GS87] est un autre éditeur partagé synchrone. Son principal intérêt est d'avoir été implémenté en Argus [LS83], un langage explicitement conçu pour écrire des programmes distribués. Cependant, Argus est basé sur l'hypothèse qu'il n'y a pas de contrainte temps-réel forte, ce qui n'est pas le cas pour les systèmes à manipulation directe.

GroupDesign [BLK92] est un éditeur de dessin structuré écrit par Alain Karsenty au LRI. GroupDesign est une application synchrone, c'est à dire que lorsqu'un utilisateur créé un objet, les autres utilisateurs le voient apparaître immédiatement sur leur écran. GroupDesign possède une architecture *purement répliquée*.

2.2.3 Rencontres informatiques

Nous décrivons dans cette section différentes applications ayant trait à la prise de décision en groupe et à la conférence électronique (dont les principaux éléments d'implémentation sont la connection de terminaux (talk) et la réplication de fenêtres [EGR91]). On trouvera également quelques exemples intéressants dans [Bux91].

LIZA [Gib89] est une boîte à outils permettant d'écrire du collecticiel (ayant une structure centralisée). Plusieurs applications collecticielles ont été développée avec LIZA afin de prouver son utilisabilité; parmi eux : un outil de vote et et des outils montrant quels sont les utilisateurs en session et ce qu'ils font.

MPCAL et RTCAL (Multi-Person CALendar et Real-Time CALendar) sont des agendas multi-utilisateurs écrits en CLU, à structure centralisée. RTCAL notifie les utilisateurs concernés au fur et à mesure que des dates sont entrées.

Les mondes virtuels. Ce concept a émergé très récemment et ses contours sont encore assez flous, comme souvent en matière d'interface homme-machine. Voici quelques définitions :

> "Les interfaces utilisateurs du style de la réalité virtuelle permettent à l'utilisateur de manipuler des objets virtuels dans un environnement 3d en utilisant des périphériques d'entrée 3d." [SLGS92].
>
> "Un monde virtuel est un environnement tri-dimensionnel, interactif et multisensoriel où les interactions homme-machine sont basées sur la façon dont nous interagissons avec le monde réel" [CJK+92]

Un exemple de système réellement implémenté est Rubber Rocks [CJK+92] qui est bati autour d'un moteur plus général décrit dans [LKL91]. Il s'agit d'un jeu qui se joue à deux par entrée multi-modale (gant, tracker, reconnaissance vocale, etc.) dans un monde simulé d'objets déformables qui rebondissent. Les évènements en entrée (par exemple le flot de données venant d'un gant de donnée, d'un tracker, d'une entrée vocale) sont traduits en évènements de haut niveau ("prendre un objet") par des filtres. Un processus, qui joue le rôle d'UIMS, collecte tous les évènements de haut niveau en entrée dans le système et pilote d'autres processus (simulation, interaction, rendu graphique). Les calculs liés à l'affichage sont répartis sur plusieurs stations pour des questions de performances, mais le système est fondamentalement centralisé.

XTV . XTV est un programme écrit à l'université de Caroline du Nord à Chapel Hill. C'est un programme qui permet de lancer des applications X quelconques et de les rendre multi-utilisateurs, dans le sens où tous les participants de la session voient les mêmes fenêtres. Un utilisateur doit lancer un programme de supervision préalablement au lancement de toute application. Cet utilisateur, appelé le *président* de la session jouit de droits spéciaux. Un seul utilisateur à la main à un instant donné. Si un autre utilisateur la veut, il doit la demander et attendre que celui qui la détient la libère (sauf le président qui peut reprendre la main d'autorité). Les demandeurs peuvent être mis en file d'attente. C'est évidemment une application centralisée sous X.

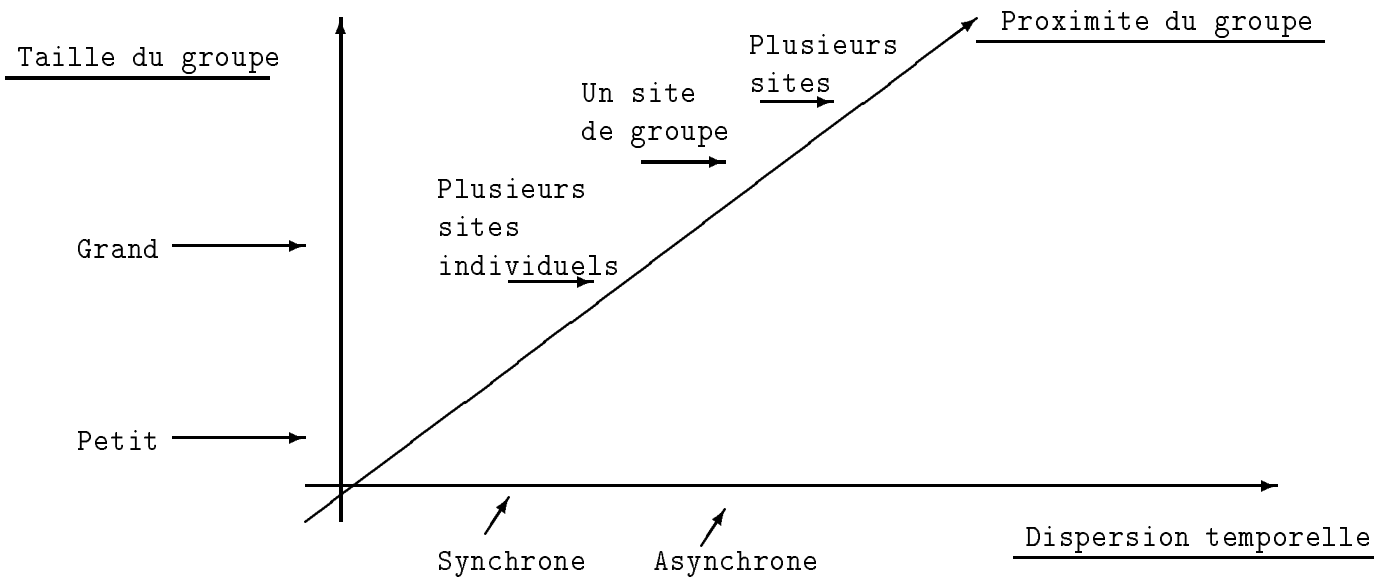

Figure 5: Taxonomie du collecticiel

VideoDraw et le VideoWhiteBoard. Ceci est une intéressante expérience de collecticiel sans informatique mené au Xerox PARC [TM91]. VideoDraw est constitué de deux tables à dessins. Sur chaque table, on projette l'image vidéo de l'autre table. Il est donc possible de mixer les deux dessins et de voir sur chaque table le travail effectué sur l'autre. Le VideoWhiteBoard est un tableau partagé utilisant un mécanisme similaire.

2.3 Analyse du collecticiel

Dans cette section, nous nous efforçons de mettre en évidence les caractéristiques clés du collecticiel.

2.3.1 Différence entre le collecticiel et d'autres applications multi-utilisateurs.

La différence entre une application collecticielle et une application distribuée "classique" est qu'une application collecticielle regroupe plusieurs utilisateurs qui travaillent ensemble et simultanément sur le même document ou dans la même session. En ceci, une application collecticielle s'oppose fortement à un système d'exploitation par exemple, dans lequel un programme permet un accès *transparent* des ressources logicielles à différents utilisateurs. Un système temps partagé s'efforce de donner à chaque utilisateur l'impression de disposer seul des ressources de calcul, de stockage, d'impression, etc., de la machine, alors que ces ressources sont en réalité partagées par tous les utilisateurs. Au contraire, dans une application collecticielle on va s'efforcer de restituer à des utilisateurs qui disposent généralement chacun de leurs ressources propres, l'illusion de l'existence d'un "espace de travail" commun.

Une application collecticielle doit permettre à chaque utilisateur d'être conscient de la "présence" des autres utilisateurs avec lesquels il collabore pour accomplir sa tâche. Par exemple, au cours d'une téléconférence, on veut donner à chaque utilisateur l'illusion de la présence directe des autres utilisateurs.

2.3.2 Taxonomie du collecticiel

Classiquement, on distingue les différentes catégories de collecticiels selon les trois axes de l'*espace*, du *temps* et de la *taille du groupe* (figure 5) ([EGR91], [Flo91]).

Si on cherche à discrétiser, on peut dire que la taille du groupe peut être "petite" (quelques personnes à une dizaine de personnes), ou "grande" (une dizaine à plusieurs milliers de personnes). La proximité peut être divisée en trois classes : "plusieurs sites individuels", qui correspond à plusieurs utilisateurs, chacun derrière sa station de travail; "un site de groupe", qui correspond à une salle de conférence, et "plusieurs sites de groupes", qui correspond à des salles de conférences et/ou des utilisateurs isolés dispersés dans le monde. Enfin, le *couplage* (c'est à dire le fait qu'une action d'un utilisateur soit rapidement perçu par les autres utilisateurs) peut être fort, fort relaxé ou faible.

Notons que ces trois variables sont fortement correlées : il est impensable d'avoir un très grand groupe (plusieurs milliers de personnes) devant la même console, de même qu'il n'est pas réaliste avec les réseaux actuels de demander à deux machines sur deux continents différents d'afficher la même chose au même instant.

Il faut ajouter une quatrième composante à cette taxonomie, invisible de l'utilisateur mais cruciale pour le programmeur : la structure de l'interface. L'interface peut être centralisée ou répartie à des degrés divers. Les trois cas possibles sont :

- Il existe quelque part un *unique* processus qui détient les données décrivant l'état de l'application.
- A l'autre extrémité du spectre, l'architecture peut être purement répliquée : dans ce cas chaque interface utilisateur est associée avec un processus local qui détient l'état de l'application. Les processus qui détiennent localement cet état doivent communiquer afin que toutes leurs représentations soient consistentes.
- Enfin, dans une architecture hybride, les vues sont réparties, mais un processus connaît l'état du système et joue le rôle de serveur de transactions.

Cette dernière dimension est également dépendante des autres. Plus une application est dispersée géographiquement, moins elle sera centralisée. En effet, compte tenu des réseaux qui sont actuellement disponibles, il est impensable d'obtenir des performances correctes si chaque fois qu'un utilisateur clique dans un menu à New York, la commande doit aller jusqu'à Orsay et retour. C'est pourtant ce qui se produit quand on lance une application à distance sous X.

2.3.3 Le problème des performances

Cette distinction entre les architectures distribuées ou centralisées ne correspond pas à une vision abstraite de l'application mais à l'endroit où sont physiquement effectués les calculs. Ceci est crucial, car aujourd'hui une bonne interface homme-machine est une interface homme-machine dont la dialogue est notablement graphique et utilise la manipulation directe : l'interface fournit une métaphore du monde réel (comme la métaphore du bureau popularisé par le Macintosh). Or, ce type d'applications graphiques dévore la puissance de calcul, et notamment la bande passante graphique, surtout si la machine sur laquelle le programme tourne ne dispose pas de hardware spécialisé (sprite, DMA, blitter). Si en cliquant sur une icône et en la déplaçant vers la poubelle on veut voir à l'écran l'icône suivre le curseur, cela implique d'être capable de déplacer en mémoire des milliers de points couleurs plusieurs fois chaque seconde.

Ainsi, pour les interfaces à manipulation directe en général et les mondes virtuels en particulier, la performance est absolument cruciale. Il est noté dans [LKL91], qu'un rendu plus naturel et plus intuitif pour l'utilisateur dépend plus d'un mouvement doux et régulier et d'un temps de latence faible, que des performances en vitesse pure (c'est à dire ici la vitesse d'affichage du système). Il n'est pas réaliste de faire effectuer les calculs d'affichage des interfaces d'une dizaine d'utilisateurs sur un seul ordinateur. Dans le cas du collecticiel, ce serait d'ailleurs maladroit, car, les applications étant intrinsèquement distribuées, on peut espérer disposer d'une puissance de calcul considérable puisqu'elle est répliquée. Obtenir de bonnes performances en vitesse pure est donc plutôt plus réaliste qu'avec des systèmes centralisés.

Certains problèmes ne peuvent être réglés que par l'application. Ainsi, pour des applications de mondes virtuels, on envisage d'avoir des "pièces" différentes. Lorsqu'un utilisateur passe d'une pièce à une autre, il faut transmettre les attributs des objets situés dans la nouvelle pièce. Ceci comprend aussi bien les

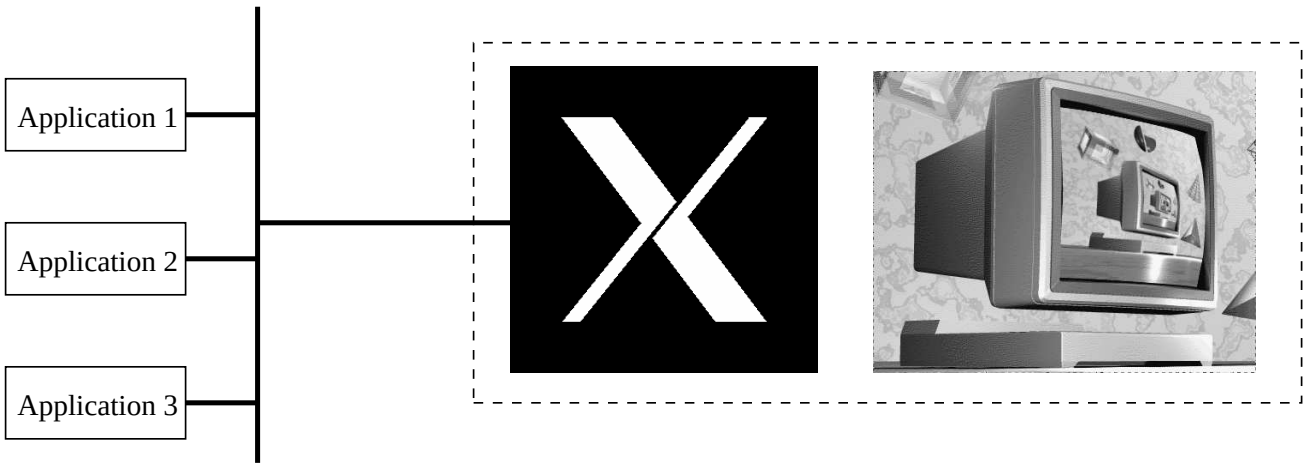

Figure 6: Le modèle de X

coordonnées des objets que leurs formes, les équations de leurs déplacements, voire leurs comportements ou leurs textures, ce qui représente une quantité d'information considérable. Quand des objets se déplace dans une pièce éloignée, transmettre leur coordonnées en permanence constitue un gaspillage de bande passante. Il faut effectuer une décomposition hiérarchique de la structure du monde virtuel, par exemple ne transmettre les déplacements que pour les objets se trouvant dans les pièces adjacentes. Ceci n'est clairement possible que si l'on possède une bonne connaissance de l'application.

2.3.4 Architecture répliquée contre architecture centralisée

C'est à la lumière de ces questions de performance qu'il faut examiner les différentes architectures possibles. Jusqu'ici, le fonctionnement des systèmes a été très influencé par leur architecture interne. Ainsi, aucun système actuel ne possède de composantes asynchrones et d'autres composantes synchrones car, dans le désert actuel d'outils de conceptions de collectiels, décider de faire une application synchrone est un choix fondemental qui va dicter l'ensemble de la structure de l'implémentation.

On peut noter que dans le cas du collecticiel homogène le choix d'une architecture centralisée ou répartie dépend fortement du réseau qui a été retenu : architecture répartie sur un réseau AppleTalk et architecture centralisée sous X [MAH90]. Nous allons nous attarder quelques instants sur les deux standards AppleTalk et X/IP qui sont au centre de pratiquement toutes les réalisations en matière de collecticiel.

AppleTalk. Dans la dernière version du système d'exploitation du Mac Intosh, appelée Système 7 [Apple C91], le réseau est explicitement pris en compte. Dans Système 7, toute application est une boucle qui lit les évènements de la file des évènements (qui est commune à toutes les applications d'une machine donnée). Les évènements peuvent être aussi variés que des clicks venant de la souris, des impulsions d'horloge, des signaux venant du système d'exploitation ou des évènements venant du réseau. Système 7 contient toute un jeu d'instructions (la PPCToolBox, pour Program-to-Program Communication Tool Box) qui permettent de localiser une application sur une machine distante et de lui envoyer explicitement un évènement. Ceci favorise donc l'écriture d'applications réparties, puisque chaque machine à une vision "locale" de son état.

X Window System. Le système de fenêtrage X Window System (plus connu sous le nom de X) est basé sur une architecture client/serveur qui communiquent au travers du réseau (figure 6). Chaque machine fait tourner son propre serveur qui reçoit des requêtes graphiques depuis les applications, qui sont les clients. Une application peut travailler avec plusieurs serveurs; ceci conduit donc naturellement à des applications centralisées. X a été conçu dès le départ pour être transparent au réseau. Il est donc assez facile d'y écrire des applications "multipostes", et donc multi-utilisateur (voir par exemple[BL90a], [Pat91], [CJK+92], [DC91a], également [LFK88]).

Le principal avantage d'une architecture répliquée est le temps de réponse, pratiquement identique à celui d'une application pour un seul utilisateur, puisque les commandes lancées en local sont executées en local. Pour une architecture hybride, cet avantage est conservé si le réseau n'est pas trop étendu. En revanche, si on a une application hybride sur un réseau étendu ou une architecture centralisée, grossièrement le temps de réponse croît avec les distances dans le réseau (exprimées en débit moyen et temps de latence sur les chemins empruntés par les communications).

Les inconvénients de l'architecture répliquée sont aussi évidents que ses avantages : on se heurte au problème de maintenir la cohérence entre les différentes représentations de l'état de l'application.

Un autre problème est celui des mécanismes de diffusion de messages vers tout le monde (broadcast) ou vers une liste d'applications (multicast). La plupart des réseaux ne disposent pas de telles facilités, sauf les réseaux locaux (genre Ethernet). Lorsqu'elles existent, ces facilités ne sont pas forcément commodes à utiliser : comment utilise-t'on le broadcast sur un réseau Ethernet depuis Unix ??? Donc, dans les applications actuelles, le multicast est souvent codé "à la main" dans l'application.

Ainsi, dans le cas d'une application centralisée écrite sous X, l'accès à X se fait via la Xlib, une bibliothèque écrite en C qui réalise l'interfaçage entre le client et le serveur. La Xlib ne sait pas utiliser le multicast. L'affaire est donc entendue : avec n serveur, le client enverra n messages.

Dans le cas d'une architecture purement répliquée, le problème est plus complexe. Dans le cas de GroupDesign par exemple, un message est envoyé chaque fois qu'un objet est créé, contenant le type de l'objet (rectangle, ovale), ses coordonnées et d'autres attributs. Chaque machine possède la liste de toutes les autres et envoie un message à chaque machine distante. Ainsi, si on suppose un débit moyen en sortie pour chaque machine (correspondant à l'activité de l'utilisateur), le nombre de messages circulant dans le réseau croît comme le *carré* du nombre de machines, ce qui risque de devenir rapidement inacceptable. Il serait possible pour chaque machine d'envoyer un message, disons à son voisin de gauche (en admettant qu'on ait créé un anneau logique sur le réseau), et de détruire le message quand il lui revient (technique du jeton). Ainsi, l'augmentation du nombre de messages serait seulement proportionnel au nombre de machines en session, mais le délai de transmission (qui est de "un lien traversé" avec la version actuelle de GroupDesign), passerait à $n/2$ liens, ce qui devient rapidement inacceptable, sans parler de ce qui se passe si une machine tombe en panne.

Une solution possible à ce problème sera examinée dans une lointaine section (section 4.5.3, page 33).

2.3.5 Problèmes spécifiques issus d'une interface répartie

Inadaptation des environnements actuels. Les systèmes d'exploitation et les réseaux actuels sont relativement inadaptés aux application collecticielles synchrones : Unix ne permet pas d'exécuter une instruction en temps borné, Ethernet (qui est en général le dernier tronçon d'un réseau IP) est un réseau de transmission à temps non borné.

Un environnement collecticiel nécessite l'emploi constant de la notification : des processus doivent être avertis de l'occurrence d'évènements dont ils dépendent. L'idéal serait que ceci soit pris en compte au niveau du système d'exploitation, mais un système comme Unix (que certains considèreront comme archaïque, mais qui est très bien implanté et pour encore longtemps), est plutôt pauvre dans ce domaine. L'unique primitive de communication asynchrone est l'instruction `kill`, qui ne peut notifier qu'un processus ou un groupe de processus, un processus ne pouvant appartenir qu'à un seul groupe de processus.

Par ailleurs, certains réseaux ne peuvent pas fonctionner sur une grande échelle, par exemple le Macintosh collecte les adresses des autres machines par diffusion, ce qui est évidemment impossible dans un réseau de grande dimension.

Enfin, des liens étroits unissent le collecticiel et le multimédia. Pour supporter et renforcer les protocoles sociaux (c'est à dire des protocoles entre utilisateurs), des liens audio et vidéo sont utiles, qu'ils soient ou non pilotés par ordinateur. De tels moyens de communication ne sont pas encore largement répandus [EGR91].

Complexité du logiciel. Le CSCW (Computer Supported Cooperative Work) réclame un mélange particulièrement riche d'éléments empruntés aux bases de données, aux langages de programmation (types), aux

techniques réseaux, aux interfaces graphiques, etc. Une étude récente [MR92], indique que le développement de l'interface utilisateur occupe en 1992 toujours une part importante du volume de code a écrire dans une application, plus de la moitié du temps de développement, et par voie de conséquence une part considérable du coût de développement. En fait, comme le note les auteurs de cet article, il semble qu'il y ait une difficulté *intrinsèque* à l'écriture des interfaces, une difficulté qui est toujours mal dominée : écrire une interface homme-machine, c'est généralement se heurter à des problèmes de logiciel complexe. Ecrire une interface multi-utilisateur, c'est se heurter à des problèmes encore plus complexes. Ainsi, Patterson écrit :

> "Although in single-user applications abstraction is often a nicety, in multi-user applications it is a necessity." [Pat91]

La relative nouveauté du domaine implique un manque d'outils de développement. Pour établir un parallèle avec le développement des outils graphiques, nous n'en sommes qu'à la manipulation des pixels individuels, aux fonctions d'accès au réseau écrites "à la main". Il n'existe pour le moment rien de comparable à une toolkit comme Motif en ce qui concerne le collecticiel, les services offerts par LIZA, par exemple, restant très sommaire.

Transférer les données sur un réseau peut être difficile, particulièrement des données très structurées. Les auteurs de [GS87] notent qu'un des principaux problèmes dans le développement (et dans les performances) de MPCAL et RTCAL a été le besoin de gérer les données hors de l'espace d'adressage, c'est à dire de les transmettre aux applications distantes (dans le cas de RTCAL).

Il est généralement considéré comme une bonne pratique de la programmation des interfaces de séparer la *vue* de l'interface de son abstraction, c'est à dire les valeurs qui sont représentées de leur représentation réelle. Patterson [Pat91] note que, s'il est en général difficile de déterminer "ce qu'il faut mettre dans l'abstraction" et "ce qu'il faut mettre dans la vue", c'est en général plus facile pour des applications multi-utilisateurs : les valeurs que tout le monde doit connaître sont "dans l'abstraction" et celle qui peuvent varier d'une vue à l'autre sont... dans la vue ! Ceci n'est cependant qu'une première approximation, et une connaissance de l'application finale est encore nécessaire pour faire exactement la séparation.

Afin de lutter contre la complexité de conception et de réalisation du collecticiel, on espère ne pas écrire entièrement une application à partir de rien. Par exemple GroupDesign à été écrit à partir d'une application existante, MetaDesign [BLK92]. Ceci nécessite néanmoins que l'application "individuelle" soit suffisamment extensible, ce qui est rarement le cas.

2.3.6 Facteurs humains

Un des problèmes du collecticiel est d'éviter que l'activité d'un utilisateur ne trouble l'activité d'un autre utilisateur engagé dans la même session. L'exemple classique est celui de l'éditeur partagé. Considérons deux personnes en train de travailler sur la même page. Chacun d'eux s'efforce de polariser son attention sur sa partie, mais risque d'être distrait par des modifications continuelles liées à l'activité de l'autre utilisateur.

Diverses solutions ont été expérimentées. Par exemple dans GROVE [EGR91], le texte nouvellement entré est en bleu et vire progressivement au noir, ce qui permet à l'utilisateur de se préparer à son arrivée. Des tâches de couleurs indique les zones de changement de texte, sans effectuer immédiatement les modifications. Ainsi un utilisateur est conscient des modifications sans être tenté de les suivre au fur et à mesure. Dans GroupDesign, des zones de couleur indique le recouvrement d'une fenêtre par celle d'un autre utilisateur, est donc la zone dans laquelle ce dernier est susceptible d'intervenir. Ceci ne doit pas être trop intrusif ni trop distrayant. Par exemple, dans CES [GS87], l'écran de l'éditeur partagé est couvert d'informations concernant les auteurs des différents fragments, le noeud dans lequel on se trouve, etc. ce qui le rend illisible.

Lacunes des métaphores d'interaction actuelles. Les objets interactifs qui sont utilisés aujourd'hui dans les interfaces graphiques sont le résultat de plusieurs années de réflexions dans le domaine de l'interaction mono-utilisateur. Il faut peut-être imaginer de nouveaux objets interactifs; est-ce qu'une scrollbar a un sens si elle peut être manipulée par plusieurs personnes ?

2.4 Les problèmes techniques : état de l'art

2.4.1 Les conflits

Des conflits apparaissent inévitablement lorsque plusieurs personnes travaillent sur le même document. Par exemple, deux personnes peuvent vouloir déplacer un même objet dans deux directions différentes. Dans ce cas il y a deux façons de régler le problème : laisser les utilisateurs le régler, ou implémenter dans l'application un mécanisme pour le régler. On parle dans le premier cas de protocole social, et dans le second de protocole matériel.

La première solution n'est pas forcément la plus mauvaise. Elle a au moins l'avantage d'être simple à implémenter : il n'y a rien à faire ! Il est noté également dans [DKR90] que ceci favorise le développement par les utilisateurs de leurs propres protocoles d'utilisation de l'outil collecticiel qui leur est confié. On peut également avertir l'utilisateur et le laisser régler le conflit. Par exemple dans MPCAL (rappelons qu'il s'agit d'un système de gestion de calendrier), si des contraintes de rencontre sont incompatibles, les utilisateurs impliqués dans le conflit sont notifiés et doivent résoudre le conflit. Dans le cas d'accès concurrents au même document, les utilisateurs sont avertis. Ils peuvent choisir d'ignorer l'avertissement, en sachant que le document sera alors modifié à leurs risques et périls [GS87].

On cherche cependant à laisser la machine régler le maximum de conflits. Pour leur résolution automatique, [EGR91] relève plusieurs méthodes possibles :

- Verrouillage
- Transactions
- Les utilisateurs prennent la main chacun à leur tour ("floor")
- Contrôleur centralisé
- Détection de dépendances
- Exécution réversible
- Transformation des opérations

Verrouillage. Le problème du *verrouillage* se pose dans toutes les applications collecticielles, et particulièrement dans les éditeurs partagés. Etant donné que plusieurs utilisateurs peuvent accéder simultanément à un même document, il faut une méthode qui assure la consistence, c'est à dire que chaque utilisateur ait bien le même document. Ce problème de consistence se pose à deux niveaux que nous appelerons la consistence système et la consistence utilisateur. Afin de mettre la différence en lumière, imaginons que deux utilisateurs écrivent un texte à l'aide d'un éditeur ayant une structure répartie. Imaginons également que les deux utilisateurs, à la suite d'un manque de coordination, commencent ensemble à écrire l'introduction. Les caractères qu'ils tapent se mélangent pour former une bouillie incompréhensible. Si les deux utilisateurs voient la même bouillie, il y a *consistence système*. Mais un collecticiel digne de ce nom doit faire plus. Lorsque deux personnes utilisent une application pour écrire le même texte, elles s'attendent sans doute à ce que le texte résultant soit cohérent. Généralement, les éditeurs collecticiels divisent le texte en unités logiques (des *fragments*) auxquels sont appliqués des verrouillages [EGR91] : pour chaque fragment, un nombre illimité de personnes ont le droit de lire si personne n'écrit, et si à un instant une personne écrit, le fragment ne peut être lu ni écrit par quelqu'un d'autre.

Ces verrouillages posent cependant des problèmes : ils prennent beaucoup de temps, il est difficile de savoir à quel niveau les faire; faut-il verrouiller une lettre, un mot, un paragraphe ? Si on laisse le choix à l'utilisateur, il risque d'avoir beaucoup de décisions à prendre, ce qui amènera une charge cognitive qui devrait être évitée. Si on a des verrous trop fins (lettre par exemple), le temps des verrouillages et déverrouillage devient prohibitif, et on retombe sur les problèmes de "consistence-utilisateur" évoqué plus haut. Avec un grain trop fort, des utilisateurs risquent d'être inutilement bloqués parce que l'un d'entre eux accède à une portion du document. Choisir le bon niveau de verrouillage nécessite donc de connaître l'application.

Enfin un utilisateur qui ne rend pas le verrou (parce qu'il oublie de le faire par exemple) peut bloquer l'accès à une partie du document inutilement. Pour éviter ceci, le verrou peut être libéré après un certain temps, ou on peut donner aux autres utilisateurs une commande pour détruire le verrou ([EGR91], [GS87]).

En ce qui concerne les éditeurs graphiques, pour lesquels on veut s'efforcer d'obtenir un aspect de manipulation directe, devoir mettre et retirer des verrous introduit un hiatus qu'il faut éviter (quoi que l'éditeur de graphe écrit avec LIZA fonctionne sur ce principe : chaque noeud du graphe à le statut public, partagé ou privé [Gib89]).

Transactions. C'est un modèle issu des bases de données. Une transaction est une unité de programme qui accède et met à jour (potentiellement) des variables. Au cours d'une transaction, toutes les variables concernées sont lues exactement une fois et sont écrites au plus une fois. On impose que si les données (qui représentent l'état de l'application) sont cohérentes avant la transaction, elles le sont encore après (mais peuvent ne pas l'être pendant). Le principal défaut de ce mécanisme est d'augmenter le temps de réponse de façon potentiellement inacceptable pour l'utilisateur, surtout dans le cas du collecticiel synchrone.

"Floor control". Un moyen très simple pour éviter les conflits est d'empêcher qu'il y en ait. Avec le mécanisme des floors (mis en œuvre par exemple dans XTV) un seul utilisateur à la main. Si un autre utilisateur la veut, il doit la prendre ou la demander et attendre qu'on la lui donne. La main peut être retirée automatiquement à quelqu'un qui la détient depuis trop longtemps.

Le principal inconvénient de ce système est évidemment qu'à un instant donné, un seul utilisateur à la main. Cependant introduire plusieurs "mains" permet d'obtenir une certaine souplesse [LA90] : un par application partagée ou un par fenêtre). On peut également autoriser plusieurs personnes à avoir la main. Le prix à payer pour tout ceci est évidemment un surcroît de complexité du logiciel, un temps de calcul plus élevé et une consommation plus élevée de bande passante.

Contrôleur centralisé. Ceci correspond à l'architecture centralisée. Toutes les opérations sont effectuées séquentiellement par un unique processus. Ce modèle a deux avantages : on sait le mettre en œuvre, et c'est particulièrement commode avec X. Les inconvénients sont les inconvénients habituels des mécanismes centralisés : augmentation du temps de réponse, goulet d'étranglement, et manque de fiabilité, une panne du contrôleur central entrainant l'arrêt de toute la session.

Détection de dépendances On remarque qu'en séquentialisant tout, on oblige de nombreuses opérations à s'exécuter l'une après l'autre alors qu'elles pourraient se dérouler en parallèle . Par exemple, pour un éditeur de dessin structuré, si deux personnes désirent déplacer deux objets distincts dans deux zones différentes, il n'y a aucune raison pour séquentialiser. En détectant les dépendances, il est possible d'effectuer le maximum d'opérations en parallèle et de ne séquentialiser que les opérations qui doivent l'être. Ceci requiert évidemment un logiciel d'autant plus complexe que les dépendances que l'on veut détecter sont à un niveau fin : il est facile de dire que bouger deux ellipses dans deux zones différentes ne posera pas de problème, en revanche, vérifier que leurs chemins ne se croiseront pas peut être très difficile, et même impossible si les utilisateurs manipulent leurs objets en temps réel.

Exécution réversible, C'est une technique très élégante. On effectue immédiatement les opérations. Si une opération entre en conflit avec une des opérations précédentes, les dernières opérations sont défaites jusqu'au conflit, qui est résolu, puis on refait toutes les opérations qui ont été défaites. La mise en œuvre de cette technique est complexe. Il faut être capable de défaire toute opération. Une pile est nécessaire, et il peut être difficile, voire impossible, de borner sa taille.

D'autre part cette technique pose des problèmes en terme de facteurs humains : supposons qu'il y ait un télépointeur (un curseur manipulé à distance par un utilisateur pour désigner des positions à l'écran de l'utilisateur local), l'utilisateur distant peut désigner un objet qui n'est pas encore apparu à l'écran.

GroupDesign utilise l'exécution réversible avec détection des dépendances pour minimiser la partie des opérations à défaire. Il est possible pour GroupDesign de borner la taille de la pile de ces opérations.

Transformation des opérations. Cette technique, qui est une extension des deux précédentes, est la plus sophistiquée. Lorsque deux opérations sont conflictuelles, on s'efforce de les composer pour obtenir le résultat voulu. Cette technique est évidemment très difficile à mettre en œuvre.

2.4.2 Couplage

Les premiers systèmes multi-utilisateurs étaient strictement WYSIWIS (What You See Is What I See). Mais un *couplage plus faible* peut être nécessaire [DC91a]. Par exemple, si un utilisateur remplit un formulaire électronique qui est observé par un autre utilisateur, ce dernier ne veut probablement pas être troublé par les fautes de frappes du premier : un champ ne devrait être transmit que lorsqu'il est remplit. Le couplage peut nécessiter un contrôle assez fin. Supposons un enseignant qui suit sur un écran le code écrit par quelques étudiants, au fur et à mesure qu'ils l'écrivent. L'enseignant pourra vouloir être avertit de toutes les modifications apportées par des étudiants débutants, susceptibles de faire beaucoup d'erreurs; pour des étudiants plus avancés, il ne veut être notifié que des modifications affectant la syntaxe, et pour des étudiants encore plus avancés, capables de comprendre seuls leurs erreurs de syntaxe, il ne voudra être notifié que des modifications susceptibles de changer la sémantique. Tout ceci permet à l'utilisateur de concentrer son attention sur les points cruciaux et de le décharger des détails sans importances (ce qui est évidemment toujours l'objectif en matière d'interface utilisateur). Dans le système Suite de [DC91a], plusieurs personnes peuvent visualiser, remplir et modifier simultanément des formulaires (correspondant par exemple à un budget familial). Il est possible de configurer l'instant où les autres utilisateurs sont notifiés d'une modification : automatiquement, quand un champ est validé, sur demande de l'utilisateur, etc.

Dans Quilt [LFK88], plusieurs personnes peuvent travailler simultanément sur un article. Lorsqu'elles ont fini, le document est envoyé à un relecteur (sur demande des auteurs). Le relecteur peut annoter le document à l'attention des auteurs, ou pour son usage privé. Lorsqu'il a terminé, il le signifie au système qui enverra aux auteurs les commentaires qui leur sont destinés. Il serait absurde d'envoyer ces commentaires avant que le relecteur est fini, alors qu'il peut encore décider, par exemple, que l'un d'entre eux est superflu.

Dans le cadre des systèmes à manipulation directe, la situation est légèrement différente. La manipulation directe induit des *contraintes temps-réel* qui peuvent être importantes. Si on prend le cas d'un simulateur d'avion de chasse avec au minimum deux personnes, il est clair qu'en phase de combat toute action doit être instantanément perçue par toutes les personnes engagées dans la simulation. Pour un éditeur graphique, la situation est moins critique.

Dans tous les cas, l'important est que l'utilisateur ne puisse pas faire la distinction entre une action transmise instantanément et une action qui arrive avec un certain retard à l'interface d'un utilisateur. Dans le cas d'un simulateur de combat, un délai dans les transmissions mettra rapidement le système dans un état incohérent. Dans le cas d'un éditeur partagé (par exemple), il est possible de simuler une propagation instantanée de l'information [BLK92]. Avec une telle méthode cependant, toutes les vues de l'état du système ne sont pas dans un état cohérent à chaque instant. Il est même possible qu'au cours d'une session, deux vues ne soient *jamais* dans le même état (bien qu'il y ait convergence de toutes les vues vers le même état en temps fini si le système est au repos).

En conclusion, la question de savoir si la notification doit être sous le contrôle de l'utilisateur ou être effectuée automatiquement par le système ne peut être réglée sans une connaissance de l'application.

2.4.3 Sécurité et confidentialité

Le collecticiel amène plusieurs personnes à partager un "espace de travail commun". Ces personnes peuvent souhaiter utiliser dans cet espace des données confidentielles (des numéros de téléphone privés par exemple); une entreprise peut souhaiter que des documents ne soient accessibles qu'aux personnes habilités, etc. Il importe donc que les applications qui détiennent ces documents coopèrent avec le système d'exploitation sous-jacent pour garantir des accès réservés aux documents.

Ces problèmes sont peu abordés pour le moment, et difficiles à traiter de toute façon : un système comme X ne prend pas les problèmes de sécurité en compte, et comme il est maintenant extrêmement répandu, et que, malgré ses défauts, il est suffisament adaptable pour couvrir une large gamme de besoins, il est pour

ainsi dire certain que X sera pour de nombreuses années encore LE système graphique (David Rosenthal l'appelle dans [Ros90] "le FORTRAN des systèmes de fenêtrage" !).

Les droits d'accès doivent souvent varier dynamiquement à cause de l'entrée et de la sortie de la session de nouveaux utilisateurs par exemple, ce que les systèmes d'exploitation actuels ne savent pas forcément faire.

Pour les systèmes qui reposent sur une base de données à laquelle on a rajouté des fonctionnalités multi-utilisateurs (et de support de travail collaboratif), les problèmes sont un peu mieux cernés et sont résolus essentiellement par le système de base de données. [GS87] signale qu'en général, les fichiers doivent être read/write pour tout le monde (puisque tout le monde doit accéder aux fichiers décrivant l'état de l'application). Les droits d'accès aux informations sont alors gérés par l'application collecticielle, ce qui la rend vulnérable à des attaques malveillantes (accès direct au fichier sans passer par l'application).

Les problèmes les plus difficiles à résoudre surviennent dans le cas d'une interface répartie lorsque l'interface et/ou l'application doivent détenir pour fonctionner des informations que l'on souhaite cacher à l'utilisateur. Par exemple, dans MazeWar chaque exemplaire du jeu doit disposer de la position des autres joueurs pour calculer la distance à l'adversaire le plus proche (qui est fournie au joueur). Un joueur qui connaît le protocole peut décoder les messages qui parviennent à son exemplaire et connaître la position des autres joueurs.

3 Hétérogénéité

3.1 Pourquoi écrire du collecticiel hétérogène

Les utilisateurs doivent travailler sur des plate-formes informatiques variée : MS-DOS, Macintosh, Unix, VMS. Les raisons de telles variations sont diverses : disponibilité de telle ou telle application, coût des machines, environnement déjà existant, performance... et cet état de chose persistera sans doute encore longtemps. Les utilisateurs sont habitués à un petit ensemble d'applications qu'ils utilisent quotidiennement. Si, comme c'est le cas sur le Macintosh, le système qu'ils utilisent s'efforce d'offrir une présentation et des commandes ("look and feel") communes à toutes les applications, l'utilisateur est également habitué au look and feel de son système, et en changer représente pour lui une charge cognitive supplémentaire qu'il importe de lui épargner.

Des utilisateurs travaillant sur des plate-formes variées peuvent néanmoins avoir besoin de travailler ensemble. On introduit alors la notion d'*application collecticielle hétérogène*. Celui qui veut écrire des applications de ce genre ne peut plus s'appuyer sur un système d'exploitation, un système graphique ou un modèle d'interaction unique connu de tous les utilisateurs.

3.2 Identifier l'hétérogénéité

Une application interactive collecticielle peut-être décomposée en couches.

Application
Bibliothèque d'interfaçage
Bibliothèque graphique
Système d'exploitation
Transport

Application
Bibliothèque d'interfaçage
Bibliothèque graphique
Système d'exploitation
Transport

L'hétérogénéité d'une application collecticielle hétérogène peut donc se loger en divers niveau. Au niveau le plus bas, celui du transport, il est clair que pour écrire une application collecticielle, il faut avoir au minimum une homogénéité point à point, c'est à dire que les ordinateurs doivent pouvoir communiquer. La couche la plus élevée extérieure à l'application est celle de la bibliothèque d'interfaçage, qui détermine le look and feel de l'application. A ce niveau, on peut très bien avoir une application collecticielle qui tourne sous Unix et sous X avec deux utilisateurs qui utilisent deux window managers différents (figure 7. Dans ce cas-là, l'hétérogénéité est ressentie par l'utilisateur mais est transparente au programme.

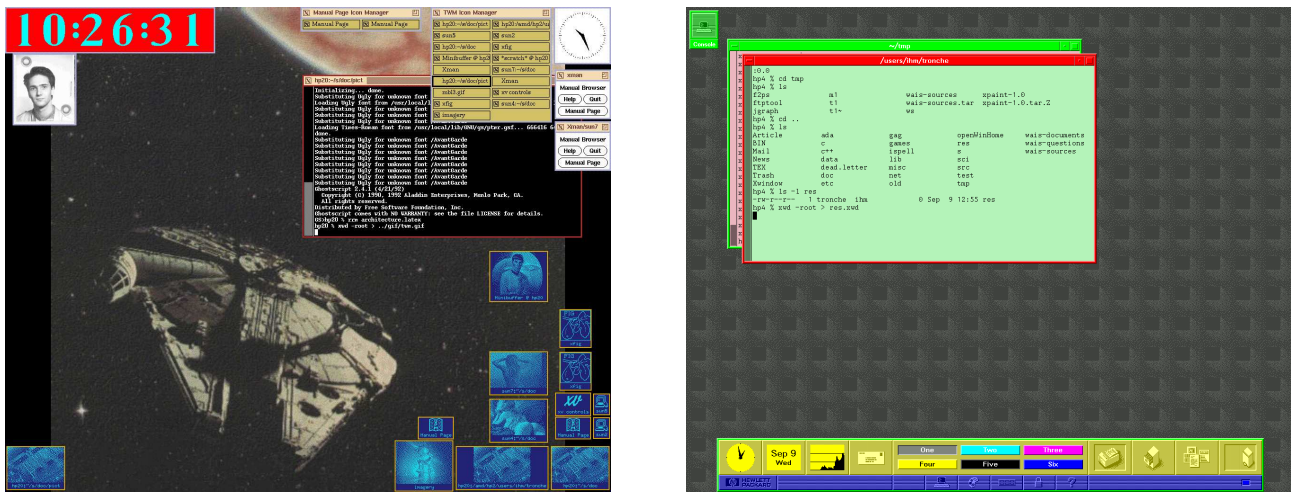

Figure 7: Deux window managers

3.3 Problèmes spécifiques du collecticiel hétérogène

Nous ne ferons qu'effleurer les problèmes dans cette section. Ils seront abordés plus en profondeur dans les sections décrivant comment ces problèmes se sont posés au cours de l'écriture du ShareBoard Unix et de l'éditeur de dessin structuré Unix, et quelles sont les solutions apportées.

3.3.1 Inadaptation des environnements actuels

Encore une fois, sans parler de l'absence de toolkits adaptées (le domaine étant trop récent), les systèmes et les réseaux existants sont aussi inadaptés à l'écriture de collecticiel hétérogène qu'à l'écriture de collecticiel homogène. Quelques problèmes supplémentaires se posent. Ainsi le système du Macintosh (non préemptif) est catastrophique pour écrire du multi-utilisateur (cf aussi [Pat91]).

3.3.2 Architecture répliquée

Nous avons déjà noté (section 2.3.4, page 13) que l'architecture purement répliquée imposait de trouver un compromis entre le nombre de liens traversés par un message, qui détermine son temps de transmission et qui est donc très important pour l'utilisateur, et le nombre de messages qui vont circuler dans le réseau. Il est important de voir que passer au collecticiel hétérogène met l'accent sur les problèmes de performances du réseau : la vitesse dans un chemin dans le réseau est la vitesse du lien le plus lent.

Dans un réseau hétérogène, des réseaux différents et incompatibles sont connectés par des gateways ou des routeurs. Ces gateways ne peuvent aller plus vite que le plus *lent* des réseaux auxquels ils sont connectés. Ils est donc important de minimiser le trafic réseau, spécialement le trafic qui traverse les gateways. Pour réussir ceci, il importe de dissocier les problèmes de *transports* des problèmes d'interfaces.

3.3.3 Nommage

Le problème du nommage dans un réseau hétérogène est important.

Un réseau homogène possède évidemment un mécanisme qui permet de nommer n'importe quelle machine à travers le réseau : ainsi IP stocke ses adresses sur 32 bits, Ethernet sur 48, etc. Une application sur le Macintosh peut être citée par un triplet donnant son nom, le nom de la machine sur laquelle elle tourne et le domaine AppleTalk dans lequel se trouve la machine. En IP, les adresses sont des triplets formés par l'adresse physique de la machine (sur 32 bits), le domaine (TCP ou UDP) et un numéro de port sur 16 bits.

Dans un réseau hétérogène, on veut évidemment pouvoir nommer des machines sur des réseaux pour lesquelles les conventions de nommage ne sont pas les mêmes.

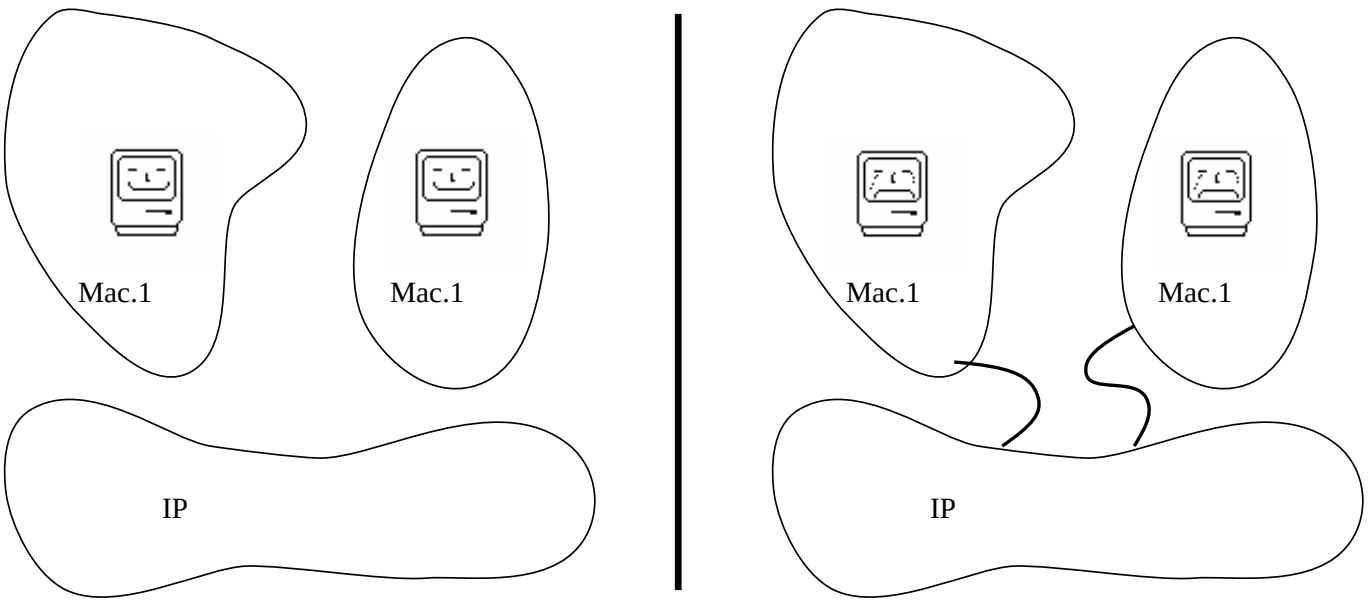

Figure 8: Collision de noms lors d'une connection de réseaux

La connexion de réseaux pose des problèmes de nommage supplémentaire. Si deux machines, chacune sur un sous-réseau, portent le même nom, une collision de nom se produit lorsque les deux sous-réseaux sont connectés. En principe, ceci ne doit pas se produire dans IP, pour lequel les noms reflètent la topologie physique du réseau. Ce n'est pas le cas pour AppleTalk. Des problèmes encore plus complexes peuvent survenir si les deux sous-réseaux ne sont pas connectés directement, mais par l'intermédiaire d'un troisième réseau de type différent, par exemple deux réseaux AppleTalk connectés au travers de IP (figure 8).

3.3.4 Formats

Les applications s'échangent des informations de formes très diverses : texte, image, son, etc. sous des formats eux-même très divers. Certains formats sont universellement connus, comme le langage de description de page PostScript. D'autres sont très spécifiques à un matériel, un système d'exploitation, voire une application. Travailler dans un environnement hétérogène impose la conversion de formats de représentation de données en d'autres formats.

4 Ecrire du collecticiel hétérogène

Dans cette section, nous allons passer en revue le travail d'implémentation réalisé dans le cadre de ce stage et dans le domaine du collecticiel hétérogène. Notre intérêt se porte sur les interfaces ayant une structure distribuée.

Nous possèdons déjà des application multi-utilisateurs tournant sur Macintosh : ShareBoard et GroupDesign [BLK92]. Nous allons écrire dans le monde Unix les applications correspondantes. Le tableau ci-dessous résume la situation :

Machine	Stations de travail	Apple Macintosh
Réseau	IP	AppleTalk
Protocole	UDP	AppleEvents
OS	Unix	Apple Macintosh
Système graphique	X11	Quickdraw
Application	dessin structuré	GroupDesign

Il s'agit ici de faire du "100 % hétérogène" puisque toutes les couches sont différentes (et incompatibles).

Communiquer. Au niveau le plus bas, le premier problème est de disposer d'au moins une machine qui soit capable de "parler" le langage des deux mondes AppleTalk et Unix. Ceci est obtenu en permettant à un Macintosh d'utiliser IP pour discourir avec l'extérieur, les liaisons locales de Macintosh à Macintosh continuant à se faire sur AppleTalk.

La liaison est obtenue physiquement par un câble Ethernet. La bibliothèque MacTCP permet de programmer le Macintosh en exploitant IP.

4.1 Retour sur IP et AppleTalk

Nous avons vu précédemment (section 2.3.4, page 12) comment le Macintosh intégrait élégamment le réseau dans son modèle d'ordonnancement, via les AppleEvents. X, s'appuie en revanche sur IP (Internet Protocol) dans le monde Unix, et utilise également un mécanisme basé sur des évènements. Quelques détails sur IP et AppleTalk faciliteront la compréhension de la suite.

Les principales caractéristiques qui permettent de classer un réseau du point de vue du protocole sont généralement [San91] :

- connexion / sans connexion : avec connexion, il est nécessaire d'initier la session au départ et de la terminer à la fin. Sans connexion, chaque message contient l'adresse du destinataire.
- conservation de la limite des messages : si elles ne sont pas conservées, le réseau délivre un service de type flot de données.
- Réception des messages dans l'ordre.
- Duplication / perte de messages.

Les AppleEvents sont des messages sans connexion, sans perte, sans duplication, avec conservation des limites du message et de l'ordre des messages.

IP peut utiliser deux protocoles : UDP (User Datagram Protocol) fournit, des datagrammes c'est à dire un mode sans connexion. C'est le service de base : les limites des messages sont conservées, les messages peuvent être perdus ou dupliqués et n'arrivent pas forcément dans l'ordre. TCP (Transmission Control Protocol) offre lui un service avec connexion, flot de données, les données étant reçues dans l'ordre sans duplication ni perte.

Dans la catégories des datagrammes un autre élément entre en jeu : la taille maximale d'un message qui peut être échangé. Un AppleEvent peut être aussi grand que le permet la mémoire. Pour UDP, la situation est beaucoup plus complexe. IP permet de relier des réseaux différents qui travaillent à des vitesses différentes et utilisent des ordinateurs, des systèmes d'exploitation très différents. Supposons qu'un très gros datagramme cherche à entrer dans un sous-réseau qui ne peut pas le recevoir. Le datagramme va être fragmenté, et les fragments vont circuler dans le sous-réseau, pour être réassemblés à la sortie de ce sous-réseau. Que se passe-t'il si un des fragments est perdu ? Le réassemblage échoue et le paquet est perdu. S'il s'agissait d'un paquet TCP, il existe quelque part un processus qui attend ce paquet et qui va redemander la transmission (de toute façon, TCP s'arrange en principe pour que les paquets soient assez petits pour passer). Si en revanche il s'agissait d'un paquet UDP, il est perdu pour de bon. La conclusion est que la probabilité de passer un datagramme diminue quand sa taille augmente.

Typage. Les données qui transitent dans un AppleEvent sont des données *structurées*. Un AppleEvent possède une *classe* (un mot 32 bits) à laquelle il appartient, et un *type* (un autre mot 32 bits) qui indique à l'application comment elle doit traiter l'évènement. Enfin l'AppleEvent peut contenir des données structurées : listes, tables, enregistrements, construits à partir d'autres types structurés ou de types de bases. Les types de bases sont extrêmement riches sur le Macintosh (c'est une des spécificité de son système d'exploitation). Un type de base peut évidemment être un entier ou une chaîne de caractère, mais aussi une application distante, un fichier, un son ou un dessin animé !

IP, en revanche, ne possède aucun mécanisme de typage. Avec IP, on transmet des flots d'octets d'un côté (des paquets d'octets dans le cas d'UDP), et on récupère de l'autre côté les mêmes octets. Il faudra donc que l'application gère elle même la structuration des données.

4.2 Le ShareBoard et son implémentation

Nous possédons déjà le ShareBoard écrit par Alain Karsenty (décrit à la section 2.2.2, page 7) pour le Macintosh (figure 9). Pour obtenir une application hétérogène, j'ai donc écrit la contrepartie Unix du ShareBoard. Elle se présente sous la forme de trois commandes : sb permet de lire un ShareBoard sur Macintosh depuis une station Unix, pastetext permet de coller du texte dans un ShareBoard Macintosh depuis une station Unix et pasteImage permet de coller une image de la même façon.

4.2.1 Le ShareBoard Macintosh

Le mode d'utilisation d'un micro-ordinateur individuel, est très différente de celui d'une station de travail : une station de travail connaît l'utilisateur car ce dernier a du donner son nom en se logguant. Un micro-ordinateur ne dispose pas de cette information. L'utilisateur voit un ordinateur individuel comme sa propriété privée. Il peut le mettre en marche et l'arrêter à volonté, créer et détruire des fichiers où et quand il le désire, etc. Notamment, il n'y a pas de notion de "home directory", ni d'endroit qui soit caractéristique d'un utilisateur, si ce n'est la machine complète elle-même.

Noms d'utilisateurs. Rappelons que le ShareBoard est "orienté utilisateur" dans le sens où on ne s'intéresse pas au ShareBoard d'adresse 129.175.15.120, port 12354, mais à celui où Untel a stocké les images que je lui ai demandé. Dans le cas du ShareBoard Macintosh, on connait le nom de la machine sur laquelle il faut aller le chercher (qui ne varie pas). Nous avons choisi d'assimiler un Macintosh à un utilisateur.

L'utilisateur est donc identifié dans le monde Unix par son nom de login, ce qui permet de stocker les fichiers qui le concerne dans un endroit facile à retrouver, alors qu'il est identifié dans le monde Macintosh directement par la machine. Il y donc exactement un ShareBoard Macintosh qui tourne sur chaque Macintosh.

Structure du ShareBoard Macintosh. Ce ShareBoard se présente sous la forme de deux programmes : ShareBoard lui-même qui est le programme permettant l'interrogation, c'est à dire de lire le fichier (un par machine) contenant les données et d'afficher chacune des pages. Ce programme offre également une interface qui permet de stocker des images dans un autre ShareBoard.

L'autre processus, nommé ScrapWatch est celui qui réalise effectivement l'insertion. Il est le seul processus qui manipule le fichier de donnée en écriture (ce qui assure la cohérence). Les requêtes sont sous forme d'AppleEvents et passent par la file des évènements de la machine distante, c'est à dire celle sur laquelle le document va être stocké. Il y a un exemplaire de ScrapWatch qui tourne en permanence sur chaque Macintosh doté d'un ShareBoard. ScrapWatch est donc l'équivalent Macintosh d'un démon Unix (figure 10).

4.2.2 Le ShareBoard Unix.

Le fonctionnement est extrêmement simple : pour obtenir le contenu d'un ShareBoard, on s'adresse au Macintosh concerné et on lui demande le fichier correspondant. Deux problèmes se posent donc : à qui parler sur ce Macintosh et à quel Macintosh parler.

Un correspondant sur le Macintosh. Le premier problème est simple à résoudre dans le principe : on va adjoindre à chaque processus ScrapWatch un processus frère appelé r. Lorsqu'une machine Unix veut s'adresser à ScrapWatch, elle envoie une requête à r sous la forme d'une chaîne de caractères. r lit cette requête, la décode, construit la requête correspondante sous forme d'AppleEvents et l'envoie à ScrapWatch. r peut également lire le fichier local et le transmettre, ce qui permet à la machine Unix de l'obtenir. Le fichier est transmit sur une connexion TCP, car il peut être volumineux.

Dans la pratique, des problèmes d'implémentation se posent : le système du Macintosh est "multi-tâche non préemptif", ce qui signifie qu'un processus comme r doit rendre la main régulièrement sous peine de bloquer toutes les autres applications de la machine (et notamment ScrapWatch). L'écoute du port UDP se fait par interruption. Quand r est dans le traitement d'une telle interruption, il ne peut envoyer d'évènement à ScrapWatch car ceci nécessite de manipuler la mémoire, ce qui est impossible en mode interruption sans prendre le risque de laisser la mémoire dans un état incohérent. r doit donc reprendre la main régulièrement et vérifier s'il n'a pas de requête (arrivée par UDP) à traiter. La communication entre les modes "interruption"

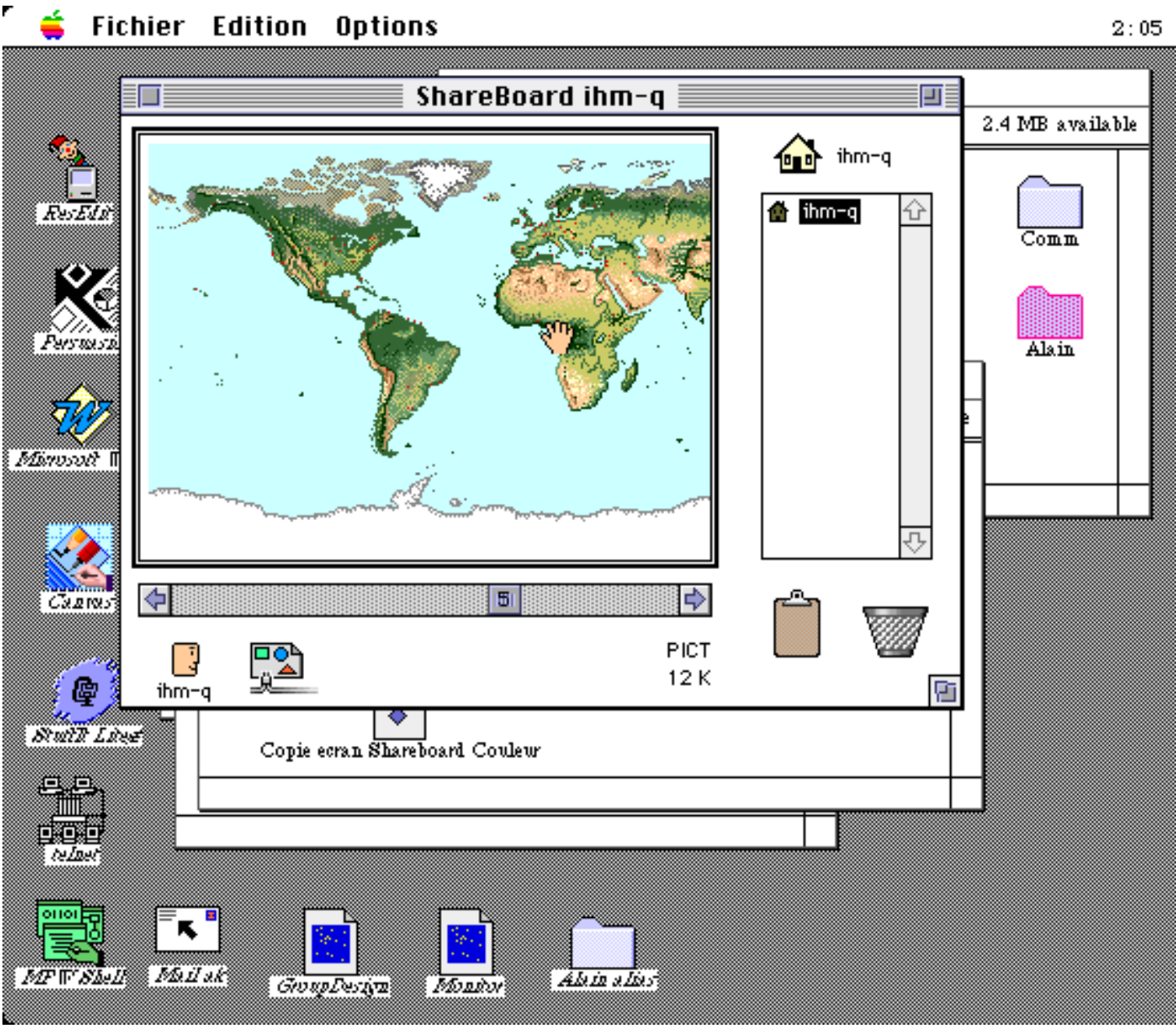

Figure 9: Le ShareBoard Macintosh

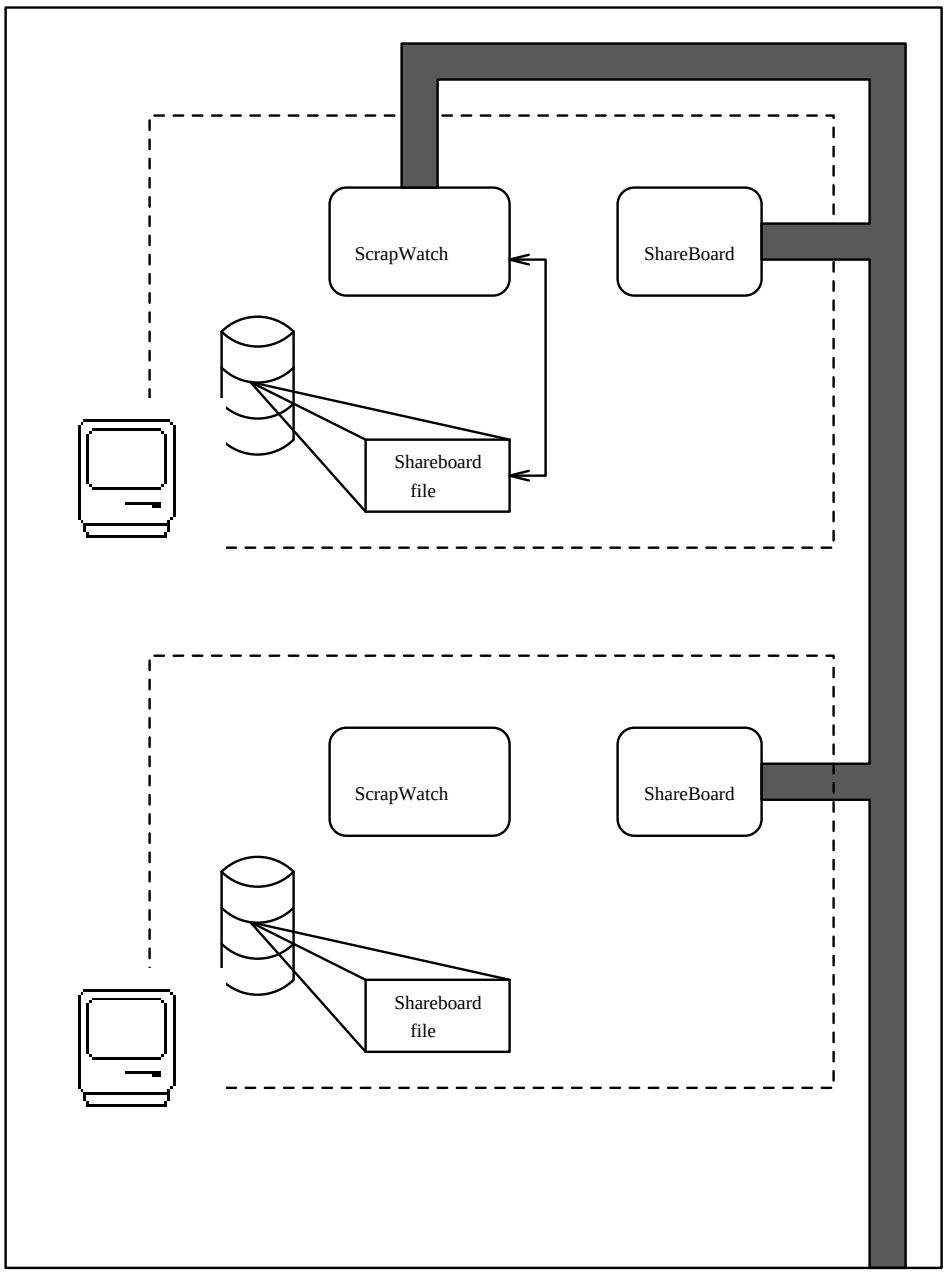

Figure 10: Architecture du ShareBoard Macintosh

et "non interruption" se fait par variable partagée. D'autre part, le fait de reprendre régulièrement la main consomme inutilement du temps de calcul (c'est de l'attente active). Ce problème, inhérent au système du Macintosh, n'épargne pas ScrapWatch.

Résoudre les adresses Etant donné que nous avons fait le choix d'avoir les Macintoshs comme machine "bilingue", il faut une adresse IP sur le Macintosh où envoyer des messages, c'est à dire une adresse physique, un domaine et un numéro de port.

Nous avons choisi UDP pour transmettre tous les messages pour plusieurs raisons. Tout d'abord, c'est le protocole qui s'approche le plus de la notion de "message", c'est à dire qu'on peut transmettre l'information sous forme de bloc, et sans qu'il soit nécessaire d'établir une connexion. Une autre raison est qu'UDP charge moins le réseau (dans TCP, des paquets sont régulièrement échangés entre les processus pour s'assurer que la connexion est toujours valide et pour réguler le flot).

En principe, l'adresse physique de la machine peut être choisie par l'utilisateur (à l'intérieur du domaine qui lui est alloué). En pratique, ce n'est pas possible sous MacTCP pour de sombres raisons techniques : la machine s'alloue toute seule une adresse la première fois que le driver IP est accédé.

Un dernier problème provient du fait que le système d'exploitation du Macintosh est assez "fragile" : il n'est pas rare, lorsque des utilisateurs développent une application par exemple, qu'il faille le rebooter. A chaque fois l'adresse physique est susceptible de changer.

En ce qui concerne le numéro de port, on peut (toujours en principe), le choisir. En pratique, certaines adresses sont réservées (par tradition) à des applications connues : un certain nombre de services IP (comme le courrier ou les connexions à distance), les serveurs X, etc. Imposer un numéro de port, c'est courrir le risque d'une collision avec une autre application. D'autre part, une fois qu'un port est refermé, il ne peut être utilisé pendant un certain temps ("Quiet Time"), qui est supérieur à la durée de vie des paquets dans le réseau ("Time To Live"). Ceci permet d'éviter qu'une application reçoive un paquet qui était destiné à une autre application qui écoutait précédemment sur le même port.

Conclusion : il faut laisser l'application s'allouer un numéro de port.

BATMAN. En résumé, le problème est de réussir à établir une correspondance entre des noms d'utilisateurs et/ou de Macintoshs et des adresses IP qui changent fréquemment et erratiquement, tout en réservant le moins d'adresses possible. Pour cela il suffit d'un processus qui joue le rôle d'un serveur de noms. Nous avons appelé ce processus BATMAN (pour BAsic AppleTalk MANager). BATMAN se trouve sur une machine connue de tous à une adresse connue de tous. C'est l'unique adresse réservée parmi toutes les applications. Lorsqu'un Macintosh est redémarré, r annonce à BATMAN son adresse et son numéro de port (qu'il a obtenu du système d'exploitation) ainsi que le nom de la machine locale. BATMAN stocke également le nom de l'application correspondante, ce qui lui permet d'être un serveur pour de multiples applications.

Lorsqu'une application (comme sb) veut communiquer avec une autre application, elle interroge BATMAN en lui passant le nom de l'application et le nom de l'utilisateur. BATMAN répond en communiquant l'adresse IP correspondante.

Format d'échange des données. Rappelons que le système du Macintosh possède une grande variété de types de données. Notamment, une image est un type particulier, qui fait partie du système et qui est en principe opaque aux utilisateurs et aux applications. Les uns comme les autres peuvent manipuler les images, les insérer dans d'autres documents, en sélectionner certaines portions, les modifier, etc. sans jamais rien connaître de leur format interne. Ce format est une est une description de l'image en Quickdraw, un langage graphique spécifique au Macintosh.

Le monde Unix ignore tout de Quickdraw ou des types de données du Macintosh. Il faut donc un programme sous Unix qui soit capable de convertir le format Quickdraw en un format exploitable, par exemple une image écran. Ceci s'inscrit dans la ligne générale des problèmes de conversion de format en milieu hétérogène. Le format interne de Quickdraw, comme c'est souvent le cas pour des formats internes, est très mal documenté. Travailler avec un tel format peut tourner à la "devinette". C'est aussi prendre le risque d'incompatibilités futures, puisque Apple garantit seulement la compatibilité de ses futurs produits avec le format Quickdraw de haut niveau, non la représentation interne.

pasteImage et pasteText. Pour coller des images ou du texte dans un ShareBoard, il suffit d'obtenir l'adresse du r associé auprès de BATMAN et d'adresser la requête correspondante à r, qui effectuera le collage (via ScrapWatch) par le processus déjà décrit (section 4.2.2, page 22).

4.2.3 Un ShareBoard pour stocker des documents sous Unix

sb, pasteText et pasteImage permettent d'interroger et de stocker des information dans un ShareBoard Macintosh depuis une station Unix. L'étape suivante est un programme qui permette de stocker les informations sur une machine Unix. Le programme correspondant s'appelle lui aussi ScrapWatch. On peut lancer ScrapWatch sur plusieurs machines Unix. La commande subscribe permet à un utilisateur de s'enregistrer auprès d'un ScrapWatch en indiquant dans quel fichier doit être stocké son ShareBoard.

Les données échangées utilisent le même format que le ShareBoard du Macintosh (y comprit les données graphiques), ce qui rend les échanges totalement transparents, mais est un peu lourd compte tenu de l'étroitesse du standard graphique du Macintosh. L'architecture globale est donc celle de la figure 11

Comme le montre l'exemple de la section 4.4, page 29, l'application est simplement nommée par un couple <nom d'application, nom d'utilisateur>. BATMAN effectue la conversion en adresse physique. Ainsi, non seulement l'utilisateur n'a pas besoin de savoir si le ShareBoard qu'il interroge se trouve sur une machine Unix ou sur un Macintosh, mais même le *programmeur* de l'application n'a pas a s'en soucier.

4.3 Le moteur réactif

L'expérience de l'écriture du ShareBoard Unix a permis d'identifier certains des problèmes à résoudre. Fondamentalement, il faut réussir à abstraire certaines structures de données et certaines opérations :

- Tous les éléments d'un collecticiel à architecture répartie se présentent sous la forme d'une entité communicante constituée d'une boucle qui attend des requêtes et les exécute.
- Présenter les requêtes sous la forme de chaînes de caractères est trop pauvre, il faut pouvoir les structurer (en champs par exemple).
- Il faut un mécanisme qui permette d'interroger automatiquement BATMAN.

Afin de tester facilement les idées correspondantes, j'ai développé une bibliothèque C++ qui implémente les abstractions correspondantes : le moteur réactif.

4.3.1 Mécanismes de communication

Les mécanismes de communications IP de Unix sont de trop bas niveaux pour être maniable. UCH [BL90b] constitue un premier pas en avant. C'est une bibliothèque de classes C++ qui s'efforce de s'affranchir des mécanismes de communication de bas niveau pour en proposer une version abstraite.

Le premier type d'objet du moteur réactif est le `datagramPlug`. Construit au-dessus de UCH, le `datagramPlug` est une "prise" sur laquelle on se branche pour lire et écrire des `datagrams`. Un `datagram` est simplement un objet qui, à l'instar d'un AppleEvent, possède un type, qui est une chaîne de caractères, et une série de champs, qui contiennent chacun une chaîne de caractères. Un `datagram` est donc un conteneur pour un nom de méthode et ses paramètres. Il est possible de donner une réponse à un `datagram` : il suffit d'y appliquer la méthode `reply` avec un `datagram` réponse et le `datagram` question fait automatiquement parvenir la réponse à l'émetteur de la question.

4.3.2 Réaction à des méthodes venues de l'extérieur

Quand on passe à une architecture hétérogène, il devient nécessaire d'avoir une caractérisation plus abstraite des objets afin de s'affranchir des caractéristiques particulières d'un système graphique ou d'un système d'exploitation.

Il est utile de voir une application comme un ensemble d'entités communicantes : les entités ne sont connues de l'extérieur que par les messages auxquels elles savent répondre; une telle entité est un objet réactif. L'abstraction la plus naturelle pour les entités dispersées à travers un réseau est celle d'acteur, dans

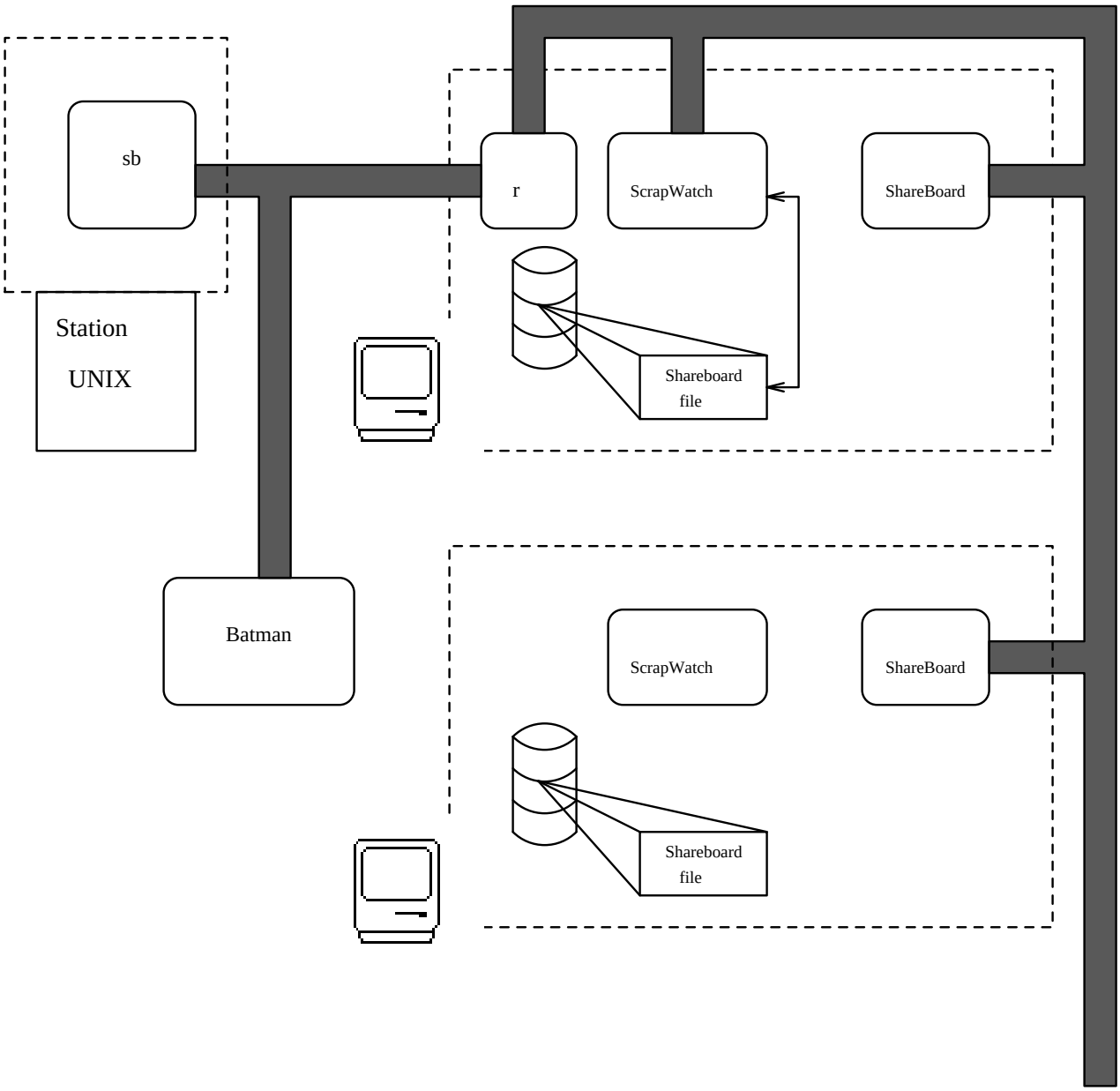

Figure 11: Architecture complète du ShareBoard

le sens de Hewitt. Un acteur est une entité qui reçoit des messages et qui y répond en envoyant d'autres messages. Cette abstraction englobe jusqu'à l'utilisateur lui-même. Pour Patterson [Pat91], l'utilisateur doit être vu comme un objet au sens de la programmation par objets. Une interface homme-machine n'est rien d'autre qu'un mécanisme sophistiqué pour passer des messages de et vers l'utilisateur. Patterson appelle un *rôle* un "type abstrait d'utilisateur".

Le deuxième type d'objet du moteur réactif est le `scheduler`. Le `scheduler` est la partie active d'un acteur. C'est un objet qui est lié à un `datagramPlug` (qui lui sert de port d'entré/sortie). Le `scheduler` attend un message sur son `datagramPlug`. Lorsqu'un message est envoyé au `scheduler` par un `scheduler` distant sous la forme d'un `datagram`, il exécute la méthode correspondante. De ce point de vue, il agit comme un *serveur*.

Chaque méthode est définie par son nom et sa sémantique (le corps de la méthode), qui est codée sous la forme d'une classe C++. Lorsqu'il reçoit une méthode, le `scheduler` retrouve le corps de la méthode correspondante dans son dictionnaire des méthodes et l'appelle en lui passant les arguments qu'il a trouvé dans le `datagram`. Si la méthode est une méthode avec résultat, le `scheduler` récupère les résultats, les encapsule dans un `datagram` et envoie la réponse au `scheduler` (distant) qui avait envoyé la méthode initiale.

Un `scheduler` possède un nom d'utilisateur et un nom d'application. C'est sous ce nom qu'il s'enregistre automatiquement à sa création auprès de BATMAN

Mettre au point des applications sous formes d'acteurs communiquant dispersés dans le réseau s'est révélé être particulièrement long et pénible. Pour faciliter la tâche, le `scheduler` possède un dictionnaire interne dans lequel la recherche des méthodes se poursuit lorsqu'elle a échoué dans le dictionnaire principal. Ce dictionnaire interne contient deux méthodes prédéfinies : `whatMethodsDoYouKnow` qui permet d'obtenir la liste des méthodes connues d'un `scheduler` et `reportOn:`, qui l'oblige à envoyer une trace de toute son activité sur le numéro de port passé en paramètre et d'où elle poura être envoyé à l'écran.

Cohabitation avec d'autres sources Dans le cadre du collecticiel, il est souvent nécessaire d'accepter des entrées venant de plusieurs sources. Par exemple, le moteur réactif est en attente sur son `datagramPlug` et une boucle est en attente d'évènements du serveur X (local). Ceci n'est pas possible directement avec le `scheduler`, puisque celui-ci est en attente sur une seule entrée, son `datagramPlug`. Pour pallier ceci, on utilise le `CHANNEL_SET` d'UCH. Un `CHANNEL_SET` est un ensemble de canaux (dérivant du type `CHANNEL` d'UCH) qui peut se mettre en attente sur une série de canaux. On peut donc demander au `scheduler` de faire une exécution non bloquante (c'est à dire de ne pas entrée dans sa boucle), et le stocker dans un `CHANNEL_SET`. Lorsqu'un message arrive, le moteur réactif est automatiquement réveillé par le `CHANNEL_SET`.

4.3.3 Acteurs distants

Le dernier type d'objet du moteur réactif est la `remoteApplication`. Une `remoteApplication` matérialise l'acteur distant. Elle possède comme le `scheduler` un nom d'utilisateur et un nom d'application. Lorsqu'elle est créée, elle interroge automatiquement BATMAN pour obtenir l'adresse du `scheduler` correspondant. On peut appliquer à une `remoteApplication` les méthodes `method`, qui permet d'envoyer un message à l'application distante, `methodWithReply` qui permet d'envoyer un message et de rester bloqué en attendant la réponse, implémentant ainsi une sémantique *d'appel de procédure à distance*. Enfin on peut appliquer la méthode `methodWithConnection`, qui initie automatiquement une connexion.

Les noms de méthodes et les paramètres sont échangés sous la forme de chaînes de caractères. Les champs ne sont pas typés. Toutes les valeurs sont représentées "en clair" : 12543 est représentée par la chaîne de cinq caractères "12543". Cette représentation à deux avantages. Le premier est que la dépendance vis-à-vis de la représentation interne des données par les machines est réduite, ce qui est évidemment particulièrement important dans le cas de réseaux hétérogènes. Par exemple pour un nombre, on ne dépend pas de l'ordre dans lequel la machine manipule les octets d'un mot. Cet avantage est cependant tout relatif, car le moteur réactif ne peut fonctionner que si toutes les machines utilisent l'ASCII, ce qui n'est pas une contrainte très forte. Le deuxième avantage concerne la mise au point : la simple impression des messages permet d'en identifier le contenu. Si les messages se présentaient sous une forme interne quelconque, ce serait évidemment plus difficile.

```
 1 #include "classes.h"
 2 #include "communicationManager.h"
 3 #include "reactiveEngine.h"

 4 char commandName[] = "ex1";

 5 class compute: public abstractStringWithReplyParameter {
 6   private:
 7     void processQuery(const char *const *inData, const char **outData);
 8 };

 9 void compute::processQuery(const char *const *inData, const char **outData)
10 {
11   int val;
12   static char buffer[MAX_COMMAND_SIZE];

13   sscanf(inData[0], "%d", &val);
14   sprintf(buffer,"%d", val*val);
15   outData[0] = buffer;
16 }

17 main()
18 {
19   datagramPlug      sock;
20   remoteApplication batman(&sock);
21   scheduler         s("square", "King Arthur", &sock, &batman);

22   s.addMethod("squareOf:%s is:%r", new compute);
23   s.run();
24 }
```

Figure 12: Le serveur

4.4 Un exemple simple

Voici un exemple très simple qui montre la facilité avec laquelle on peut écrire un serveur et un client avec le moteur réactif. Le serveur calcule le carré d'un entier qui lui envoyé par le client.

4.4.1 Le serveur

Le listing du serveur apparaît sur la figure 12.

Les lignes `1–3` incluent divers fichiers; la ligne `4` donne un nom utilisé en cas de sortie avec erreur.

Le corps de la méthode est déclarée aux lignes `5–16`. Les lignes `5–8` indiquent que la méthode hérite de `abstractStringWithReplyParameter`, qui correspond aux méthodes avec réponse.

Le corps proprement dit de la méthode se trouve aux lignes `9–16`. La méthode recevra les données en entrée dans la table `inData`, et doit mettre la réponse dans `outData`. A la ligne `13`, le paramètre (qui, rappelons-le, se présente sous la forme d'une chaîne) est converti en entier. Le résultat est calculé à la ligne `14` et renvoyé à la ligne `15`.

Les lignes `17–24` constituent le serveur proprement dit. A la ligne `19` est déclaré le `datagramPlug` qui est le point de communication avec l'extérieur. A la ligne `20`, le serveur de nom BATMAN est déclaré. Enfin, on déclare à la ligne `21` un `scheduler s` qui fait tourner l'application "square" de l'utilisateur "King Arthur". `s` utilisera `batman` comme serveur de nom et communiquera en utilisant le `datagramPlug sock`.

```
 1 #include "communicationManager.h"
 2 #include "message.h"
 3 #include "remoteObject.h"
 4 #include "tools.h"

 5 char commandName[] = "application caller";

 6 main(int argc, const char *const *argv)
 7 {
 8   datagramPlug      sock;
 9   remoteApplication batman(&sock);
10   remoteApplication a("square", "King Arthur", &sock, &batman);
11   remoteResponse    *r;
12   message m;

13   if (argc != 2) errorExit("usage : ex1 n");
14   m << "squareOf: " << argv[1] << " is:%r";
15   r = a.methodWithReply(m);
16   printf("%s\n", ((*r)[0]));
17   delete r;
18 }
```

Figure 13: Le client

Le lien entre la méthode de nom `squareOf:is:` et son corps est délaré à la ligne `22`. Les symboles `%s` et `%r` indiquent les emplacements de la question et de la réponse, respectivement, dans la chaîne constituant la méthode avec ses paramètres.

Enfin le serveur démarre à la ligne `23`.

4.4.2 Le client

Le listing du client apparaît sur la figure 13.

Les lignes `1`–`5` jouent le même rôle que dans le serveur, de même que les lignes `8`–`9`.

La ligne `10` déclare l'application distante de nom "square" de l'utilisateur "King Arthur" sous le nom local `a`.

La ligne `11` déclare un pointeur sur une `remoteResponse` qui servira à contenir la réponse. La ligne `12` déclare un `message` qui va contenir la question.

La ligne `13` vérifie qu'il y a bien un paramètre.

La ligne `14` construit la chaîne de caractère `squareOf:` *n* `is:` `%r`, qui est le message correspondant à la question.

La ligne `15` envoie le message à l'application distante `a`. `methodWithReply` indique qu'il y a une réponse, qui est stockée dans `r`.

La ligne `16` imprime le résultat sur la sortie standard.

Enfin la ligne `17` libère l'espace mémoire occupé par la réponse.

4.5 Edition de dessins structurés

Tout comme le ShareBoard a servi de support aux expérimentations dans le domaine du collecticiel asynchrone, l'éditeur de dessins structurés multi-utilisateurs GroupDesign va permettre d'expérimenter des solutions dans le domaine du collecticiel synchrone (figure 14).

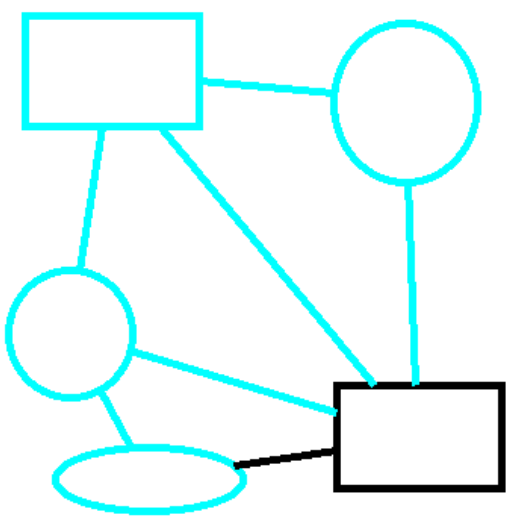

Figure 14: Dessin structuré sur Macintosh et sous UNIX.

GroupDesign est un programme à architecture purement répartie, comme indiqué à la section 2.2.2, page 9. Chaque exemplaire de l'application engagé dans une sesion possède la liste de toutes les machines engagées dans la même session. Chaque fois qu'un objet est créé ou modifié, l'application locale d'une part met à jour sa propre vue du dessin et d'autre part envoie à chaque machine de la session un message indiquant la modification effectuée.

Pour entrer dans une session, une commande permet d'afficher la liste de toutes les machines. L'utilisateur clique alors sur une machine déjà en session, appelons-la la machine source. Il clique ensuite sur l'application correspondant a GroupDesign pour entrer dans la même session. La machine source transmet alors à la machine locale la liste de toutes les applications. Enfin, la machine locale envoie à toutes les machines un message les informant de son entrée en session. Toute les machines ajoutent alors la machine locale dans leur liste des machines en session.

4.5.1 Un editeur de dessin structuré sous Unix

J'ai réalisé dans un premier temps gd, un éditeur de dessins structurés simplifié : il permet de tracer des rectangles et des ellipses, de les relier et de les modifier.

4.5.2 Passer à l'hétérogène

Connecter GroupDesign et gd nécessite tout d'abord de définir le protocole qui leur est commun.

On se souvient que sb communiquait avec le ShareBoard Macintosh au travers de r, et que r construit pour chaque message venu de sb, pasteText ou pasteImage l'AppleEvent correspondant. Ce mécanisme est très lourd : pour chaque méthode susceptible d'être envoyé par sb, pasteText ou pasteImage, il doit exister un code correspondant dans r. Toute modification de ces méthodes implique une modification dans r, ce qui ne peut pas se faire sur la même machine, r étant écrit et compilé sur le Macintosh.

CAT-WOMAN. Un pas supplémentaire est donc franchi en rendant le processus faisant le pont entre Unix et Macintosh capable de traduire les messages transitant entre les machines Unix et les Macintoshs sans qu'il soit nécessaire de lui décrire tous les messages. Ce processus, qui tourne sur le Macintosh, reçoit le nom de CAT-WOMAN (Converting AppleTalk WithOut Any Notice).

La seule chose que connaisse CAT-WOMAN est le protocole d'entrée en session : elle doit s'annoncer afin de recevoir les messages correspondant à la session du côté des Macintosh. Cette traduction est possible, car Système 7 fournit les outils nécessaire pour obtenir le nombre de champs d'un enregistrement, leurs noms et leurs types. Les types de données utilisés par GroupDesign se limitent à des nombres en virgule fixe pour la représentation des coordonnées et des entiers. Si des types plus spécialisés du Macintosh, comme les sons ou les dessins animés, avaient été utilisés, un transcodage aurait été nécessaire à ce niveau.

Un champ d'un AppleEvent est traduit en une portion de chaîne de caractères constituée par la concaténation du nom du champ, du marqueur "`#`", du type du champ, du séparateur "`:`", et enfin du contenu du champ. Rappelons que les nombres utilisé par le moteur réactif sont des chaînes de caractères. La conversion est effectuée par CAT-WOMAN. Le type entier (le plus courant), est considéré comme le type par défaut, ce qui permet de l'omettre. On obtient ainsi des datagrammes beaucoup plus courts. Prenons l'exemple d'un champ entier nommé `KXco` (ce champ représente généralement l'abcisse d'un objet dans GroupDesign) qui contient un nombre 32 bits de valeur 10. La conversion donne la chaîne `KXco: 10`. Si un champ porte le nom `name`, possède le type caractère (`kChr`) et contient la valeur '`a`', le résultat de la conversion sera la chaîne `name#kChr:a`.

CAT-WOMAN concatène la classe et le type de l'AppleEvent ainsi que toutes les chaînes issues des conversions des différents champs dans une chaîne de caractères qui est le message dans un format compréhensible par le moteur réactif.

Les messages utilisés par dgd (la version répartie de gd) sont donc les messages de GroupDesign tels que les formatte CAT-WOMAN. Quelques précautions supplémentaires sont nécessaires pour effectuer la transformation des coordonnées de GroupDesign (en virgule fixe), vers le système de coordonnées utilisé par gd (coordonnées entières de X).

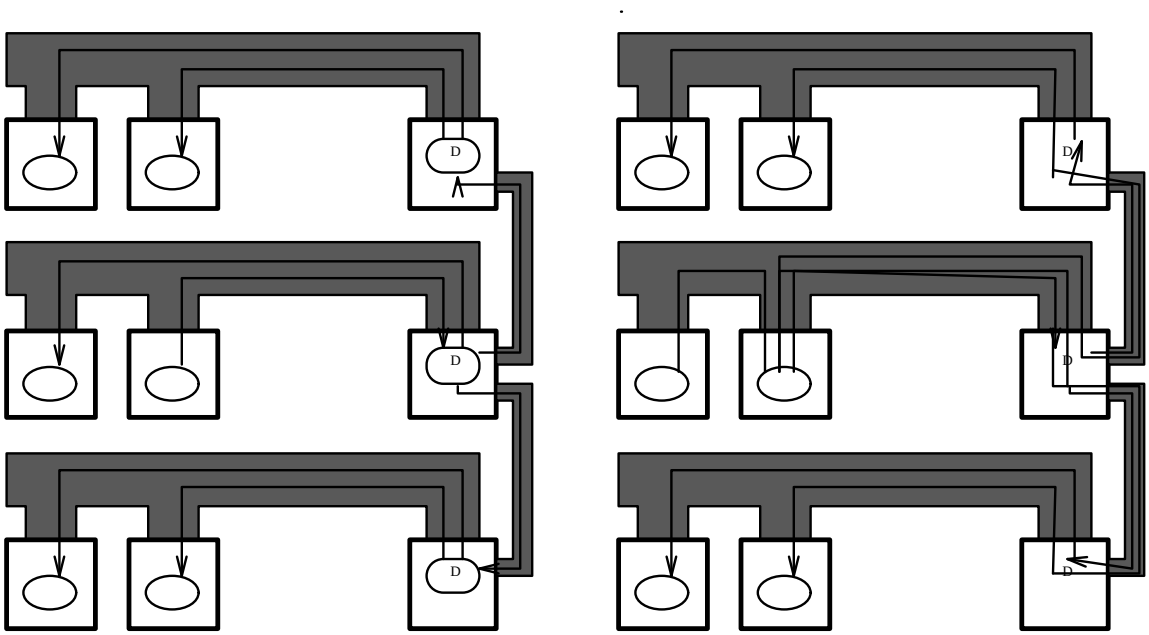

Figure 15: Avec et sans les `dispatchers`

4.5.3 Transmission

Les machine Unix sont situées sur des réseaux Ethernet, qui sont des réseaux à haut débit. AppleTalk est également un réseau rapide, bien que nous ne disposions pas de données techniques précises à son sujet.

Inadaptation de l'architecture purement répliquée. Il a déjè été souligné (section 3.3.2, page 19), qu'un gateway est toujours au mieux aussi rapide que le plus lent des réseaux qu'il connecte. Il faut donc limiter le trafic traversant ces jonctions. Obtenir une bonne performance des transmissions est pour cette raison plus difficile en environnement hétérogène. Ceci est particulièrement vrai pour des réseaux de grande dimension. Supposons qu'on ait onze exemplaires d'un programme à architecture purement répliquée : un à Orsay et dix à Toronto. Une machine d'Orsay envoie *dix* messages à travers l'Atlantique, ce qui est totalement inefficace. Ce qu'il faut réussir, c'est l'adaptation de la topologie des flux de messages à celle du réseau.

Dispatcher. Pour atteindre cet objectif, on introduit un nouvel acteur, le dispatcher. Le dispatcher n'est rien d'autre qu'un "relais logiciel" qui connait la topologie du réseau et la répartition du groupe sur cette topologie. Son fonctionnement consiste simplement à renvoyer les messages qu'il reçoit vers les autres relais et vers des applications (figure 15).

Ce modèle conserve l'avantage de l'architecture purement répliquée, qui est un temps de réponse identique à celui d'une application individuelle. Ce modèle est même plus efficace puisque chaque application n'a besoin d'envoyer qu'*un seul* message (vers le dispatcher), alors que dans une architecture purement répliquée, chaque exemplaire de l'application doit envoyer ses messages à tous les autres.

Son inconvénient est d'augmenter le nombre de liens qu'un message doit traverser pour aller d'une machine à une autre. Cependant, il ne doit pas être nécessaire de traverser un grand nombre de liens : un seul suffit pour passer d'un continent à un autre. Le ralentissement résultant de l'augmentation du nombre de liens reste donc dans des limites raisonnables.

C'est un de ces dispatcher qui distribue les messages en provenance de CAT-WOMAN vers chacun des exemplaires de dgd en session. A chaque session correspond un réseau de dispatcher.

```
dispatcher = new remoteApplication("dispatcher", argv[1], &sock1, &batman);
controlledScheduler s("xxx", "yyy", &sock1, dispatcher);
```

Figure 16: Un fragment de dgd

Structure d'une application utilisant les dispatchers. BATMAN n'a pas besoin de connaitre les dgd en session, seulement les dispatchers. On peut remarquer qu'un dispatcher se comporte face à un dgd comme une application standard se comporte face à BATMAN : l'application envoie un message lorsqu'elle est créée pour signifier sa naissance et son nom, et un autre lorsqu'elle disparait. Cette similitude est exploitée dans dgd, comme le montre le fragment de code de la figure 16. Comme on le voit, le dispatcher a été enregistré auprès de BATMAN, ce qui permet à dgd de le retrouver. `argv[1]`, le nom d'utilisateur pour BATMAN, et en fait ici le nom de la session. `s`, le `scheduler` de dgd, utilise dispatcher comme un "BATMAN". Les noms `xxx` et `yyy` indiquent que le nom d'application et le nom d'utilisateur ne sont pas relevant pour le dispatcher, puisqu'un nouveau réseau de dispatchers est créé pour chaque session. Le `controlledScheduler` indique simplement que le `datagramPlug` du `scheduler` n'est pas la seule source d'entrée (l'autre source est le serveur X local dans le cas de dgd).

Le fait de séparer le transport de l'interface simplifie la conception de cette dernière, puisque tout le code lié au transport se trouve dans le dispatcher. Ceci renforce la modularité de l'application.

Le futur. Actuellement, les dispatchers doivent être entièrement placés sur le réseau par un utilisateur avant le lancement de l'application. Dans une prochaine étape le dispatcher devra démarrer seul et surtout se placer seul au point du réseau où il minimisera le débit.

4.6 Aller plus loin : des entités autonomes

Dans la mesure où une application est vue comme un ensemble d'acteurs, il est possible d'introduire des programmes qui agissent sur le document.

L'idée n'est pas entièrement originale. Par exemple on trouve dans [DKR90] la réalisation d'un jeu de Black Box distribué. Certain des joueurs sont des systèmes experts. Les auteurs de [CJK+92] sont persuadés que les mondes virtuels seront peuplés aussi bien d'utilisateurs humains que de composants ayant un comportement autonome, qu'ils appellent des *simulations*.

Si on prend l'exemple d'une salle de conférence informatique, l'ordinateur peut-être utilisé en tant que média de l'information, pour projeter par exemple l'image de chaque participant sur l'écran de tous les autres, ou permettre à chaque participant de montrer aux autres des images, des animations ou tout autre élément de la panoplie du multimédia. Il peut fournir des outils permettant aux participants de voter. Il peut servir comme un agent passif, par exemple gérer un agenda de tous les participants en s'assurant que les différents rendez-vous n'entrent pas en conflits. Il peut être un agent actif, et régler d'autorité les conflits de calendrier. Et on peut même envisager qu'il soit lui-même un agent du débat, par exemple un système expert peut fournir des informations au fur et à mesure de l'avancée de la discussion.

4.6.1 bugGame

L'acteur bugGame est une première illustration de ce principe. Placé à l'écoute d'une session GroupDesign/dgd, il enregistre toutes les créations d'objets afin de posséder leurs coordonnées, puis il déplace aléatoirement un objet à chaque seconde. Un programme totalement inutile, mais parfaitement crispant pour ceux qui essaient de travailler !

4.6.2 Simulateur de réseau de Pétri

Tout comme bugGame, cet acteur écoute une session GroupDesign/dgd afin de connaître l'état du graphe qu'il considère comme un réseau de Pétri. Il peut alors simuler ce réseau de Pétri en intervenant lui-même

dans le dessin.

5 Comment écrire facilement une application collecticielle hétérogène

Dans sa thèse d'habilitation, Michel Beaudouin-Lafon écrit :

> L'aspect artisanal de la construction d'interfaces se traduit à tous les niveaux : conception à l'aveuglette, développement sans outils appropriés, tests réduits au strict minimum, maintenance hasardeuse. [BL92]

Comme le remarque par exemple [DC91b], pour le collecticiel, la situation est encore pire. Pour le collecticiel hétérogène, elle est franchement catastrophique. Malgré plusieurs essais dans ce sens ([DC91b], [LS83]), nous ne disposons même pas d'un *langage de programmation* largement reconnu et accepté pour écrire des applications collecticielles. Une des raisons importantes est que les langages de programmation dont les programmeurs ont l'habitude (comme C, qui sert généralement à programmer pour X, via la Xlib), sont presque totalement inadaptés à l'écriture de programmes distribués. Même des langages qui permettent d'exprimer la concurrence (comme Ada), permettent surtout de créer des tâches qui s'exécutent "localement" (c'est à dire que l'extension maximum du réseau est celle du réseau de communication d'un multiprocesseur).

Quand on traite les éléments répliqués d'une application comme des entités communicantes, on est obligé de réécrire une interface pour chaque sous-systèmes d'interfaçage utilisateur différent.

Une idée au départ était qu'à l'aide du moteur réactif et de quelques ingrédients, on devrait pouvoir facilement prendre une application "individuelle", comme un éditeur de dessin structuré, et en faire une application collecticielle. En fait, même muni de ces outils, il faut encore écrire beaucoup de code répétitif pour obtenir une application, ce qui est particulièrement pénible.

Il serait souhaitable qu'une application collecticielle se présente sous la forme d'un seul programme, et non de cinquante mille programmes qui communiquent (en fait, ou voudrait *ignorer* le protocole de communication).

5.1 Une petite expérience de programme réparti en environnement hétérogène

Les étapes de la conception et de la programmation d'une application collecticielle sont actuellement les suivantes :

1 Prendre deux applications permettant de faire un travail commun (par exemple, un simulateur de réseau de Pétri et un éditeur de dessin structuré) dans deux environnement différents.

2 Définir un protocole de communication.

3 Ecrire le boîtier logiciel qui réalise l'interface de l'application avec le protocole sous un environnement.

4 Courrir vers une autre machine (avec un autre environnement).

5 Recommencer au point 3 avec la nouvelle machine et le nouvel environnement.

6 Et ainsi de suite pour chaque machine et chaque environnement...

Il serait beaucoup plus simple pour le programmeur d'écrire tous son code sur la même machine, sans se préoccuper de la machine sur laquelle les différentes parties de l'application vont tourner. La répartition de la compilation entre les différents types de machine devrait être fait de faççon automatique.

Le point important est de voir que ceci est *possible*. Voici une petite expérience qui le met en évidence : nous avons écrit un ensemble de shell scripts qui permet d'effectuer des opérations sur les machines du LRI au travers du réseau. Supposons par exemple que l'on cherche la capacité totale de stockage disque du labo, avec le détail machine par machine. Si l'on est sur une machine donnée, la liste des disques et leur capacite peut s'exprimer ainsi :

```
hp20 % df | awk '/^\/dev\// { print $2"\t"$6 }'
1045329 /
1276742 /sources
```

Si on veut la réponse pour toute les machines, on entre ceci :

```
hp20 % mkdir tmp
hp20 % mkdir res
hp20 % cat > tmp/prog1
df | awk '/^\/dev\// { print $2"\t"$6 }'
hp20 % chmod 744 code
hp20 % everyMachine $PWD/tmp prog1 $PWD/res/espace_disque < activeMachineList.data
```

`everyMachine` est un itérateur qui applique la commande (ici `prog1`) sur toutes les machines de la liste (`activeMachineList.data`). `$PWD/tmp` est le *contexte* de la commande : tous les fichiers qui s'y trouvent seront "transportés" avec lui. `$PWD/res/espace_disque` contiendra les résultats :

```
espace_disque.sun1 espace_disque.sun2
hp20 % more *
::::::::::::::
espace_disque.sun1
::::::::::::::
57131 /
288447 /usr
9667 /tmp
37315 /mail
264031 /users/goinfre2
253155 /local.sun1
124255 /spare/125MO
133473 /usr.sun3.old
69023 /prodsun
::::::::::::::
espace_disque.sun2
::::::::::::::
9207 /
147907 /usr
19339 /tmp
74175 /var
106895 /export/root
220247 /export/swap
372393 /local.sun2
```

Certains fragments du script, mais également (possiblement) certains fragments de la commande sont écrits en C. Ils est impossible de transmettre ces fichiers tels qu'ils sont et d'espérer les voir s'exécuter, car les différentes machines ont des processeurs différents, des bibliothèques différentes, des versions différentes du système d'exploitation (bien qu'il s'agissent dans tous les cas de Unix), etc.

Voici comment procède le script : pour chaque machine de la liste, le contexte et les différents fichiers du script sont copiés sur la machine distante dans un fichier temporaire. Le script *recompile* ensuite les fragments en C (ou dans un autre langage) sur la machine distante et y lance l'exécution de la commande. Le processus est ensuite détaché, ce qui permet à l'itérateur de continuer avec la machine suivante, parallèlement à l'exécution de la commande. Une portion du script exécuté sur la machine distante prend en charge la recopie des résultats sur la machine de départ et la destruction finale des fichiers temporaires (les commandes s'exécutent donc en parallèle sur les différentes machines).

Il est important de comprendre que tout ceci est possible car il est possible de disposer d'un espace de travail et d'exécuter un certain nombre de commandes *à distance*. Identifier quelles sont les commandes indispensables fait partie du travail qui reste à faire. Il y a néanmoins quelques évidences :

Le problème de l'écriture des applications collecticielles hétérogènes est de réussir à transmettre la *sémantique* des méthodes. Dans le cas d'une application homogène, il suffit de transmettre un exécutable, ce qui est évidemment impossible dans le cas du collecticiel hétérogène. Actuellement, on ne sait pas exprimer la sémantique de programme "réels" autrement que par la donnée du listing de leur code (ce que d'aucun pourrait considérer comme une méthode "pauvre"). Il est évidemment possible de transmettre le listing d'un programme comme n'importe quel type de texte. Mais le problème est que la machine distante réussisse à l'exécuter. Dans le cas d'un langage interprété (comme avec les scripts), il suffit de disposer d'un interpréteur sur la machine distante. Dans le cas d'un langage compilé, il faut être capable de recompiler et d'exécuter à distance le code. Quelques précautions supplémentaires seront nécessaires. Par exemple, il faut être certain que les mots mémoire ont la même dimension, sont écrits dans le même sens, etc. sur toutes les machines, ou du moins s'assurer que ceci ne change pas la sémantique (ce qui dans le cas d'un langage comme C semble a priori impossible).

5.2 Quelques solutions possibles

5.2.1 Améliorer le moteur réactif

Il est évidemment possible d'améliorer le moteur réactif, notamment selon deux directions :

- Typer les messages.
- Transmettre la sémantique.

5.2.2 Un langage avec une architecture dataflow

L'abstraction que nous avons utilisé est celle d'acteurs s'échangeant des messages. Les transformations peuvent être relativement simples, par exemple le dispatcher ne fait que retransmettre les messages qu'il reçoit. Il peut être tentant d'utiliser des langages *flots de données*, dont il existe plusieurs types. Lustre et Signal sont de tels langages. Malheureusement, leur modèle utilise l'hypothèse du *synchronisme*, qui n'est pas réaliste dans notre contexte. Ce synchronisme la n'est pas ce que nous avons informellement définit comme le synchronisme. Pour nous, l'intervalle de temps qui séparent deux évènements "synchrones" peut être... arbitrairement grand si on le fait passer par un réseau à temps de transmission non borné comme Ethernet.

Le plus répandu des langages flots de données est sans doute VHDL [ABOR90], un langage de description de systèmes matériel+logiciel. Il est possible d'écrire :

```
entity actor_tee is
port(in1 : in string;
     out1 : out string;
     out2 : out string);
end actor_tee;

architecture a1 of actor_tee is
out1 <= in1;
out2 <= in1;
end a1;
```

Ceci est une façon de déclarer une entité VHDL, que nous pourrions aussi bien appeler un acteur. Cet acteur est un T qui duplique son entrée sur ses deux sorties. La flèche <= indique que chaque fois qu'un des signaux impliqué dans l'expression à droite (ici de type "string") de la flèche change, la nouvelle valeur devient celle du signal à gauche de la flèche.

Les flots entre les entités s'expriment en les connectant par des *signaux* (figure 17). Un tel langage permet donc d'exprimer facilement des ensembles coopérants d'acteurs communiquant, ce qui faciliterait grandement l'écriture d'applications collecticielles.

```
entity actor_three is
port (tin1 : in string;
      tout1 : out string;
      tout2 : out string;
      tout3 : out string);
end actor_three;

architecture a1 of actor_three

signal internal : string;
signal s1 : string;
signal s2 : string;
signal s3 : string;
signal s4 : string;

c1: actor_three port map (in1 => s1,
  out1 => internal,
  out2  => s2);

c2: actor_three port map (in1 => internal,
  out1 => s3,
  out2 => s4);

end actor_three;
```

Figure 17: Connection d'entité en VHDL

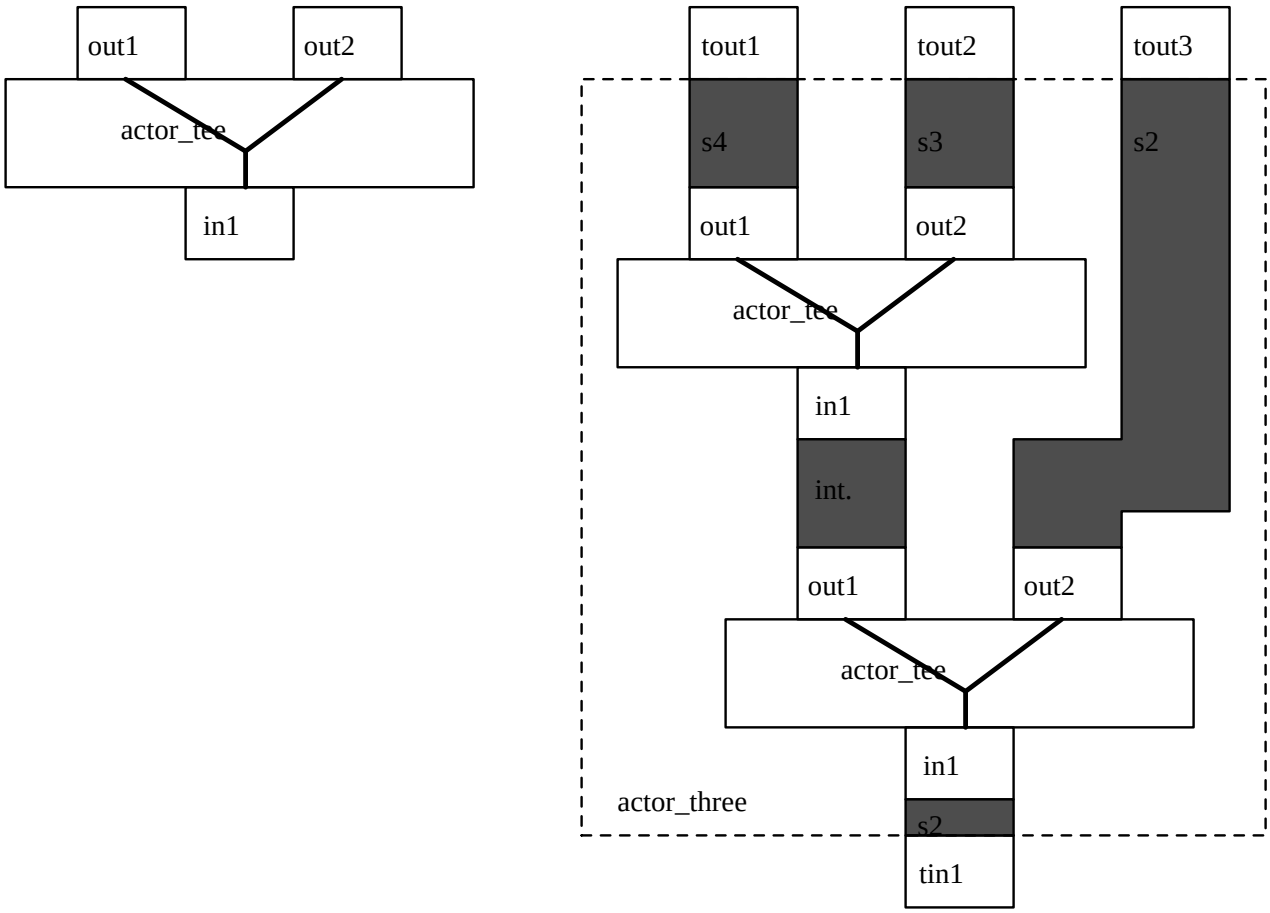

Figure 18: `actor_tee` et `actor_three`

5.2.3 Ada9x

Bref historique. Ada a été créé à la suite des efforts du Department Of Defence (Dod) américain qui souhaitait un langage puissant pouvant couvrir tous ses besoins, supportant les concepts de modularité et d'abstraction. Ces effort aboutissent à la définition d'un langage en 1983. Ada a depuis fait la preuve de sa valeur en tant que langage de programmation.

En Janvier 1988, le DoD entreprenait la révision du langage afin d'y incorporer les concepts les plus modernes des langages de programmation apparus dans l'intervalle, notamment les concepts de classes et d'héritage. Le travail de révision, étendu sur plusieurs années, a engagé des dizaines de personnes sur toute la surface de la planète, résultant dans l'effort de conception le plus important jamais entrepris en matière de langage de programmation. Le langage résultant porte actuellement le nom de Ada9x, signifiant que le langage devra apparaître dans les années 90.

Ada9x et les applications distribuées. Ada9x intègre de nombreuses fonctionnalités orientées vers les applications distribuées. Le chapitre 8 de [Ada90] ("Requirements for distributed processing") décrit les possibilités que devront supporter les implémentations qui prétendent se conformer à la partie du standard concernant les systèmes distribués. Les spécifications correspondantes du langage sont décrites dans le chapitre 8 de [Ada92a] : "Distributed systems". Le concept central est celui de *partition*.

Un programme est formée d'une ou plusieurs partitions, chaque partition étant susceptible de s'exécuter sur une machine différente. L'interface entre les modules distants se fait à travers des packages. Lorsqu'un package est situé dans une partition distante, le compilateur génère automatiquement des appels vers un module de communication, que l'utilisateur peut contrôler au travers d'un module UPCS (User Provided Communication Subsystem) et qui implémente une sémantique d'appel de procédure à distance "au plus un", c'est à dire que chaque message arrive au plus une fois (contrairement au moteur réactif, dans lequel l'application doit gérer les messages perdus et dupliqués). Il est donc particulièrement facile d'écrire des applications réparties en Ada9x. Les paramètres passés par appels distants sont typés, contrairement à ceux du moteur réactif (pour le moment). Enfin, des exceptions peuvent être propagées depuis des procédures distantes.

Dans le cas de Ada-9x, le standard ne précise pas comment les partitions sont placées sur les machines distantes, ni comment la consistence est assurée entre les différentes partitions. Cependant un outil *devra* permettre de placer les partitions. On peut imaginer de transmettre un arbre d'analyse syntaxique décoré vers la machine distante où un simple démon peut prendre en charge la phase finale de la compilation, l'édition de lien et le chargement.

Gestion du temps. La gestion du temps est cruciale dans une interface homme-machine. La plupart des langages sont dépourvues de mécanismes permettant d'expliciter ou de gérer des contraintes temporelles. Jusqu'ici les programmeurs ont adoptés une démarche pragmatique et ne se sont pas préoccupés des problèmes temporels en espérant que "ça irait assez vite". Cette attitude ne sera sans doute plus suffisante dans le cas de collecticiels hétérogène à grande échelle géographique, mettant en présence un grand nombre d'utilisateurs, ou s'exécutant sur des réseaux ultra-rapides.

Ada contient déjà des outils qui permettent de gérer le temps, comme les attentes de tâche avec chien de garde. Ada9x comportera des mécanismes supplémentaires ([Ada90], chapitre 5 : "Real time requirements" et [Ada92a], chapitre 5 :"Real-time systems"), notamment la détection des échéances manquées.

Conclusion sur Ada9x. Le principal défaut d'Ada9x est de n'être pour le moment qu'un langage papier. New York University et Free Software Foundation sont sous contrat depuis Juin 1992 pour la réalisation du premier compilateur pour le langage noyau, qui devrait être livré en Août 93. Le standard n'est pas encore fixé, hors du langage noyau. Ainsi, la dernière version de l'annexe I, concernant les systèmes distribués, est disponible depuis la mi-Juillet 1992 [Ada92b].

Malgré ces défauts, Ada-9x est actuellement le seul langage "réaliste" (i.e. qui ait des chances de s'implanter industriellement), qui permette de répartir un programme sur un réseau d'ordinateurs faiblement couplés, c'est à dire sans mémoire commune, et muni de liens de communication lents). Ses qualités en feront probablement un langage de choix pour l'écriture d'interfaces homme-machine collecticielles.

6 conclusion

Si l'écriture de collecticiel est possible, elle reste très difficile. L'étude du collecticiel hétérogène a néanmoins permis de mettre en lumière un certain nombre de points :

Les acteurs se sont révélés une abstraction très utile. Si l'on dispose d'un outil de base comme le moteur réactif, il est possible de résoudre les problèmes au fur et à mesure en écrivant de nouvaux acteurs. Ainsi BATMAN résoud le problème d'abstraction des noms, tandis que le dispatcher résoud les problèmes de transport.

Il est indispensable de découpler les problèmes de transport des interfaces, ce qui n'est possible ni pour une architecture purement centralisée, ni pour une architecture purement répliquée. En revanche, c'est possible en utilisant un réseau de relais logiciels construit en utilisant les dispatchers. Dans l'idéal, le placement des relais sur la topologie du réseau devrait être automatique.

Il est pour ainsi dire indispensable de disposer de langage de programmation qui permette de programmer facilement une application distribuée. Cette application doit être vue préférentiellement comme un seul programme, et non comme une collection de programmes répartis. ada9x semble le seul candidat qui ait à la fois les qualités requises en terme de fonctionnalité, et un potentiel de développement académique et industriel suffisant pour que de véritables applications soient construites.

Le futur L'existence des problèmes d'adaptation de la topologie des `dispatchers` à celle du réseau est lié à la lenteur des réseaux de grande extension actuels. Si ces réseaux devenaient beaucoup plus rapides (comme Renater), le problème serait radicalement transformé.

Pour connaître les potentialités de tels réseaux, il faut savoir quels types de services ils peuvent délivrer. Par exemple, un réseau qui est capable de fournir 130 Mbits/s mais qui nécessite 10 s pour initier une transmission sera un réseau dificile à exploiter pour faire du collecticiel temps réel. L'autre question est de savoir jusqu'à *quelle distance* ces réseaux sont rapides.

Les hypothèses hypothèses habituelles sont [LS83] le temps de transmission d'un message est significativement plus long que le temps de lecture d'une variable mémoire. C'est également l'hypothèse que nous avons faite, mais elle pourrait ne plus être valable au moment de l'apparition des réseaux ultra rapides (de type ATM).

Néanmoins, un réseau rapide ne résoudra évident pas tous les problèmes, et beaucoup de travail reste à faire. Aucun réseau informatique "rapide" ne l'est encore suffisamment pour passer de la vidéo ou du son, pourtant indispensables au support du travail collaboratif...

Liste des Figures

1 Architectures d'interface . . . 4
2 Information interne et information abstraite . . . 5
3 Wscrawl . . . 6
4 mazewar . . . 8
5 Taxonomie du collecticiel . . . 10
6 Le modèle de X . . . 12
7 Deux window managers . . . 19
8 Collision de noms lors d'une connection de réseaux . . . 20
9 Le ShareBoard Macintosh . . . 23
10 Architecture du ShareBoard Macintosh . . . 24
11 Architecture complète du ShareBoard . . . 27
12 Le serveur . . . 29
13 Le client . . . 30
14 Dessin structuré sur Macintosh et sous UNIX. . . . 31
15 Avec et sans les `dispatchers` . . . 33
16 Un fragment de dgd . . . 34
17 Connection d'entité en VHDL . . . 38
18 `actor_tee` et `actor_three` . . . 39

Bibliographie

[ABOR90] Roland Airiau, Jean-Michel Berge, Vincent Olive, and Jacques Rouillard. *VHDL, du langage à la modélisation*. Presses Polytechniques Romandes, 1990.

[ACM89] ACM SIGCHI. *Hypertext'89*, November 1989. Proc. of Hypertext'89, November 5–8, Pittsburgh, Pennsylvania.

[ACM91] ACM SIGGRAPH and SIGCHI. *Proceedings of the fourth Symposium on User Interface Software and Technology*. ACM Press, November 1991.

[Ada90] Ada Joint Program Office. Ada 9x *Requirements*, December 1990.

[Ada92a] Ada Joint Program Office. Ada 9x *Mapping, Volume II, Mapping Specification and Rationale (Annexes), Abridged*, March 1992. version 4.1.

[Ada92b] Ada Joint Program Office. *ds_annex_Jul92.doc,* Ada 9x *Mapping, Volume II, Annex I*, July 1992.

[Apple C91] Apple Computer, Inc. *Inside* Macintosh*, Volume VI*, April 1991.

[BBL92] Penny Bauersfeld, John Bennett, and Gene Lynch, editors. *CHI'92 Conference Proceedings*. ACM SIGCHI and SIGGRAPH, May 1992. Proc. of the ACM Conference on Human Factors in Computing Systems, May 3–7, 1992, Monterey, California.

[BL90a] Michel Beaudouin-Lafon. Collaborative development of software. In Gibbs and Verrijn-Stuart [GVS90], pages 103–114.

[BL90b] Michel Beaudouin-Lafon. The Unix Channel – C++ communication library. technical report, LRI, May 1990. version 2.0.

[BL92] Michel Beaudouin-Lafon. Construction d'interfaces et nouvelles dimensions de l'interaction homme-machine. Internal Report 768, LRI, July 1992. présentation des travaux en vue de l'Habilitation à diriger des recherches.

[BLK92] Michel Beaudouin-Lafon and Alain Karsenty. Transparency and awareness in a real-time groupware system. In *Proc. ACM Symposium on User Interface Software and Technology (UIST)*, November 1992.

[Bux91] William Buxton. Telepresence: Integrating shared task and personnal spaces. In Hendriks [Hen91], pages 27–36.

[CBY89] Timothy Catlin, Paulette Bush, and Nicole Yankelovich. Internote: Extending a hypermedia framework to support annotative collaboration. In *Hypertext'89* [ACM89], pages 365–378. Proc. of Hypertext'89, November 5–8, Pittsburgh, Pennsylvania.

[CJK+92] Christopher Codella, Reza Jalili, Lawrence Koved, Bryan J. Lewis, Daniel T. Ling, James S. Lipscomb, David A. Rabenhorst, Chu P. Wang, Alan Norton, and Greg Turk. Interactive simulation in a multi-person virtual world. In Bauersfeld et al. [BBL92], pages 329–334.

[DC91a] Prasun Dewan and Rajiv Choudary. Flexible user interface coupling in a collaborative system. In Robertson [Rob91], pages 41–48.

[DC91b] Prasun Dewan and Rajiv Choudary. Primitives for programming multi-user interfaces. In *Proceedings of the fourth Symposium on User Interface Software and Technology* [ACM91], pages 69–86.

[DKR90] C. De Koven and T. Radhakrishnan. An experiment in distributed group problem solving. In Gibbs and Verrijn-Stuart [GVS90], pages 61–76.

[EGR91] Clarence A. Ellis, Simon J. Gibbs, and Gail L. Rein. Groupware: some issues and experiences. *Communications of the ACM*, 34(1):38–58, January 1991. Si vous lisez un seul article sur le groupware, c'est celui-la.

[Flo91] Gert H. Florijn. Groupware research at serc. In Hendriks [Hen91], pages 77–87.

[Gib89] Simon J. Gibbs. Liza: An extensible groupware toolkit. In *CHI'89 proceedings*, pages 29–35. ACM, May 1989. Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems, (Austin, Texas, April 30 – May 4).

[Gru91] Jonathan Grudin. Groupware and CSCW: Why now ? In Hendriks [Hen91], pages 3–14. Extension d'un papier paru dans [Rob91]. Le point sur la situation du groupware en 1991 et la logique de son développement depuis les débuts de l'informatique.

[GS87] Irene Greif and Sunil Sarin. Data sharing in group work. *ACM Transactions on Office Information Systems*, 5(2):187–211, April 1987.

[GVS90] Simon J. Gibbs and A.A. Verrijn-Stuart, editors. *Multi-User Interfaces and Applications*. IFIP WG 8.4, North-Holland, September 1990. Proc. of the IFIP WG 8.4 Conference on Multi-User Interfaces and Applications, Heraklion, Crete, Greece, 24–26 September, 1990.

[Hen91] P.R.H. Hendriks, editor. *The potential of team and organizational computing*. Software Engineering Research Centre (SERC), SERC, 1991. Proceedings of Groupware 1991, Utrecht, The Netherlands.

[KK88] Kenneth L. Kraemer and John Leslie King. Computer-based systems for cooperative work and group decision making. *ACM Computing Surveys*, 20(2):115–145, June 1988.

[LA90] J. Chris Lauwers and Lantz Keith A. Collaboration awareness in support of collaboration transparency: Requirements for the next generation of shared window systems. In J. Carasco Chew and J. Whiteside, editors, *Human Factor in Computing Systems*, pages 303–311. ACM, April 1990.

[LFK88] Mary D. P. Leland, Robert S. Fish, and Robert E. Kraut. Collaborative document production using Quilt. pages 206–215, 1988.

[LKL91] Bryan J. Lewis, Lawrence Koved, and Daniel T. Ling. Dialogue structures for virtual worlds. In Robertson [Rob91], pages 131–136.

[LS83] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed, programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[MAH90] Risto Mäntylä, Jary Alasuvanto, and Heikki Hämmäinen. PAGES: A testbed for groupware applications. In Gibbs and Verrijn-Stuart [GVS90], pages 37–47.

[MR92] Brad. A. Myers and Mary Beth Rosson. Survey on user interface programming. In Bauersfeld et al. [BBL92], pages 195–202.

[Pat91] John F. Patterson. Comparing the programming demands of single-user and multi-user applications. In *Proceedings of the fourth Symposium on User Interface Software and Technology* [ACM91], pages 87–94.

[Rob91] Scott P. Robertson, editor. *Reaching through technologies*. ACM, May 1991. Proc. of the ACM conference on human factors in computing systems, New Orleans, Louisiana, April 27–May 2, 1991.

[Ros90] David S. H. Rosenthal. X11: Surviving the success disaster. presented at the European X Window System Conference, London, November 1990.

[San91] Michael Santifaller. *TCP/IP and NFS^{TM}, Internetworking in a UNIX^{TM} Environment.* Addison-Wesley Publishing Company, 1991. Translated from the German edition *TCP/IP und NFS in Theorie und Praxis: UNIX in lokalen Netzen* by Stephen S.Wilson.

[SCG89] Frank M. III Shipman, R. Jesse Chaney, and G. Anthony Gorry. Distributed hypertext for collaborative research: The virtual notebook system. In *Hypertext'89* [ACM89], pages 129–135. Proc. of Hypertext'89, November 5–8, Pittsburgh, Pennsylvania.

[SG85] Sunil Sarin and Irene Greif. Computer-based real-time conferencing systems. *IEEE Computer*, 18(10):33–45, October 1985.

[SLGS92] Chris Shaw, Jiandong Liang, Mark Green, and Yunqui Sun. The decoupled simulation model for virtual reality. In Bauersfeld et al. [BBL92], pages 321–328.

[TM91] John C. Tang and Scott L. Minneman. Videowhiteboard : Video shadows to support remote collaboration. In Robertson [Rob91], pages 315–322.