Troisième Partie

Mise en œuvre : science et technologie informatiques

Modèles d'architecture

Si j'avais appris la technique, je serais technicien. Je fabriquerais des objets compliqués. Des objets très compliqués, de plus en plus compliqués, cela simplifierait l'existence.

Eugène Ionesco

Modèles d'architecture

6.1. Introduction

Les disciplines dont nous avons décrits les apports dans les chapitres de la première partie contribuent à l'analyse des besoins et à l'établissement des spécifications du système à réaliser. Lorsque vient le moment de construire effectivement le système à partir des spécifications, d'autres guides centrés sur le génie logiciel deviennent nécessaires. La complexité croissante des systèmes informatiques et la diversité des techniques logicielles ont mis en évidence depuis quelques années l'utilité de l'architecture logicielle pour la conception et la réalisation des systèmes. Pour répondre à l'aphorisme de Ionesco, il est vrai que nous créons aujourd'hui des objets informatiques de plus en plus compliqués afin de simplifier l'existence des utilisateurs. Mais nous visons, principalement grâce aux modèles d'architecture logicielle et aux outils, à simplifier aussi l'existence des concepteurs et réalisateurs de ces objets informatiques complexes.

Dans ce chapitre, nous nous intéressons d'abord aux modèles d'architecture logicielle en général et identifions les rôles qu'ils doivent remplir. Nous précisons ensuite les dimensions de l'espace problème des systèmes multi-utilisateurs en termes d'architecture logicielle. Quelles particularités et quels concepts de ces systèmes doivent être pris en compte par un modèle d'architecture logicielle ? A la lumière de cette analyse, nous présentons les modèles d'architecture les plus significatifs pour la conception logicielle des systèmes multi-utilisateurs. Au chapitre suivant, nous présentons notre propre contribution.

6.2. Architecture logicielle : définition

L'architecture logicielle est un domaine récent et il n'existe pas de vrai consensus sur une définition. A défaut, plusieurs auteurs sont cependant d'accord sur le rôle attendu d'une architecture logicielle (voir par exemple [Abowd 1994], [Bass 1994], ou [Garlan 1993]). Une architecture logicielle doit communiquer quatre perspectives sur le système et son organisation : fonctionnelle, structurelle, allocation et coordination.

- La *perspective fonctionnelle* décrit les services du système comme une collection de services élémentaires. Le terme "service élémentaire" est à prendre avec précaution car la granularité d'un service élémentaire dépend du domaine d'application, de sa maturité, mais aussi du contexte de développement. Un domaine mature admet une partition canonique en services. Abowd cite comme domaine d'application mature l'exemple des compilateurs [Abowd 1994]. L'ensemble des services d'un compilateur classique est maintenant bien connu :

analyseur lexical, analyseur syntaxique, analyseur sémantique, générateur de code et service d'optimisation. Cependant, pour un domaine d'application moins bien cerné comme les compilateurs pour les systèmes massivement parallèles, on pourra préciser cette décomposition fonctionnelle et affiner par exemple la phase de génération de code en une phase d'analyse des dépendances et d'allocation des processeurs suivie de la génération de code pour chacun des processeurs.

- La *perspective structurelle* décrit de façon statique l'organisation des services. Traditionnellement représentée sous forme de diagrammes, elle fait intervenir des composants, des connecteurs et des configurations. Un *composant* est un élément qui réalise une activité de calcul. Un objet, un agent ou un processus sont des exemples de composants. Un composant a la faculté de communiquer avec l'extérieur c'est-à-dire avec d'autres composants ou avec des ressources matérielles. Cette communication s'effectue par l'intermédiaire de *connecteurs* qui relient plusieurs composants et définissent un protocole entre ces composants. Des connecteurs classiques sont par exemple l'appel de procédure, l'envoi de message ou un protocole réseau comme TCP/IP. Notons que l'ensemble des connecteurs possibles est bien identifié, au contraire de l'ensemble des composants qui dépend fortement du domaine. Enfin une *configuration*, comme celle de la figure 6.1, rassemble les instances de composants et de connecteurs constituant un système particulier à un instant donné. En général, un système est décrit à l'aide de plusieurs configurations complémentaires. Ces différentes configurations décrivent soit différents ensembles indépendants de composants et de connecteurs, soit l'évolution d'un ensemble de composants et de connecteurs au cours du temps.

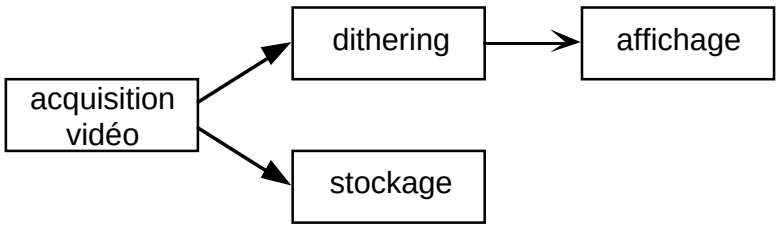

Figure 6.1. Un exemple de configuration pour un module de capture et d'affichage d'images vidéo. Les rectangles représentent des composants identifiés par le nom du service qu'ils réalisent. Les flèches représentent des connecteurs. Le style d'architecture (voir plus loin) est du type flot de données et les connecteurs assurent une simple fonction "copie".

- La *perspective allocation* définit l'allocation des services aux constituants de la structure. Autrement dit, la perspective allocation décrit le lien entre les deux perspectives précédentes, c'est-à-dire de quelle façon la structure véhicule les

fonctions identifiées. C'est principalement à cette étape que sont faits des choix dont l'influence sur la qualité du logiciel sera déterminante. Une stratégie d'allocation adaptée permet en particulier de satisfaire les propriétés de modifiabilité ou de réutilisabilité.

- La *perspective coordination* traduit les aspects dynamiques du système. Deux caractéristiques importantes du fonctionnement du système sont décrites par cette perspective : les protocoles utilisés par les connecteurs et la création dynamique de composants et de connecteurs. Il est important de constater que la dimension coordination est orthogonale à la dimension fonctionnelle. La dimension fonctionnelle prend en compte les services rendus par les composants alors que la dimension coordination s'intéresse au comportement dynamique de l'ensemble des composants et connecteurs.

Notons que cette dernière perspective n'est pas toujours mentionnée (voir par exemple [Kazman 1994]). Pourtant, dans le domaine des interfaces homme-machine il est indispensable de considérer la dimension coordination. La création dynamique de nouveaux éléments d'interaction est souvent occultée dans les architectures pour les interfaces : la responsabilité de la création de nouveaux éléments ou le moment auquel ils doivent être créés ne sont pas toujours clairs. Pour les systèmes multi-utilisateurs, l'analyse fonctionnelle fait apparaître une dimension purement dynamique (la dimension coordination du trèfle du chapitre 1). D'autre part, la communication et donc les connecteurs sont un fondement indispensable d'un système multi-utilisateur. Une architecture logicielle adaptée doit tenir compte de ces deux aspects et pour nous cette perspective coordination est donc particulièrement importante.

Pour être complète, une architecture logicielle doit être accompagnée d'une légende : le *style d'architecture*. Le style est constitué d'un vocabulaire et de règles d'organisation. Le style définit comment doivent être lus les diagrammes et conduit à les interpréter avec un modèle usuel d'organisation de composants : flux de données (*data flow*), client-serveur, tableau noir, etc. Mentionnons que sans le style correspondant, une architecture est ambiguë : suivant le style choisi, elle pourra être interprétée différemment et elle ouvre la voie à des implémentations aux propriétés bien différentes [Shaw 1995]. Pourtant, le style est parfois implicite dans une architecture. En effet, la nature des composants et des connecteurs permet souvent de lever les ambiguïtés. Cependant, sans le style qui sert de guide de lecture, une architecture est incomplète.

Une architecture logicielle répond à un double objectif : organiser le code et les composants du système à construire et documenter cette organisation. Un *modèle*

d'architecture logicielle vise à généraliser les architectures logicielles pour un domaine donné et propose une architecture canonique adaptée au domaine considéré. Au contraire d'une architecture pour un système donné, un modèle d'architecture présente l'avantage de codifier le savoir et donc d'être réutilisable. C'est aussi un objet scientifique : il est possible de raisonner sur ses propriétés et ses qualités. Pour concevoir l'architecture d'un système donné, le modèle d'architecture est instancié avec les composants et connecteurs particuliers au système considéré. Le modèle Arch [Bass 1992] ou le modèle PAC [Coutaz 1987] sont des exemples de modèles d'architecture pour les systèmes interactifs mono-utilisateurs.

6.3. Caractéristiques requises d'une architecture

Une architecture est profondément liée au contexte de développement. Les rôles attendus d'une architecture que nous venons de présenter doivent être modulés par le contexte particulier du système à développer ainsi que par l'expertise des acteurs du développement. Comme le soulignent [Bass 1994] et [Kazman 1994], il n'y a pas de "bonne" ou de "mauvaise" architecture dans l'absolu. Une architecture répond à des critères et à des besoins dans un contexte donné. Toutefois, dans le cadre des systèmes interactifs et des systèmes multi-utilisateurs, on peut identifier des caractéristiques générales qu'une architecture et donc un modèle d'architecture doivent satisfaire.

[Nigay 1994] identifie quatre règles qui traduisent les bénéfices attendus d'un modèle d'architecture logicielle pour les interfaces utilisateur. Ces règles sont aussi applicables au cas des systèmes multi-utilisateurs :

① Un modèle d'architecture logicielle implique une organisation spécifique du code de l'interface utilisateur. Cette organisation doit faciliter les modifications dues, pour l'essentiel, à la mise au point itérative des interfaces.

② Un modèle d'architecture logicielle doit réduire la complexité de réalisation d'une interface utilisateur en proposant des éléments de structuration.

③ Un modèle d'architecture logicielle doit faciliter la décomposition du travail.

④ Un modèle d'architecture doit tenir compte de l'existence des outils logiciels de développement d'interfaces utilisateur et doit englober la plate-forme physique d'accueil. En d'autres termes, un modèle d'architecture doit prendre en compte les contraintes logicielles et matérielles stipulées dans le cahier des charges.

La règle ① impose une contrainte importante due au domaine particulier des systèmes interactifs : elle met en exergue la modifiabilité du code développé en suivant le modèle d'architecture. Nous lui ajoutons volontiers une propriété liée : la réutilisabilité. La satisfaction de ces propriétés est garantie par une allocation adéquate des fonctions à la structure comme dit au paragraphe précédent. La règle ② est contenue en partie dans la perspective structurelle d'une architecture. Mais la réduction souhaitée de la complexité dépend de la pertinence de l'analyse fonctionnelle du domaine. La règle ③ découle des deux précédentes mais souligne un aspect important d'une architecture : en structurant l'ensemble du code à développer, elle permet d'allouer le travail de réalisation des composants à plusieurs individus ou à plusieurs équipes. Cette règle plaide pour une spécification précise des connecteurs afin de préparer l'intégration qui sera nécessaire à l'issue de la phase de réalisation. Nous avons progressé dans cette direction avec le modèle CoPAC présenté au chapitre suivant et avec la classification IMPACT du chapitre 8. Enfin la règle ④ dénote la validité dans le monde réel du modèle. Cette règle est particulièrement importante pour qu'un modèle soit opérationnel. C'est elle qui garantira qu'un modèle est véritablement utilisable, *in fine*, par les praticiens.

Un problème important est la vérification des caractéristiques annoncées par un modèle d'architecture. La méthode SAAM [Kazman 1994] propose une méthodologie permettant ces vérifications. Nous l'avons appliquée à notre modèle CoPAC, en particulier pour vérifier les règles ① et ④. Ces résultats sont présentés avec le modèle CoPAC au chapitre 7.

Pour les systèmes multi-utilisateurs, des facteurs supplémentaires sont à considérer. Premièrement, comme ces systèmes présentent une interface utilisateur à chacun des participants, il faut envisager la duplication de certains composants. Identifier clairement dans l'architecture les composants dupliqués permet de factoriser leur réalisation et réduit ainsi le coût de développement. Notons que cette éventuelle duplication de composants est indépendante du type d'architecture et n'est pas propre à l'architecture répliquée présentée plus loin. Nous proposons donc la règle suivante :

❶ Un modèle d'architecture pour les systèmes multi-utilisateurs doit permettre de factoriser la réalisation en identifiant les composants identiques.

Deuxièmement, étant donné la grande variété des systèmes multi-utilisateurs et la diversité des groupes auxquels ils s'adressent, un modèle d'architecture doit indiquer pour quel facteur d'échelle il est valide. Un modèle peut être adapté à un groupe restreint et se révéler inopérant à plus grande échelle. D'où la règle :

❷ Un modèle d'architecture doit mentionner ses limitations en termes de taille du groupe et indiquer à quelle échelle il est applicable.

Mais ces règles générales sur les modèles d'architecture ne sont pas d'une aide suffisante. Il nous faut maintenant expliciter les composantes de l'espace problème des architectures pour les systèmes multi-utilisateurs afin d'être à même d'évaluer les différents modèles proposés dans la littérature.

6.4. Espace problème

L'objectif de notre espace problème est d'identifier un ensemble de concepts pertinents pour les modèles d'architecture des systèmes multi-utilisateurs et de les organiser. Cet espace problème pourra ensuite servir de base structurante à notre réflexion sur les modèles d'architecture existants.

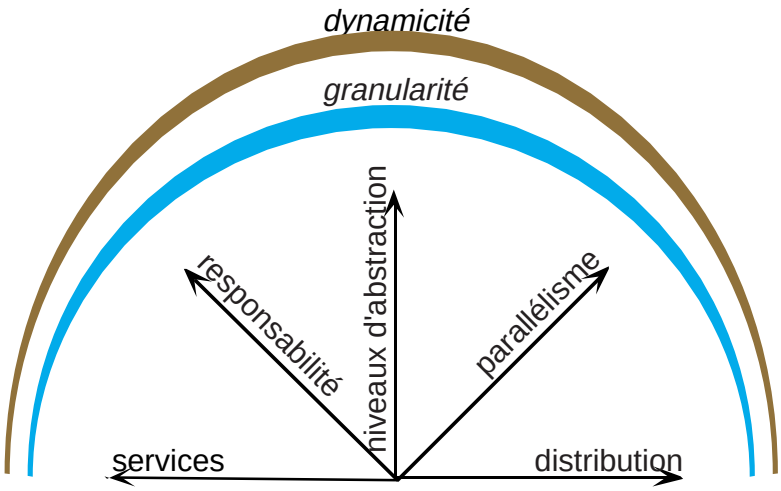

Figure 6.2. Espace problème des modèles d'architecture pour les systèmes multi-utilisateurs. Les arcs représentent des modificateurs qui s'appliquent à chacun des cinq axes.

Nous avons identifié cinq dimensions pour l'espace problème des modèles d'architecture des systèmes multi-utilisateurs : services, responsabilité, niveaux d'abstraction, parallélisme et distribution. A ces cinq axes sont accolés deux modificateurs : granularité et dynamicité. La combinaison d'un modificateur et d'un axe donné permet de moduler le champ d'application de l'axe considéré. La figure 6.2 présente notre espace problème.

6.4.1. Services

La première dimension représente les *services* qu'une architecture d'un système multi-utilisateur doit prendre en compte. Cet axe correspond à la perspective fonctionnelle de

l'architecture. Pour préciser la dimension des services, nous empruntons à Ellis une classification fonctionnelle des services des systèmes multi-utilisateurs. [Ellis 1994] distingue quatre classes de services : les dépositaires, les synchronisateurs, les communicateurs et les agents actifs ("keepers, synchronizers, communicators and agents"). Nous ne conservons que les trois premières catégories, la quatrième étant une combinaison des trois précédentes modulée par la dimension responsabilité de notre espace problème. Nous y reviendrons lorsque nous détaillerons cette dimension.

Les *dépositaires* sont des composants d'une architecture dont le rôle est de maintenir des données. Des exemples typiques de dépositaires sont les systèmes de gestion de fichiers ou les bases de données. Rapprochés du modèle du trèfle du chapitre 1, les dépositaires implémentent ce qui a trait à l'espace de production. On peut aussi faire l'analogie entre les dépositaires et le noyau fonctionnel des systèmes interactifs. Pour spécifier complètement un dépositaire, il convient de s'interroger sur la structuration des données maintenues, du partage de ces données et du contrôle d'accès. La granularité des données maintenues par un dépositaire est aussi un aspect à considérer.

Les *synchronisateurs* assurent la synchronisation des tâches du groupe. Un synchronisateur peut être par exemple la procédure régissant un système de type workflow, ou un composant gérant le contrôle d'accès à un dépositaire. Vis-à-vis du modèle du trèfle, les synchronisateurs implémentent l'espace de coordination. On peut aussi faire le parallèle entre les synchronisateurs et le contrôleur de dialogue des systèmes interactifs. Les synchronisateurs doivent être examinés en lien étroit avec les dimensions responsabilité et parallélisme de notre espace problème.

Les *communicateurs* ont la charge de la communication entre les utilisateurs. Les systèmes de courrier électronique ou les mediaspaces reposent essentiellement sur des communicateurs. Un communicateur peut aussi être vu comme un connecteur particulier qui ne modifie pas le contenu du message qu'il fait transiter : il assure une simple fonction "copie". Les communicateurs implémentent l'espace de communication du modèle du trèfle. Le tableau de la figure 6.3 résume les liens entre ces trois classes de services, le modèle fonctionnel du trèfle et les systèmes interactifs mono-utilisateurs.

Les dépositaires, synchronisateurs et communicateurs catégorisent les services utiles pour un système multi-utilisateurs. Pour être complet, il faut toutefois rajouter les services de l'interface utilisateur. Nous sommes donc amenés à ajouter la classe des *éléments d'interaction* qui est constituée par les composants avec lesquels l'utilisateur interagit. Les services de cette classe sont analogues à ceux de la composante interaction ou présentation des systèmes mono-utilisateurs. A ces services, il faut toutefois ajouter de nouveaux

mécanismes pour garantir des propriétés spécifiques : par exemple un télépointeur ou des barres de défilement (*scrollbars*) multi-utilisateurs.

Classes de services	Espace du trèfle	Systèmes interactifs
Dépositaires	Production	Noyau fonctionnel
Synchronisateurs	Coordination	Contrôleur de dialogue
Communicateurs	Communication	(Pas d'équivalent)

Figure 6.3. Équivalence entre les classes de services des modèles d'architecture, les espaces du modèle fonctionnel du trèfle et les systèmes interactifs mono-utilisateurs.

6.4.2. Responsabilité

La deuxième dimension de notre espace problème définit la *responsabilité* de la prise en charge d'un service donné. Un service peut être réalisé par un des deux types d'acteurs : acteur humain ou acteur logiciel. Dans son acception pour les systèmes mono-utilisateurs, cette distinction traduit le fait qu'une tâche peut être déléguée par l'utilisateur au système. Nous avons trouvé l'exemple suivant dans le Finder du Macintosh [Apple 1994] : si l'on veut copier un élément vers un disque et que la place nécessaire ne peut être obtenue qu'en vidant la corbeille, le système propose de le faire pour l'utilisateur (figure 6.4). Si l'utilisateur répond "OK", il délègue au système la tâche de vider la corbeille. On trouve déjà cette délégation dans Eager lorsque Eager est certain qu'il à détecté des tâches répétitives de l'utilisateur [Cypher 1991].

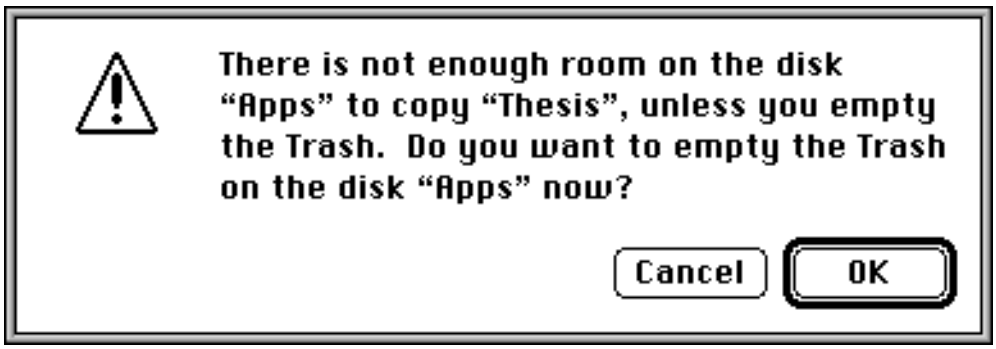

Figure 6.4. Une boîte de dialogue du Finder du Macintosh. Le système demande à l'utilisateur de confirmer qu'il lui délègue la tâche de vider la corbeille.

En fait, la délégation est un phénomène général qui peut se produire entre différentes entités : entre l'utilisateur et le système, entre composants du système et entre différents utilisateurs.

La délégation entre l'utilisateur et le système peut prendre plusieurs formes. A un extrême, l'utilisateur peut se voir déléguer une tâche par le système (par exemple, "insérez la disquette n° 2"). Un exemple commun de délégation est fourni par Eudora, logiciel de courrier électronique [Dorner 1995] : on peut modifier les réglages pour déléguer à Eudora la tâche de relever la boîte-aux-lettres à intervalles réguliers (figure 6.5).

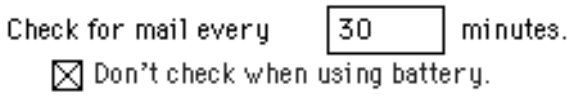

Figure 6.5. Dans Eudora, l'utilisateur délègue au système la tâche de relever la boîte-aux-lettres toutes les 30 minutes. La délégation peut être faite de façon conditionnelle suivant la valeur d'une variable du contexte (l'ordinateur fonctionne sur batterie ou sur le secteur électrique).

A l'autre extrême, les agents actifs (ou agents "intelligents"), informés par l'utilisateur, exécutent pour lui une tâche parfois complexe. Knowledge Navigator [Apple 1988] laisse entrevoir les possibilités de ce type d'outils. Les agents actifs d'Ellis relèvent de ce type de délégation. Les trieurs de courrier électronique tels MAXIMS [Metral 1994] ou Information Lens [Malone 1986] font aussi partie de cette catégorie.

Le phénomène de délégation se produit aussi entre les composants logiciels constituant le système. La délégation sémantique définie par [Coutaz 1990] est un cas particulier dans lequel une partie du rôle du noyau fonctionnel est déléguée au contrôleur de dialogue. Cet enrichissement du contrôleur de dialogue en connaissances sémantiques permet de réutiliser un noyau fonctionnel sans le modifier. La délégation sémantique remplit aussi un autre rôle : la réparation sémantique, c'est-à-dire l'introduction dans les composants dialogue et présentation de concepts inconnus du noyau fonctionnel. Appliquée aux composants logiciels, la notion de délégation est un élément important pour définir la perspective allocation d'un modèle d'architecture.

Pour les systèmes multi-utilisateurs, la délégation existe aussi entre utilisateurs. Cette *délégation sociale* nécessite un support système plus ou moins spécifique. Les mécanismes requis peuvent aller d'un mécanisme général comme le changement des droits d'accès à un fichier jusqu'à des mécanismes plus ad hoc comme la possibilité de se mettre à la place d'un autre utilisateur, par exemple dans les systèmes de réalité virtuelle. Il faut aussi distinguer la délégation inter- et intra-rôles qui peuvent demander des mécanismes différents. Enfin, la dimension responsabilité permet de prendre en compte les différents rôles que doit implémenter le système.

6.4.3. Niveaux d'abstraction

Les services se situent à différents *niveaux d'abstraction*, depuis les primitives du système d'exploitation jusqu'aux mécanismes du noyau fonctionnel. Cette dimension rappelle la règle de validité du modèle d'architecture vis-à-vis du monde réel, c'est-à-dire la plate-forme d'accueil. Tout comme le modèle Arch [Bass 1992] pour les systèmes mono-utilisateurs prend en compte explicitement les boîtes à outils graphiques, un modèle pour les systèmes multi-utilisateurs doit tenir compte des possibilités offertes par l'environnement. Typiquement, un système d'exploitation à objets distribués comme GUIDE [Krakowiak 1990] offre des services de plus haut niveau qu'un système classique comme Unix. En relation avec l'axe responsabilité et modulé par le modificateur dynamicité, la dimension niveaux d'abstraction permet d'exprimer pour un service, la capacité de migration dynamique entre les différents niveaux d'abstraction du système.

6.4.4. Parallélisme

Déjà importante dans les systèmes mono-utilisateurs, en particulier multimodaux [Coutaz 1993], la dimension *parallélisme* prend une nouvelle importance pour les systèmes multi-utilisateurs. Elle permet de décrire les relations temporelles non seulement pour un utilisateur donné, mais pour l'ensemble des utilisateurs. Ces relations temporelles, en relation avec l'axe niveaux d'abstraction, peuvent porter sur les groupes de tâches, les tâches ou les actions des utilisateurs (suivant le modificateur granularité). Cette dimension englobe aussi la traditionnelle distinction entre systèmes multi-utilisateurs synchrones et asynchrones.

6.4.5. Distribution

La dimension *distribution* est à rapprocher de la perspective structurelle d'une architecture. Elle exprime la localisation physique des composants logiciels sur l'ensemble des sites[1] constituant le système multi-utilisateur. Combinée au modificateur granularité, cette dimension demande de considérer quelles entités seront distribuées : objet ou même partie d'objet pour un système d'exploitation à objets, composants logiciels ou combinaisons de composants. Combinée au modificateur dynamicité, elle nous permet de prendre en compte les déplacements d'objets d'un site vers un autre qui se produisent dans les systèmes à objets. Cette dimension permet aussi de réfléchir à la réplication des composants et répond donc au besoin de mise en évidence de la factorisation que nous avons introduite au paragraphe 6.3. A un plus haut niveau de granularité, la dimension distribution permet de prendre en compte le nombre de sites

[1] Nous utilisons ici "site" dans le sens "unité de traitement informatique indépendante". En général, un site est une station de travail.

participant au système multi-utilisateur. Du point de vue de l'architecture, il est souvent plus pertinent de considérer le nombre de sites plutôt que le nombre d'utilisateurs, mais ces deux caractéristiques sont bien sûr liées. On peut toutefois trouver des cas où plusieurs utilisateurs partagent le même site.

Les cinq axes de notre espace-problème, services, responsabilité, niveaux d'abstraction, parallélisme et distribution, modulés par les modificateurs granularité et dynamicité, nous donnent un cadre de réflexion pour les modèles d'architecture pour les systèmes multi-utilisateurs. Cet espace-problème dont nous avons explicitées les dimensions va nous permettre d'évaluer les modèles usuels proposés pour les systèmes multi-utilisateurs.

6.5. Modèles d'architecture usuels pour les systèmes multi-utilisateurs

Après avoir identifié ce que doit communiquer une architecture et les concepts pertinents pour les systèmes multi-utilisateurs, nous passons maintenant en revue dans cette double perspective quelques modèles usuels proposés dans la littérature. Nous commençons par le triple modèle classique centralisé/répliqué/hybride. Puis nous présentons trois modèles pour les systèmes synchrones qui privilégient l'information partagée (l'espace de production) comme les éditeurs partagés. Ces trois modèles, ALV, le modèle à états partagés et les User Display Agents, ont des couvertures fonctionnelles différentes. Nous détaillons ensuite deux modèles plus adaptés à des systèmes privilégiant la communication.

6.5.1. Le triple modèle classique : centralisé/répliqué/hybride

Le modèle triple centralisé/répliqué/hybride fait quasiment office de modèle de référence dans la littérature sur les architectures des systèmes multi-utilisateurs. Il identifie trois cas présentés figures 6.6, 6.7 et 6.8.

Dans le cas centralisé, un processus unique s'exécute sur un site serveur et gère l'ensemble de l'application multi-utilisateur, c'est-à-dire le noyau fonctionnel et l'ensemble des interfaces des utilisateurs. Ce modèle est bien adapté à un système graphique client-serveur comme X-Window. La figure 6.6 présente une extension de ce modèle dans laquelle différentes interfaces correspondant à différents rôles sont pris en compte.

Le *modèle centralisé* présente l'avantage de la simplicité : comme l'application est réduite à un seul processus, les questions de communication et de synchronisation entre

processus ne se posent pas. Or elles constituent une difficulté de la conception logicielle des systèmes multi-utilisateurs. Mais cette simplicité se paie en termes de fiabilité et de flexibilité de l'architecture. Le problème de fiabilité est connu : comme toute l'architecture repose sur un seul processus, sa défaillance ou son manque de performance ont des conséquences gênantes pour l'ensemble des utilisateurs. D'autre part, un processus central ne permet pas de garantir pour tous les utilisateurs un temps de réponse acceptable ni même constant. Les propriétés de stabilité et de conformité du temps de réponse sont compromises. Cette architecture présente aussi l'inconvénient de ne pas prendre en compte tous les types de systèmes. Un système qui requiert un parallélisme important tel MMM (Multi-device, Multi-user, Multi-editor) [Bier 1991] est limité par une architecture centralisée. Notons que cette limitation sur le parallélisme peut être levée en utilisant des processus légers ou threads comme en proposent de nombreux systèmes d'exploitation modernes. Cependant, les problèmes de fiabilité et de vivacité de l'interface persistent. Notons aussi que la grosse granularité de l'architecture centralisée, qui raisonne au niveau du processus, limite son intérêt pour un système comme MMM qui requiert un grain plus fin, de l'ordre de l'objet, voire de la méthode d'objet.

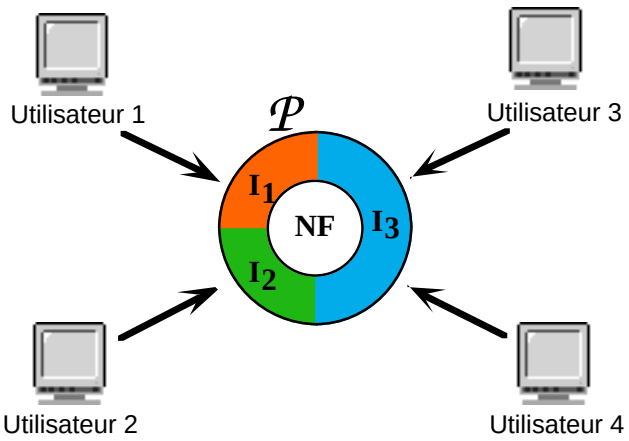

Figure 6.6. Le modèle centralisé. Exemple de configuration avec quatre utilisateurs, quatre terminaux et un serveur. Les flèches signifient "est client de". $\mathcal{P}$ est l'unique processus du système. Il gère le noyau fonctionnel **NF** et une ou plusieurs interfaces utilisateur { $\mathbf{I_i}$ }. Une interface I_i présente et permet d'utiliser les services correspondant au rôle i. Sur la figure, les utilisateurs 3 et 4 ont le même rôle. Dans le modèle centralisé originel, i = 1.

En résumé, vis-à-vis de notre grille d'analyse, l'architecture centralisée privilégie les dépositaires au détriment des autres classes de services. Remarquons aussi que des systèmes qui sont apparemment des systèmes de communication, comme les messageries multi-utilisateurs des serveurs Minitel ou les MUDs (Multi-User Dimensions) adoptent une architecture centralisée. En fait, du point de vue de l'architecture, ces systèmes sont bien des systèmes orientés vers la production : les MUDs par exemple, permettent une communication de chaque utilisateur vers l'ensemble des utilisateurs sous forme d'un

espace texte partagé. Ils sont en fait plus proches des éditeurs partagés. Lorsque des communications entre utilisateurs un à un sont possibles, l'architecture assigne un processus à chaque utilisateur et l'on retrouve le modèle répliqué que nous présentons maintenant.

Le *modèle répliqué* est une perspective radicalement opposée. Cette fois, un processus est alloué à chaque utilisateur et les éventuelles données partagées sont dupliquées sur tous les sites. La figure 6.7 présente une généralisation du modèle répliqué qui prend en compte différents rôles.

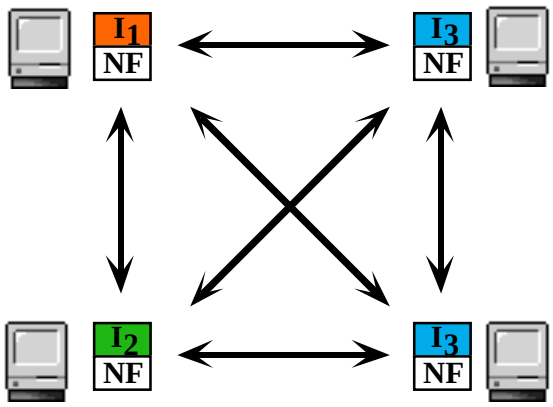

Figure 6.7. Le modèle répliqué. Exemple de configuration à quatre utilisateurs. Un processus par utilisateur gère une copie du noyau fonctionnel **NF** répliqué et l'interface locale $\mathbf{I_i}$ adaptée au rôle de l'utilisateur. Chaque processus peut communiquer directement avec n'importe quel autre. Les connecteurs représentés par des flèches sont un protocole réseau de communication inter-applications. Ils relient deux à deux les noyaux fonctionnels et les interfaces.

Ce modèle semble a priori mieux adapté aux systèmes multi-utilisateurs : la duplication des composants de l'architecture permet d'améliorer la vivacité de l'interface. La duplication permet aussi une fiabilité accrue : en cas de défaillance de l'un des processus, les autres processus continuent leur activité. Au cas où le processus défaillant doit communiquer avec d'autres processus, des techniques classiques (communication asynchrone, délai de garde—ou "timeout") permettent de détecter la défaillance et de mettre "hors circuit" le processus fautif. Cependant, l'architecture répliquée, si elle est bien adaptée aux services de la classe communicateurs, présente l'inconvénient de dupliquer les dépositaires. Les données étant répliquées sur tous les sites, il devient indispensable de mettre en place des mécanismes de maintien de la cohérence des données répliquées. Cette approche présente les inconvénients suivants.

- Les communications n'étant pas instantanées, des phénomènes de "zone grise" peuvent se produire : la modification d'une donnée sur un site ne sera répercutée sur les autres sites qu'après un intervalle de temps Δt dépendant du réseau de

communication. Un réseau de communication (tel Ethernet) étant non-déterministe, Δt est en général imprévisible. Des conflits peuvent donc apparaître si plusieurs utilisateurs modifient la même donnée simultanément. Pour résoudre ce problème, des protocoles de communication garantissant l'ordonnancement global des messages, tels les protocoles multicast sont envisageables. L'inconvénient de cette technique est qu'un protocole de ce type a un temps de réponse qui croît exponentiellement en fonction du nombre de sites. Cette solution n'est donc pas envisageable pour un grand nombre de sites. D'autres protocoles, comme ceux fondés sur des algorithmes optimistes [Karsenty 1994], sont toutefois des solutions moins coûteuses et prometteuses.

- En cas de défaillance d'un processus ou du réseau, l'intégrité des données locales au processus fautif (ou de l'ensemble des données dans le cas du réseau) est compromise. Il est indispensable d'envisager des mécanismes robustes de reprise après panne. Ces mécanismes peuvent être analogues aux mécanismes de fusion de versions de documents et nécessitent souvent dans ce cas l'intervention des utilisateurs pour résoudre les conflits. D'autres solutions reposant sur l'utilisation d'un historique des modifications sont aussi utilisées, mais requièrent le maintien d'un historique local à chaque processus.

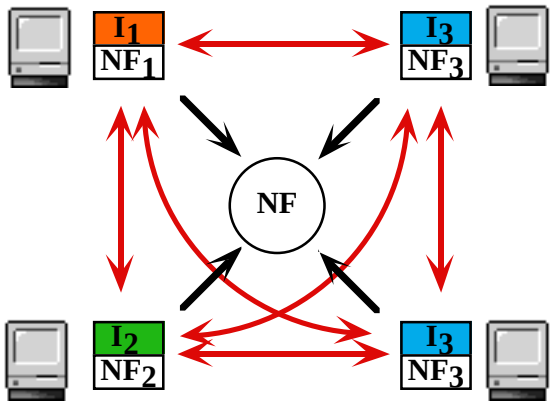

Figure 6.8. Le modèle hybride. Exemple de configuration à quatre utilisateurs. Un processus gère la station de chaque utilisateur (noyau fonctionnel local $\mathbf{NF_i}$ et interface locale $\mathbf{I_i}$) et un processus central gère le noyau fonctionnel commun **NF**. Les communications entre les processus combinent les possibilités du cas centralisé avec le processus central et celles du cas répliqué pour les processus utilisateurs.

Le *modèle hybride* apporte une solution au problème du maintien de la cohérence des données sans sacrifier la vivacité du temps de réponse de l'interface. Cette approche combine les deux cas précédents en décomposant un même système en deux parties, l'une centralisée, l'autre répliquée. La figure 6.8 présente le modèle hybride étendu au cas d'un système à plusieurs rôles.

Cette solution permet d'isoler dans la partie centralisée les données partagées et conserve le modèle répliqué pour la gestion locale. Notons qu'il n'est pas nécessaire de centraliser les données partagées. On peut utiliser le composant centralisé comme un simple synchronisateur et garder des dépositaires locaux à chaque processus utilisateur. Le synchronisateur central est alors un "chef d'orchestre" qui gère le contrôle d'accès aux données partagées répliquées. Cette solution garantit de meilleurs temps de réponse du système lorsqu'un utilisateur accède aux données partagées, comme souligné par [Karsenty 1994]. Mais en contrepartie, les données partagées doivent être maintenues cohérentes.

A l'examen de ce triple modèle fondamental centralisé/répliqué/hybride, plusieurs constatations s'imposent : la validité de ces modèles doit être appréciée selon les dimensions de notre espace problème et selon leur couverture fonctionnelle du modèle du trèfle du chapitre 1.

Les deux modèles extrêmes (centralisé et répliqué) tiennent plus du cas d'école que du modèle utilisable en pratique. Ces modèles présentent toutefois un intérêt dans des cas particuliers : lorsqu'il faut tenir compte de contraintes logicielles ou matérielles ou si l'on s'intéresse à un système multi-utilisateur limité. Un environnement logiciel particulier peut conduire à adopter un modèle centralisé pour faciliter l'implémentation et si les contraintes du cahier de charges (par exemple, on souhaite du WYSIWIS strict) s'y prêtent. On peut citer comme exemple SharedX qui repose sur X-Window [Garfinkel 1989] ou le serveur mediaspace IIIF [Buxton 1990]. Dans ce deuxième exemple, l'utilisation d'un réseau audio/vidéo analogique construit autour d'un répartiteur piloté par un serveur conduit à une architecture centralisée. Nous présentons au chapitre 7 l'architecture du système NEIMO dans lequel d'autres contraintes ont imposé de centraliser certains services. Les deux modèles centralisé et répliqué sont chacun bien adaptés à un extrême de l'éventail des systèmes multi-utilisateurs. Le modèle centralisé répond bien aux exigences de systèmes permettant principalement des tâches de production. De plus, l'utilisation de ce modèle pour un système asynchrone lève la restriction que nous avons émise sur la satisfaction de la propriété de vivacité. S'il n'y a pas de parallélisme entre les tâches des utilisateurs, on retrouve le cas d'un système utilisé successivement par différents utilisateurs. Si l'on admet du parallélisme entre les tâches des utilisateurs, et si la synchronisation est entièrement de la responsabilité du système, on retrouve un autre cas commun : les systèmes du type base de données multi-utilisateurs. Le modèle répliqué, lui, convient mieux aux systèmes privilégiant les tâches de communication. Nous reviendrons au chapitre 7 sur l'architecture de notre mediaspace VideoPort qui s'inspire de l'architecture répliquée.

D'un point de vue pratique et pour un système multi-utilisateur quelconque, le modèle hybride est le plus réaliste. Mais ce modèle est schématique et n'offre qu'une aide limitée pour guider la décomposition des services de haut niveau comme le noyau fonctionnel. Le modèle hybride est aussi simplificateur : il ne représente les systèmes hybrides existants qu'à un haut niveau d'abstraction. En pratique, une partie centralisée peut être complexe (par exemple une base de données répartie gérée par une collection ou une hiérarchie de serveurs comme l'annuaire électronique sur Minitel ou un réseau hypertexte comme World-Wide Web).

En fait, vis-à-vis de notre espace problème, le triple modèle de référence centralisé/répliqué/hybride ne couvre correctement que la dimension distribution pour une granularité de l'ordre de la combinaison de composants. Ce modèle envisage clairement les différentes localisations possibles des composants noyau fonctionnel et interface. En revanche, il montre des lacunes pour toutes les autres dimensions. En ce qui concerne les services, il n'identifie pas de services spécifiques aux systèmes multi-utilisateurs, à part peut-être un mécanisme de contrôle de la cohérence des données répliquées. Mais il ne positionne pas ce service dans les composants logiciels qu'il préconise et ne dit rien de sa réalisation. Malgré l'existence de modèles de référence simples comme le modèle Arch, le modèle centralisé/répliqué/hybride se contente de la distinction élémentaire entre interface et noyau fonctionnel. Nous verrons plus loin avec ALV ou au chapitre suivant avec notre modèle CoPAC que cette distinction peut être affinée. Les trois modèles ne permettent pas de prendre en compte de façon utile les dimensions responsabilité ou niveaux d'abstraction de notre espace problème. Le parallélisme peut être envisagé en termes de fonctionnement simultané au niveau du processus, mais sans plus de détail. Remarquons que les lacunes de ce modèle et d'une façon générale son trop haut niveau d'abstraction le rendent vite inopérant. Dans les figures 6.6, 6.7 et 6.8, nous avons essayé d'étendre le modèle en introduisant la notion de rôle. Nous avons distingué différentes combinaisons de composants de l'interface correspondant à différents rôles. Cette extension n'est toutefois pas entièrement satisfaisante : le noyau fonctionnel ne doit-il pas être également adapté au rôle ? Un composant d'interface ne peut-il pas être partagé entre plusieurs rôles ? La "grosse" granularité des composants dans ce modèle d'architecture empêche une analyse précise. Le modèle CoPAC que nous présentons au chapitre 7 vise à répondre à ces questions.

Nous avons vu que le modèle précédent pêche par manque de détail dans la définition et la structuration des composants. Le modèle ALV que nous présentons maintenant est une tentative pour y remédier.

6.5.2. Le modèle ALV

Le modèle d'architecture ALV (Abstraction-Link-View) [Hill 1992], utilisé dans le système RendezVous™, vise les systèmes multi-utilisateurs synchrones construits autour d'un noyau fonctionnel centralisé. Ce composant central est dénommé "Shared Abstraction" dans le modèle. Chaque utilisateur dispose d'un composant interface qui lui est propre et adapté à son rôle ("Personal View"). Des composants de liaison, "Link", font le lien entre l'abstraction partagée et chaque vue. Les Link maintiennent des systèmes de contraintes et propagent les modifications de l'un des deux composants vers l'autre. La figure 6.9 présente une configuration du modèle ALV pour quatre utilisateurs.

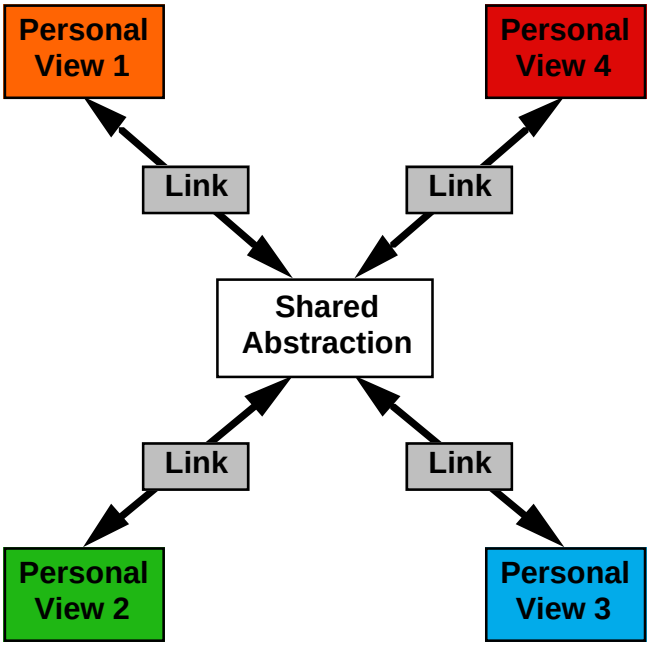

Figure 6.9. Le modèle ALV. Les flèches représentent des contraintes liant un composant abstraction partagé à des vues propres à chaque utilisateur. Les composants Link maintiennent et propagent les contraintes entres les composants abstraction et vue. Sur la figure quatre utilisateurs partagent une même abstraction.

Le modèle ALV rappelle le modèle PAC pour les interfaces mono-utilisateur : tous deux distinguent un composant abstraction et un composant d'interface et tous deux insistent sur la nécessité d'introduire un composant intermédiaire qui maintient la cohérence. Ce composant rend indépendants abstraction et interface conformément au principe de séparation [Coutaz 1990]. Toutefois, on peut noter des différences entre les deux conceptions de cette séparation.

Le modèle PAC préconise une organisation des agents PAC en une hiérarchie à l'intérieur du contrôleur de dialogue. ALV ne semble pas identifier clairement un composant de contrôle du dialogue à part le Link. Or le rôle d'un contrôleur de dialogue est double : il maintient la cohérence entre abstraction et présentation et il gère aussi l'ordonnancement

des tâches. Si le premier aspect est bien pris en compte par les Link de ALV, le deuxième aspect n'est pas explicite dans le modèle. D'autre part, ALV ne recommande pas une vraie structuration des composants. A part pour les composants vue où il est dit explicitement qu'ils sont composés d'une hiérarchie de vues, les composants constituant un Link en particulier ne sont pas organisés et sont tous au même niveau. En revanche, la hiérarchie PAC exprime la coordination entre agents. Cette analyse met en évidence deux points faibles du modèle ALV : sa couverture fonctionnelle est incomplète puisqu'il n'identifie qu'une partie du contrôle du dialogue. Du point de vue structurel, l'organisation des composants n'est indiquée que pour les vues.

Par rapport à notre espace problème, ALV insiste sur un service spécifique des systèmes multi-utilisateurs : le partage de composants abstraction. Notons qu'ALV, au moins dans une version préliminaire [Patterson 1990], identifiait un service de mise en place de session reposant sur un serveur de sessions. Ce type de service, que nous verrons plus en détail avec l'exemple de GroupKit, augmente notre éventail de services spécifiques aux systèmes multi-utilisateurs. ALV distingue deux niveaux d'abstraction (abstraction et vue) et précise comment les relier. ALV permet aussi de réfléchir à la distribution des composants, et dans une moindre mesure, aux possibilités de parallélisme. Mais ALV est un modèle qui privilégie le partage de composants abstraction ; il est donc bien adapté aux systèmes privilégiant l'aspect production et peut être vu comme un affinement du modèle centralisé.

Notons que [Croisy 1994] propose un affinement du modèle ALV dans lequel le contrôleur de dialogue est explicite. Plus précisément, ce modèle distingue un composant "multi-dialogue" géré par un superviseur et composé de "micro-dialogues". Un micro-dialogue est dédié à l'abstraction et communique avec autant de micro-dialogues de présentation qu'il y a de vues. Afin d'assurer la rétroaction de groupe, les micro-dialogues peuvent communiquer directement entre eux. Mais comme ALV, ce modèle est adapté aux systèmes privilégiant l'espace de production. D'autre part, l'indépendance des micro-dialogues n'est pas clairement garantie dans le modèle. En l'absence d'un mécanisme d'abonnement comme GroupKit (voir paragraphe 6.5.5), l'indépendance des micro-dialogues n'est pas assurée sauf s'ils sont tous identiques. La flexibilité du modèle risque donc d'être compromise. Toutefois ce modèle a l'avantage d'intégrer explicitement le contrôle du dialogue et en propose un affinement intéressant.

6.5.3. Le modèle à états partagés

Patterson, un des auteurs de RendezVous d'où est issu ALV, propose un autre modèle qui vise les mêmes objectifs mais de façon différente. Le domaine d'application est

toujours les systèmes multi-utilisateurs synchrones dans lesquels le partage est important. Le modèle à états partagés identifie quatre niveaux d'états d'une application : File, Model, View et Display [Patterson 1994]. Il distingue deux modes de partage : par états partagés et par états synchronisés. Sur la figure 6.10, l'état du composant Model est partagé et l'état des composants View est synchronisé.

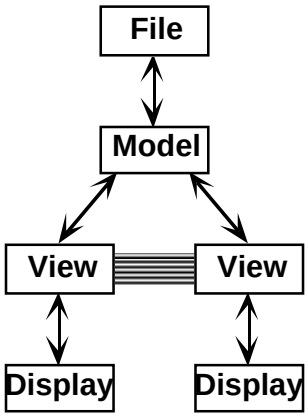

Figure 6.10. Le modèle à états partagés, d'après [Patterson 1994]. Pour faciliter la lecture, ce schéma montre une configuration à deux utilisateurs. Le composant Model est partagé ainsi que le composant File. Les deux composants View sont synchronisés (barres horizontales). Les composants Display sont indépendants.

Dans le partage d'états, si un composant est partagé, tous les composants de niveau plus abstrait sont aussi partagés (sur la figure, Model est partagé par états, donc File l'est aussi). Dans le partage par synchronisation, les deux copies des composants utilisent un protocole pour synchroniser leurs états.

A première vue, ce modèle peut apparaître comme un affinement de ALV qui prend en compte la possibilité de maintenir la cohérence entre composants répliqués, comme le permet le modèle hybride. En fait il faut noter une différence : le modèle repose sur un partage des états des composants et non sur un partage des données comme le modèle hybride du paragraphe 6.5.1. D'autre part, les quatre niveaux d'abstraction identifiés par le modèle à états partagés sont différents de ceux d'ALV. L'abstraction d'ALV est décomposée en File et Model, et la vue d'ALV est précisée en View et Display. Mais au contraire d'ALV, le modèle ne prend pas du tout en compte le contrôle du dialogue. Or l'état du dialogue devrait pouvoir aussi être partagé. Il est probable qu'une partie du dialogue est diluée dans les mécanismes de synchronisation et de partage sous-jacents. Mais le dialogue en tant que tel n'apparaît pas.

Comme ALV, le modèle à états partagés souffre d'une couverture fonctionnelle incomplète. Le contrôle du dialogue n'est pas explicite. Il semble qu'une partie du dialogue soit présente dans le mécanisme de synchronisation sous-jacent. Mais dans ce

cas, la synchronisation devrait apparaître comme un composant à part entière et non comme un simple connecteur. En fait, ce modèle nous semble supposer l'existence d'une plate-forme matérielle d'accueil spécifique, comportant en particulier des mécanismes de synchronisation de haut niveau d'abstraction. Dans ce cas, la synchronisation peut être un service offert par le système et peut être en effet vue comme un type particulier de connecteur. Cette constatation est à double tranchant : elle montre une limitation du modèle en termes de validité vis-à-vis de l'environnement de l'application. Mais elle montre aussi que le modèle distingue explicitement un service spécifique aux systèmes multi-utilisateurs : le partage et la synchronisation d'états entre composants. Vis-à-vis des autres axes de notre espace problème, le modèle n'est pas beaucoup plus complet que ALV : il présente l'avantage de mettre en évidence le service de synchronisation et affine les niveaux d'abstraction de ALV. Il permet donc de considérer plus complètement les possibilités de parallélisme et de distribution des composants. Il met aussi en évidence la migration du lieu de partage et de synchronisation.

Pour le cas des systèmes reposant sur le partage d'informations, nous avons vu plusieurs décompositions : du cas simpliste du modèle centralisé jusqu'à la décomposition à quatre niveaux du modèle à états partagés. Il convient ici de citer le modèle SLICE [Karsenty 1994] qui identifie sept niveaux et est à notre connaissance le plus complet. Sans être à proprement parler un modèle d'architecture, SLICE (Sharing Layers In Cooperative Editing) propose une structuration en sept couches : document abstrait, représentation du document, manipulation directe, représentation des vues, manipulation des vues, manipulation indirecte, curseurs. Chacune des couches peut être partagée entre les utilisateurs de façon indépendante. Cette décomposition fournit un cadre de réflexion pour le partage d'informations des systèmes d'édition coopérative.

Notons que ni ALV, ni le modèle à états partagés ne tiennent compte des services de la classe des communicateurs. ALV privilégie les dépositaires et le modèle à états partagés privilégie, quoiqu'incomplètement, les synchronisateurs. Cette limitation est peut-être induite par le domaine d'application qui conduit d'abord à s'interroger sur le partage de l'information. Le modèle des User Display Agents que nous présentons maintenant est une autre approche de ce même domaine.

6.5.4. Le modèle des User Display Agents

Le modèle des User Display Agents [Bentley 1994], comme les modèles précédents, s'intéresse aussi au cas des systèmes synchrones. Il décompose un système en une partie centralisée et une partie répliquée. La partie centralisée s'articule autour de l'Object Store, un dépositaire d'objets partagés. L'Object Store Server est le composant qui gère l'Object

Store : il est composé de l'Update Handler qui transmet toutes les modifications des objets et gère l'accès aux objets. Ses autres constituants sont des User Display Agents (UDA). Un UDA a la charge de gérer un élément d'interaction en relation avec l'Object Store. Ce type d'agent est proche d'un agent de type PAC ou MVC, avec la différence que sa partie présentation ou vue est un ensemble de composants répliqués, les UDAs délégués. Les UDAs délégués sont répliqués sur chacune des stations utilisateur.

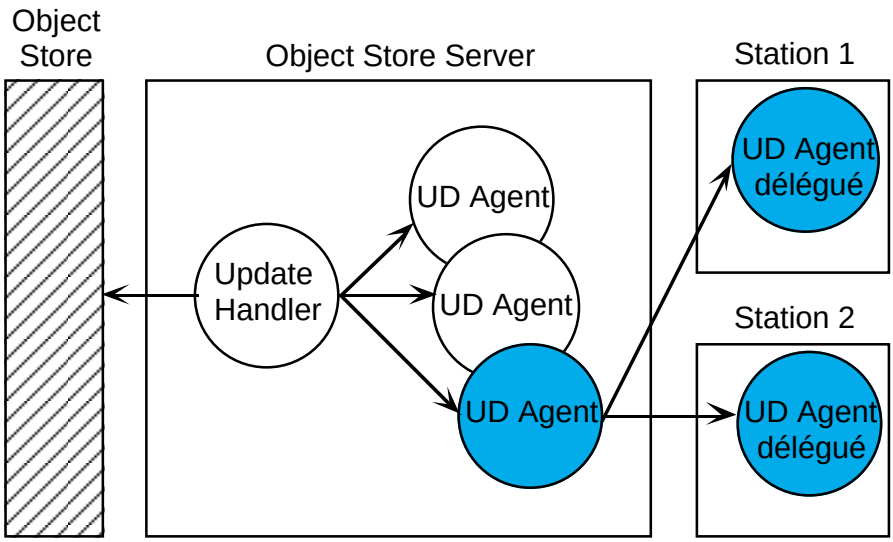

Figure 6.11. Le modèle des User Display Agents (UDA). Un UDA est localisé sur le serveur central et gère un composant d'interface. Sur chaque station utilisateur, des agents UDA délégués reflètent à l'interface les changements d'état de leur UDA "maître". Le Update Handler répartit les changements des objets de l'Object Store aux UDAs concernés.

Ce modèle nous fait penser à la gestion des vues multiples d'une interface mono-utilisateur avec le modèle d'architecture PAC : ce modèle recommande d'assigner à un agent la gestion d'une vue donnée et de chapeauter ces agents vue par un agent père chargé de maintenir la cohérence entre les vues. Le modèle des UDAs est une sorte de généralisation de la gestion des vues multiples.

Le modèle des UDAs permet de raisonner à un niveau d'abstraction plus fin que les modèles que nous avons vus précédemment. Il introduit des agents qui modélisent des composants en charge d'un élément d'interaction donné. Cette décomposition plus fine permet aussi de réfléchir à la délégation entre composants. Le modèle fait intervenir un service spécifique aux systèmes multi-utilisateurs : l'Object Store Server gère la session et permet l'abonnement et le désabonnement des UD délégués. Dans ce modèle encore, l'existence d'une infrastructure système particulière (système à objets) est supposée. Mais au regard de notre espace problème, le modèle des UDAs est le plus complet de ceux que nous avons présentés. Toutefois, lui non plus ne prend pas en compte les composants de la classe communicateurs.

6.5.5. GroupKit

Les modèles que nous avons vus jusqu'à présent s'intéressent au cas des systèmes multi-utilisateurs qui privilégient la dimension production. Nous présentons maintenant le modèle d'architecture de GroupKit [Roseman 1992], qui repose sur la notion de conférence et met en avant la dimension coordination. GroupKit est une boîte à outils de haut niveau pour la construction des systèmes multi-utilisateurs et non pas un modèle d'architecture proprement dit. Mais nous verrons au chapitre 8 que les outils de haut niveau reposent sur un modèle d'architecture implicite sous-jacent. Nous explicitons maintenant le modèle d'architecture sous-jacent de GroupKit, présenté figure 6.12.

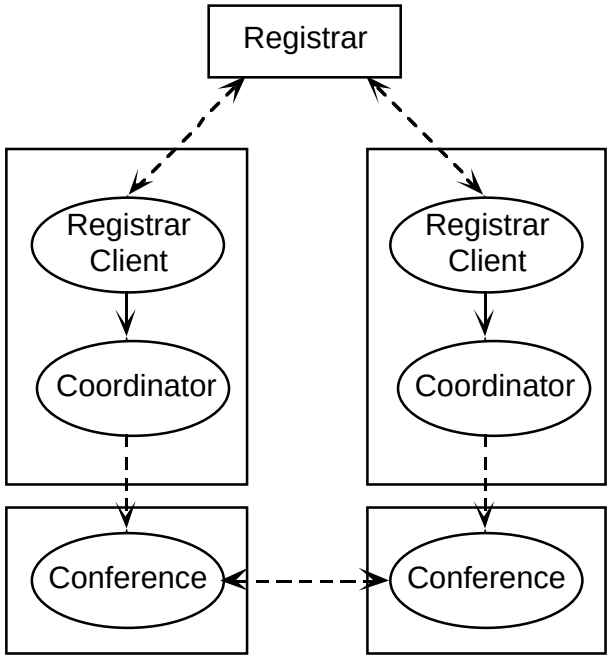

Figure 6.12. Le modèle sous-jacent de GroupKit. La figure présente une configuration pour deux utilisateurs. Les rectangles représentent des processus et les ovales représentent des objets. Les flèches pleines matérialisent les envois de messages entre objets et les flèches pointillées sont des communications inter-applications via des sockets.

En haut de la figure 6.12, le Registrar est un composant centralisé avec lequel les Registrar Client communiquent pour initier, se joindre à, ou quitter une conférence. Le composant Coordinator gère des processus Conference. Sur la figure, nous n'avons fait figurer qu'un seul processus Conference attaché à chaque Coordinator, mais un Coordinator a la capacité de gérer plusieurs Conference qui peuvent être de natures différentes. Une Conference est une application partagée et peut être par exemple, un module d'édition de texte ou de dessin partagé ou un module de communication audio ou vidéo. Une Conference contient à la fois le noyau fonctionnel, le dialogue et l'interface utilisateur.

On peut regretter dans ce modèle qu'il ne détaille pas plus l'objet Conference. Le modèle de ce fait ne garantit pas l'observation de principes élémentaires comme la distinction entre noyau fonctionnel et interface utilisateur. Cependant, le modèle de GroupKit a le mérite de mettre en évidence un service important pour les systèmes multi-utilisateurs : l'abonnement des utilisateurs à une session. Vis-à-vis de notre découpage fonctionnel, les composants Registrar, Registrar Client et Coordinator assurent l'essentiel de la facette coordination. Les aspects production et communication sont couverts par le composant Conference. Un autre aspect intéressant du modèle de GroupKit est qu'il distingue deux types de connecteurs : les messages de contrôle entre objets représentés par des flèches pleines et les communications entre processus représentées par des flèches pointillées. Toutefois, nous pensons que cette distinction n'a été faite que pour renforcer la distinction entre processus et objets sur le schéma d'architecture. Par exemple, la communication entre deux objets Conference sera toujours matérialisée par des flèches pointillées, quelle que soit la nature des échanges : messages adressés au noyau fonctionnel distant dans le cas d'un éditeur de dessin ou messages dont le contenu n'est pas compris par le système comme dans le cas d'une vidéoconférence. Dans le modèle CoPAC présenté au chapitre suivant, nous proposons de distinguer explicitement les messages destinés au noyau fonctionnel de ceux dont la sémantique n'est pas interprétée par le système comme les flots audio/vidéo.

Contrairement aux modèles que nous avions vus jusqu'ici, le modèle de GroupKit détaille peu les composants en charge de l'espace de production et insiste particulièrement sur la facette coordination. Nous allons voir maintenant un modèle qui recouvre l'aspect communication.

6.5.6. Le modèle de communication de Gemma

Gemma est une architecture pour un système de fenêtrage distribué qui étend X-Window [Freeman 1993]. Gemma dans son ensemble s'intéresse particulièrement à des systèmes faisant un usage intensif de multiples dispositifs d'entrée. L'architecture globale proposée rappelle le modèle à états partagés de Patterson mais se limite à une synchronisation au niveau View ou Display. Un intérêt de ce modèle est qu'il prend aussi en compte la communication entre utilisateurs via des médias continus comme le son ou la vidéo. Le modèle adopté pour la communication est assez classique : on retrouve une architecture similaire dans [Koegel Buford 1994]. Notre bibliothèque de communication UserLink que nous présentons au chapitre 8 adopte une approche analogue. Le principe directeur de cette architecture repose sur la distinction entre informations à communiquer et contrôle du canal de communication (figure 6.13).

Une source génère un flot d'un média continu, comme un système de capture vidéo. Un puits est un récepteur d'un flot continu, comme une fenêtre d'affichage. Un gestionnaire de capture contrôle la source. Par exemple, c'est ce composant qui a la charge d'initier le flot vidéo ou de le stopper. Le gestionnaire de présentation a la responsabilité d'initier le puits, c'est à dire de lui indiquer sa surface d'affichage, et gère les actions de l'utilisateur sur la surface d'affichage. Par exemple un clic sur la fenêtre vidéo peut signifier qu'il faut stopper le flot vidéo. C'est le gestionnaire de présentation qui effectue ce traitement. Le gestionnaire de présentation et le gestionnaire de capture communiquent directement entre eux pour échanger des informations de contrôle et de synchronisation. Le gestionnaire de synchronisation va, par exemple, répercuter sur le gestionnaire de capture une modification de la taille de la surface d'affichage.

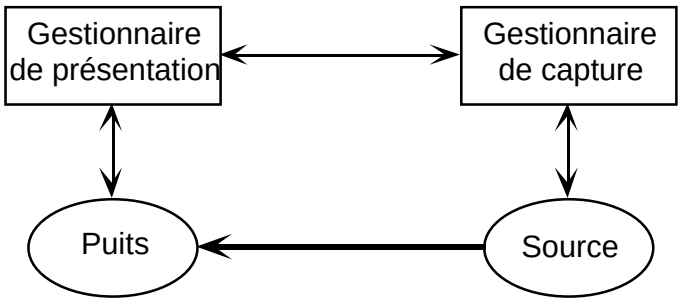

Figure 6.13. Le modèle de communication de Gemma. Le trait gras indique un flot de données. Les autres flèches indiquent des flots de contrôle.

Ce modèle usuel est bien adapté à l'intégration des médias continus dans une architecture. La séparation qu'il impose entre flots de données et contrôle traduit les propriétés différentes qui sont requises des deux types de communication. En particulier, des mécanismes adaptés au niveau des sources et des puits permettent d'assurer par exemple les propriétés de régularité des flots ou de synchronisation. On peut regretter que l'architecture de Gemma ne propose pas d'intégration entre la communication et les autres services pour les systèmes multi-utilisateurs. La communication est vue comme un aspect indépendant. La plupart des modèles d'architecture pour les systèmes multi-utilisateurs qui intègrent l'aspect communication adoptent aussi cette approche. Or elle rend plus difficile une meilleure intégration des services de communication avec les autres services des systèmes multi-utilisateurs. Nous avons tenté avec le modèle CoPAC présenté au chapitre suivant de mieux intégrer la communication avec les deux autres facettes de l'espace fonctionnel des systèmes multi-utilisateurs, production et coordination.

6.6. Conclusion

Dans ce chapitre, nous avons vu qu'une architecture logicielle communique quatre perspectives sur le système qu'elle décrit : fonctionnelle, structurelle, allocation et

coordination. Une architecture n'est pas bonne ou mauvaise dans l'absolu mais répond à des critères et un contexte particulier. Pour réfléchir sur les modèles d'architecture pour les systèmes multi-utilisateurs, nous avons donc énoncé des caractéristiques générales attendues d'un modèle d'architecture pour ce type de systèmes, puis nous avons défini un espace-problème pour servir de cadre à notre analyse de modèles d'architecture existants. Nous avons passé en revue quelques modèles d'architecture représentatifs des modèles d'architecture pour les systèmes multi-utilisateurs et avons constaté que ceux-ci ne couvrent qu'incomplètement notre espace-problème. En termes de couverture fonctionnelle au sens du modèle du trèfle du collecticiel, les modèles d'architecture favorisent généralement l'un des trois espaces, production, coordination ou communication au détriment des deux autres.

Références

[Abowd 1994] G. D. Abowd. *Defining reference models and software architectural styles for cooperative systems*, Workshop on Software architectures for cooperative systems at CSCW'94, ACM Conference on Computer-Supported Cooperative Work, Chapel Hill, North Carolina, USA, 1994.

[Apple 1988] *The Knowledge Navigator*, vidéo. Apple Computer Inc., Cupertino, California, USA, 1988.

[Apple 1994] *Finder 7.5*. Logiciel pour Macintosh. Apple Computer Inc., Cupertino, California, USA, 1994.

[Bass 1994] L. Bass et G. Abowd. *Software Architecture: A Tutorial Introduction*, Software Engineering Institute, Carnegie-Mellon University, Tutorial, 1994.

[Bass 1992] L. Bass, R. Little, R. Pellegrino, S. Reed, R. Seacord, S. Sheppard et M. R. Szczur. *The UIMS Tool Developers' Workshop: A Metamodel for the Runtime Architecture of an Interactive System*, in *SIGCHI Bulletin*, 24(1), janvier 1992. pp. 32-37.

[Bentley 1994] R. Bentley, T. Rodden, P. Sawyer et I. Sommerville. *Architectural Support for Cooperative Multiuser Interfaces*, in *IEEE Computer*, 27(5), mai 1994. pp. 37-46.

[Bier 1991] E. A. Bier et S. Freeman. *MMM: A User Interface Architecture for Shared Editors on a Single Screen*, UIST'91, ACM Symposium on User Interface Software and Technology, Hilton Head, South Carolina, USA, 1991. pp. 79-86.

[Buxton 1990] W. Buxton et T. Moran. *EuroPARC's Integrated Interactive Intermedia Facility (IIIF): Early Experiences*, IFIP Conference on Multi-User Interfaces and Applications, Heraklion, Crète, 1990.

[Coutaz 1987] J. Coutaz. *PAC, an Implementation Model for Dialog Design*, Interact'87, IFIP Conference on Human-Computer Interaction, Stuttgart, 1987. pp. 431-436.

[Coutaz 1990] J. Coutaz. *Interface homme-ordinateur : conception et réalisation*, Dunod, Paris, France, 1990.

[Coutaz 1993] J. Coutaz, L. Nigay et D. Salber. *The MSM Framework: A Design Space for Multi-Sensori-Motor Systems*, in *Human-Computer Interaction, Third International Conference, EWHCI'93, Moscow, Russia, August 3-7, 1993, Selected Papers*. Lecture Notes in Computer Science n° 753, L. J. Bass, J. Gornostaev et C. Unger, (eds.). Springer-Verlag, Berlin, 1993. pp. 231-241.

[Croisy 1994] P. Croisy. *Modèle multi-agent et conception d'applications coopératives interactives*, IHM'94, Sixièmes journées sur l'Ingénierie des Interfaces Homme-Machine, Lille, France, 1994. pp. 139-144.

[Cypher 1991] A. Cypher. *EAGER: Programming Repetitive Tasks by Example*, CHI'91, ACM Conference on Human Factors in Computing Systems, New Orleans, Louisiana, USA, 1991. pp. 33-39.

[Dorner 1995] *Eudora 1.5.1*. Logiciel pour Apple Macintosh. Steve Dorner & Qualcomm, 1995.

[Ellis 1994] C. Ellis. *Keepers, Synchronizers, Communicators and Agents*, Workshop on Software architectures for cooperative systems at CSCW'94, ACM Conference on Computer-Supported Cooperative Work, Chapel Hill, North Carolina, USA, 1994.

[Freeman 1993] S. M. G. Freeman. *An Architecture for Distributed User Interfaces*. Ph.D Thesis, Darwin College, University of Cambridge, 1993.

[Garfinkel 1989] D. Garfinkel. *The Shared X Multiuser Interface User's Guide*, Hewlett-Packard, Research Report, STL-TM-89-07, 1989.

[Garlan 1993] D. Garlan et M. Shaw. *An introduction to software architecture*, in *Advances in Software Engineering and Knowledge Engineering.* World Scientific Publishing Company, 1993.

[Hill 1992] R. D. Hill. *The Abstraction-Link-View Paradigm: Using Constraints to Connect User Interfaces to Applications*, CHI'92, ACM Conference on Human Factors in Computing Systems, Monterey, California, USA, 1992. pp. 335-342.

[Karsenty 1994] A. Karsenty. *GroupDesign : un collecticiel synchrone pour l'édition partagée de documents.* Thèse de doctorat, Université d'Orsay Paris-Sud, 1994.

[Kazman 1994] R. Kazman, L. Bass, G. Abowd et M. Webb. *SAAM: A Method for Analyzing the Properties of Software Architectures*, ICSE-16, Sorrento, Italy, 1994.

[Koegel Buford 1994] J. F. Koegel Buford. *Middleware System Services Architecture*, in *Multimedia Systems*. J. F. Koegel Buford, (ed.) Addison-Wesley, New York, New York, USA, 1994. pp. 221-244.

[Krakowiak 1990] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill et C. Roisin. *Design and Implementation of an object-oriented strongly typed language for distributed applications*, in *Journal of Object-Oriented Programming*, septembre 1990.

[Malone 1986] T. W. Malone, K. R. Grant et F. A. Turbak. *The Information Lens: An Intelligent System for Information Sharing in Organizations*, CHI'86, ACM Conference on Human Factors in Computing Systems, 1986. pp. 1-8.

[Metral 1994] M. Metral. *MAXIMS: A Learning Interface Agent For Eudora (A User's Guide to the System)*, MIT Media Lab, Technical report, 1994.

[Nigay 1994] L. Nigay. *Conception et réalisation des systèmes interactifs: Application aux Interfaces Multimodales.* Thèse de doctorat, Université Joseph Fourier Grenoble I, 1994.

[Patterson 1994] J. F. Patterson. *A Taxonomy of Architectures for Synchronous Groupware Applications*, Workshop on Software architectures for cooperative systems at CSCW'94, ACM Conference on Computer-Supported Cooperative Work, Chapel Hill, North Carolina, USA, 1994.

[Patterson 1990] J. F. Patterson, R. D. Hill, S. L. Rohall et W. S. Meeks. *Rendezvous: An Architecture for Synchronous Multi-User Applications*, CSCW'90, ACM Conference on Computer-Supported Cooperative Work, Los Angeles, California, USA, 1990. pp. 317-328.

[Roseman 1992] M. Roseman et S. Greenberg. *GROUPKIT: A Groupware Toolkit for Building Real-Time Conferencing Applications*, CSCW'92, ACM Conference on Computer-Supported Cooperative Work, Toronto, Canada, 1992. pp. 43-50.

[Shaw 1995] M. Shaw et D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*, Prentice Hall, 1995.