Chapitre 2 - Architecture logicielle et construction d'applications client-serveur

«Toute technologie suffisamment avancée est indiscernable de la magie» (Arthur Clarke)

Résumé

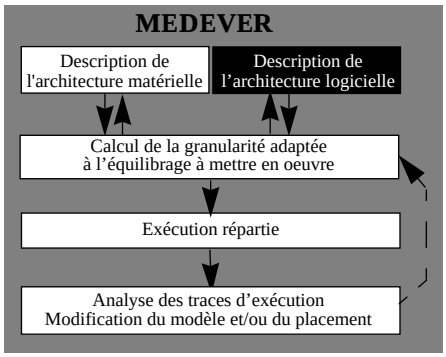
La méthodologie MEDEVER a été conçue pour s'appuyer sur une description d'architecture logicielle d'application client-serveur. Nous commençons notre étude par une définition des différents termes utilisés pour la description générale d'une architecture. Puis, nous orientons notre étude sur la définition des architectures de type client-serveur.

Cette description architecturale doit de manière statique gérer la granularité des entités logicielles et leur schéma de coopération. Elle est ensuite utilisée pour calculer le placement statique et garder la trace des évolutions dynamiques des entités logicielles lors de l'exécution (Etape 3 et 4 de notre méthode). Ce chapitre est dédié à l'étude des langages, des outils et des environnements susceptibles de fournir ces services, toujours en suivant deux axes d'étude : un axe langage et un axe système. Le premier axe sert à l'étude des mécanismes de construction d'architectures logicielles à travers les langages de programmation objets concurrents. Ces derniers ont été choisis car ils correspondent parfaitement aux types d'applications que l'on désire prendre en compte dans MEDEVER. Une application étant alors constituée de composants logiciels répartis communiquants. Le second axe étudie les diverses infrastructures et plate-formes facilitant la mise en oeuvre d'architectures client-serveur. Enfin, le dernier chapitre est consacré à la synthèse de ces études et à la recherche d'un langage de description d'architecture logicielle adapté aux besoins de génération, d'exécution et de déboggage des applications parallèles et réparties.

Apports scientifiques

Dans ce chapitre, nous proposons un état de l'art des techniques de description d'architectures logicielles. Nous avons tenté de classer et de définir chacun des termes liés à cette description (tels que les gabarits de conception, le squelettes d'implémentations, etc.). Nous avons aussi tenté de comprendre à travers l'étude des outils et des langages de description d'applications client-serveur comment était réalisée la construction d'une application répartie ou parallèle. Nous avons ainsi pu mettre en exergue l'importance prise par les technologies objets (et notamment les objets répartis) et les technologies médiateurs (*middleware*).

Plan



Description d'architectures logicielles

Définition et description d'une architecture logicielle
Définition d'une architecture logicielle de type client-serveur et du middleware
Construction selon un axe Langage Les langages objets concurrents Les langage d'acteurs Les langages de configuration Les langages de description d'architectures matérielles Les langages d'interconnexion de modules Les langagesde définition d'architectures logicielles
Construction selon un axe système Les architectures logicielles basées sur des systèmes ouverts Les plate-formes d'intégration Les architectures à base de composants logiciels Les architectures logicielles basées sur des ORB Les architectures logicielles basées sur des systèmes répartis

Mots clefs

Architectures logicielles, applications réparties, bus logiciel, client-serveur, langage objet concurrent, systèmes ouverts, objets répartis, systèmes répartis.

Les concepts d'architecture logicielle et de programmation structurée sont difficiles à mettre en oeuvre pour des applications complexes. La première étape dans la construction d'une application complexe consiste à ne pas la concevoir comme un tout, mais comme un multitude de composants qui sont liées dans un environnement donné. Cette approche modulaire peut être à la fois de type *Top-Down* (du tout on extrait les parties) ou *Bottom-Up* (on commence par les parties et on les agrège).

Si la modularité facilite le découpage fonctionnel des composants de l'application, elle entraîne en contrepartie la gestion du contrôle et de l'interopérabilité des composants dans le cas d'applications réparties et/ou parallèles. Au niveau des composants, il faut aussi prévoir leur gestion (naissance, vie et mort), des schémas d'interactions internes (à l'intérieur du composant) et externes (interfaces avec l'environnement). Enfin, la recherche de la granularité de ces composants (qu'on représente souvent par leur taille en octets et/ou le nombre de services qu'ils offrent) n'est pas triviale et dépend à la fois de critères fonctionnels, mais aussi de critères opérationnels [Ozanne 96]. Ces critères opérationnels conditionnent l'implantation de tels composants.

Les techniques d'implantations de composants sont nombreuses et basées sur des composants logiciels fixes (tels que des objets) ou mobiles (par exemple des agents). Dans ce dernier cas, la distribution de code est alors un pré-requis à l'exécution répartie d'une application basée sur des objets mobiles qui coopèrent et se déplacent dans un espace d'adressage réparti. La granularité de ces composants dépend fortement des langages et des compilateurs utilisés. Il en va de même de la granularité du contrôle, car une grande partie du comportement et de la sémantique de communication de ces composants est définie à travers leur environnement, donc de manière contextuelle. On en déduit alors que la granularité de ces objets est fonction du contexte et du temps.

La mise en chantier d'une application répartie passe donc préalablement par la compréhension du canevas cohérent de coopération entre plusieurs composants et de l'environnement cible. Ainsi, par exemple, dans les architectures actuelles supportant les communications GSM, la cellule est l'unité de base de cohérence. Cette cohérence est assurée si on ne dépasse pas une certaine portée et un certain intervalle de temps. Le grain du temps est caractérisé par une tranche consistante d'histoire durant laquelle un canevas de coopération entre composants est cohérent. Ce canevas forme alors la base de l'architecture logicielle de l'application dans laquelle on tente de trouver le grain de parallélisme adapté à chaque composant en respectant le grain de cohérence nécessaire pendant une portée temporelle.

C'est pourquoi, la conception d'une architecture logicielle, bien que d'une importance capitale pour la réussite d'un projet informatique, est encore trop souvent liée au savoir faire d'une personne (ou d'un groupe de personnes) spécialiste(s) d'un domaine précis. Comment alors exprimer ce savoir-faire, tout en l'intégrant dans une architecture aux contraintes multiples et qui évoluent au cours du temps ? La réponse à cette question n'est pas unique, car au concept de modularité déjà cité s'ajoutent :

- **la réutilisabilité** qui est la capacité à rendre générique des composants et à construire des boites noires susceptibles de fonctionner avec des langages et/ou des environnements donnés. Généralement, des composants réutilisables possèdent des interfaces d'accès clairement définies, dont la bonne utilisation est liée au respect d'un mode d'emploi ou d'un contrat. Ainsi, par exemple, tout composant Opendoc, fonctionne s'il se trouve dans un container, (même s'il doit s'y adapter) et tout composant Java Beans fonctionne dans une BeanBox.
- **la maintenabilité** qui est la capacité de modifier et de maintenir en production une application sur une période de vie assez longue. Une architecture bien conçue doit aussi être maintenable, tout au long de son cycle de vie, c'est pourquoi, la prévision de l'intégration d'extensions nécessaires à l'architecture et la correction des erreurs sont primordiales. Sinon, tous les bénéfices du développement originel risque d'être perdus par des rajouts indispensables, mais peu conformes à l'architec-

ture. La vitesse d'intégration de nouveau composant et de correction des erreurs varie aussi en fonction du degré de réutilisabilité des composants de l'architecture.

- **la performance** qui est souvent caractérisée par le temps mis par une application à répondre à une requête donnée. Les performances d'une application dépendent de l'architecture logicielle choisie, mais aussi de son environnement d'exécution, de son implémentation et de la puissance des infra-structures utilisées (puissance et taux d'occupation des machines et débit du réseau utilisé). Néanmoins, chaque architecture doit, pour certaines fonctionnalités requises, en optimiser les performances.

La granularité de chacun des éléments d'un système influe à la fois sur sa modularité, sur sa réutilisabilité, sur ses performances et sur sa maintenabilité. Un composant à granularité trop grande sera difficilement réutilisable du fait du nombre important de fonctions (et donc difficilement maintenables) qu'il offre. Par contre un composant de granularité trop faible peut entraîner de faible performance et augmenter les coûts de gestions et la complexité de description du modèle global (la complexité du schéma de coopération entre objets croissant avec le nombre de petits objets présents).

Contexte de l'étude

Notre propos n'est pas de définir une nouvelle démarche de développement d'architecture logicielle, mais plutôt de compléter les démarches de développement existantes en prenant en compte des composants logiciels de granularité variables. Nous cherchons à renforcer les éléments de description de l'architecture, pour pouvoir en déduire des informations susceptibles de faciliter la génération, l'exécution et le déboggage des composants logiciels constituant une application parallèle ou répartie. Pour cela, nous nous sommes intéressés aux architectures logicielles utilisant un schéma d'interaction client-serveur. Le choix du paradigme client-serveur a été réalisé pour deux raisons majeures :

- la migration des applications des grands systèmes vers des réseaux de stations de travail interconnectées (*Downsizing*) est réalisée en transformant des architectures logicielles monolithiques en une collection d'architectures logicielles de types client-serveur.
- des travaux sur la conception d'applications client-serveur au niveau des modèles d'architecture fonctionnelle et opérationnelle ont été déjà été menés dans notre équipe [Ozanne 96] et méritaient d'être complétés et étendus.

Dans le domaine des architectures logicielles client-serveur réparties et/ou parallèles, nos travaux se sont concentrés sur deux axes :

- **l'axe langage** : les langages de programmation d'applications réparties et/ou parallèles et les environnements d'exécutions qui les accompagnent. **Nous avons d'ailleurs fait le choix de ne prendre en compte que les langages orientés objets, qui intègrent le mieux la notion de composant logiciel mobile (tel que Java par exemple) et de concurrence.** Nous verrons par la suite que ces langages induisent des composants logiciels de granularité variable et que l'utilisation d'une méthode comme MEDEVER apporte de nombreux avantages.
- **l'axe système** : les systèmes d'exploitation et les plate-formes d'exécution offrent des services susceptibles de contraindre ou d'étendre l'architecture logicielle d'une application. Nous verrons que les composants logiciels gérés sont de granularité variable et qu'ils sont susceptibles d'évoluer dynamiquement en cours d'exécution. Cette étude vise avant tout à évaluer les services qu'il faut prendre en compte dans les étapes 3 et 4 de notre méthode MEDEVER (qui sont respectivement dédiées à l'exécution répartie et à la collecte et à l'analyse des traces).

Le premier chapitre définit la notion d'architecture logicielle et présente un état de l'art des méthodes de description récentes. Le second chapitre introduit les architectures logicielles de type client-serveur, leur classification, les médiateurs (*middleware*) qu'elles

utilisent et les outils de développement nécessaires. Le troisième chapitre présente un état de l'art des langages de programmation orientés objets d'application client-serveur (axe langage) et le chapitre quatre un état de l'art sur les architectures client-serveur basées sur des systèmes ouverts (axe système). Enfin, le dernier chapitre est consacré à la synthèse de ces études et à la recherche de langages et d'outils de description d'architecture logicielle adapté à notre méthode MEDEVER.

1. Définition et description d'architectures logicielles

Le rôle principal d'une architecture est d'entraîner une démarche de conception, à la fois formelle (permettant par le calcul mathématique de valider ses propriétés) et opérationnelle (lors de sa conception, de sa mise au point et de sa maintenance). La définition d'une architecture logicielle passe donc par la réponse à la question suivante : Quelles règles doit-on suivre pour réussir à développer des systèmes qui répondent à toutes les exigences fonctionnelles (les services à rendre) et non fonctionnelles (performance et extensibilité du système par exemple) ?

Globalement, une architecture logicielle peut-être vue comme :

- un ensemble de briques de base : *les composants* ;
- un ensemble de règles d'utilisation de ces briques de base : *le mode d'emploi ou gabarit d'utilisation* ;
- un ensemble de recettes et de conseils pour combiner et gérer les interactions entre les différents composants : *le savoir faire* ;
- un ensemble de principes directeurs qui aident le concepteur dans ses décisions si besoin est : *les assistants.*

Ceci nous amène alors à la définition générale d'une architecture logicielle :

Définition 3 : Architecture logicielle

Une architecture est une infrastructure composée de modules actifs, d'un mécanisme d'interaction entre ces modules et d'un ensemble de règles qui gouvernent cette interaction [Boasson & al. 95].

Lors de la définition d'une architecture logicielle par une approche *Top-Down*, on tente :

1) de trouver une décomposition pertinente de l'application en un ensemble de composants logiciels ;
2) de répartir et d'allouer les exigences fonctionnelles et non fonctionnelles sur les composants logiciels créés (recherche des services à offert et attendu et des interfaces d'accès à ces services) ;
3) de valider les éléments de conception (chaque composant logiciel est donc d'abord validé indépendamment des autres) ;
4) de composer les éléments de conception validés (en utilisant leur mode d'emploi) ;
5) de valider la composition des éléments de conception (par validation de propriétés globales sur l'architecture).

A ces cinq points, présentés dans [Ozanne 96], on peut rajouter les points suivants :

6) de distribuer physiquement les composants de l'application (déploiement);
7) d'ajuster les performances à celles requises, quitte à recommencer au point 1.

Ceci nous conduit alors à raffiner la définition d'une architecture logicielle répartie.

Définition 4 : Architecture logicielle répartie

Une architecture logicielle répartie est un ensemble organisé d'entités de calcul, appelées composants, et d'entités de description des interactions entre ces composants appelées connecteur [Garlan & al. 93].

Un composant est une abstraction statique qui dispose de ports d'interconnexion, d'une longue durée de vie et d'un stockage persistant [Nierstrasz & al. 96]. Un composant est une abstraction, car comme une boîte noire, il cache sa logique et ses particularités d'implémentation. Il dispose de ports d'interconnexion pour interagir et communiquer avec d'autres composants via des échanges de paramètres ou de messages. Plus on se situe en amont du cycle de développement et plus le niveau d'abstractions des entités manipulées sera fort.

Nous étudions dans la suite de cette section comment décrire une architecture logicielle. Nous présentons dans un premier temps le principe de la séparation des préoccupations, qui simplifie la conception d'une architecture, en séparant le noyau fonctionnel de ses facettes techniques. Nous montrons ensuite qu'il existe pour chaque type d'architecture des styles architecturaux adaptés. Nous en déduisons alors, que ce n'est pas une mais deux descriptions d'architectures logicielles qu'il est nécessaire de gérer. La première, de haut niveau, s'abstrait des choix d'implémentation et conduit à la création de gabarits de conception réutilisables. La seconde, plus proche de l'implémentation, manipule des squelettes d'implémentation prêt à l'emploi. Nous proposons ensuite, un bref état de l'art sur les différents types de langages de description d'architectures logicielles. Enfin, nous présentons les conclusions que nous avons tiré de cette étude, en nous replaçant dans le cadre de notre méthode MEDEVER.

1.1. Séparation des préoccupations et vues sur l'architecture

La description de composants, de connecteurs et de leurs relations est en général fonction de la granularité des éléments manipulés, mais aussi des axes d'intérêt choisis. Ces axes d'intérêts sont des vues partielles sur une architecture et sont utilisés pour étudier l'architecture selon des critères spécifiques. L'utilisation de ces critères spécifiques est liée à une volonté, la séparation des préoccupations (*separation of concern*) [Hürsch & al. 95].

La séparation des préoccupations consiste à découpler les facettes techniques, afin de faciliter la conception du noyau fonctionnel. Par ce moyen, on sépare formellement l'algorithme principal et les questions annexes telles que la gestion de la distribution, la gestion des synchronisations ou l'organisation des classes d'objets manipulés et leurs interactions [Kiczales & al. 97]. La description, la mise en place de l'architecture et sa maintenabilité en sont alors facilitées. Ainsi, la conception de l'architecture d'un éditeur de texte partagé, consiste à séparer la logique applicative (la gestion du texte, des polices, etc.), des services de travail coopératif (l'accès et de partage des documents par exemple).

Le niveau conceptuel et le niveau implémentation forment deux niveaux de séparation des préoccupations. Le niveau conceptuel fournit une définition claire et une identification conceptuelle de chaque préoccupation qui se distingue des autres. On s'assure alors que chaque concept individuel est primitif, dans le sens ou il n'est pas dû à la composition de plusieurs concepts. Le niveau de l'implémentation fournit une organisation adéquate qui isole les préoccupations. L'idée est de séparer les blocs de code qui adressent les différentes préoccupations et fournissent un couplage faible entre elles. Le passage d'un niveau à un autre se fait par l'utilisation de langage de programmation et est souvent automatisé (par des techniques de prototypage par exemple) [Budinski & al. 96].

La séparation des préoccupations peut être réalisée par l'utilisation de vues.

Définition 5 : Une vue

Une vue définit un raffinement d'une architecture, d'un environnement ou d'un système en fonction d'un axe d'intérêt.

L'architecture est donc étudiée selon un point de vue particulier (aussi, appelée une facette), ce qui facilite sa définition et sa mise eu point. Les modèles d'architectures ouvertes tels qu'ANSA, ensuite repris dans ODP, sont des exemples d'utilisation de vues.

Ces modèles se basent sur un ensemble de cinq vues, chacune offrant des concepts et des règles représentant :

1) le modèle de l'entreprise *(Enterprise Model)* pour exprimer les limites du système, ses règles et son utilité. Il décrit l'organisation de l'entreprise et ses changements;
2) le modèle d'information *(Information Model)* pour exprimer la signification de l'information répartie et des processus qu'on lui applique;
3) le modèle de traitement *(Computational Model)* exprimant la décomposition fonctionnelle de l'application en unités de distribution;
4) le modèle d'ingénierie *(Engineering Model)* décrivant les composants et les structures dont on a besoin pour supporter la distribution (les fonctions de l'infra-structure);
5) le modèle technologique *(Technology Model)* qui décrit la construction de l'application en terme de composants, avec des règles de conformance pour sa réalisation.

Il existe des modèles d'architecture de plus bas niveaux et moins complexes à mettre en oeuvre. Le modèle de la société Rationale Software [Kruchten 95] est décomposé, lui aussi, en cinq vues :

1) une vue logique (*logical view*) qui décrit les besoins fonctionnels, c'est à dire les services à rendre aux utilisateurs.
2) une vue procédurale (*process view*) qui décrit les aspects de concurrence, de synchronisation, de tolérance aux fautes et de performance.
3) une vue de distribution (*physical view*) qui décrit le placement du logiciel sur le matériel et décrit les aspects répartis.
4) une vue développement (*development view*) qui décrit l'organisation statique des composants logiciels et de l'environnement de développement.
5) les scénarios qui sont des exemples de mise en pratique des quatre vues précédentes.

L'utilisation de toutes les vues n'est pas obligatoire. Chacune de ces vues utilisent ses propres notations et un ou plusieurs styles architecturaux. Chaque architecte gère donc différentes vues contenant des éléments qui sont adaptés à son métier. Il en résulte donc une simplicité de conception (chaque vue isolant une seule facette du problème) et un meilleur travail en groupe (chaque spécialiste d'une facette d'un système concevant son modèle en utilisant un style architectural adapté et pas forcément uniforme à toutes les vues).

1.2. Style architectural

Un style architectural est un véhicule générique d'aide à l'expression de solution structurelles [Coutaz & al. 96]. Il comprend un vocabulaire d'éléments conceptuels (les composants et les connecteurs), impose des règles de configuration entre les éléments du vocabulaire (ensemble de contraintes) et véhicule une sémantique qui donne un sens (non ambiguë) à la description structurelle [Shaw & al. 95 & 96]. Ainsi, un système est

vu de manière abstraite comme une configuration des composants et des connecteurs [Perry & al. 92].

Les composants sont les entités de stockage et de calcul principales du système (tels que les bases de données, les serveurs et les outils logiciels). Les connecteurs déterminent les interactions entre les composants (tels que le protocole client-serveur et les communications avec RPC). Ces abstractions sont couramment représentées graphiquement par des boites (les composants) et des arcs (les connecteurs) qui les relient. Des notations formelles commencent à voir le jour [Allen & al. 94] ou [TSE 95].

Chaque style véhicule des propriétés logicielles spécifiques adaptées à des critères retenus pour un système. Ainsi, dans une organisation client-serveur :

- un serveur représente un processus qui fournit des services à d'autres processus appelés clients;
- le serveur ne connaît pas à l'avance l'identité et le nombre de clients;
- le client connaît (ou peut trouver via un autre serveur) l'existence du serveur.
- le client accède au serveur, par exemple via des appels de procédures distants.

La description de systèmes complexes impliquent la combinaison de styles. Cette combinaison est réalisable de trois manières [Garlan & al. 93] :

1) hiérarchiquement si le composant ou le connecteur d'un système, organisé selon un style, bénéficie d'une structure interne développée dans un style différent.
2) en laissant un composant utiliser des connecteurs provenant de différents styles architecturaux (notamment pour la gestion de l'hétérogénéité).
3) en décrivant complètement un niveau d'une architecture dans un style différent.

1.2.1. Cohabitation des styles et incompatibilités architecturales

La cohabitation des styles offre une grande richesse de conception, mais conduit parfois à des incompatibilités architecturales. Ces incompatibilités sont liées à des hypothèses erronées faites par un composant sur son environnement. Les conflits sont en partie dûs à des présomptions architecturales sur un composant ou un environnement qui sont souvent implicites et difficiles à analyser avant la construction du système.

Ces présomptions sont liées à [Garlan & al. 95] :

- **la nature des composants** : incompatibilité d'infrastructure, du modèle de contrôle et/ou de données des composants.
- **la nature des connecteurs** : incompatibilité au niveau des protocoles et/ou du modèle de données utilisés pour les communications.
- **la structure de l'architecture globale** : hypothèses erronées concernant la topologie et la présence ou l'absence de composants et de connecteurs.
- **le processus de construction de l'architecture** : l'instanciation des composants et la combinaison des briques de bases ne sont pas effectuées dans le bon ordre.

La réponse à ces incompatibilités architecturales n'est pas unique. Dans la suite de ce paragraphe, nous traitons l'exemple des incompatibilités de style aux interfaces et nous présentons brièvement les solutions possibles.

1.2.2. Un exemple : résoudre les incompatibilités de style aux interfaces

Au niveau de ses interfaces, chaque composant réutilisable peut interagir [Garlan & al. 95] :

- avec des composants d'un niveau applicatif supérieur qui réutilisent ce même composant comme leur propre infrastructure (*interface at the top*).
- avec des infrastructures applicatives de plus bas niveaux que lui (*interface at the*

bottom).

- avec des composants qui se trouvent au même niveau d'abstraction (*side interface*).

L'interopérabilité des interfaces entre des services offerts et requis se fait donc à trois niveaux d'abstraction différents. Le modèle de référence de l'OSI est un exemple de tentative de normalisation des communications entre composants appartenant à des couches de communication de niveaux différents.

De manière générale, l'utilisation du paradigme d'appel de procédure à distance (*Remote Procedure Call*) [Birrell & al. 84] est la solution la plus simple à ce problème. Ce paradigme est basé sur une hypothèse implicite : les procédures du serveur offrent exactement les fonctionnalités requises par le client. Dans le cas contraire, il faut adapter l'appel de service pour que la procédure puisse être appelée avec les bons paramètres dans un codage cohérent pour le serveur.

Polylith [Purtilo & al. 91 et Purtilo 94] est un exemple de cette approche. L'intégration de composants hétérogènes dans le système Polylith est réalisée grâce à NIMBLE, un langage de génération de passerelles d'interfaces. Un programmeur décrit alors avec des directives NIMBLE, la correspondance entre les paramètres d'un service requis et offert. Le système génère alors automatiquement le code qui réalise les adaptations au moment de l'exécution. Il faut néanmoins modifier le style, s'il n'existe pas de correspondance une à une entre les interfaces.

Une autre approche consiste non pas à générer un module d'adaptation des interfaces à la volée, mais au contraire de standardiser l'accès aux interfaces. Ainsi, à partir des interfaces générées dans l'environnement d'exécution local, on accède aux services disponibles dans d'autres environnements. C'est la cas de SLI [Wileden & al. 91], qui définit la compatibilité des types en spécifiant les propriétés des objets et en cachant les différences de représentation. Les types partagés par des programmes qui interopèrent sont décrits en respectant un modèle de type unifié (*Unified Type Model ou UTM*). L'implémentation d'un nouveau service avec SLI passe par la spécification d'une interface de service avec UTM et fournit de nouvelles définitions de type et un langage de liaison. Une fois ce travail fait, la génération automatique de l'interface est réalisée, dans le langage d'implémentation spécifié.

On retrouve cette approche de standardisation des interfaces dans CORBA, qui spécifie les interfaces à l'aide d'un langage appelé IDL (*Interface Definition Language*) et génère automatiquement les modules de code clients et serveurs (appelés des talons ou *stubs*) d'invocation des interfaces. L'inconvénient majeur de SLI et CORBA est que l'interface de service doit être incluse dans le code des clients. En cas de modification des interfaces du serveur, il faut modifier tous les clients.

La résolution des incompatibilités de style au niveau des interfaces passe actuellement par l'utilisation de passerelles d'interfaces ou par la standardisation des interfaces [Konstantas 96]. L'essor de la programmation orientée objet a entraîné des recherches d'interopérabilité non plus au niveau des procédures, mais directement au niveau des objets et à l'encapsulation des constructions (*wrapper construction*) [Yellin & al. 94].

1.2.3. Synthèse

L'utilisation de styles pour décrire les éléments d'une vue peut conduire à des incompatibilités architecturales. Plutôt que de résoudre par des méthodes spécifiques chaque type d'incompatibilité architecturale, il est préférable lorsque cela est possible d'essayer de trouver une solution globale. On cherche alors une solution qui rend explicite les choix et les hypothèses de conception qui sont implicites dans la tête de l'architecte travaillant dans un domaine donné (une facette du problème). Le but étant finalement de réduire au minimum les choix architecturaux implicites non documentés et non discutés. Pour cela, on tente de codifier et de disséminer des principes et des règles de composition des com-

posants logiciels à l'intérieur d'un style, entre styles et parfois même entre vues (mais ce dernier cas est plus rare car on réduit de manière drastique le pouvoir de description).

En fait, l'expérience révèle l'existence de problèmes de conception récurrents auxquels les architectes répondent systématiquement par l'utilisation d'un ou plusieurs styles. C'est la capitalisation de cette connaissance que supporte les gabarits de conception.

1.3. Gabarit de conception

Le premier à avoir utilisé la notion de gabarit de conception est Alexander [Alexander 79], mais dans le domaine de l'architecture des bâtiments. Son idée était de proposer des composants de base réutilisables (comme des portes des fenêtres, des colonnes) et des règles d'assemblage de ces composants. Ce n'est que récemment que Coad [Coad 92] fit un lien entre les gabarits d'Alexander et les besoins actuels dans le domaine de l'architecture de logiciels. Ce lien est aussi le résultat de la maturité atteinte dans le domaine des technologies orientées-objets. D'ailleurs, pour Coad, les gabarits de conception sont identifiés en observant les briques de bases les plus élémentaires, à savoir l'objet, la classe et leurs relations.

Définition 6 : Un gabarit de conception (Design Pattern)

Un gabarit de conception décrit un problème de conception récurrent, propose un schéma de solution prédéfini et inclut des règles heuristiques indiquant les clauses de son applicabilité [Gamma & al. 94].

Cette définition des gabarits de conception est liée à l'observation de systèmes déjà construits. Les gabarits de conception sont donc descriptifs et destinés à des architectes de systèmes. Ils ne sont par contre pas destinés à être exécutés ou analysés par des ordinateurs, ce sont **des gabarits de conception passifs**, aussi appelés gabarits de conception de Gamma [Gamma & al. 93]. Un gabarit de conception garde la trace de la démarche de conception et des choix alternatifs qui ont été envisagés.

Description de gabarits de conception

Il n'existe pas de norme pour la description d'un gabarit de conception. Néanmoins, le Tableau 16 présente des attributs qui apparaissent souvent dans la littérature [Coplien & al. 95] et [Gamma & al. 93].

Tableau 16: Attributs de description d'un gabarit de conception passif

Attributs	Description des attributs
Nom	Nom du gabarit de conception.
Raison d'être	Description non abstraite des motivations de la création du gabarit.
Applicabilité	Ensemble des règles indiquant dans quel contexte il s'applique.
Classification	Aide à la recherche d'un gabarit adéquat si on n'est pas un expert.
Participants	Liste des participants et des collaborateurs.
Représentation	Représentation graphique du gabarit.
Méthodologie	Étapes pour construire le gabarit.
Exemples	Exemple d'utilisation de ce gabarit.
Contraintes	Contraintes pour l'application correcte du gabarit.
Voir Aussi	Liens vers d'autres références ou gabarits.

Gabarits de comportements : les contrats

Des descriptions formelles des éléments d'un gabarit de conception existent, mais ne sont pas destinées à la vérification de propriétés. Elles forment des gabarits décrivant le comportement (*behavioral pattern*) de chaque participant, pour lesquels des obligations contractuelles doivent être définies [Holland 92].

Définition 7 : Un contrat

Un contrat [Helm & al. 90] est une construction définissant explicitement les interactions et les responsabilités entre un groupe d'objets.

Une fois la description des services offerts par chacun des participants réalisée, il reste à déclarer quels sont les invariants du contrat et sous quelles condition la mise en place de ce contrat est possible. Ainsi par exemple, chaque participant à des obligations de types (lors de l'instanciation de variables et de types) et des obligations causales (les actions et les conditions associées à chaque obligation de types). Il n'existe pas de consensus pour l'utilisation de langages spécifiques pour la description de contrats. On retrouve par exemple la notion de contrat dans le langage Eiffel [Meyer 92b]. L'approche formelle n'est d'ailleurs pas toujours applicable de manière simple. Les notations utilisées demandent une exactitude que l'architecte ne maîtrise pas obligatoirement et noient les aspects essentiels de l'architecture dans des détails de bas niveaux (en fait proche des langages de programmation). La notion de contrat rajoute des attributs à la description d'un gabarit. Nous en présentons quelques un dans le Tableau 17.

Tableau 17: Attributs de description supplémentaires à ceux d'un gabarit de conception passif

Attributs	Description des attributs
Collaboration	Description des collaborations entre participants pour assumer les responsabilités.
Comportement dynamique	Illustration du comportement typique, utilise en général des diagrammes OMT [Rumbaugh & al. 95].
Implémentation	Directives d'implémentation du gabarit (pseudo-code ou langage de programmation).
Variantes	Subtilités d'utilisation du gabarit pour traiter des cas spécifiques.

Catalogues de gabarit

Des catalogues de gabarits commencent à voir le jour et sont composés de gabarits logiquement organisés. On trouve ainsi par exemple des catalogues de gabarits de création d'objets, de structures et de comportement [Gamma & al. 94], [Coad & al. 95] et [WWW-Pattern 97]. Ces catalogues présentent une collection de solutions indépendantes à des problèmes de conception communs. La prochaine étape annoncée est celle du regroupement des gabarits inter-corrélés pour former des langages de gabarits.

Langages de gabarits

Définition 8 : Langage de gabarits

Un langage de gabarits [Gabriel 94] est représenté par une collection structurée de gabarits construits les uns au dessus des autres pour transformer les besoins et les contraintes en architecture [Coplien & al. 95 &Vlissides & al. 96].

Ainsi, un gabarit peut être utilisé pour augmenter à son tour un autre gabarit (cf. Figure 10).

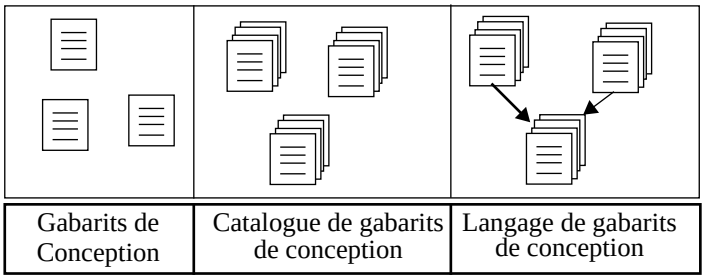


Figure 10 : Des gabarits de conception aux langages de gabarits.

Ainsi, par exemple le langage de gabarits de Buschmann [Buschmann & al. 96] offre un modèle architectural basé sur la granularité des composants et spécifie les relations entre gabarits. Son modèle est décomposé en trois catégories : les idiomes, les gabarits de conception et les gabarit d'architecture. Un idiome décrit comment implémenter un composant particulier d'un gabarit, ses fonctionnalités ou ses relations avec d'autres composants. Dans le modèle objet, le niveau idiome concerne les objets, les classes et la manière de les implémenter. Le niveau gabarit de conception décrit comment créer des composants logiciels de plus haut niveau à partir de ceux qui sont fournis [Buschmann & al. 96]. Enfin, le gabarit d'architecture exprime un paradigme fondamental pour la structuration de systèmes logiciels. Il fournit un ensemble de sous-systèmes prédéfinis ainsi que des règles et des directions pour organiser les relations entre eux. Un gabarit d'architecture coordonne les activités entre diverses applications (interopérabilité et partage de services entre applications) notamment par l'utilisation de l'IDL de l'OMG.

Gabarits de code

Les langages de gabarits de conception utilisables pour la génération automatique d'architecture logicielle, sont appelés **génératifs** (*generative pattern*) [Budinsky & al. 96]. Ils s'appuient quelquefois sur un modèle de programmation basé sur des gabarits de code (*coding pattern*) [Coplien 92]. Un gabarit de code décrit comment une structure particulière ou une séquence d'actions est réalisable en utilisant les mécanismes spécifiques d'un langage de programmation [Wild 96]. Ils démontrent des combinaisons utiles de concepts du langage et forment une base de standardisation de la structure du code source et du nommage.

Les gabarits de code sont classés en trois catégories [Lieberherr & al. 95] :

1) les gabarits de propagation (*propagation pattern*) qui définissent des opérations sur les données à partir d'algorithmes.
2) les gabarits de transport (*transportation patterns*) qui rendent abstrait le concept de paramétrisation. Ils sont utilisés dans les gabarits de propagation pour faire transiter les paramètres en entrée et en sortie des opérations.
3) les gabarits de synchronisation qui définissent la synchronisation entre objets dans des applications concurrentes.

Ainsi, chaque tâche à accomplir est spécifiée par un gabarit de propagation, ainsi qu'avec d'autres gabarits utiles. Cet ensemble de gabarits associé à un graphe de classe (définissant la structure de l'objet sur lequel le gabarit s'applique) est ensuite utilisé par un compilateur de gabarits pour obtenir un programme orienté objet [Budinsky & al. 96].

Synthèse

Les gabarits de conception concentrent un savoir-faire et sont regroupés pour définir des catalogues plus ou moins complexes. Ainsi, par exemple, l'ouvrage de [Mowbray & al. 97] présente un catalogue de gabarits de conception pour la construction d'applications CORBA et [Mahmoud 97] pour la construction de programmes en langage Java. Ces catalogues de gabarits se reflètent au niveau technique par la réutilisation quasi systéma-

tique de morceaux de programmes utilisés comme briques de construction. On se rapproche alors de la notion de composants métiers qui avait été mise en oeuvre dans Taligent. Dans cette optique, la réutilisation de gabarit de conception seule ne suffit pas, car elle n'a trait qu'à l'architecture et à la conception et absolument pas au code. Pour être productifs, les développeurs doivent aussi réutiliser les algorithmes, les interfaces, les détails d'implémentation, etc. Les squelettes d'implémentations (*framework*) répondent de manière opérationnelle à un besoin de réutilisation du logiciel.

1.4. Squelette d'implémentation

L'utilisation de langages objets, tel que SmallTalk [Goldberg & al. 83, Decouchant 86], ont d'abord permis l'utilisation de squelettes d'implémentation [Deutsch 89]. Un squelette d'implémentation est alors défini comme un ensemble de classes coopérantes qui déterminent un schéma de conception réutilisable pour une classe de logiciel [Johnson & al. 88]. Par la suite, le concept s'est généralisé pour aboutir à la définition ci-dessous.

Définition 9 : Squelette d'implémentation, aussi appelé motif (framework)

Un squelette d'implémentation est un environnement logiciel qui est conçu pour simplifier le développement d'applications et la gestion de systèmes pour un domaine d'application donné [Bernstein 96]. Il est défini par une API, une interface utilisateur et divers outils.

Les squelettes d'implémentation sont donc des applications semi-complètes et adressent un domaine d'application particulier. Les classes d'objets par contre sont en général indépendantes d'un domaine (on trouve ainsi par exemple des classes de gestion de tableaux, de nombres complexes, etc.).

Dans le monde objet, un squelette d'implémentation [Gamma & al. 94] définit :

1) la structure de l'application ;
2) le partitionnement de l'architecture en classes et en objets, on obtient alors un squelette d'implémentation fonctionnel. Un squelette d'implémentation fonctionnel définit et organise des services logiques et des fonctions qui satisfont aux exigences d'un système. On parle aussi, dans ce cas, d'architecture générique ou réutilisable ;
3) les responsabilités de chacun des objets ;
4) la collaboration entre objets et classes ;
5) le flot de contrôle.

C'est pourquoi, la réutilisation à ce niveau entraîne une **inversion du contrôle entre l'application et le logiciel sur lequel elle est basée**. L'approche conventionnelle du développeur d'applications consiste à écrire le corps de son application, puis à faire appel à du code contenu dans des bibliothèques externes pour accéder à des fonctionnalités pré-programmées. Avec un squelette d'implémentation, on réutilise le corps d'une application et on écrit le code que le squelette d'implémentation appelle. Une application est alors développée en héritant et en instanciant des composants de squelettes d'implémentations. Les applications ainsi crées ont des architectures similaires, ce qui réduit le temps de développement, facilite la maintenance et permet la réutilisabilité. Par contre, cela restreint le développeur au niveau de sa créativité. L'utilisation de classes se fait uniquement par invocation des méthodes des classes.

Nous présentons dans la section suivante des exemples de langages de description d'architectures logicielles par composition de composants logiciels (homogènes ou hétérogènes).

1.5. Langages de description d'architectures logicielles

Lors de la création d'une application répartie, on peut séparer la structure de l'application en instances de processus interconnectés d'une part et en algorithmes décrivant les différents types de processus d'autre part. Ainsi, chaque processus est décrit comme indépendant du contexte, mais possède des interfaces clairement définies. On oppose alors à la notion de programmation centralisée et monolithique (*programming-in-the-small*) à la programmation modulaire et ouverte (*programming-in-the-large*) [Burns & al. 90].

Les efforts de recherche dans le domaine des langages de configuration ont porté sur la spécification et la génération de composants d'une architecture, sur leur composition et sur les outils de génération automatique de code, de validation des description et d'exécution répartie. Le processus de développement d'une application est alors découpé en deux phases : une phase de description de l'architecture logicielle et une phase de déploiement et de configuration des éléments logiciels mis en place [Bellissard & al. 96].

La configuration de composants logiciels s'exécutant dans des environnements hétérogènes couvre donc deux phases. La première est basée sur la description de la configuration de l'application. On définit alors les instanciations des composants logiciels et leurs liens de communications en utilisant, en général, un langage déclaratif. A partir de ce langage, et dans un second temps, il est possible de générer des exécutables en interfaçant des composants logiciels déjà existants. Le programme réparti résultant est exécutable dans une environnement système spécifique (comme dans Olan) ou généré (comme dans Aster). La gestion de la distribution est alors statique ou dynamique (comme dans Darwin ou Olan).

Dans la suite de ce paragraphe nous étudions deux types de langage de description : les langages d'interconnexion de modules (*Module Interconnection Language ou MIL*) et les langages de définition d'architecture logicielle (*Architectural Description Language ou ADL*).

1.5.1. Les langages d'interconnexion de modules (MIL)

Les MIL sont des langages de haut niveau dans lesquels on décrit l'interconnexion des composants, indépendamment de la plate-forme de déploiement. On sépare alors la phase de programmation de composants logiciels pris individuellement, de la phase de configuration d'une application qui consiste à assembler ces mêmes composants selon un schéma d'interaction propre à l'application. Un environnement d'exécution est utilisé pour encapsuler les communications entre les modules et réaliser l'adaptation des données (si bien que les exigences d'interopérabilité sont découplées des exigences fonctionnelles). Nous présentons dans la suite la notion de modèle d'assemblage, puis réalisons un rapide état de l'art avant de décrire le cycle de développement particulier induit par l'utilisation de MIL.

Le modèle d'assemblage

Dans un MIL, chaque composant est une entité logicielle (en fait l'unité de modularité logicielle) qui encapsule des ressources et qui dispose d'interfaces concernant les services offerts et fournis. La configuration consiste à réaliser la liaison entre les opérations des différents composants au niveau de leurs interfaces. Un composant est aussi lié à un ou plusieurs fichiers d'implémentation et s'exécute séquentiellement (processus) ou concurremment (groupe de processus ou processus légers [Demeure & al. 94]). Les composants sont des entités indépendantes qui interagissent avec d'autres composants par envoi de messages. Chaque message est typé et tout message envoyé par un composant doit être compatible avec celui qui est attendu par un autre composant. Les différentes implémentations des modèles d'assemblage sont présentés dans l'état de l'art qui suit.

Etat de l'art sur les MIL

Le modèle d'assemblage, bien que simple dans sa description, a été mis en oeuvre de différentes manières. Les exemples les plus connus de MIL sont Conic [Magee & al. 89], Darwin, Durra et Polylith [Purtilo & al. 91, Purtilo 94].

Polylith dispose d'un environnement d'exécution qui gère l'interconnexion et la reconfiguration des interactions entre des composants logiciels s'exécutant dans des environnements hétérogènes. Il encapsule les communications entre les modules et les transformations de données pour automatiser l'adaptation des messages aux interfaces. La composition de composants dans Polylith n'est par contre pas possible, ce qui est un inconvénient majeur.

Cet inconvénient est aussi présent si on utilise Durra, qui est un MIL plutôt adapté au développement d'applications réparties ayant des contraintes temps réels. Durra est fortement lié au langage ADA, c'est pourquoi la structure des composants logiciels qu'il gère est dérivée des contraintes d'implémentation.

Ces limitations ne sont pas présentes dans le langage de configuration Darwin, de conception plus récente. En effet, il gère des composants complexes (ie. contenant des composants dit primitifs) et instanciables dynamiquement durant l'exécution. L'application est alors vue comme une hiérarchie de composants qui évoluent dynamiquement au cours du temps. Cette application s'exécute dans un environnement d'exécution et de gestion des éléments de communication entre composants, appelé Regis. Regis utilise la description de la configuration écrite en Darwin, pour la gestion une collection limitée de types de composants et l'exécution de multiples instances de ces types.

Enfin, OCL (*Olan Configuration Language*) étend les principes de Darwin en ajoutant des services dynamiques au niveau des interactions entre les composants. OCL, intégré au sein de l'environnement Olan, permet la gestion dynamique de groupes de composants, la nommage associatif des composants et la déclaration de contraintes de placements. Tous ces outils suivent globalement le même cycle de développement.

Le développement d'applications avec des MIL

L'approche générale à adopter pour le développement d'applications avec des langages de configuration d'architecture logicielle est la suivante :

1) on identifie les composants exécutables et on produit une description de la structure du flot de données.
2) on introduit le contrôle (les synchronisations) entre les composants et on raffine la configuration, la description et les spécifications d'interface des composants.
3) on élabore des composants types et on les assemble le plus souvent hiérarchiquement pour former des composants complexes. Les composants qui résultent de cet assemblage sont principalement constitués de composants primitifs fournis avec un code d'implémentation.

L'étape de conception de l'architecture se termine lorsqu'une description structurelle du système désiré est réalisée et qu'un ensemble de types de composants primitifs sont décrits dans un langage de programmation. Reste alors à construire le système réparti en invoquant les directives de compilation, de liaison et d'outils appropriés. Cette activité de construction utilise des informations non fonctionnelles telles que la localisation des composants, des contraintes de disponibilité, de besoins en ressources, etc. Ainsi, par exemple, Polylith utilise l'outil Polygen pour la génération automatique d'applications à partir de configurations hétérogènes. Enfin, l'étape d'analyse permet de valider des choix architecturaux et de mesurer l'influence de la modification de paramètres spécifiques. Cette étape s'effectue à priori sur le modèle (ie. avant sa construction) ou directement sur la maquette produite par simulation ou exécution réelle.

Synthèse

Les langages MIL ont comme principaux défauts d'imposer une configuration de manière statique et de ne pas supporter des composants écrits dans des langages divers. Ainsi, Conic imposait un Pascal augmenté et Durra ne supportait qu'Ada 83. Des extensions ont par la suite été réalisées pour résoudre ces problèmes. Ainsi, de mots clefs spécifiques ont été rajoutés dans Durra, alors que Darwin a du être intégré dans l'environnement de développement et d'exécution Regis.

La prise en compte de la dynamicité et le développement des recherches sur les architectures logicielles ont conduit à l'émergence d'un nouveau type de langage : les langages de définition d'architecture logicielle.

1.5.2. Les langages de définition d'architecture logicielle (ADL)

Les langages de définition d'architecture logicielle (*Architectural Description Language ou ADL*) identifient différents styles d'architectures logicielles. Ce sont en général des contraintes topologiques qui définissent les composants logiciels utilisables et les interactions entre ceux-ci. Une architecture est alors décrite par des composants (aussi appelés des tâches ou des modules) reliés logiquement entre eux par des connecteurs (aussi appelés canaux) modélisant les communications.

La description d'un composant est similaire à celle disponible dans les MIL. Par contre, la description des connecteurs passe par la définition au niveau de leur interface des protocoles d'interaction. La phase de configuration consiste à réaliser la liaison des opérations des composants via les connecteurs et leurs règles d'utilisation. En fonction de l'architecture sous-jacente, les interactions entre composants se font par l'intermédiaire d'une mémoire partagée (répartie ou non) et par envoi de messages.

Parmi les ADL les plus récents, on trouve Unicon, Rapide et Aesop. Rapide et Unicon sont des langages d'architectures généraux qui fournissent une plate-forme universelle de conception d'architecture. Aesop au contraire exploite les styles architecturaux pour fournir le support de familles de systèmes construits dans un même style. Dans Aesop, des entités architecturales peuvent avoir de multiples descriptions (grâce à la notion de représentation) représentant des implémentations, des spécifications et des vues alternatives. Tous ces langages supportent la hiérarchie au niveau des composants et plus rarement au niveau des connecteurs entre composants (comme dans Aesop). Les composants sont alors de deux types : primitifs (s'ils ne sont pas conteneur) ou composites (s'ils sont conteneur d'autres composants).

Les langages ADL mettent à la disposition de l'architecte des outils de validation plus au moins puissants. Ainsi par exemple, Rapide dispose d'un système réactif géré par des règles alors qu'Aesop propose la spécification de contrainte de topologie ou de style.

A partir des langages de description d'architecture, on peut aussi générer automatiquement des environnements d'exécution (Aesop), des programmes répartis exécutables (Unicon) ou simulables (Rapide).

1.5.3. Synthèse

Nous avons classé tous les langages MIL et ADL cités dans notre état de l'art en fonction de six critères présentés dans le Tableau 18. Les critères choisis sont les suivants :

1) **granularité** : indique la granularité des composants logiciels gérés dans la description de l'architecture ;
2) **connecteur** : indique le nom ou la mise en oeuvre des liens de communication entre composants de l'architecture ;
3) **hiérarchie** : décrit si les composants ou les connecteurs disposent de plusieurs niveaux de description ;
4) **vérification et validation** : indique si le MIL ou l'ADL dispose d'outils théoriques

ou empiriques de tests de la cohérence et du bon fonctionnement de l'architecture logicielle ;

5) **en entrée** : indique de quelle manière on décrit préalablement l'architecture logicielle avant tout traitement ;

6) **en sortie** : quels résultats on peut obtenir en donnant une description conforme à l'outil utilisé en entrée.

A la lecture de ce tableau on se rend compte qu'il existe quasiment autant de solutions que de produits examinés. D'autre part, on remarque que la granularité usuelle des composants réels d'une architecture logicielle, à savoir un objet ou un morceau de code, est inadéquate pour la construction et la configuration d'architectures logicielles. C'est pourquoi, on crée des entités de plus haut niveau qui permettent de définir des interactions communes et des règles d'administration pour des groupes de composants de granularité variable lors de la conception et lors de l'exécution.On aboutit alors à deux types d'objets ayant des granularités différentes :

- **les modules** qui représentent du code compilé, mais pas lié au programme. L'exemple le plus connu est celui des librairies dynamiques qui sont compilées séparément, mais liées dynamiquement.
- **les composants** qui représentent une spécification séparée, qui peut-être héritée et comporter zéro ou plusieurs modules.

Tableau 18: Comparaison des langages de description d'architectures et de configuration

Langages (type)	Granularité	Connecteur	Hiérarchie au niveau des	Vérification et validation	En Entrée	En Sortie
Aesop (ADL)	composant	connecteur	composants et des connecteurs	contraintes de topologie du style	un style d'architecture	un environnement exécutable (une fable)
Aster (MIL)	composant	système d'exécution spécialisables	composant	appariement de spécifications formelles	une description Aster	Un système spécialisé pour une application
Durra (MIL)	tâches	canal	tâches	à la compilation	une description DURRA	un programme réparti exécutable
Olan (ADL)	composant	connecteur	composants	contraintes aux interfaces	une description OCL	structures réparties et interprétées
Rapide (ADL)	module	connecteur	interfaces	systèmes réactif à base de règles	un programme Orienté Objet	une architecture simulable
Regis/ Darwin (MIL)	composant	liaison (bind)	composants	formelle et basé sur le π-calcul	un programme Darwin	un programme réparti exécutable
Unicon (ADL)	composant	connecteur	composants	Vérification de types	un système	un programme exécutable

1.6. Synthèse

Les gabarits de conception ne sont pas des méthodes de développement de logiciels, ce sont des vecteurs de transmissions entre concepteurs de solutions logicielles connues. Ils complètent des méthodes déjà existantes et réduisent la complexité du logiciel à différentes phases du cycle de vie du logiciel.

Les gabarits de code sont des composants logiciels de haut niveau, quasiment indépendants des autres gabarits et de l'organisation des données. La séparation de la spécification du gabarit et de la structure de la classe entraîne la non modification de tous les autres gabarits. Le paradigme de séparation des préoccupations augmente donc la flexibilité et la compréhension des applications orientées-objets.

Un squelette d'implémentation est un pari fait par le développeur sur la capacité qu'une architecture sera adaptée à plusieurs applications d'un domaine spécifique. Tout changement important du squelette d'implémentation à des conséquences sur la réutilisabilité et dans le cas de gabarit de conception génératif, sur la rétro-ingénierie des applications.

Les gabarits de conception et les squelettes d'implémentations sont des concepts ayant des synergies. Un gabarit de conception peut être vu comme une description abstraite d'un squelette d'implémentation. Similairement, un squelette d'implémentation peut être vu comme une réalisation concrète d'un gabarit de conception qui facilite la réutilisation directe du schéma de conception et du code. En fait, un squelette d'implémentation contraint l'architecture d'une application. Mais, **un gabarit de conception est un élément architectural de granularité moindre qu'un squelette d'implémentation**. Un squelette d'implémentation comprend en général plusieurs gabarits, **mais l'inverse est toujours faux**. Enfin, l'utilisation d'un gabarit de conception n'entraîne pas de choix de conception, contrairement à la description d'un squelette d'implémentation. Il est décrit indépendamment d'un langage, alors qu'un squelette d'implémentation est implémenté dans un langage particulier. La prochaine génération de squelettes d'implémentations orienté objet intégrera explicitement plusieurs gabarits de conception et les squelettes d'implémentations seront largement utilisés pour documenter la forme et le contenu de gabarits de conception [Johnson 92] et [Beck & al. 94].

Il est à noter que l'utilisation des gabarits de conception et des squelettes d'implémentation n'est pas liée à une approche de composition (Bottom-Up) ou de décomposition (Top-Down) de composants. Cette possibilité de va et vient continuels entre les deux approches facilite le travail en équipes qui n'ont pas toujours la même culture ou les mêmes méthodes de conception. Chaque architecte pouvant utiliser l'approche qui lui convient et éventuellement affiner le travail d'un autre architecte ayant suivi une approche inverse.

Les terminologies des constituants les plus fréquents d'une architecture logicielle sont résumées dans le Tableau 19.

Tableau 19: Terminologie liée à la description d'architectures logicielles

Composant	Description
un méta-gabarit	Ensemble de gabarits de conception qui décrivent comment construire des squelettes d'implémentation indépendamment d'un domaine spécifique.
un gabarit de conception	Décrit un problème de conception récurrent, propose un schéma de solution prédéfini et inclut des règles heuristiques indiquant les clauses de son applicabilité.
un squelette d'implémentation	Définit et organise un ensemble de concepts pour satisfaire un ensemble d'exigences pour un système.

Tableau 19: Terminologie liée à la description d'architectures logicielles

Composant	Description
une architecture	Raffine et spécifie un squelette d'implantation fonctionnel en terme d'un ou plusieurs modèles.
un modèle	Est une abstraction d'entités du monde réel en un ensemble de concepts prédéfinis et de relations entre eux. On parle alors de modèle de données, de modèle de communication ou de modèle de nommage.
un modèle de référence	Est un modèle général définissant des termes, des concepts et des relations dans un domaine particulier. L'exemple le plus connu est celui de l'ISO (Interconnexion de Systèmes Ouverts) de l'OSI [AFNOR 88].
un environnement	Est l'instanciation d'une ou plusieurs architectures en intégrant des points annexes tels que les outils et les langages de programmation dont l'architecture est inconnue ou inexistante. OSF/DCE ou ANSaware en sont deux exemples.
un système	Est l'implémentation d'un environnement ou son intégration au sein d'autres environnements déjà existants.
un idiome	Décrit la manière d'implémenter un composant particulier (où une partie de ce composant), ses fonctionnalités et ses relations avec les autres composants. Ils sont spécifiques pour un langage de programmation particulier.
une vue	Définit un raffinement d'une architecture, d'un environnement ou d'un système en fonction d'un axe d'intérêt.

Vers une hiérarchie de description des niveaux architecturaux

Richard Helm [Helm 95] a été le premier à proposer une définition d'une hiérarchie de niveaux architecturaux pour des gabarits de conception. Cette définition a été reprise et étendue par [Mowbray & al. 97].

Global - Standards, Internet	[Mowbray & al. 97]
Entreprise - Modèles de références, infrastructures, politiques	[Mowbray & al. 97]
Système d'information - Interfaces horizontales et verticales	[Pree 95]
Applications - Styles, programmation, intégration d'objets	[Garlan & al. 94]
Macro-architecture - Squelettes d'implémentation, hiérarchie de classes	[Johnson & al. 88]
Micro-architecture - Gabarits de conception	[Gamma & al. 93]
Objets - Objets et classes d'objets	[Coad 92]

Figure 11 : Sept niveaux de description d'une architecture logicielle

Ainsi, un langage de gabarit est organisé en sept niveaux architecturaux résumés sur la Figure 11 (sur laquelle nous avons ajouté les références des travaux décrivant le mieux la couche en question) qui définissent une architecture complexe :

1) au niveau le plus bas de l'architecture, on trouve **les objets**. A ce niveau le développeur est concerné par la définition et le développement d'objets et de classes d'objets réutilisables (bibliothèques écrites en C++ [Stroustrup 86], en Smalltalk [Goldberg & al. 83], en Ada ou en Java par exemple). C'est à ce niveau que se situe les gabarits de Coad [Coad 92]. Les objets et les classes sont dépendantes d'un lan-

gage, sauf si la définition des classes est réalisée avec le langage IDL de l'OMG (cf. OMA) et que la déclaration des interfaces est compatible avec celle de CORBA.

2) **le niveau microarchitecture** est centré sur le développement de composants logiciels qui résolvent des problèmes logiciels récurrents. C'est à ce niveau qu'on trouve les gabarits non génératifs de [Gamma & al. 94].

3) **le niveau macroarchitecture** concerne l'organisation et le développement des squelettes d'implémentation d'application, impliquant une ou plusieurs micro-architectures. La plupart des squelettes d'implémentation de Taligent sont à ce niveau.

4) **le niveau architecture d'application** concerne l'organisation d'une application développée pour répondre à un ensemble de besoins des utilisateurs. Il est composé de plusieurs classes d'objets, de multiples microarchitectures et de une ou plusieurs macro-architectures. La notion de style développée par [Shaw & al. 96] s'inscrit à ce niveau.

5) **le niveau système d'information** ajoute l'interopérabilité entre les applications (middleware) et la gestion du cycle de vie et des évolutions du système. Trois principes clés sont disponibles à ce niveau : les interfaces horizontales (interfaces communes à une organisation), les interfaces verticales (interfaces paramétrées en fonction du domaine et des besoins) et les métadonnées (utilisées pour décrire les services et les données du système). Des gabarits sont utilisés à ce niveau pour décrire des passerelles (*gateway*), des entrepôts de données (*data repository*), des systèmes de nommage, etc.

6) **le niveau architecture de l'entreprise** est dédié à la coordination et à la communication à l'intérieur d'une même entreprise. Une entreprise pouvant être dispersée géographiquement et disposer de logiciels et de matériel hétérogènes. Des gabarits de conception à ce niveau décrivent des modèles de références standards, des ORB, la gestion des systèmes, etc.

7) **L'architecture globale** : choix de conception applicables à tous les systèmes. On trouve à ce niveau des gabarits liés à Internet, aux langages et aux standards édictés par l'industrie ou par l'entreprise.

1.7. Notre vision d'une architecture logicielle dans MEDEVER

L'application du principe de séparation des préoccupations nous a amené à découpler la description de l'architecture logicielle (les composants et leurs interactions), de son implémentation (par programmation ou génération de code), de son déploiement (initialisation et placement de différents composants) et de son exécution (basée sur des environnements et des plate-formes différentes).

C'est pourquoi, dans MEDEVER, nous distinguons deux types de description pour une même architecture logicielle. La première est basée sur une description conceptuelle des éléments d'une architecture (composants et connecteurs), sur leurs relations et leurs responsabilités (les contrats à respecter). Nous appelons cette description conceptuelle, **une architecture logicielle de définition**. La seconde est plus opérationnelle et est directement liée au type d'équilibrage que l'on désire mettre en oeuvre (équilibrage d'application ou équilibrage de charge). Nous appelons cette description, **une architecture logicielle d'exécution**. Elle est obtenue par un développement idoine du code [Mahmoud 97] ou par des techniques de génération automatique de code [Budinsky & al. 96].

Dans MEDEVER, il est important que le style architectural utilisé soit unique pour ces deux types d'architectures et prenne en compte à la fois l'aspect conceptuel (les gabarits de conception) et opérationnel (les squelettes d'implémentation) des architectures logicielles. Cette unification évite de gérer les incompatibilités architecturales,

sans imposer de contraintes sur les méthodes de spécification et de construction d'applications réparties.

Dans le chapitre suivant, nous spécialisons notre étude sur des architectures basées sur le paradigme client-serveur. Nous mettons alors en exergue le rôle des médiateurs (middleware), aussi bien au niveau de l'architecture logicielle de définition (via l'utilisation d'un certain nombre de services standards) que d'exécution (via l'utilisation d'outils mettant en oeuvre ces services).

2. Les architectures logicielles de type client-serveur

Originellement, le terme client-serveur décrivait les interactions entre deux programmes d'une architecture logicielle [Ghernaouti 94]. Les programmes résidaient alors sur la même machine hôte. La signification du terme client-serveur a complètement changé lors de l'apparition d'outils de distribution des traitements en fonction des ressources logicielles et matérielles. Désormais, une architecture client-serveur s'appuie sur des architectures matérielles et logicielles qui **interagissent**. On tente alors de séparer des problèmes de communication de bas niveau des problèmes de contrôle.

Définition 10 : Le paradigme client-serveur

Le paradigme client-serveur consiste à structurer un système en termes d'entités clientes et d'entités serveurs qui communiquent par l'intermédiaire d'un protocole de communication à travers un réseau informatique.

Ces entités interagissent de manière à assurer l'ensemble des fonctionnalités que le système fournit à son environnement [Ozanne 96], ainsi :

- un service correspond à un regroupement de fonctions liées à la gestion d'une ressource applicative, physique ou logique ;
- un serveur est une entité exécutable qui réalise, seule ou en coopérant avec d'autres entités du type serveur, les fonctionnalités du service ;
- un client correspond à une entité exécutable qui émet une requête auprès d'un serveur pour utiliser les fonctionnalités d'un service.

La première section présente deux classifications, l'une conceptuelle et l'autre plus opérationnelle, d'application client-serveur. La section deux décrit les différentes générations d'outils de développement client-serveur et leur évolution. Enfin la section trois, est dédiée à l'étude du médiateur (*middleware*) de plus en plus utilisé pour interconnecter des architectures logicielles client-serveur.

2.1. Classification des architectures client-serveur

Plusieurs classifications des architectures client-serveur existent à ce jour. La première, et la plus connue, est une classification issue des travaux du Gartner Group. La seconde est plus récente et prend en compte les évolutions matérielles qui influence le déploiement des applications client-serveur.

2.1.1. La classification du Gartner Group

Le Gartner Group a établi un découpage en six vues distinctes montrant les différentes possibilités de répartition entre clients et serveurs des trois principales strates logicielles :

- la couche de présentation chargée de la mise en forme et de l'affichage des informations reçues;

- la couche de logique applicative mettant en oeuvre les fonctions de l'application;
- la couche de gestion des données.

Ces vues ne constituent pas obligatoirement les étapes possibles d'un projet client-serveur, visant à faire passer des applications centralisées sur terminal passif connecté à un ordinateur central à des applications réparties et multi-serveurs. Chacun pourra donc s'arrêter à la vue qui correspond le mieux à son organisation et aux moyens financiers et humains dont il dispose.

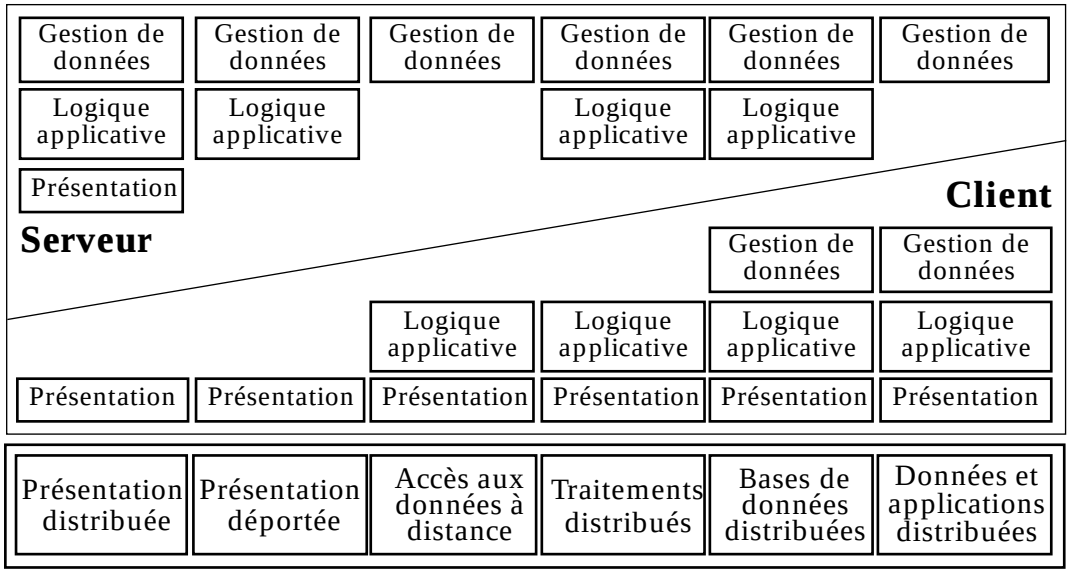


Figure 12 : Taxinomie des architectures client-serveur

Les six vues mises en exergue sont (cf. Figure 12) :

1) **la présentation répartie** qui est le premier niveau du modèle client-serveur. Concrètement, on laisse la gestion des données, la logique applicative et la présentation à la charge du serveur, en ne demandant au client qu'un habillage neuf des données transmises (*revamping*). Ce procédé est utilisé lorsqu'on ne peut se séparer de ses ordinateurs centraux et des logiciels qui fonctionnent dessus, mais que l'on dispose de postes clients puissants (affichage graphique sous windows, au lieu d'un affichage en mode caractère).
2) **la présentation déportée** qui est plutôt dédiée à des architectures avec des serveurs puissants et des postes clients très simples (comme des terminaux X).
3) **les applications avec des traitements répartis** qui laissent la couche de présentation et de logique applicative à la charge du client et celle de la gestion des données à celle du serveur (pour des raisons de sécurité et d'accessibilité). C'est le modèle classique des applications SQL émettant des requêtes complexes vers des bases de données.
4) **l'accès aux données à distance** qui est un compromis architectural. Le client comme le serveur dispose d'une couche de logique applicative qui doit tout faire pour minimiser le débits de données transférées sur le réseau. La couche de présentation reste à la charge du client (pour éviter des temps de transfert trop important) et la couche de gestion des données sur le serveur (toujours pour des raisons de sécurité). Ce modèle s'applique aux applications utilisant des réseaux informatiques longues distances (MAN ou WAN) ou de faible débit (comme les postes nomades utilisant le réseau téléphonique).
5) **les bases de données réparties** laissent le gros du travail au client et accèdent à des serveurs disposant d'une logique applicative de gestion des transaction et des

traitements locaux (telles que les procédures stockées).

6) Enfin, **le modèle décentralisé** intègre des données et des applications réparties.

Cette classification n'est intéressante que si l'on considère que les éléments mis en jeu dans les différentes couches sont hétérogènes. Cette description reste néanmoins trop schématique et à comme seul intérêt de présenter une vison en couches des différentes composantes d'une architecture client-serveur (comme le modèle en 7 couches de la norme de ISO adoptée par l'OSI).

2.1.2. Une autre vision des architectures client-serveur

Une autre vision des architectures client-serveur, nous ramène à un classement en trois grandes catégories : le client-serveur de présentation, le client-serveur à deux niveaux (ou client-serveur de données) et le client-serveur à trois niveaux.

Le client-serveur de présentation place la logique de présentation sur le poste client, mais laisse les traitements sur le serveur.

Le client-serveur à deux niveaux (*fat client-thin server*) est actuellement le plus répandu et le mieux maîtrisé. Il laisse au client la charge de la couche de présentation et de la logique d'application, le serveur abritant la gestion des données. Ce modèle d'architecture comporte des limitations majeures à savoir :

- il génère un trafic sur le réseau élevé. L'utilisation de réseaux à haut débit peut dans certains cas régler ce problème ;
- il est difficile de se connecter à plusieurs applications ou plusieurs bases de données en même temps. Une solution à ce problème est d'utiliser des passerelles hétérogènes (comme EDA/SQL d'Information Builders) ;
- une mauvaise extensibilité qui à pour conséquence de limiter le nombre d'utilisateurs à une centaine ;
- des temps de réponse qui peuvent se révéler désastreux ;
- la création de clients «trop gros» qui disposent de toute la logique applicative et dont la maintenance coûte cher (à chaque modification, il faut mettre à jour tous les postes clients).

Le client-serveur à trois niveaux ôte la logique et le traitement de l'application du poste client pour les mettre sur un serveur spécialisé. Le client conserve néanmoins la présentation incluant le dialogue avec l'utilisateur et plus généralement l'interface homme-machine. La communication entre le client et le serveur d'applications et entre le serveur d'applications et le serveur de données s'établissent par l'intermédiaire d'une couche médiateur. Cette couche gère les requêtes émises par les clients, répartit la charge entre les serveurs, gère la sécurité, etc. Cette architecture logicielle à trois niveaux se combine du coup avec une architecture matérielle elle aussi à trois niveaux où :

- le poste client de type PC-Windows gère la présentation ;
- un serveur d'application de taille intermédiaire (comme Windows NT ou OS2 Warp Server) gère la logique d'application ;
- un serveur de données de type gros système ou serveur départemental offrant des performances élevées gère les requêtes multiples aux bases de données.

L'avantage du modèle à trois niveaux (aussi appelé trois tiers) est d'enlever l'application du poste client, là où elle est le plus difficilement maintenable et sécurisable. Par contre, l'ajout de serveurs intermédiaires complique l'architecture finale déployée et les règles de gestion associées. Ainsi, au niveau de l'architecture de l'application, il est difficile de partitionner simplement les services offerts et rendus, de placer les programmes sur l'architecture matérielle et de gérer les accès hétérogènes aux données.

2.1.3. Synthèse

La classification du Gartner Group est avant tout conceptuelle et correspond à ce que nous appelons une architecture logicielle de définition. Cette classification tente de mettre en application, sur des architectures d'application client-serveur, le principe de séparation des préoccupations. Il est toutefois important de nuancer le terme logique applicative, qui dans ce cas, ne correspond pas toujours à l'ensemble minimal des besoins fonctionnels de l'application. Par contre, on assimile sans difficulté, la couche de présentation et de gestion des données à des facettes techniques. La seconde classification des architectures client-serveur présentée est plutôt opérationnelle et correspond à ce que nous appelons une architecture logicielle d'exécution (chaque client et chaque serveur étant avant tout lié à des machines distinctes).

Ces deux classifications d'application client-serveur privilégient une vision séquentielle de l'accès à des traitements sur des données réparties.

Notre vision du paradigme client-serveur est beaucoup plus large, dans le sens où rien n'empêche un client de disposer de plusieurs fils d'activités en parallèle et de coopérer avec d'autres clients pour réaliser un but. La partie gestion de données étant dans ce cas beaucoup moins importante que la partie contrôle qui gère la synchronisation des communications, la gestion de la coopération, etc. Cette vision séquentielle d'une architecture client-serveur était aussi liée, en partie, aux générations antérieures d'outils et d'environnements de développement d'applications client-serveur qui étaient fortement orientés données. Ce n'est aujourd'hui plus le cas, comme nous le montrons dans la section suivante.

2.2. Les environnements de développement client-serveur

Les architectures logicielles d'applications client-serveur ont évolué d'un modèle centré sur le client, vers un modèle centré sur le serveur, puis vers un modèle centré sur des composants logiciels mobiles. Ces évolutions ont été rendues possibles grâce à l'augmentation du débit des réseaux informatiques et à la montée en puissance des mini et micro-ordinateurs. Nous étudions dans la suite l'évolution de ces architectures et des outils utilisés pour les mettre en oeuvre.

2.2.1. Première génération

La première génération d'environnement de développement client-serveur ne considérait que des architectures centralisées. Les applications étaient de tailles importantes et généralement écrites par des experts du domaine. L'utilisateur accédait alors, grâce à son terminal passif (le fameux VT100), à des applications sur un ordinateur central.

Par la suite, grâce à des techniques d'interconnexion de réseaux, de routage multi-protocole et de logiciels d'émulation de terminaux, il devint possible d'ouvrir plusieurs sessions sur des ordinateurs centraux distincts. Mais les interactions possibles avec le système étaient toujours réduites au stockage de données et à l'utilisation de ressources externes. Enfin, pour les systèmes gérant un grand nombre de transactions, la seule solution était de recourir à des moniteurs transactionnels.

2.2.2. Seconde génération

Les systèmes client-serveur de première génération ne permettaient pas d'effectuer des traitements sur les serveurs. L'architecture des applications mise en place imposait alors des traitements lourds sur des stations pas toujours très puissantes. C'est que l'on a appelé le syndrome du client «obèse» *(fat client)*. Dans les années1990, certains éditeurs ont eu l'idée de déporter certains traitements exprimés en langage SQL sur des serveurs. Le processus employé était basé sur la notion de déclencheurs (*Trigger*) activables à distance et qui lançaient l'exécution de procédures stockées (*stored procedure*) dans le gestionnaire de base de données. Mais ces améliorations ne considéraient que les applications client-serveur orientées données.

La deuxième génération d'outils s'est donc focalisée sur le partitionnement dynamique et transparent des traitements dans le but d'accroître la tolérance aux fautes de ces applications et surtout d'en augmenter les performances. Systématiquement fondée sur l'approche objet, ces outils créent des services applicatifs qui se comportent à la fois comme des clients et des serveurs.

Un service applicatif est une fonction (en général un objet), éventuellement répartie sur le réseau, qui peut être partagée par différentes applications. L'emplacement physique des services applicatifs sur les postes clients et serveurs n'est réalisé qu'au moment du déploiement. Si certains éditeurs offrent déjà ce partitionnement dynamique, d'autres ont préféré opter pour une partitionnement statique utilisant des communications via des RPC [Birrell & al. 84] (ce dernier choix simplifiant considérablement les outils de développement, mais compliquant les mises à jour de l'application). Cette seconde génération fut le point de départ de la migration massive des architectures centralisées vers des architectures client-serveur sur réseaux locaux *(Downsizing).*

Différents cabinets de consultants ont établi une liste des critères fonctionnels et techniques, résumés sur le Tableau 20, décrivant les outils client-serveur de seconde génération.

Tableau 20: Critères fonctionnels et techniques caractérisant la seconde génération

Critères fonctionnels	**Critères techniques**
Gestion d'applications stratégiques à l'échelle de l'entreprise	Partitionnement dynamique, équilibrage de charge, tolérance aux pannes (par duplication de services), référentiel multidéveloppeur
Gestion de débits transactionnels élevés	Partitionnement dynamique, gestion des moniteurs transactionnels
Facilité de déploiement et de redimensionnement en cours de production	Partitionnement dynamique, outils de déploiement automatique, infrastructure de gestion des services répartis
Déploiement en environnement hétérogène, ouverture	Portabilité sur divers environnements graphiques et systèmes, compilation du code de L4G, génération de code C ou C++, gestion de nombreux médiateur, gestion des technologies Internet (microprogrammes mobiles)
Couverture de l'ensemble du cycle de vie de l'application	Outils de conception, de déploiement, de test, de surveillance en temps réel
Performances	Génération de code C ou C++ ou intégration d'un compilateur, duplication de services
Réutilisation de composants standards orientés techniques ou métiers	Orientation Objet

2.2.3. Troisième génération : vers un client-serveur universel

La troisième génération d'outils client-serveur est née en premier lieu de la vague Internet, mais aussi de limitations apparues au cours de la seconde génération. La lourdeur des développements, les performances souvent médiocres et les dépassements en temps et en budget des projets de développement ont milité vers une architecture d'applications encore plus portable et modulaire.

Des modifications en droite ligne d'internet

Les technologies mises en oeuvre sur Internet, et plus particulièrement sur ce qu'on appelle le WWW (World Wide Web) [Berners-Lee & al. 94] sont basées sur :

- un protocole client-serveur simple appelé HTTP (*Hyper Text Transfer Protocol*) définissant les modalités du dialogue entre un client et un ou plusieurs serveurs de données ;
- un système de désignation universel appelé URL (*Uniform Resource Locator*) composé du type de protocole utilisé (HTTP pour accéder à des documents hypertextes ou FTP pour faire du transfert de fichier de site à site) et d'un chemin d'accès aux données ;
- un langage de description de document hypertextes (*Hyper Text Markup Language*) ;
- un nouveau type de client universel appelé navigateur (*Browser*).

Tout client connecté sur le WWW désirant obtenir un document simple ou composite en fait la demande à un serveur, par l'intermédiaire d'un navigateur (cf. Figure 13). Le serveur traite la requête et renvoie les informations dans un format prédéfini (généralement une page de texte comprenant des balises de mise en forme HTML). Le navigateur reçoit les données, les interprète et les affiche à l'écran. Si on se replace dans une architecture client-serveur, on peut assimiler le client au navigateur. En effet, le navigateur se charge du contrôle, de la gestion de flux, de la gestion des données, de l'affichage et de la sécurité (d'ailleurs pas toujours de manière satisfaisante).

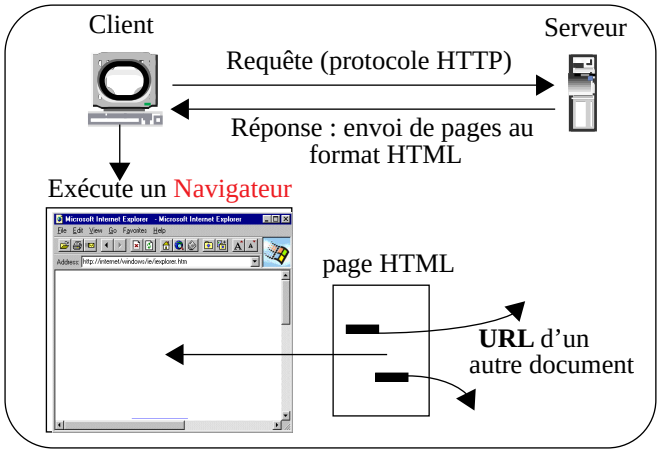


Figure 13 : Architecture client-serveur pour l'accès aux informations sur le WWW

Retour du client obèse, maintenant boulimique

Le navigateur, qui n'était à ses débuts qu'un simple logiciel de demande, de décodage et d'affichage de fichiers HTML, se transforme en une plate-forme système à lui tout seul. Ainsi, Netscape (avec *Constellation*) et Microsoft (avec *Active Desktop*) tente de fondre le navigateur et le système d'exploitation. L'idée majeure est de rendre l'accès aux informations totalement uniforme. Ainsi, qu'elles soient locales ou distantes les données sont parcourues, manipulées et traitées avec les mêmes logiciels. On en revient donc à la notion de client obèse (*fat client*), à la différence près que les services systèmes d'accès à des plate-formes hétérogènes et réparties sont désormais complètement intégrées.

Les navigateurs jusqu'à présent uniquement utilisés pour l'accès aux pages HTML, deviennent donc les supports d'applications client-serveur. Ils intègrent des passerelles vers des bases de données, des interpréteurs de langage de script (tels que Javascript ou VBscript) et des machines virtuelles pour l'exécution de code mobile (tels que Java ou ActiveX). Leurs fonctionnalités sont de plus extensibles de manière modulaire, par

l'ajout de composants additionnels (*plug-in*) respectant des API propriétaires (le format de composant additionnel du *Navigator* de Netscape étant bien entendu différent de celui d'*Internet Explorer* de Microsoft).

Le client obèse est même devenu par la force des choses, **boulimique**. En effet, pour bénéficier de la transparence d'accès aux informations et à l'exploitation de leur contenu, l'utilisateur doit sans cesse récupérer les dernières versions des composants additionnels nécessaires et leurs lots d'erreurs de programmation et d'incompatibilité.

2.2.4. Synthèse

Pour éviter d'avoir un client sans cesse plus gros, sans perdre son coté universel, il a fallu rendre le code mobile. Les clients ont donc suivi une cure d'amaigrissement grâce à des procédures de téléchargement de programmes à la demande. Un client universel standard sur tous les postes de travail télécharge donc, au fur et à mesure de ses besoins, des microprogrammes mobiles sur un réseau. Dans ce modèle, les traitements se rapprochent automatiquement des systèmes qui en ont besoin. Ce nouveau modèle présente néanmoins des risques :

- le client universel est modulaire et évolutif, ce qui implique des configurations non uniformes (chaque client peut rajouter les microprogrammes qu'il veut), des mises à jour fréquentes et des incompatibilités logicielles ;
- les temps de téléchargement variables nuisent aux performances générales ;
- les temps d'exécution sont souvent longs car les microprogrammes compilés dans un code intermédiaire (appelé *bytecode* en Java par exemple) sont interprétés sur les machines virtuelles intégrées au client. Pour améliorer la vitesse, il est possible d'utiliser un optimiseur lors de l'interprétation adapté à chaque type de machines cibles (on les appelle à tord des compilateurs à la volée dans le cas de Java).

Si on se place au niveau architectural, on constate qu'au cours du temps, le recours à des composants logiciels de liaison cachant la complexité des couches basses et des protocoles de coordination entre composants n'ont cessé de se développer. Ces médiateurs sont chargés, entre autres, de gérer les services de nommage, de communication, d'exécution, de stockage des composants logiciels mobiles qui évoluent dynamiquement dans un espace d'adressage immense. C'est pourquoi, nous étudions les médiateurs et les services qu'ils offrent dans la section suivante.

2.3. Le médiateur : colle des architectures logicielles client-serveur

Définition 11 : Médiateur

Le médiateur est un logiciel qui permet à des éléments d'une application d'interopérer à travers des réseaux informatiques, en dépit des différences de protocoles de communication, d'architectures systèmes, de systèmes d'exploitation, de bases de données, etc.

Autres définitions possibles :

Ciment (ou colle) entre les applications et les systèmes;

Technologies qui permettent de déployer des systèmes d'information répartis;

Ensemble des éléments logiciels qui s'interposent entre une application et les spécificités de son environnement.

Dans les années 1990, le médiateur(1) était uniquement basé sur SQL et sur un accès uniforme à des bases de données. Malheureusement ces outils peu extensibles, disposaient

(1) Aucune traduction française du mot middleware étant reconnu, nous reprenons la définition de [Gardarin 96] utilisant le mot médiateur. Médiateur à l'avantage de souligner le rôle d'intermédiaire et de négociateur du middleware.

de peu de fonctions (SQL forms d'Oracle en est une preuve flagrante) et n'offraient pas toujours de bonnes performances.

Aujourd'hui, le scénario se répète avec les outils de développement client-serveur dit de seconde génération (Forté, Natstar, Dynasty), pas encore tout à fait opérationnel. Néanmoins, les produits médiateur ont beaucoup évolué passant de simples couches systèmes spécialisées à des environnements répartis complexes et riches en fonctionnalités. Leur importance dans le contexte de développement des architectures client-serveur est telle que le choix du médiateur est aujourd'hui devenu plus important que celui des systèmes d'exploitation. En effet, il n'est plus concevable de changer de plate-forme médiateur avec chaque nouvelle application client-serveur conçue. Les applications actuelles sont donc développées indépendamment du matériel et doivent interopérer avec d'autres applications de manière simple et performante.

Enfin, la structure organisationnelle de l'entreprise a contribué à semer la confusion dans l'offre des outils médiateur. Il y a quelques années, on vendait des produits SQL pour les administrateurs de bases de données, des outils de développement pour les programmeurs et des outils d'administration et de déploiement aux administrateurs systèmes et réseaux. Aujourd'hui, les applications complexes nécessitent des équipes regroupant toutes ces compétences et des outils multi-fonctions intégrant toute la chaîne de développement.

La couche médiateur à pour but de lier logiquement les éléments d'une application répartie, en offrant une interopérabilité maximale. De par sa nature même, le médiateur est compliqué. Il doit absorber la complexité générée par l'hétérogénéité de l'environnement et assurer la transparence d'accès aux réseaux, aux bases de données et dans une certaine mesure au langage d'accès, tout en déchargeant l'utilisateur des tâches d'ordre techniques. C'est pourquoi, on en recense de nombreux types, fonction des applications à mettre en oeuvre :

- le médiateur composé d'API d'accès direct aux ressources systèmes et réseaux (tels que Net2000 de Novell ou WIN32 de Microsoft) ;
- les médiateurs d'accès à des bases données via le langage SQL (tels que ODBC de Microsoft ou DAL/DAM d'Apple) ;
- les médiateurs d'accès à des multibases (tels que DRDA d'IBM ou EDA/SQL d'Information Builders) ;
- les médiateurs propres à la gestion des transactions (tels que BEA-Tuxedo de BEA Systems ou Top End d'AT&T-NCR) ;
- les médiateurs propres aux objets répartis (tels que Corba de l'OMG ou DCOM de Microsoft) ;
- les médiateurs propres au Groupware (en général des protocoles qui doivent être supportés tels que MAPI, IMAP4 ou POP3) ;

La première section présente l'architecture générique des applications médiateur. La section deux, propose une taxonomie des modèles d'architectures logicielles utilisant le médiateur. La section trois, décrit les modèles de communication entre des éléments d'une application utilisant une couche médiateur. Avant de faire une synthèse du chapitre, nous étudierons brièvement deux exemples d'outils médiateur, à savoir les médiateurs orientées messages (MOM) et les moniteurs transactionnels.

2.3.1. Les composants du médiateur

Tous les produits médiateur ont les mêmes composants de base [Rymer 96], à savoir (cf. Figure 14) :

- les outils de développement qui tendent de plus en plus à être intégrés dans des Ateliers de Génie Logiciel (AGL) ;
- l'environnement d'exécution composé de certains services de base (comme le

nommage ou la sécurité), de services de communication (par l'intermédiaire d'API) et les services d'applications (telles que l'accès aux bases de données avec SQL, la gestion du courrier électronique, la gestion et l'exécution des transactions et la gestion des documents) ;

- les outils d'administration et de déploiement des applications créées.

Si on se concentre sur les couches médiateur d'accès au système de gestion de bases de données (SGBD), une nouvelle classification est possible [Gardarin 96]. Le médiateur est alors décomposé en trois types : propriétaire, éditeur et indépendant (cf. Figure 15).

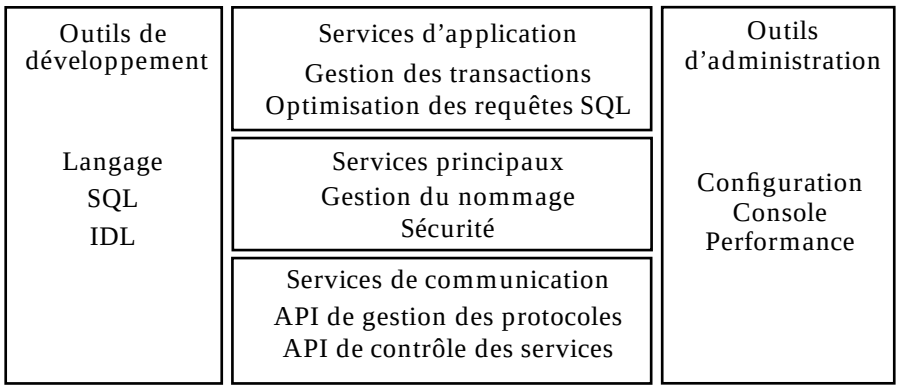


Figure 14 : Les trois composants de base du médiateur

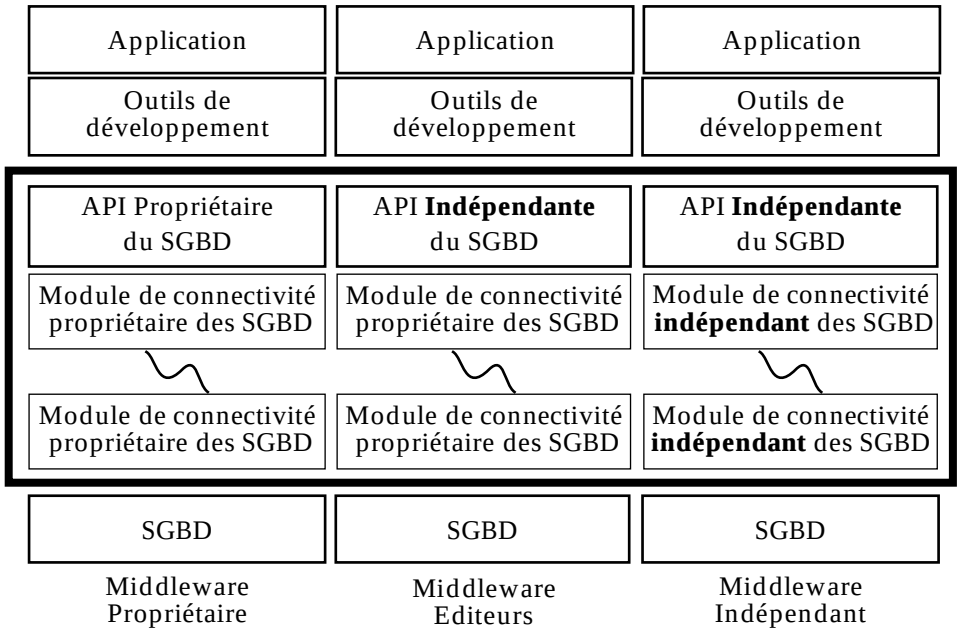


Figure 15 : Accès aux systèmes de gestion de bases de données par médiateur

Les médiateur propriétaires sont construits par le fabriquant du SGBD et ne sont utilisables que dans un environnement fixe. Les médiateurs éditeurs sont avant tout des interfaces applicatives de communication avec les SGBD qui s'appuient sur des protocoles d'accès et des formats définis par le constructeur du SGBD. Enfin, les médiateur indépendants sont fournis par un éditeur tiers différent de celui du SGBD et des outils de développement. Ils permettent ainsi des accès à des multibases et sont plus ou moins compatibles avec des outils de développement du marché.

Au niveau de l'architecture générale des médiateurs, la tendance actuelle est de laisser l'utilisateur choisir les services de bases (*core services*) dont il a besoin plutôt que de tous les intégrer dans un même noyau [Clark 95]. Ainsi, il est possible de rajouter un gestionnaire de mémoire répartie (comme TreadMarks [Amza & al. 96]) ou un annuaire réparti conforme à ITU-T X500 dans une application répartie de manière relativement simple.

2.3.2. Une taxonomie des modèles d'application du médiateur

Le médiateur met aujourd'hui à la disposition des programmeurs dix modèles d'application allant du plus concret (au début du Tableau 21) vers le plus abstrait (à la fin du Tableau 21) [Rymer 96].

Tableau 21: Les dix modèles d'application du médiateur

Type	Description
Stocker et envoyer Publier et s'abonner	Les messages sont envoyés directement au destinataire
Gestion de flux	Les messages sont envoyés au destinataire en fonction de conditions spécifiques
Transactions réparties	Gère plusieurs mises à jours simultanées sur des bases de données multiples sous le contrôle de la transaction
Accès à des fichiers distants	Redirige les demandes de fichiers à travers le réseau
Accès à des bases de données distantes	Transmet des requêtes SQL à travers le réseau vers des serveurs
Interactions entre objets répartis	Offre l'envoi de messages entre objets à travers le réseau
Accès à des fonctions distantes	Redirige les appels de fonctions à travers le réseau
Gestion de bases de données réparties	Maintient une seule base de données locale au dessus de plusieurs bases de données physiques
Réplication de bases de données	Synchronise les copies d'une base de donnée
Affichage réparti	Offre à des clients distants l'accès à des fonctions de présentation d'une application.

Les cinq premiers sont généralement connus des utilisateurs, alors que les cinq derniers sont plutôt du fait des développeurs d'applications. En voici un brève description :

- le modèle **stocke et envoie** (*store and forward*) est le modèle le plus simple et le plus connu, c'est celui du courrier électronique. Le modèle publier et s'abonner (*publish and subscribe*) est plutôt lié à une problématique intégration de système. Dans le premier cas, les applications échangent des données avec l'environnement extérieur, alors que dans le second cas, l'échange se fait entre des modules connus de l'application.
- le modèle de **gestion de flux** (*Workflow*) oriente l'information entre applications en fonction de conditions prédéfinies. C'est le modèle utilisé pour l'échange de documents informatisé (EDI).
- le modèle de **transactions réparties** utilise un protocole de validation pour mettre à jour de multiples données de manière atomique (elles sont toutes mises à jour ou alors aucune d'elle ne l'est) et cohérente.

- les modèles **d'accès à des fichiers distants et à des bases de données distantes** se sont énormément développés lors de la mise en place des réseaux locaux dans les entreprises. Ils permettent un accès transparent à des données distantes en respectant des règles de sécurité.
- **le modèle d'interactions entre objets répartis** et **le modèle d'accès à des fonctions distantes** sont intimement liés. Lorsqu'un développeur utilise des technologies de développement orienté objet, avec des objets répartis, il utilise des appels de routines (ou appel de méthodes) avec accès à des fonctions distantes.
- le modèle de **gestion de bases de données réparties** et le modèle **de réplication de bases de données** sont tous deux utilisés pour synchroniser des données réparties. Le développeur crée donc un vue unique d'une base de données virtuelle calquée sur de multiples bases de données physiques (répliquées ou non).
- le modèle **d'affichage réparti** offre l'accès à des applications distantes sur des terminaux graphiques ou textuels. Seules les fonctions de présentation sont gérées par le terminal, le reste étant à la charge de l'application hôte.

2.3.3. Modèles de communication du médiateur

Un modèle de communication (première colonne du Tableau 22) spécifie un type d'interaction entre des éléments de l'application (deuxième colonne du Tableau 22)

Tableau 22: Modèles de communication du médiateur

Modèles de communication	Type d'interaction	Exemples
Datagramme	Un message envoyé	Notification d'un matériel réseau
Question-réponse	Un message envoyé et un message en retour	Transfert de fichier
Requête	Un seul message envoyé et des réponses en chaînes en retour	Navigation sur le WWW
Asymétrique	Messages envoyés et réponses multiples, une seule session	Une application de publication sur le WWW
Symétrique	Multiples demandes et réponses sur deux canaux de communication différents	Système de réservation à haut débit

On dénombre cinq modèles distincts de communication au sein d'une couche médiateur [Rymer 96] :

- **le modèle datagramme** (*datagram*) est unidirectionnel et sans mémoire. Une application envoie un message sans attendre de réponse. Ce mode est aussi appelé «envoyer et prier» (*send and pray*).
- **le modèle question-réponse** (*one-shot*) implique que chaque message envoyé doit entraîner un message unique en retour.
- **le modèle requête** (*Query*) implique que chaque message envoyé doit entraîner une chaîne de messages en retour. Ce sont typiquement les modèles de communication pour les clients interrogeant des bases de données (avec SQL par exemple) ou pour les clients navigateurs du World Wide Web.
- **le modèle asymétrique** (*Asymmetric*) et **le modèle symétrique** (*Symmetric*) offrent des débits d'échanges de messages entre applications élevés et limités par la bande passante disponible sur les liens du réseau. Dans le cas asymétrique, les messages en entrée et en sortie sont véhiculés sur le même médium, alors que dans

le cas symétrique, ils utilisent deux canaux de communication distincts. Face au nombre important de messages échangés, la gestion des sessions de communication entre les applications est primordiale.

Ces modèles ont tous un point commun, ils nécessitent une liaison sur un réseau informatique physique. D'autre part, les types d'interaction sont dépendants ou pas du réseau sous jacents. Ainsi, les applications disponibles sur le World Wide Web sont indépendantes (en principe) du réseau utilisé. Il n'en est pas de même de certaines bases de données liées à des réseaux spécifiques (comme SQL Server sur Windows NT).

2.3.4. Deux outils médiateur «standards»

Il nous semble important, au travers de deux exemples, de montrer à quel point les services offerts et les technologies employées peuvent varier au sein de la couche médiateur. Nous allons donc présenter les médiateur Orienté Message (MOM) et les moniteurs transactionnels.

Le médiateur Orienté Message

Les couches médiateur mettant en oeuvre des systèmes répartis à base de messages sont encore peu répandues. Elles permettent un échange de messages asynchrones entre clients et serveurs par l'intermédiaire de files d'attente. Elles offrent une grande flexibilité car clients et serveurs sont activés à des moments différents, selon leur rythme. L'inconvénient est qu'on ne peut pas garantir un temps de réponse fixe.

Parmi les produits les plus vendus on trouve Dec MessageQ (Digital), Message Express (momentum Software), MQSeries (IBM), Pipes (Peer Logic - Texas Instrument), etc.

Le moniteur transactionnel

Un système transactionnel fait en général référence à un système regroupant des générateurs d'application, des gestionnaires de données, des réseaux de communication, des bases de données et des systèmes d'exploitation. Le coeur de ces systèmes est composé d'un ensemble de services qui gèrent et coordonnent les flots de transactions à travers le système. Une transaction est une unité de travail qui vérifie des propriétés ACID :

- **A**tomicité : une transaction est indivisible, soit est réalisée, soit elle échoue.
- **C**ohérence : une transaction fait passer le système d'un état cohérent à un autre.
- **I**solation : une transaction n'est pas affectée par d'autres transactions.
- **D**urée : les modifications dues à une transaction terminée sont durablement garanties.

Précédemment inclus dans tous les environnements grands systèmes (*mainframe)* comme CICS et IMS sur IBM, les moniteurs transactionnels [Bernstein 90] se positionnent dorénavant comme une couche indépendante des systèmes d'exploitation. L'origine de cette évolution vient du modèle XTP de l'X/Open qui définit l'ouverture et la capacité du moniteur à supporter et à s'interfacer avec les composants du système réparti. Le moniteur transactionnel fournit des services de niveau applicatif et orienté transactionnel, il se différencie en cela des autres outils médiateurs.

Le rôle de moniteurs transactionnel a évolué considérablement pour prendre en compte les nouveaux besoins des architectures client-serveur à trois niveaux (le serveur intermédiaire disposant d'un moniteur transactionnel). Ainsi, outre la supervision et la synchronisation des transactions, les moniteurs transactionnels offrent des services pour :

- répartir la charge entre différents serveurs ;
- fédérer des systèmes de bases de données (SGBD) hétérogènes ;
- apporter l'indépendance du développement par rapport aux environnements où ils seront utilisés.

Les modes de communication généralement supportés sont le mode synchrone, conversationnel et par files d'attentes. En effet, toute transaction maintient un état global et un historique de tout ce qui s'est passé durant son exécution. Les contraintes de mise en oeuvre sont donc fortes et entraînent un surcroît de gestion, qui n'est pas généralement acceptable dans le cas de d'un environnement asynchrone réparti.

Les principaux moniteurs transactionnels sont BEA-Tuxedo (BEA Systems), Top End (AT&T NCR), CICS (IBM), Encina (IBM-Transarc), Unikix (Bull) et MTS (Microsoft).

2.3.5. Synthèse

Le médiateur a pour beaucoup été le remède miracle aux problèmes d'intégration et d'interopérabilité entre applications client-serveur. Or, s'il est vrai que cette couche logicielle s'adapte bien aux applications transactionnelles, il n'en est pas de même pour des applications basées sur des milliers d'objets qui coopèrent et éventuellement se déplacent. De plus, l'empilement des couches de communication, de nommage et de médiateur dégrade souvent les performances globales de l'application dés qu'une certaine charge est atteinte ou dés qu'un service tombe en panne. Pire, il est presque impossible de prévoir les performances d'une application client-serveur à priori. Des campagnes de tests sont alors obligatoires avant tout déploiement.

Néanmoins, la construction d'applications réparties passe souvent par au moins un médiateur, qui gomme la complexité de gestion de systèmes d'exploitation et de ressources multiples installées sur des sites distants (parfois même hétérogènes).

Au niveau des applications parallèles, l'utilisation de médiateur est quasiment obligatoire lorsque l'architecture matérielle cible est spécifique et propriétaire. La mise en oeuvre d'architecture logicielle parallèle et répartie est donc à la fois simplifiée par la richesse et la diversité des produits offerts (communication asynchrone avec les MOM, synchrone avec les moniteurs transactionnels) et compliquée par le fait que les médiateurs de différents types sont souvent incompatibles entre eux (tels que les MOM et les moniteurs transactionnels). A cela s'ajoute le fait que les médiateurs manipulent des entités logicielles qui sont de granularité très variable (un message électronique pour les MOM et des bases de données hétérogènes pour les moniteurs transactionnels).

On rejoint alors à la problématique de MEDEVER, qui est de fournir un cadre méthodologique pour la gestion de la granularité des éléments d'une application durant son cycle de vie. **Nous distinguons dans MEDEVER les médiateurs qui jouent un rôle d'adaptateur de données (par exemple en réalisant un pont entre un composant OLE et OPENDOC) et ceux qui offrent de réels services systèmes (placement, gestion des interruptions, temps réel).** Le problème qui se pose alors est de savoir si ces services médiateurs sont exclusifs entre eux ou sont cummulables (ie. interopérables) pour permettre d'offrir des facettes techniques différentes à un même noyau fonctionnel.

2.4. Conclusion

Les architectures logicielle de type client-serveur actuelles nécessitent la recherche du meilleur découpage d'une application en composants logiciels, ainsi que la répartition efficace de ces composants. Ces composants sont alors organisés en une architecture logique multi-tiers, constituée d'entités logicielles clientes et/ou serveurs. Malheureusement, la granularité de chacun de ces composants, liée à ses données et à la complexité de sa logique interne, n'est pas universellement quantifiable. **Notre méthode MEDEVER, séparant l'architecture logicielle de définition et de celle d'exécution, est une réponse à ce type de problème.**

Ainsi, l'architecture logicielle de définition est déduite des spécifications des modèles de haut niveau utilisés en amont du cycle de vie. On cherche alors à calculer théoriquement la granularité des composants en fonction de leur fréquence d'utilisation, des services qu'ils offrent et de leur rôle au sein de l'architecture logicielle. Puis l'architecture d'exé-

cution est mise en oeuvre, en conservant la traçabilité avec les composants logiciels de l'architecture de définition. Des procédures de tests et d'ajustements des performances sont ensuite effectuées sur l'architecture d'exécution. Les résultats de ces études sont ensuite propagées jusqu'au modèle. Le va et vient entre ces deux architectures permet la mise au point incrémentale de l'application.

Au niveau de l'architecture d'exécution, le médiateur agit comme un rideau de fumée cachant la complexité et l'hétérogénéité de l'architecture, ce qui entraîne :

1) un fossé entre le modèle et la réalisation ;
2) une difficulté nette à prévoir le coût et la durée des développements ;
3) des offres très propriétaires à choisir parmi des centaines de possibilités (technologie client x technologie serveur x technologie médiateur).

Le passage de l'architecture logicielle de définition à celle d'exécution est donc rendu difficile par l'usage de médiateurs, dont les services sont difficiles à modéliser et à intégrer au sein de méthodes de spécification. **Notre méthode MEDEVER tente d'intègrer les différentes technologies médiateurs uniquement au niveau de l'architecture d'exécution, tout en contrôlant leur influence sur l'architecture logicielle de définition.**

Enfin, force est de constater que les architectures d'applications client-serveur basées sur des composants logiciels nomades, utilisent des technologies et des langages objets dont nous présentons les principaux aspects dans le chapitre suivant.

3. Les architectures client-serveur selon un axe langage

Les travaux récents sur la construction et la validation des applications client-serveur se basent sur une structuration des informations en un ensemble d'objets. En général, il existe autant de types d'objets que de langages de programmation. Ces langages de programmation ne comportant pas tous des extensions de gestion de la concurrence, il est souvent necessaire de les étendre, voire d'en réécrire de nouveaux. La granularité des objets gérés, la granularité de la concurrence et les architectures logicielles qui en résultent sont donc extrêmement variées.

La première section rappelle les définitions de base du modèle objet et pose la problématique de la gestion de la concurrence. La section 2 présente une classification des langages objets généraux. La section 3, étudie l'introduction de la concurrence dans les langages objets et la section 4, les nouveaux types d'objets qui en résultent. Enfin, nous concluons notre étude sur le constat de l'expansion des langages acteurs et des systèmes basés sur des agents mobiles.

3.1. Rappels sur le modèle objet

Dans un objet, on intègre de manière unique des données et des traitements qui leurs sont associés. Ainsi un objet est toujours composé des données qui constituent l'état de l'objet et du code des méthodes permettant de manipuler l'état de l'objet. Un objet est donc caractérisé par l'encapsulation de son comportement interne et ne communique que par ses interfaces d'accès [Meyer 88].

Le type d'un objet définit des ensembles d'objets qui ont les mêmes interfaces (donc le même comportement), mais leur réalisation, leur structure interne et leur programme peuvent être différents Pour les différencier entre eux, on utilise alors leur signature. On appelle signature d'un objet le n-uplet comprenant son type, ses méthodes et ses paramètres d'appels.

3.1.1. Définition d'une classe d'objets

La notion de classe a été introduite par le langage Simula [SSI 87] pour désigner des entités utilisant conjointement données et procédures et rendant inutile la réécriture de petits segments de code similaires.

Définition 12 : Classe d'objets

Une classe d'objets est à la fois une structure de données et un ensemble d'opérations sur ces données.

Une classe est une abstraction correspondant à un ensemble d'objets ayant les mêmes caractéristiques dans le monde réel. Ces caractéristiques, variables de champs et de méthodes, deviennent celles de la classe. Une classe est un gabarit de conception permettant de construire des objets ayant ses caractéristiques [Moreau 95]. En tant que gabarit une classe doit indiquer :

- les champs des objets à créer. Un attribut étant alors une caractéristique d'une classe auquel correspond un champ dans chacun des objets qu'elle permet de construire. Un attribut n'a donc pas de valeur par lui même, les valeurs sont attribuées aux variables de champs dans les objets. Par contre, un attribut possède des propriétés telles que son nom, le type des variables de champs qui lui sont associés, etc.
- ses méthodes. Les unes, dites méthodes de classe, sont celles qui lui permettent de créer des objets. Les autres, dites méthodes de représentants (*instance method*) seront utilisées par les objets créés à partir de la classe pour accéder ou modifier les variables de champs.

On distingue deux types de classes : les classes abstraites et concrètes.

Classe concrète

Définition 13 : Classe concrète

Une classe d'objets dont on peut définir une représentation opérationnelle des entités. Ce sont des classes possédant toutes les informations nécessaires à leur instanciation.

Un représentant (instance) d'une classe est un objet qui a les mêmes caractéristiques que celles de la classe, mais dont une valeur a été affectée à toutes ses variables de champs. Chaque représentant est donc toujours dans un certain état et possède sa propre identité.

Classe abstraite

Les classes qui définissent un comportement commun ne sont pas des classes instanciables (donc concrètes), mais des abstractions de ces dernières. Ainsi, les propriétés de classes similaires (variables et méthodes de la classes) sont définies dans une superclasse commune. On appelle cette superclasse **une classe abstraite**.

Dans une classe abstraite on admet que des méthodes ne soient pas implémentées. Néanmoins, elles sont quand mêmes spécifiées sous la forme d'une prototype de fonction (nom de la fonction et paramètres) pour que ses descendants ne puissent pas changer les interfaces de méthodes. L'instanciation de classes abstraites n'a donc aucun sens car elle n'est pas complètement définie.

Dans une classe abstraite toutes les composantes liées à la définition de l'objet ne sont pas présentes. Ce type de classe est utilisé comme méta-modèle et offre une représentation fonctionnelle des entités manipulées. Ainsi, plus une classe est abstraite plus elle est réutilisable.

Définition 14 : Classe abstraite

C'est une classe générique, ancêtre de classes concrètes. Elle définit des propriétés communes et certains des comportements généraux des classes concrètes par leur nom seul et en l'absence de définition précise commune.

3.1.2. Définition de l'héritage de classes

Certains langages proposent des techniques de partage de code et/ou des données différentes de celles existantes entre une classe et ses représentants. Ces techniques, appelées héritage, permettent la spécialisation et la généralisation de classes.

Définition 15 : Héritage de classes

Une classe A hérite d'une classe B lorsque les caractéristiques de B reprennent celles de A et sont enrichies.

Généralement, toutes les classes sont les descendantes d'une classe unique, appelée la racine. Si une classe possède dans l'arbre d'héritage un seul père, on parle d'héritage simple, sinon, on le qualifie d'héritage multiple. Idéalement, une bonne hiérarchie de classes est caractérisée par la présence de classes abstraites à tous les niveaux, sauf au niveau des feuilles composées de classes concrètes. L'héritage est une des notions qui permet de classifier les langages objets, comme nous le montrons dans la section suivante.

3.2. Classification des langages objets

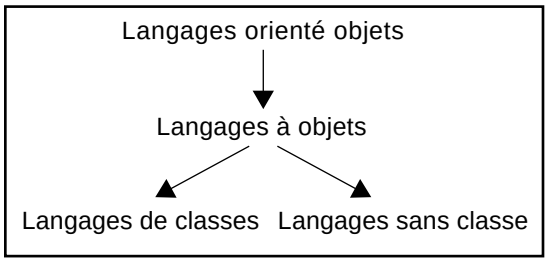


Figure 16 : Les catégories de langages décrivant des objets

On classe en général les langages objets en quatre catégories [Wegner 90] (cf. Figure 16) :

1) **les langages orientés objets** introduisent une relation entre les classes par un mécanisme d'héritage simple ou multiple. Ce mécanisme d'héritage et le principe de la délégation permettent la réutilisation de code. La délégation permet à un objet donné d'en créer un second d'une classe différente, son délégué. Ainsi, chaque propriété non définie dans l'objet de base sera relayée à son délégué.
2) **les langages à objets**, comme Ada 83 ou Modula 2. Dans ces langages, un objet est en réalité un type abstrait dont la structure interne est cachée à son utilisateur (clause PRIVATE en Ada 83) et qui n'est manipulable que par des fonctions dites publiques.
3) **les langages de classes** comme CLU. Un objet est alors créé à partir d'une classe, par instanciation. Les fonctions de manipulation de l'objet sont réalisées par des méthodes de classes, et il est possible de partager des informations par le biais des variables de classe.
4) **les langages sans classe**, comme SELF [Ungar & al. 87, Smith & al. 95]. Dans ces langages, chaque objet est l'équivalent d'une classe et l'héritage est réalisé par clo-

nage ou par prototypes.

La gestion de la concurrence est donc dépendante du type de langage utilisé et de son niveau de conformité avec le modèle orienté objet.

3.3. Révision du modèle objet en vue de l'intégration de la concurrence

Les langages objets concurrents ont été créés en partant d'un double constat :

1) l'approche objet est une méthode de structuration des données en objets, mais initialement séquentielle elle ne s'intéresse pas à la structuration d'un problème en termes d'activités parallèles. Plus précisément, un seul objet est actif à un instant donné et une seule méthode peut être exécutée. L'invocation d'une méthode équivaut alors à l'appel d'une procédure (distante ou locale). ;
2) l'approche par processus communiquants est une méthode de structuration d'un problème en activités concurrentes et en leur mode de coopération, mais ne s'intéresse que très peu à la structuration des données.

Les deux approches étant complémentaires, l'idée de les associer pour créer de nouveaux langages est alors née.

L'adjonction de la notion de processus au modèle objet semble simple, car les similitudes sont fortes [Meyer 93] :

1) **existence de variables locales**. On les appelle attributs d'une classe dans le modèle objet, et variables dans le cas d'un processus.
2) **utilisation de données persistantes**. Elles gardent leur valeur entre différentes activations.
3) **encapsulation du comportement**. On définit un certains nombres de routines pour une classe et un processus possède un cycle.
4) **restriction sur la manière d'échanger de l'information entre modules**. La communication est réalisée par passage de messages.

3.3.1. Les trois approches d'intégration

On distingue trois grandes approches pour introduire la concurrence dans les langages objets [Karaorman & al. 93] :

1) on crée de toute pièce un nouveau langage *(comme par exemple Hybrid [Nierstrasz 87], Pool, SR, ABCL/1, Docase, Dowl*).
2) on étend un langage orienté objet déjà existant d'une des façons suivantes :
   - par héritage d'une classe gérant la concurrence qu'un compilateur modifié reconnaît *(comme dans Eiffel //, Epée et PRESTO [Bershad & al. 88a et 88b]).*
   - par ajout de mots clefs spécifiques ou par l'utilisation d'un pré-processeur, qui permettent d'étendre et de modifier la syntaxe et la sémantique du langage *(comme dans Ada 95 par extension d'Ada 83, dans C* par extension du C, dans CEiffel par extension d'Eiffel [Meyer 92a], dans Garf par extension de Smalltalk, dans μC++ [Buhr & al. 92a et 92b] par extension de C++).*
   - par extension de la syntaxe et la sémantique du langage pour obtenir un paradigme plus général pour la gestion de la concurrence *(tel que le modèle acteur : Act++ [Kafura & al. 90] et Actalk [Briot 89]).*
3) on crée de nouvelles bibliothèques qui gèrent la concurrence *(comme par exemple par rajout de classes Eiffel dans [Geib & al. 91] ou de la classe Process dans Choices).*

L'ordre de présentation de ces approches correspond aussi à l'ordre chronologique de leur apparition. Ainsi, les premiers langages orientés objets séquentiels furent-ils créés de

toute pièce, pour permettre de tester et d'améliorer les techniques de programmation et de développement. Une fois ces techniques maîtrisées, les langages orientés objets séquentiels furent étendus pour exprimer la concurrence. L'approche bibliothèque est la plus récente des approches et donc celle qui bénéficie de tous les acquits précédents dans le domaine.

Néanmoins, dans certains cas, l'intégration de la concurrence dans les langages objets entraîne un conflit entre les notions d'héritage et de synchronisation.

3.3.2. Concurrence et héritage

Ce phénomène a été identifié sous le terme d'anomalie d'héritage dans [Matsuoka & al. 93]. Il a ainsi été prouvé que les parties de code où se trouve défini le comportement de l'objet concernant le contrôle de ses contraintes de synchronisations ne peut pas être hérité sans redéfinition non triviales de ses classes. L'occurrence des anomalies d'héritage est donc fortement lié au schéma de synchronisation utilisé dans le langage de programmation. Trois niveaux d'intégration de l'héritage dans les langages objets concurrents sont alors possibles :

1) aucune intégration : la concurrence et la notion d'objets coexistent alors dans un même langage, comme par exemple Concurrent C++ [Gehani & al. 88].
2) une intégration partielle : la concurrence est intégrée dans le langage orienté objet, mais la hiérarchie ne s'applique pas aux classes d'objets actifs et/ou asynchrones. Guide est un bon exemple d'intégration partielle.
3) une intégration complète : chaque classe étant considérée comme active, comme par exemple dans POOL-I.

Une des solutions possibles est l'utilisation du polymorphisme [Kafura & al. 89]. Le lecteur intéressé par ce problème trouvera une très bonne synthèse dans [Matsuoka & al. 93] et [Meseguer 93]. Une autre solution est d'opter pour un langage d'objet mobile et autonome basé sur le modèle acteur.

3.3.3. L'émergence des langages acteur

Les différents exemples de mise en oeuvre présentés ci-dessus montrent que le grain de concurrence est fonction du type d'implémentation, mais aussi du nombre de méthodes exécutables concurremment que contient l'objet. Le modèle de concurrence est donc centré sur l'objet. Si on étend cette vision pour prendre en compte les schéma de coopération et d'évolution d'une communauté d'objets, chacun étant indépendant des autres, on aboutit alors au modèle acteur.

Le modèle acteur offre un moyen d'expression de la concurrence, dans lequel un problème est résolu par une communauté d'acteurs, entités indépendantes, qui communiquent entre elles et coopèrent pour la résolution du but commun [Hewitt 77] et [Agha 86a].

La communication entre acteurs repose uniquement sur la notion d'envoi asynchrone de messages, unidirectionnels (seul l'émetteur connaît le destinataire), en point à point et sans perte (tout message arrive dans un délai fini). Le modèle acteur offre une approche modulaire de la concurrence et permet [Arcangeli & al. 94] :

- de définir et d'interconnecter les composantes (acteurs) d'un système ouvert (parallélisme à gros grain) de façon dynamique,
- de décomposer chacun de ces acteurs en acteurs plus simples dont le comportement peut être lui même concurrent,
- de supporter la notion d'abstraction de données,
- d'offrir une concurrence interne entre les opération élémentaires de calcul et celles d'envoi de messages.

Un langage d'acteurs doit obligatoirement implémenter les constructions suivantes [Agha 86b] :

- définition de comportements,
- création d'acteur avec un comportement initial (CREATE),
- transmission asynchrone de messages (SEND TO),
- changement de comportement (BECOME).

Les diverses implémentations du modèle se différencie par le type de langage décrivant le comportement des acteurs :

- langage impératif pour ABCL/1 ;
- langage fonctionnel pour PLASMAII ;
- langage objet pour Actalk [Briot 89] et Mering IV [Ferber 83] ;
- lambda-calcul pour décrire la sémantique du modèle de Agha (Act1 [Theriaut 82 et Lieberman 87], Act2 [Theriaut 83]);
- π-calcul pour décrire la sémantique de l'algèbre des processus [Millner 89], dans le langage ACL [Kobayashi & al. 94].

3.3.4. Synthèse

L'introduction de la concurrence dans les langages objets nécessite la prise en compte des problèmes suivants :

1) synchronisation et protection des variables d'instances dans le cas d'une communication intra-objets.
2) synchronisation et protection des variables de classes dans le cas d'une communication inter-objets.
3) création explicite d'objets actifs et passifs permettant une expression claire des points de contrôle de concurrence. Un objet actif est un processus et la communication entre objets actifs se fait par échanges de messages. Un objet passif contient des données et sa communication avec d'autres objets passifs se fait par mémoire partagée. Un processus n'est alors rien d'autre qu'une instance de classes (il est donc typé et donc statique). Ainsi, dynamiquement un objet peut être associé à un processus.
4) résolution du problème du partage d'un objet passif par plusieurs objets actifs.

Le placement et l'exécution répartie des objets sont fortement dépendants du langage et de l'environnement d'exécution. Néanmoins, la programmation orienté objet permet d'attacher facilement la description des actions (routines) à la description des objets (classes). Reste alors à prendre en compte la notion de processeurs. C'est ce que propose [Meyer 92a], en ajoutant la clause *separate* [Meyer 93] dans le langage Eiffel. Une classe ou un objet déclaré séparé, entraîne une exécution distante sur un autre processeur virtuel que celui ou l'objet est créé. On crée donc un lien de répulsion entre deux classes ou objets. L'allocation d'un processeur virtuel à un processeur physique sera géré par le système. Notons que dans le cas d'une machine mono-processeur, on se ramène au cas d'un système multitâche.

Les langages acteurs sont utilisables comme des langages de bas niveau pour la réalisation de systèmes complexes, mais aussi comme un modèle d'interconnexion d'applications hétérogènes [Agha & al. 92a]. Les langages acteurs intègrent les notions clefs de l'objet pour permettre de combler le manque de structuration des données dans les acteurs et d'introduire des mécanisme d'héritage. On peut alors soit partir d'un modèle orienté objet et l'enrichir avec le modèle acteur (comme dans Mering IV [Ferber 83] où le rôle des acteurs est limité à l'exécution des méthodes des objets) ou partir d'un modèle acteur et l'enrichir avec un modèle objet (comme dans HAL [Agha & al. 92b] où la com-

posante objet ne sert qu'à définir le comportement des acteurs). Il faut néanmoins prendre en compte le fait que les modes de communications différent entre les langages objets concurrents et acteurs. L'invocation d'une méthode est un appel procédural dans le modèle objet. La transmission de message est asynchrone dans le modèle acteur et la synchronisation est basée sur le principe de continuation.

Enfin, la mise en oeuvre de la concurrence dans les langages objets et acteurs passe, en général, par l'utilisation de quatre classes d'objets.

3.4. Mise en oeuvre de la concurrence

De notre étude sur les langages objets et acteurs concurrents, nous avons déduit qu'il existe quatre classes d'objets pour la mise en oeuvre de la concurrence : les objets passifs, les objets actifs, les acteurs et les objets mandataires.

3.4.1. Les objets passifs

Le modèle d'objet passif est caractérisé par le fait qu'un processus n'est pas restreint à un unique objet. La notion importante est l'activité constituée d'une chaîne d'invocations de méthodes. Cette chaîne constitue un flot de contrôle supporté par un seul processus, mais associés à des objets distincts au fur et à mesure des invocations. En d'autres termes, un seul processus est utilisé pour traiter toutes les opérations de réalisation d'une action quelques soient les objets utilisés.

En cas d'invocation de méthodes d'un objet distant, deux solutions sont envisageables :

1) un objet relais est créé sur le site distant pour transférer le flot de contrôle. Le processus appelant est alors bloqué, jusqu'au retour de l'invocation. Le modèle d'objets passifs dispose dans ce cas d'un schéma d'invocation synchrone ;
2) l'objet est migré sur le site distant, ce qui évite de déplacer le flot de contrôle. Cette solution a été mise en oeuvre dans Guide (*primitives co_begin/co_end*) et Emerald.

En fait, tout se passe comme si virtuellement on ne disposait que d'un seul processus. De plus, l'objet ne prenant pas part à l'activation de ses méthodes, on augmente le degré de concurrence inter-objet (due aux activités concurrentes) et intra-objet (due aux activités présentes simultanément au sein d'un même objet).

3.4.2. Les objets actifs

Un objet actif est en général l'association :

- d'une file d'attente destinée à recevoir les messages ;
- d'un mécanisme de contrôle de concurrence déterminant quelles méthodes peuvent s'exécuter à un instant donné et sous quelles conditions ;
- d'une ensemble de variables d'instance ;
- d'une ensemble de méthodes.

L'objet gère l'acceptation des messages en fonction de son état et des méthodes en cours d'exécution. Un objet actif peut être considéré comme un processus spécialisé, appelé serveur et qui joue le rôle d'un serveur d'invocation de méthodes. Ainsi, un ou plusieurs processus sont assignés à chaque objet afin de gérer et contrôler les invocations d'objets qu'elles soient locales ou distantes. Contrairement à un objet passif, une chaîne d'invocations de méthodes met en jeu un processus à priori différent à chaque appel. Les invocations de méthodes sont asynchrones et réalisées par passage de messages.

Le nombre de serveurs est statique ou dynamique. Dans le premier cas, le degré de concurrence est limité au nombre de processus serveurs. Dans le second cas, chaque requête reçue entraîne la création d'un processus. Dans le cas où un processus serveur est bloqué en attente de la terminaison d'une méthode invoquée, il ne peut y avoir de parallélisme

intra-objet. Le parallélisme inter-objet lui est par contre maximisée, au prix d'un fort coût de gestion de processus.

3.4.3. Les acteurs

La notion d'acteur n'est pas récente [Hewitt & al. 73], mais a connu un essor considérable suite aux travaux de G. Agha [Agha 86a, 86b et Agha et al. 87]. Dans ce modèle, tout objet appelé un acteur, connaît un autre acteur à qui il envoie un message de continuation quand il a terminé son travail. Le flot de contrôle est donc transféré d'acteur en acteur et porte le nom de passage de continuation. Un acteur est identifié de manière unique par une référence. Cette référence, ainsi que son comportement courant et l'ensemble des messages non encore traités permettent d'obtenir son état. La prise en compte d'un message par l'acteur destinataire est appelé un événement. La continuation est contenu dans un message et correspond à la référence d'un acteur client auquel il faudra renvoyer une réponse. La notion de continuation a été introduite dans le premier langage d'acteurs séquentiels PLASMA [Hewitt & al. 73] et dans ses dérivés Mering IV [Ferber 83] et Actalk [Briot 89] entre autres.

Un script décrit le comportement autonome de l'acteur car le flot de contrôle est indépendant des appels de méthodes antérieurs. Les acteurs communiquent par échange de messages et leur comportement est composé de l'ensemble des accointances (les autres acteurs qu'ils connaissent) et des scripts qui décrivent leurs réactions en fonction des évènements reçus.

Le cycle de vie d'un acteur consiste à récupérer un message et à le traiter, ce qui peut le conduire à :

1) créer dynamiquement de nouveaux acteurs,
2) envoyer des messages aux acteurs dont il possède la référence,
3) changer son comportement courant par celui nécessaire pour traiter le message suivant. Ainsi, le traitement des messages se fait en exclusion mutuelle.

La granularité du parallélisme est donc l'objet lui même, c'est pourquoi chaque acteur s'exécute en concurrence avec tous les autres. Le modèle acteur offre donc de la concurrence au niveau inter et intra-objet.

3.4.4. Les objets mandataires

Les objets mandataires ou proxy [Shapiro 86] sont des objets délégués d'objets distants et se comportent exactement comme les objets qu'ils représentent. Les objets mandataires ont les propriétés suivantes :

- il conserve la sémantique du langage que les références aux objets soient locales ou distantes ;
- les références locales ne sont pas pénalisées par un mécanisme de gestion des références distantes ;
- certains objets sont garantis non migrables, il est donc inutile de tester leur réelle situation avant de les accéder ;
- Il n'effectue pas (en général) de contrôle des messages et de se contenter de les rerouter. Ils agissent comme des ports de communication.

Ils sont implémentés par des processus (ou des groupes de processus) et s'exécutent dans l'environnement client. Ils sont donc créés par le service distant spécifiquement pour un client.

3.4.5. Synthèse

Ces quatre types de modèles d'objets sont à la source des choix techniques des langages de programmation objet concurrent. Chacun utilisant l'un et pas l'autre ou les combinant

pour obtenir un degré de granularité et de parallélisme adapté au domaine d'intérêt du programmeur.

La distinction entre objets actifs et objets passifs a été fortement remise en question par [Meyer 93]. D'après lui, le terme objet actif sous entend qu'il n'est composé que d'un seul processus séquentiel et non pas d'un ensemble de processus qui s'exécutent en parallèle et qui sont intérruptibles. Le concept d'objet actif entraîne alors le programmeur à associer un objet et un algorithme donné. Restreindre ainsi un objet à un seul flot de contrôle (i.e. un processus), c'est se priver du coté dynamique du modèle objet (plusieurs routines pouvant s'exécuter concurremment au sein d'un même objet). Cette distinction implique aussi des contraintes sur les communications entre objets actifs. En effet, si un objet actif est en section critique et qu'un message arrive, il ne peut le traiter. Des mécanismes supplémentaires doivent donc être créés dans une couche supérieure pour gérer les accès concurrents (comme avec la clause *Accept* dans le langage Ada 83 et Ada 95 ou *Serve* dans Eiffel//), ce qui est inacceptable. Enfin, la notion d'objet actif et passif, n'a plus aucun sens dès qu'on travaille avec un modèle client-serveur généralisé, où tout client peut être serveur et vice versa (chacun jouant respectivement le rôle d'un object actif, puis passif). La solution qu'il adopte est de créer une nouvelle primitive (*Separate*) pour associer à chaque objet créé, un processeur virtuel responsable de l'exécution des invocations adressées à cet objet.

3.5. Conclusion

Les langages objets concurrents(2) et acteurs manipulent des entités logicielles qui ont un grain de concurrence et une granularité intrinsèque très variable. Le grain d'un objet étant calculé en tenant compte de sa partie contrôle (en fonction de l'état interne de l'objet et des processus qu'il gère) et de sa partie données (fonction des accès concurrents en mode synchrone ou asynchrone). Même si tous les langages ont comme granularité annoncé l'objet, cela ne veut pas dire que la granularité gérée est du même ordre de grandeur. Ainsi, Hybrid a des objets de grain moyen, Mering IV de grain fin et les objets de type OLE sont de gros grain.

Les architectures logicielles construites avec des langages orientés objets sont aussi très souvent constituées de quelques composants logiciels majeurs (les processus, les clients et les serveurs par exemple), de leurs relations (ce qui inclut les communications et le contrôle du flot de données) et d'une masse de petits objets échangés entre les modules à travers les relations (échange d'informations ou d'événements). Il est donc nécessaire de pouvoir gérer deux niveaux de granularité, le premier prenant en compte les composants de base à durée de vie longue (certains étant dit persistants) et le second, les petits objets à durée de vie limitée. Les études actuelles dans le domaine des langages objets concurrents porte sur la description de méthodes de développement d'applications réparties [Müllhaüser & al. 93] concernant d'une part le placement et la l'exécution des objets concurrents. **C'est aussi notre problématique dans MEDEVER, qui est une méthode conçue pour gérer la granularité variable de composants logiciels, qu'ils soient objets ou non.**

Les langages d'acteurs ont tracé la voie à de nouveaux types de langages, où le composant logiciel de base (l'agent), implanté par un acteur, devient autonome et mobile. Ainsi, les agents mobiles explorent leur environnement pour résoudre un problème seuls ou en coopération avec d'autres agents. L'environnement est alors organisé comme un lieu, pour permettre la mobilité. C'est le cas notamment de l'environnement Odyssey (anciennement Telescript) qui définit des agents logiciels mobiles dans un environnement organisé en places. Cet environnement, ainsi que Voyager et Aglets Workbench sont basés sur Java, un langage orienté objet et portable. La portabilité étant offerte nativement, grâce à sa compilation dans un format intermédiaire appelé Bytecode, ces environnements

(2) Le lecteur intéressé par les langages de programmations objets concurrents pourra se reporter aux articles suivants [Agha 90], [Gaudiot & al. 92], [Wyatt & al. 92], [Guerraoui 95], [Briot & al. 96] et [Papathomas 96].

s'attachent surtout à définir des règles de gestion de l'autonomie et de la coopération. Cela revient en fait à créer statiquement, lors de l'exécution de l'application, une architecture logicielle composée d'objets mobiles. Puis, il suffit de suivre l'évolution de cette architecture en fonction de la création, du mouvement et de la disparition de ces objets.

On se ramène donc, une fois de plus, à une architecture logicielle de définition issue des spécifications et à une architecture logicielle d'exécution qui évolue en fonction du mouvement de ses composants. Nous pouvons donc gérer ce type d'application dans MEDEVER.

L'architecture logicielle d'une application client-serveur est donc définissable, comme nous l'avons vu dans ce chapitre, directement au niveau du langage de programmation. La gestion de la concurrence passe alors très souvent par l'utilisation d'un environnement d'exécution spécifique (une machine virtuelle spécifique dans le cas de Java). Cet environnement d'exécution étant un sous ensemble de services systèmes greffés sur le système d'exploitation sous jacent. A contrario, il existe des systèmes d'exploitation ou des environnements systèmes suffisamment ouverts pour supporter l'intégration simple d'applications client-serveur. L'architecture logicielle est donc vue selon un axe système.

4. Les architectures client-serveur selon un axe système

Les architectures logicielles ouvertes s'opposent aux anciennes architectures propriétaires qui ne permettaient pas l'intégration aisée de composants architecturaux tiers. Cette évolution architecturale est en partie due à l'évolution rapide des technologies matérielles et des nouveaux systèmes et outils de développement logiciels. Ainsi, un système ouvert doit supporter cette ouverture selon trois axes principaux [Tsichritzis 89] :

1) la topologie : des applications ouvertes s'exécutent sur des réseaux configurables ;
2) la plate-forme : le logiciel et le matériel sont hétérogènes ;
3) l'évolution : les services à fournir sont instables et changent constamment.

Pour nous, la problématique des systèmes ouverts, des plate-formes d'intégration et des systèmes client-serveur généralisés ou universels se rejoignent par leurs objectifs. Le but à atteindre, ainsi que les méthodes pour les atteindre sont similaires, mais dans des contextes différents. D'où l'intérêt de factoriser ces techniques au sein d'une même méthode de développement. Ainsi, dans MEDEVER, on cherche à fournir ou à préserver l'interopérabilité et l'intégration d'applications client-serveur constituées de composants logiciels communiquant. Si on considère un axe système, on constate que l'exécution répartie et l'interopérabilité des applications sont avant tout gérées par les couches médiateurs et systèmes. C'est pourquoi, notre étude est découpée en cinq parties.

La première section présente les architectures de systèmes réparties ouverts que nous classons en trois familles distinctes, en fonction du niveau d'interopérabilité souhaité. Puis, dans la seconde section, nous examinons les plate-formes d'intégration et les moyens utilisés pour faire interagir des applications client-serveur entre elles. La section trois décrit les architectures à base de composants logiciels, qui mettent en oeuvre des technologies récentes et toujours en mouvement. La section quatre introduit les architectures basées sur des bus logiciels (ou *ORB*) adaptées à des objets mobiles se déplaçant dans un espace d'adressage global. La section cinq se focalise sur les avancées récentes dans le domaine des systèmes répartis et notamment sur la spécialisation de ces derniers, pour s'adapter au mieux aux besoins d'une application client-serveur. Enfin, la dernière section conclue sur l'existence de trois grandes familles de composants logiciels : les contenus exécutables, les objets répartis et les agents.

4.1. Les architectures de systèmes ouverts.

La construction d'architecture de systèmes ouverts simplifie le déploiement et la portabilité d'applications réparties complexes. Ce que l'on recherche avant tout avec ces architectures, c'est de la flexibilité. C'est pourquoi, les systèmes logiciels et matériels supports de leur exécution, doivent-être configurables indépendamment de l'application.

Ces systèmes pour être utilisables doivent satisfaire les contraintes suivantes :

- permettre à des applications différentes de coexister, voire de coopérer au sein de la même architecture de système ;
- incorporer différentes technologies aussi bien en ce qui concerne les protocoles de communication, que les systèmes d'exploitation, les plate-formes matérielles et les langages de programmation ;
- s'adapter à la taille des réseaux utilisés et au nombre de noeuds à gérer ;
- proposer un mode d'utilisation simple et efficace ;
- être flexibles et évolutifs.

L'idée sous jacente est de soulager l'application, et donc les développeurs, de la gestion des services de bases des systèmes répartis. Ainsi, la sécurité, les contrôles d'accès aux ressources, le nommage, (etc.) ne sont plus pris en compte dans l'application. En d'autres termes, on intègre les médiateurs directement au niveau du système, pour gagner en performance et en richesse fonctionnelle. Reste alors à évaluer ces architectures et leur capacité à répondre aux besoins de l'application.

Les systèmes récents qui ont attiré notre attention sont les suivants : ANSA, ODP, DCE, ATLAS, CORDS, CAE, MIA, ROSA, ISA. Une comparaison détaillée de ces architectures dépasse largement le cadre de notre étude, néanmoins un certain nombre de points clefs méritent d'être soulignés.

4.1.1. Les trois familles de systèmes mis en exergue

Globalement on peut découper en trois familles les systèmes actuels selon leur degré d'interopérabilité. Ainsi, certains se focalisent sur les systèmes d'exploitation, d'autres sur les couches intermédiaires et d'autres enfin sur les applications et leur conformité à une architecture de référence susceptible de les accueillir.

Interopérabilité au niveau du système d'exploitation

Au niveau d'interopérabilité le plus bas, on trouve les systèmes ATLAS, CAE et MIA. Ils sont basés sur des systèmes d'exploitation standards, dont on a normalisé les interfaces et les services de base, pour permettre leur coopération. Ainsi, par exemple, CAE a été développé en étendant des systèmes d'exploitation actuels (surtout les systèmes Unix), pour offrir une interface d'application unifiée. Un ensemble d'interfaces de services indépendantes de l'implémentation ont été définies, imposant la portabilité des applications au niveau du code source (plus de 1000 interfaces standards, connues sous le nom de *X-Open System Interface* sont supportées).

Interopérabilité au niveau des couches intermédiaires

Dans les systèmes dont l'interopérabilité est basée sur des médiateurs, on trouve ISA et DCE. Le plus connu, DCE *(Distributed Computing Environment)* est un environnement de développement et d'exécution de systèmes répartis. Il intègre des technologies choisies par l'OSF (*Open Software Foundation*) pour l'interconnexion et la coopération de systèmes et d'équipements différents. DCE s'appuie sur des services répartis pour assurer la coopération des applications (tels que les services de nommage et d'annuaire réparti, le service de gestion de temps, les services de sécurité, le service de gestion de fichier).

Interopérabilité au niveau d'une architecture de référence

Enfin, au niveau le plus haut du processus d'interopérabilité, certaines recommandations vont jusqu'à créer de toutes pièces une architecture de référence. Il est donc possible de tester, à priori (et donc avant implémentation), la conformité d'une architecture de systèmes en construction. Une fois validées, les applications conformes à l'architecture de référence sont raffinées jusqu'à obtenir le système final. La compatibilité par rapport à une architecture est verticale (spécifique à un domaine) ou horizontale (on décrit les services indispensables quelle que soit l'architecture). ANSA, ROSA et ODP en sont les exemples les plus connus.

ODP *(Open Distributed Processing)*, est issu d'une procédure de normalisation internationale, concernant les systèmes répartis ouverts. Son modèle de référence (*RM-ODP*) présente une collection de concepts et de termes pour décrire des architectures de systèmes réparties. La notion de point de vue, introduite dans ANSA et reprise dans ODP, se révèle pratique dans une phase de développement d'architecture. Ces points de vue sont des mises en oeuvre de la notion de vue d'une architecture présentée dans la Section 1.1. et sont au nombre de cinq :

1) **le modèle de l'entreprise** *(Enterprise Model)* pour exprimer les limites du système, ses règles et son utilité. Il décrit l'organisation de l'entreprise et ses changements;
2) **le modèle d'information** *(Information Model)* pour exprimer la signification de l'information répartie et des processus qu'on lui applique;
3) **le modèle de traitement** *(Computational Model)* exprimant la décomposition fonctionnelle de l'application en unités de distribution;
4) **le modèle d'ingénierie** *(Engineering Model)* décrivant les composants et les structures nécessaire pour supporter la distribution (les fonctions de l'infra-structure);
5) **le modèle technologique** *(Technology Model)* qui décrit la construction de l'application en terme de composants, avec des règles de conformance pour sa réalisation.

4.1.2. Granularité et type des composants gérés

L'unité de base d'opération et d'administration dans ANSA et DCE est constituée d'utilisateurs, d'ordinateurs et de ressources orientés en général vers un objectif commun. ANSA appelle cette unité de gestion de base un noeud et DCE une cellule. Les autres systèmes étudiés étant basés sur des modèles orientés objet, l'unité de gestion est l'objet. La granularité est donc infiniment plus faible.

La vision d'applications réparties inter-agissant en point à point, à conduit les membres du projet CORDS à adopter un modèle orienté processus [Strom 86] et [Bauer & al. 93], plutôt qu'orienté objet. Les processus sont alors utilisés comme brique de base et ne contiennent pas de parallélisme.

4.1.3. Gestion des communications

En conservant la sémantique des appels de procédure, grâce au paradigme de communication RPC (*Remote Procedure Call*) [Birrell & al. 84], on facilite la distribution des composants d'une application. Une application répartie s'appuyant alors sur les mêmes concepts qu'une application locale. C'est pourquoi, toutes les architectures étudiées utilisent un paradigme de communication client-serveur basé sur les RPC, tout en manipulant des composant logiciels simples (comme dans ATLAS) ou composites (comme la cellule dans DCE).

Seul CORDS ne considère pas des architectures client-serveur et utilise des communications point à point *(peer to peer)* pour offrir une flexibilité et une dynamicité maximale. Il n'existe donc pas de notion d'état global et toutes les données sont locales à un processus (il n'existe pas de variables partagées). Les données et les processus sont persistants

et un processus actif interagit avec un autre processus en créant un canal (*channel*) par l'intermédiaire de ports.

4.1.4. Services de bases et interfaces de programmation

Ces architectures comportent tous les services de bases suivants : la gestion de la sécurité (souvent via Kerberos [Steiner & al. 88]), la gestion des horloges réparties, le gestion du nommage, la duplication des données et l'administration des applications au sein de l'architecture.

Tous ces systèmes disposent d'une interface de programmation compatible avec le langage C (voire C++ en ce qui concerne ANSA, CAE) et de la gestion des processus légers (*thread*). La création des souches clientes et serveurs des programmes créés passent par l'utilisation d'un langage de description d'interface (*IDL, voir OMA*) en général propriétaire. C'est d'ailleurs pourquoi certains systèmes offriront prochainement des passerelles vers des ORB compatibles CORBA (tels que CAE et ANSA).

4.1.5. Synthèse

Les implémentations d'architectures de systèmes ouverts réparties ne respectent pas toujours le modèle en couches qui les modélise, pour des raisons de performances principalement. Par contre le modèle en couche donne une vision simple des différents services intégrées au sein du système.

La spécialisation d'un modèle de référence générique peut conduire à des modèles de références spécifiques. L'exemple le plus connu à partir d'ODP, est issu du monde des opérateurs de télécommunications, qui travaillent sur TINA (*Telecommunication Information Network Architecture*) [Brown 94 et Stephani 95], une architecture de réseaux intelligents de télécommunication. A chaque vue du modèle de référence on associe des styles architecturaux adaptés à la description d'une facette d'un système de télécommunication. Ensuite, pour éviter les écueils du passage d'un modèle de référence à une architecture réelle, la réutilisation de composants déjà existant d'un environnement est donc nécessaire. OSF/DCE et ANSAware semblent désormais être les deux solutions privilégiées pour gérer les services de bases entre applications réparties hétérogènes complexes.

Ces solutions ne nous semblent pas adaptées à notre problématique, car elles sont trop complexes à mettre en oeuvre et ont un coût d'achat prohibitif. Par contre, nous pensons que la tendance actuelle de ne conserver que certains services offerts par ces systèmes (comme les RPC d'OSF/DCE ou les services de sécurité de Kerberos [Steiner & al. 88]), afin de les utiliser comme médiateurs continuera de se développer.

Dans la suite, nous présentons une autre technologie favorisant l'intégration et la réutilisation d'applications déjà existantes au sein d'une même architecture logicielle.

4.2. Les plate-formes d'intégration

L'intégration des applications et des outils utilisés par l'entreprise au sein d'une même architecture logicielle a abouti à la création de plate-forme d'intégration. Ces plate-formes ont pour but de conserver les outils existants, tout en permettant aux utilisateurs d'y accéder de manière transparente et sécurisée. Ces architectures sont appelées des fonds de panier logiciel (*Software Backplane*) ou des grille pains (*Toaster*). Toutes ces dénominations sont issues d'une vision architecturale où les outils qui doivent coopérer sont enfichés sur un même bus, qui se trouve être dans ce cas logiciel (cf. Figure 17).

Ce bus est généralement constitué d'un serveur de messages s'appuyant sur un système d'exploitation. L'une des premières plate-formes mise au point industriellement fut AD/Cycle d'IBM [Mercurio & al. 90]. Elle offrait des services d'intégration de données et de communication inter-application par messages. Mais cette plate-forme restait trop centralisée, très difficile à gérer et à étendre et trop propriétaire pour être réellement intero-

pérable. C'est pourquoi AD/Cycle d'IBM est aujourd'hui un projet officiellement abandonné par IBM.

La norme ECMA PCTE a, par la suite, ouvert la voie de la conception modulaire de structures d'accueil suivant des axes d'intégration clairement identifiés [Wasserman 90]. La décomposition en modules fonctionnels est alors associée à une description précise des interfaces d'accès aux services. Ces interfaces et les services offerts étant stockées au sein d'un référentiel unique, accessible par tous les outils. La norme ECMA PCTE ne spécifie pas, par contre, l'extensibilité de la plate-forme elle même et présente des carences en terme de description de service de communications inter-outils. Or, la croissance du nombre d'utilisateurs et donc du nombre de requêtes à traiter rend la gestion des communications au sein de la plate-forme cruciale.

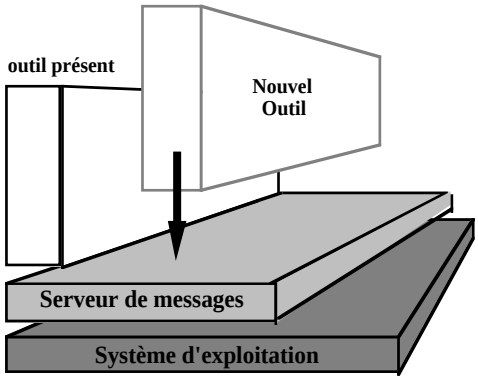


Figure 17 : Architecture d'intégration de type grille pain

C'est pourquoi ces plate-formes font désormais appels à des serveurs de messages normalisant les services communications au sein de plate-formes ouvertes et conformes (complètement ou partiellement) à la norme ECMA. Ces serveurs de messages se situent au dessus de la couche du système d'exploitation. Il en est ainsi pour le serveur de messages ToolTalk [Frankel 91], BMS (*Broadcast Message Server*) disponible dans l'environnement de génie logiciel HP-Softbench [Cagan 90] ou le serveur de messages NMS (*Network Message Server*) [Foughali 94] clef de voûte de la plate-forme d'intégration d'applications réparties Mathilda (*Method And Tools for Heterogeneous Integration and controL of Distributed Applications*) [Bonnaire & al. 96].

Synthèse

Les plate-formes d'intégration[(3)]**, bien que très utiles pour l'intégration d'applications monolithiques, sont de moindre intérêt pour la gestion d'application composées de composants logiciels communiquant à grain fin (les objets répartis par exemple).** Néanmoins, les serveurs de messages eux sont de plus en plus utilisés, car ils offrent des services transparents d'envoi de messages à des composants ou à des groupes de composants logiciels. Nous étudions d'ailleurs dans la suite les architectures logicielles à base de composants logiciels centralisés ou répartis.

4.3. Les architectures logicielles à base de composants logiciels

Les composants logiciels donnent prise à un vieux rêve de programmeur : ne pas redévelopper sans cesse à partir de rien. Une application est alors construite en respectant une architecture donnée et par assemblage des briques logicielles statiques ou dynamiques.

(3) Le lecteur intéressé par les plate-formes et la norme ECMA PCTE peut consulter les références suivantes [Lonchamps & al. 92], [Long & al. 93] et [Schefstrom & al. 93].

Définition 16 : Modèle de composant

Un modèle de composants est représenté par une architecture logicielle et un ensemble de protocoles de communication et de synchronisation permettant de combiner statiquement ou dynamiquement des composants logiciels via leurs interfaces pour créer une application. Un modèle de composants est donc un gabarit de conception.

Les composants disposent souvent d'une représentation visuelle spécifique aidant le programmeur à les dissocier et à mieux s'en servir [Arrouye 96]. La définition, la mise en oeuvre et l'utilisation de ces composants à subi de nombreuses évolutions. Dans la suite de ce chapitre, nous présentons brièvement des bibliothèques de programmation. Ensuite, nous décrivons comment elles ont permis de créer des composants additionnels à des applications ouvertes. Puis, nous analysons les impacts des technologies objets sur les composants et leurs évolutions vers un modèle hiérachique, composite supportant la mobilité.

4.3.1. Les bibliothèques logicielles réutilisables

Lorsqu'un développeur souhaite écrire un programme sur une plate-forme donnée, dans un langage de programmation donné, il lui est possible d'utiliser des bibliothèques de haut niveau, les API. Ces bibliothèques sont en général propriétaires et fournies avec un kit de développement (*SDK - Software Development Kit*).

Définition 17 : Interfaces de programmation d'applications

Les interfaces de programmation d'applications (Application Programmer Interface), ou API, sont des bibliothèques de haut niveau classées par domaines d'applications. On trouve des API de programmation graphique (OpenGL par exemple), de communication (comme les RPC) et de programmation d'applications (la bibliothèque de gestion de fichiers en langage C).

Des bibliothèques plus ouvertes et portables telles que X11 (pour l'affichage graphique) ou PVM [Geist & al. 93] (pour la communication entre programmes s'exécutant sur des plate-formes hétérogènes) sont nées par la suite, pour répondre des besoins de distribution sur des plate-formes matérielles diverses. L'intégration de ces bibliothèques se fait au niveau du code source. Il en résulte des programmes compilés ayant des tailles importantes.

Les API ont connu un second souffle lors de l'essor des langages de programmation et des méthodes objets. Des langages comme C++ ou SmallTalk ont permis très rapidement le développement de bibliothèques de classes spécifiques (graphisme, communication, gestion des données, etc.) et réutilisables. Les API ont donc été remplacées par des bibliothèques de classes qu'on parcourt visuellement avec un explorateur de classes (*class browser*). On peut citer par exemple des bibliothèques telles que les MFC pour la programmation sous Windows (*Microsoft Foundation Classes* de Microsoft), les MacApp d'Apple pour la programmation sur Macintosh et l'AppKit de Next pour la programmation au dessus du système Nextstep.

Ces bibliothèques sont donc d'assez bas niveau et dédiées au programmeur expérimenté. La construction de composants logiciels de plus haut niveau a donc été engagée.

4.3.2. Les composants logiciels additionnels

Les composants logiciels en général se contentent d'être utilisables tels quels et n'ont pas de liens directs avec des objets. Ils sont prévus pour s'insérer dans un outil de développement et encapsulent des fonctions de bas niveau. Lors de la compilation, le code de ces

composants est intégré soit directement dans le code source (comme dans Delphi 1.0 de Borland), soit déporté dans des bibliothèques dynamiques qu'il faudra joindre à l'environnement d'exécution de l'application lors de son déploiement.

L'un des exemples le plus connu de ce type de composant est lié à l'environnement de développement Visual Basic de Microsoft. Dans cet environnement, les composants s'appellent des VBX (*Visual Basic Control*) et sont obligatoirement développés en C++. Ces composants se rajoutent visuellement à ceux déjà présents dans l'environnement de développement (appelés contrôles) et à l'environnement d'exécution (sous forme de bibliothèques dynamiques). Leur utilisation n'a pas de répercussions sur la taille du code puisqu'ils utilisent des bibliothèques séparées. Même si les VBX ont ouvert la voie à l'utilisation de composants, ils avaient deux défauts de jeunesse : ils n'étaient pas toujours très utiles (ni bien documentés) et ils présentaient parfois une incompatibilité avec les outils censés les supporter. Il a donc fallu créer des composants logiciels plus robustes et mieux définis à leurs interfaces.

4.3.3. Les composants logiciels orientés objets

Les composants additionnels ont alors laissé la place à d'autres types de composants, orientés objets et d'une granularité moins forte. La granularité plus faible a entraîné des conséquences sur la modularité des applications, en imposant la mise au point d'un modèle de coopération entre objets robustes et performants. Or, la programmation orienté objet, quoi que très puissante, n'offre pas d'infrastructure standard pour que des produits de vendeurs différents interagissent dans le même espace d'adressage. A fortiori, cette constatation s'applique pour des objets dans des espaces d'adressage différents communiquant à travers le réseau et/ou sur des machines hétérogènes. Si la communication par messages est une première réponse à ce problème, elle n'induit pas de schéma de coopération élaborée entre les composants.

Comment alors définir un système tel que des composants divers provenant de sources variées soient garantis d'être interopérables au niveau binaire ? Pour créer un tel système, il a fallu répondre aux questions suivantes :

1) *interopérabilité de base* : comment être sûr que des composants seront interopérables avec d'autres ?
2) *gestion des versions* : est-ce que la mise à jour des composants n'aura pas d'effets sur les autres ? Doit-on faire des mises à jours pour tous les composants ?
3) *indépendance du langage de programmation* : Comment faire interopérer des composants écrits dans différents langages ?
4) *Interopérabilité intra ou extra processus transparente* : Comment faire s'exécuter un composant à l'intérieur d'un processus, dans un autre processus (distant) en utilisant un modèle de programmation simple ?
5) *Performance* : comment ne pas annihiler les performances d'un composant lors de son intégration avec d'autres ?

Les réponses à ces questions ont été multiples et ont entraîné la mise en chantier de nouvelles méthodes de développements et de nouvelles technologies. Dans la suite de ce paragraphe, nous présentons des solutions actuelles pour décrire des composants, pour les intégrer dans d'autres composants et pour les partager et éventuellement les déplacer dynamiquement.

4.3.3.1. Intégrer des composants dans d'autres : les documents composites

La réponse à ces questions a entraîné la naissance d'un nouveau type de médiateur : les médiateurs objet (*Object Middleware*). Un médiateur objet permet à des composants actifs d'applications d'interopérer. Le meilleur exemple de ce type de composants est représenté par l'émergence des **documents composites actifs** (*compound document*) [Wayner 94].

Un document composite actif est un document qui peut être le conteneur et/ou disposer de références sur d'autres documents. C'est un morceau de code réutilisable couplé à des données dans un format binaire intégrable dans un composant tiers. En général, les composants logiciels composites doivent adhérer à un format binaire extérieur nécessaire à leur exécution, mais sans que cela restreigne d'aucune sorte la manière dont ils sont implémentés. Ils sont soit développés avec des langages de programmation procéduraux (très rare), soit avec des langages et des *frameworks* orientés objets. Ainsi, par exemple, Opendoc/SOM est une architecture ouverte conçue pour intégrer des outils orientés objets de gestion de documents composites. Son but principal est de réduire au maximum la complexité d'une application en réutilisant des objets binaires inter-changeables.

Mais au delà de la notion de documents composites, ce que l'on voudrait, c'est créer des applications réparties basées sur des composants logiciels statiques ou mobiles.

4.3.3.2. La mobilité et le partage de composants logiciels

Pour éviter de retomber dans des architectures à base de composants logiciels propriétaires et hétérogènes, les clients ont été dotés de machines virtuelles exécutant du code mobile.

Définition 18 : Code Mobile

*Un code est dit mobile s'il est **transportable** (le lieu de son exécution est indépendant de son lieu de stockage) et **portable** (le support de ces exécutions ne se restreint pas à un type particulier de machine).*

Ce code mobile (en général précompilé) à l'avantage d'être portable, exécutable à la volée et peu gourmand en ressources. L'exécution de code mobile est réalisée soit :

1) par émulation matérielle du jeu d'instruction du programme. Ainsi, sur une plate-forme Apple, il est possible d'exécuter une application compilée pour un processeur Motorola 68K sur un processeur Power PC.
2) par interprétation d'un pseudo code indépendant des plate-formes matérielle, mais dépendant d'une machine abstraite fixée (tels que Java ou Smalltalk Agent [Quasar 94]). Il existe aussi dans ce domaine, des optimiseurs à la volée de byte-code Java qui accélèrent sensiblement son exécution (*just in time compiler*).
3) par recompilation automatique d'un pseudo code indépendant des plate-formes tel que celui défini dans Oasis.

L'utilisation de code mobile a aussi eu des répercussions sur les primitives de sécurité à mettre en place car, comme le traitement se déplace, il faut-être sur qu'il ne s'agit pas d'un programme de dégradation (virus) ou de recopie illégale (espionnage) des données.

Les deux produits phares de production de code mobile sont Java de Sun Microsystems et Telescript de General Magic. Tous les deux sont des langages de programmation proche de C++ et s'exécutent sur une machine virtuelle qui doit être présente chez le client. Du fait de leur lien très étroit avec leur machine virtuelle, ils sont incompatibles entre eux. Java supporte la création de petites applications capables de s'exécuter chez le client après téléchargement (*applet*). Telescript lui introduit le concept de mobilité ayant comme particularité de faire migrer le programme vers l'endroit où se trouve les données. L'approche de Sun avec Java est de mettre en place des applications client-serveur avec possibilité de téléchargement (ou pas) du code, avec que General Magic à une approche résolument axée sur la recherche d'information (commande *GO*) et la gestion de la coopération (commande *MEET*). Java et Telescript offrent des primitives de sécurité de haut niveau assez proches, avec néanmoins un surplus de contrôles pour Telescript du au fait que son code peut migrer. Une version Java de Telescript, nommée Odyssey, est en cours de réalisation. Odyssey n'implémentera pas la fonction GO, du fait de l'impossibilité du langage Java de sauvegarder l'état d'un processus léger en cours d'exécution avant son déplacement.

4.3.4. Synthèse

La définition de code mobile que nous avons donnée est restrictive si l'on se place dans le cadre plus général des systèmes agents. Dans ce dernier cas, ce n'est pas du code qui migre, mais un objet. L'intérêt principal du modèle objet étant justement de rendre transparent les différences entre les mécanismes d'invocation de méthodes distantes et de transfert de code (rapatriement, compilation optimisation et exécution du code source). Ainsi, du point de vue du programmeur, seuls deux objets communiquent.

Ce schéma de coopération remet donc en cause le modèle client-serveur traditionnel, dans le sens ou on ne sait plus qui est client, qui est serveur et quelles sont les informations qui transitent (code ou données). On parle alors dans ce cas de client-serveur universel.

Sun a d'ailleurs dû modifier son langage Java, pour qu'il gère des applets s'exécutant sur le serveur (appelés *servlet*). Ainsi, se plaçant dans un contexte client-serveur généralisé, l'exécution d'un programme Java est réalisé sans que l'utilisateur sache si c'est un applet ou un servlet. Pourtant, dans le premier cas, le code est transféré et exécuté sur le client, alors que dans le second, seules les informations concernant la gestion de l'affichage et des entrées sorties seront prises en compte. **C'est pourquoi dans MEDEVER nous n'avons pas voulu donner de l'importance à la notion de client et de serveur. Nous avons préféré centrer notre étude sur la notion de granularité et d'interopérabilité entre composants logiciels.** Nous sommes alors aptes à prendre en compte le paradigme client-serveur généralisé, aussi appelé universel.

En ce qui concerne l'interopérabilité, elle est limitée par la notion de machines virtuelles. Les machines virtuelles exécutant du code mobile étant propriétaires, le client comme le serveur doivent disposer de machines virtuelles multiples. L'utilisation de passerelles entre ces machines virtuelles est réalisable en utilisant des bus logiciels, aussi appelé ORB.

4.4. Les architectures logicielles basées sur des ORB

L'OMG (Object Management Group) est un consortium, qui a été fondé en 1989 à l'initiative de grands constructeurs et éditeurs informatiques (Data General, Hewlett Packard, Sun, Canon, American Airlines, Unisys, Philips, IBM, etc.) afin de normaliser les systèmes à objets répartis. Les spécifications de l'OMG contiennent un modèle objet, une architecture de référence l'OMA (*Object Management Architecture*) et un langage de définition d'interface (appelé IDL). Au centre de l'architecture OMA (*Object Management Architecture*) se trouve un bus logiciel de communication appelé ORB (*Object Request Broker*).

Définition 19 : Object Request Broker - ORB

C'est un bus logiciel par lequel des objets envoient et reçoivent des requêtes et des réponses sans connaître la localisation de leur(s) interlocuteur(s).

Le rôle de l'ORB est d'enregistrer la localisation des objets, de recevoir, d'accepter et de router les requêtes de services. L'ORB s'assure que la requête est exécutée correctement. Les ORB sont implémentables de manières différentes :

- soit ils sont programmés directement dans le client ou dans l'application.
- soit les ORB et les applications sont implémentés comme des bibliothèques résidant chez le client (comme par exemple OLE 2 de Microsoft).
- soit l'ORB est implémenté comme un serveur (un processus séparé) qui aiguillent les requêtes entre les clients et les processus de l'application (comme par exemple Orbix de Iona)

- soit l'ORB est implémenté directement dans le systèmes d'exploitation (Cairo de Microsoft ou Taligent d'IBM).

La description des opérations potentiellement déclenchables à distance sont décrites grâce au langage IDL *(Interface Definition Language)*. Ainsi, il est possible en fonction du type de l'opération, des types des arguments et du résultat passés de déduire les opérations canoniques à réaliser. IDL dissimule les mécanismes sous jacents à la programmation répartie, mais pas le fait que les composants de l'application sont sur le réseau. D'autre part, même si l'IDL de l'OMG est normalisé, la tentation d'étendre cet IDL pour exprimer des contraintes toujours plus nombreuses (comme la synchronisation, la qualité de service attendu, la tolérance aux pannes) est forte.

L'architecture adoptée pour les ORB est connue sous le nom de CORBA (*Common Object Request Broker Architecture*). C'est une infrastructure de communication pour des objets distribués hétérogènes collaborant localement (*co-located*) ou à distance (*remote*). Normalisée depuis 1992, pour faciliter l'interopérabilité entre objets de toutes sortes la norme a évolué en 1995 vers CORBA 2.0 [OMG 95] en offrant l'interopérabilité entre ORB différents. Une autre vision, propriétaire cette fois ci, de la création d'architecture logicielle via des composants logiciels répartis est donnée par le couple OLE/COM. Dans la suite de ce chapitre, nous étudions chacun de ces deux modèles d'architectures et traitons de leur interopérabilité.

4.4.1. CORBA

CORBA 1.x est une spécification d'infrastructure de communication au niveau applicatif fournissant des mécanismes au moyen desquels des objets peuvent émettre des requêtes et recevoir des réponses. Des applications actives sur différentes machines d'un environnement réparti hétérogène coopèrent par appel d'opérations sur des objets. L'architecture client-serveur ainsi créée s'appuie sur des mécanismes d'appels de procédure à distance.

Dans CORBA, le client est un programme qui a accès à l'environnement CORBA et qui a été écrit dans un langage possédant une projection IDL. Un service est formalisé sous la forme d'un objet passif. L'implémentation définit l'information décrivant le service (données et traitements). Le serveur est l'entité qui donne accès à un ou plusieurs services en allouant les ressources et en les instanciant. C'est un conteneur de services objets (équivalent à la notion de capsule ODP).

Lors d'une communication entre un client et un serveur, le noyau de l'ORB transporte les requêtes sur le réseau et assure l'uniformité des accès aux objets. L'adaptateur d'objet (*Object adapter*) reçoit les requêtes adressées aux objets et adapte la sémantique d'invocation des objets CORBA à celle de l'implantation des objets. Les squelettes d'opérations décodent les requêtes CORBA pour invoquer le code des objets (c'est le complément de l'adaptateur d'objet).

Autour de l'ORB, se greffent deux types d'interfaces client pour accéder aux objets distants : l'interface d'invocation statique (*Static Invocation Interface*) et dynamique (*Dynamic Invocation Interface*). La première est un ensemble d'opérations, fonctions ou procédures applicatives, accessibles dans le langage du client, utilisée pour l'invocation d'un objet distant de manière transparente. La seconde offre la découverte d'objets, la découverte de leurs interfaces, la création des requêtes, la réalisation de l'invocation distante du service et la réception des réponses. Le concurrent le plus sérieux de CORBA est donné par le couple OLE/COM de Microsoft.

4.4.2. OLE/(D)COM

La définition et l'implémentation des services OLE ont tellement évolué ces dernières années, qu'il est difficile aujourd'hui de s'y retrouver. Néanmoins, OLE est avant tout un standard de communication entre applications provenant de différents vendeurs et un moyen de gestion des relations complexes entre des documents.

Dans OLE, un service (aussi appelé un composant) est fourni à travers un ou plusieurs objets, chaque objet consistant en un groupe logique offrant certaines spécificités du composant. Ces objets forment les liens de communication entre l'utilisateur de ces objets (du code) et le fournisseur de ces objets. Un client qui maintient la cohérence d'objets persistants est appelé un conteneur (*container)*. Le fournisseur d'un composant (et les objets qui le constituent) est appelé un serveur. Pour résumer, un client utilise un composant offert par un serveur et la communication se fait à travers des objets OLE. OLE reste orienté poste de travail car il utilise le mécanisme d'appel à distance LRPC (*Lightweight RPC*), qui n'autorise pas de communication à distance mais seulement l'émission de requêtes sur le serveur.

Les technologies OLE de Microsoft, ne s'appliquent pas réellement à des objets mais à des composants propriétaires. Ainsi, par exemple, les composants OLE 1.0 ne contiennent aucun traitement et disposent uniquement d'un lien vers l'application source appelée lorsqu'ils sont accédés. Avec OLE 2.0, le mécanisme d'activation sur place permet de modifier un objet sans quitter l'application, sans qu'on puisse étendre des composants ou modifier leur comportement. Il n'existe pas non plus d'héritage, ce qui prouve définitivement que les composants OLE ne sont pas des objets.

Lorsqu'on édite un «objet» OLE 2.0, ce n'est pas une fonction spécifique qui est appelée, mais toute l'application propriétaire du composant. C'est pourquoi, dans la pratique, on constate que l'activation de plusieurs composants OLE 2.0 créés par la même application entraîne l'exécution d'autant d'instances de l'application que de composants activés. L'accès à des fonctions spécifiques, sans lancer toute l'application est quand même réalisable avec *OLE Automation*. Cette fonctionnalité n'est offerte qu'au programmeur d'application et en aucun cas à l'utilisateur final. Remarquons enfin que l'appel d'une fonction OLE entraîne l'exécution de l'ensemble de l'application Serveur d'*OLE Automation*. On ne dispose donc pas d'une modularité à grain fin.

4.4.3. Interopérabilité

Malgré les révisions de la norme CORBA, apportées par la version 1.2, et basées sur des retours d'expérience des principales implémentations industrielles, rien n'avait été fait pour permettre une interopérabilité entre ORB. Or, la réutilisation des composants objets via CORBA n'est réalisable que si l'interopérabilité est assuré entre objets, mais aussi entre ORB. C'est pourquoi, l'architecture de CORBA 2.0 peut être vue comme une extension de celle de CORBA 1.x qui reste compatible uniquement au niveau des interfaces. L'apport de CORBA 2.0 s'est fait dans deux directions : l'interopérabilité entre ORB et une meilleure intégration avec les langages de programmation C++ et Smalltalk.

En ce qui concerne la communication entre ORB, CORBA 2.0 définit :

- une architecture pour des communications entre ORB respectant l'architecture CORBA;
- une API pour ajouter des ponts entre ORB;
- un protocole de communication entre ORB, appelée UNO (*Universal Network Object*). Grâce à UNO, un client d'un ORB donné peut invoquer une opération d'un objet disponible sur un ORB différent. Pour cela deux protocoles existent :
  - GIOP (*General Inter-ORB Protocol*) qui spécifie une syntaxe de transfert standard et un format de messages pour les communications entre ORB. Il existe une spécialisation du GIOP pour Internet (ie. sur des réseaux utilisant le protocole TCP/IP) appelée IIOP (*Internet InterOrb Protocol*).
  - ESIOP (*Environment Specific Inter-Orb Protocol*). ESIOP a été créé principalement pour offrir l'interopérabilité avec des systèmes répartis existants non conformes à GIOP et est basé sur CIOP (*Common Interoperability Protocol*) de DCE.

UNO impose une représentation commune des types de données du modèle objet de CORBA. Avec cette représentation, appelée CDR (*Common Data Representation*), il est possible de coder tout type descriptible en IDL.

Interopérabilité CORBA / CORBA dans la réalité

Du point de vue du dialogue entre ORB, l'objet client doit obtenir la référence de l'objet qu'il appelle et qui devient l'objet serveur, avant de le solliciter. Cette référence, appelée IOR (*Interoperable Object Reference*) est actuellement délivrée sous la forme d'un fichier texte. L'IOR contient à la fois le gestionnaire de l'objet et l'adresse physique de l'objet serveur actif. Cet IOR est totalement transparent pour le client, qui n'a pas besoin de connaître la manière dont s'effectue la localisation d'un objet. Tout objet serveur est configuré de manière à écrire dans un fichier une représentation de son IOR. Ce point est l'une des principales difficultés pour l'administration des objets CORBA, car le transfert des fichiers IOR entre ORB est difficilement gérable à grande échelle. Des services de nommage inter-ORB doivent être mis au point et intégrés aux outils actuels pour résoudre ce problème.

Interopérabilité CORBA / OLE - COM

Les deux standards de gestions d'objets répartis, à savoir CORBA et OLE/DCOM, malgré une conception et des intérêts radicalement différents convergent l'un vers l'autre. L'objectif de l'OMG est de définir un environnement d'exploitation pour interpréter directement les abstractions propres aux langages orientés objets (classe, héritage, invocation de méthode) et les traduire au niveau de l'exécution. C'est une approche de haut niveau basée sur un modèle d'objet neutre. L'OMA définit donc les abstractions qui devront être comprises par l'environnement d'exécution, en utilisant un langage de description d'interfaces commun (l'IDL). L'approche de Microsoft avec OLE/COM, par contre, se définit d'emblée comme un standard d'interopérabilité binaire, par opposition avec tout standard d'interopérabilité de plus haut niveau. COM est un ensemble d'API donnant accès aux mécanismes à travers lesquels un client put se connecter à différents fournisseurs du service requis de manière polymorphe. Un objet COM est l'implémentation d'un ensemble d'interfaces, chacune d'entre elles étant réellement un tableau de pointeurs vers des fonctions exécutables issues de la compilation d'un langage source.

Malgré ces différences, l'interopérabilté entre CORBA 2.0 et COM est actuellement en cours de normalisation au sein de l'OMG. Le RFP5 (Request For Proposal) décrit un mécanisme d'interopérabilité à deux niveaux : soit entre OLE2 et CORBA, soit entre COM et CORBA. En attendant les résultats des ces travaux, les intégrations possibles entre OLE et CORBA sont les suivantes :

1) les fournisseurs de produits OLE implémentent CORBA, ce qui entraîne que tout objet OLE est vu comme un objet CORBA;
2) les éditeurs d'outils de développement modifient leurs produits pour qu'ils génèrent des applications OLE ou CORBA;
3) les fournisseurs de médiateur développent des passerelles logicielles entre les deux.
4) les solutions d'intégration sont mises en place au niveau des bibliothèques systèmes.

4.4.4. Synthèse

Toutes les technologies présentées utilisent des RPC pour la communication, sont compatibles ou disposent de passerelles vers des ORB et supportent la distribution des données et des services. Malheureusement, les versions 1.x de la norme CORBA ne spécifiaient pas les protocoles d'interopérabilité entre ORB de constructeurs différents. Si bien qu'on peut dire qu'une majorité des implémentations de la première génération d'ORB étaient propriétaires et non ouvertes.

La norme CORBA 2.0 a vite corrigé cette lacune et a même établi des ponts avec les technologies OLE/DCOM. Le choix de la technologie nécessaire au développement d'une application en est donc simplifié. Des études récentes montrent qu'OLE et Opendoc sont utilisés pour le développement de la partie cliente, alors que CORBA s'impose surtout sur la partie serveur (notons que le terme serveur dans la terminologie OLE, se réfère soit au serveur logiciel qui se trouve sur la même machine que le client, soit à un serveur distant). La couche intermédiaire étant un ORB compatible CORBA 2. Le Tableau 23 présente une comparaison des technologies OLE/COM et CORBA.

Remarquons enfin que les technologies de type OLE ou CORBA ne s'intéressent qu'à la technique de communication entre objets. Certes, le programmeur dispose d'une infrastructure pour mettre des objets en relation, mais ils ne savent pas se comprendre. Les objets applicatifs sont liés de manière générale au monde réel et en particulier aux métiers de l'entreprise. Il est actuellement difficile de prendre des objets applicatifs (aussi appelés «objets métiers») et de les assembler pour en faire une application. C'est ce que tente néanmoins IBM, avec *CommonPoint* issu des travaux sur Taligent (aujourd'hui abandonnés), en offrant des classes d'objets réutilisables par métiers (banque-finance, enseignement, calcul scientifique).

Tableau 23: Comparaisons des technologies OLE/COM et CORBA

Critères	**OLE/COM**	**OMG CORBA**
Standard	Propriétaire	Indépendant
Indépendance vis à vis du langage	Compatible au niveau binaire : COM-IDL	OMG IDL
Paradigme	Basé Objet	Orienté Objet
Accès aux objets	Une parmi plusieurs interfaces	Seulement à travers les interfaces définies
Référence à un objet	API ou résultat d'un appel de méthode	Résultat d'un appel de méthode
Interface de base	*IUnknown*	*Object*
Identification des objets	Unique et non persistant	Unique et référence d'objet persistante
Accès aux interfaces distantes	Proxys et talons	Talons-IDL et Squelettes-IDL
Héritage	Non supporté, utilise l'agrégation	Héritage d'interfaces multiples
Chargement	Automatique pour les DLL et les exécutables	Automatique grâce à l'adaptateur objet
Appel statique de méthode	À la main	Liaison des talons-IDL
Appel dynamique de méthodes	OLE Automation	Interface d'invocation dynamique
Codage / décodage	Empaquetage en partie à la main	IDL et ORB

Tableau 23: Comparaisons des technologies OLE/COM et CORBA

Critères	OLE/COM	OMG CORBA
Récupération d'information sur les types	Librairies de types (ODL)	Entrepôt d'interfaces (IDL)
Persistance	Fichiers composites	Service d'objets de gestion de la persistance

L'utilisation de technologies propriétaires telles qu'OLE ne sont pas compatibles avec la démarche que nous avons entreprise dans notre méthode MEDEVER. Spécifier, gérer et administrer la granularité des objets OLE étant une tâche quasi impossible dans le cadre de notre étude. En ce qui concerne CORBA, surtout dans sa version interopérable, nous pensons que son support devient indispensable lors de la mise au point d'une application client-serveur objet. CORBA est alors vue comme un médiateur offrant la transparence d'accès et l'interopérabilité au niveau binaire (comme dans SOM d'IBM ou des objets créés dans plusieurs langages peuvent coopérer). **Dans MEDEVER, utiliser CORBA, revient à choisir une facette technique et ne doit pas éloigner le concepteur de sa préoccupation principale : mettre en oeuvre un noyau fonctionnel que lui seul maîtrise.**

La construction d'applications via des architectures d'objets répartis, a donné des idées au concepteur de systèmes d'exploitation répartis, qui y ont trouvé toute la puissance et l'extensibilité dont ils avaient besoin.

4.5. Les Architectures d'applications client-serveur s'appuyant sur des Systèmes Répartis

Les systèmes répartis ont pour but d'offrir à un utilisateur, s'il le demande, un accès le plus transparent possible à des ressources réparties sur un réseau informatique [Tanenbaum & al. 85]. En général, on répertorie deux classes de systèmes répartis [Coulouris & al. 94] : ceux basés sur des objets et les autres. Dans ces deux catégories, on distingue les systèmes répartis compatibles Unix et les autres.

Au niveau architectural, les systèmes répartis ont subi une évolution, à notre avis, très proche de celle des applications réparties. Ainsi, certains sont une extension directe de systèmes d'exploitation centralisés existants (Locus [Popek & al. 85]), alors que d'autres sont de nouveaux systèmes (comme Spring). Néanmoins, l'idée émergente dans ce domaine est de spécialiser les systèmes d'exploitation pour fournir uniquement les gestionnaires de services systèmes requis pour une classe d'application donnée. Ainsi, une application ne stockant pas d'information localement n'a pas besoin d'un gestionnaire de fichiers. On se dirige donc vers des systèmes répartis minimalistes auxquels on rajoute dynamiquement ou après recompilation du noyau, des services en fonction des besoins.

Dans la suite de ce chapitre, nous présenterons brièvement les architectures de systèmes répartis qui ne sont pas basées sur des objets. Puis, nous nous intéressons aux systèmes répartis à base d'objets, qui sont actuellement l'objet de recherches intenses. Enfin, nous présentons des exemples de spécialisation de systèmes récents.

4.5.1. Les architectures de systèmes répartis qui ne sont pas basées sur des objets

En général, ces systèmes utilisent des architectures contenant au moins deux couches. La couche la plus basse est constituée du noyau du système d'exploitation réparti. Il fournit des services de base (*Core Services*) tels que ceux de gestion des processus, de gestion de la mémoire et de gestion des communications inter-processus.

La couche la plus haute est dédiée à gestion des processus utilisateurs. Ces systèmes utilisent des processus spécifiques, exécutés lors du démarrage de la machine et chargés de gérer les services répartis. Parmi ces systèmes, on trouve Accent [Rashid 86a], Amoeba [Van Renesse & al. 89], V-System [Cheriton 88] et Charlotte [Finkel & al. 89].

En ce qui concerne la compatibilité Unix, elle peut être fournie de manière radicalement différente. Ainsi, Eden [Almes & al. 85] est une surcouche à Unix, Locus [Popek & al. 85] est une extension d'Unix et Mach [Jones & al. 86 et Rashid 86b] est une émulation du noyau Unix au dessus d'un micro-noyau de systèmes réparti ouvert.

4.5.2. Les systèmes répartis basés sur un modèle objet

Le choix de l'approche objet pour la création de systèmes répartis est récente. Amber [Chase & al. 89] et Clouds [Dasgupta & al. 90] sont construits avec des objets passifs persistants. Les objets sont alors accédés par des processus de manière concurrente. Argus [Liskov 88], par contre, offre des accès concurrents à des objets actifs et aux données persistantes. Argus [Liskov & al. 87] est construit au dessus d'UNIX et dispose d'un langage de programmation appelé CLU [Liskov 85] pour le développement d'applications réparties. Par la suite, les systèmes répartis comme SOS [Shapiro 89], Emerald, Guide 2 et Gothic [Banatre & al. 91], ont permis la migration des objets actifs, pour offrir plus de flexibilité et de tolérance aux pannes.

4.5.3. Les systèmes répartis à facettes (ou spécialisables)

Plutôt que de fournir un système réparti complet et monolithique, certains préfèrent fournir un ensemble de services de bases, extensibles et spécialisables en fonctions des besoins du système et de l'application qui s'appuie dessus. Pour spécialiser un système, on distingue trois approches différentes :

- l'approche micro noyau basée sur une spécialisation macroscopique ;
- l'approche compilation basée sur une sélection des services systèmes nécessaires avant la recompilation du système ;
- l'approche réalisation de bibliothèques systèmes utilisées pour les différentes ressources.

L'approche micro noyau

L'approche micro-noyau est caractérisée par l'implémentation du système d'exploitation au dessus d'un noyau minimal. Ce noyau, appelé micro-noyau au vu des fonctionnalités minimales qu'il offre, fournit une gestion de ressources de bas niveaux et des services de base tels que la gestion de la mémoire et des processus.

Les deux micro-noyaux les plus utilisés actuellement dans la construction de systèmes répartis sont Chorus [Jacquemot 94] et Mach. Il existe de nombreuses réalisations au dessus de chacun de ces micro-noyaux (dont par exemple Cool pour Chorus et Guide 2 pour Mach). Certains systèmes récents, tel que Spring (un système réparti orienté objet mis au point par Sun), utilisent un micro-noyau propriétaire développé pour l'occasion.

Mach [Acceta & al. 86], adopté par l'OSF est largement utilisé du point de vue industriel et fait toujours l'objet de recherches poussées tant au niveau de l'extension de ses fonctionnalités [Lepreau & al. 93, Ford & al. 93], que comme support à la construction de systèmes répartis (comme Sprite [Kupfer 93]). Le micro-noyau de Chorus [Jacquemot 94] bénéficie néanmoins d'une architecture plus modulaire que celui de Mach (il utilise des serveurs et des portes d'accès aux services). Quant à Spring, il fournit une émulation pour la gestion des processus Unix. L'intégration d'applications déjà existantes passe donc par l'encapsulation de toute l'application, ce qui demande un travail non négligeable.

Certains trouvent néanmoins que le micro-noyau est de granularité trop importante et parle désormais de nano-noyau [Tan & al. 95]. Ainsi dans μChoices [Campbell & al. 95], le micro-noyau est découpé en deux parties l'une indépendante de l'architecture maté-

rielle et l'autre dépendante de l'architecture matérielle. Le nano-noyau encapsule donc le matériel et présente une architecture matérielle idéalisée au reste du système, via une interface unique.

L'approche compilation

Les systèmes suivant une approche compilation modifient l'architecture générale d'un système d'exploitation, en transférant certains services systèmes vers une couche intermédiaire proche de l'application. Ils définissent alors un médiateur système disposant d'interfaces d'accès aux applications paramétrables, rapides et fiables (le noyau du système étant toujours non corruptible). Ces systèmes donnent plus d'autonomie à l'application sans l'alourdir de services dont elle n'a pas besoin. Les deux systèmes les plus connus de ce type sont SPIN et Aegis/Exokernel.

SPIN est un système qui dispose d'un ensemble de service de base extensibles et d'un modèle d'extension. Le modèle d'extension s'appuie sur un langage de description des extensions (pour contrôler l'accès au noyau) et implémente des communications protégées utilisant des RPC. Exokernel, lui, s'appuie sur des appels systèmes sécurisés pour isoler les extensions et le noyau et pour laisser sans spécification la manière dont ses extensions sont définies ou appliquées.

Les premières mesures sur ces systèmes montrent que la majorité des appels de primitives systèmes du noyau s'exécutent entre 10 et 100 fois plus rapidement que dans des systèmes Unix monolithiques ou à base de micro-noyau.

L'approche réalisation de bibliothèques systèmes

Choices et Taligent sont deux systèmes de nouvelle génération conçus dés le départ pour être portables, maintenables et réutilisables par parties. Choices s'appuie sur un modèle purement objet (il dispose d'un arbre de hiérarchie de classes prédéfinies), alors que Taligent est décrit par un modèle en couches. Tous deux offrent l'assemblage de composants systèmes ou liés à un métier donné (et appelés *framework*) comme moyen de réalisation d'une application.

Dans Choices, l'utilisation d'une méthode orienté objet lors de la phase de conception des applications oblige le concepteur à modéliser et à valider les services systèmes avant de les implémenter. Le raffinement d'une même classe d'un service système facilite la mise au point et diminue les temps de développement. Ainsi, par exemple, une fois les caractéristiques d'un serveur de fichiers définies, développer un serveur de type NFS de Sun ou NTNFS de Microsoft revient uniquement à spécialiser la classe serveur de fichiers.

L'implémentation de Taligent a été abandonnée par IBM, mais les classes d'objets qui ont été créées sont actuellement réutilisées par IBM dans d'autres projets, ce qui prouve que l'approche modulaire pour la construction d'une application répartie était cohérente.

4.5.4. Synthèse

Les systèmes répartis(4) restent encore en majorité du domaine de la recherche, mais la tendance actuelle est de les construire en appliquant les mêmes règles que pour des applications client-serveur. **La majeure partie des systèmes répartis récents respectent le principe de la séparation des préoccupations.** Ainsi, des facettes techniques étendent les services offerts par un noyau fonctionnel minimal. Ce noyau minimal prend en charge l'accès aux ressources matérielles et peut être un micro-noyau (comme Mach et Chorus), un nano-noyau (comme µChoices), un exo-noyau (comme Aegis/Exokernel) ou une couche générique chargée de gérer les interactions avec le matériel (comme la couche d'abstraction matérielle HAL de Windows NT 3.x et 4.x). Au dessus de ce noyau minimal, on trouve un médiateur système, généralement dans l'espace d'adressage de l'utilisateur. Son rôle est de faire le lien entre les besoins de l'application et les services

(4) Le lecteur intéressé trouvera des études détaillées et comparatives de systèmes répartis majeurs dans [Goscinski 91, Mullender 93 et Coulouris & al. 94].

offerts par le matériel. L'application tire, alors, partie ou non de cette couche médiateur pour l'adapter à ses besoins (comme dans Aegis et Spin) ou proposer une machine virtuelle d'exécution aux applications (comme dans un système Unix traditionnel).

Des travaux en cours sur le système MASIX [Card & al. 94] vont même plus loin, et propose à un même poste client de disposer de multiples personnalités [Mével & al. 96]. Des applications hétérogènes (DOS, Unix et OS/2) s'exécutent alors simultanément en s'appuyant sur un médiateur réparti constitué de serveurs génériques (serveur générique de gestion des processus, d'administration et de configuration, d'accès au réseau et de pilotes de périphériques).

Des passerelles entre architectures de systèmes répartis et architectures basées sur des ORB ont aussi été mises au point. Ainsi, Cool-Orb met en place une passerelle entre l'environnement Chorus-Cool et des ORB compatibles CORBA 2.0. De même, Spring a contribué à la mise au point de l'ORB et des services objets conforme à CORBA de la gamme de produits DOE (*Distributed Object Everywhere*) de Sun Microsystems.

Enfin, pour conclure, nous tenons à indiquer que les concepts de gabarits de conception (ie. l'architecture de définition) et les squelettes d'implémentation (ie. l'architecture d'exécution) ont déjà été mis en oeuvre de manière industrielle et avec succès. Les systèmes conçus récemment, se sont de plus appuyés sur des méthodes de spécification objet et sur des outils de génération de squelettes de code. On peut citer comme exemple Choices et Taligent, qui offrent tous deux des méthodes de construction et de gestion de classes d'objets et des services de génération de code et de documentation automatique.

4.6. Synthèse

La création de logiciels client-serveur s'exécutant concurremment sur des plate-formes hétérogènes nécessite des architectures modulaires basées sur des composants logiciels. A l'heure actuelle, ces composants logiciels sont globalement de trois types : les contenus exécutables, les objets répartis et les agents (cf. Tableau 24).

Tableau 24: Comparaison des trois types de composants logiciels émergeant

	Contenu exécutable	**Objets répartis**	**Agents**
Description	application à la demande	invocation distante	agents intelligents
Mouvement du code	du serveur vers le client (*download*) et du client vers le serveur (*upload*)	pas de mouvement	autonomie
Exécution	sur le client (applet java) ou sur le serveur (servlet Java)	sur le serveur	sur les places de l'environnement
Invoqué par	le client (ou le serveur)	le client ou un objet mandataire	le client ou un agent intermédiaire
Stockage	sur le serveur (pages HTML)	sur le serveur	sur le client ou sur le serveur
Exemples	Java, ActiveX	ORB, DCOM	Odyssey, Voyager,

Les contenus exécutables sont des petits programmes mobiles comme les programmes Java de Sun (*applets*) ou les composants ActiveX de Microsoft. Les objets répartis utilisent un ORB pour effectuer des invocations distantes de services. Ainsi, JOE de Sun (*Java Object Everywhere*) dispose d'un ORB pour faire communiquer des applets. Enfin les agents, se différencie principalement des deux autres types de composant dans leur

manière de se déplacer : ils sont autonomes. Notons que les modifications futures des plate-formes de Sun et de Microsoft permettront de supporter les agents.

Ces composants sont construits en utilisant des langages ou des plate-formes de développement orientées objet. L'interopérabilité entre ces composants se fait soit par une intégration fine avec le système d'exploitation réparti (comme Aegis/ExoKernel), soit par l'utilisation et l'adaptation d'interfaces d'accès aux services (en utilisant CORBA 2.0). L'interopérabilité est obtenue non plus par appel de services distants (en utilisant des RPC) ou par des bibliothèques de communications portables (cf. Annexe E), mais par migration du code. Ce code est alors soit compatible au niveau binaire (technologie OLE/COM), soit interprété sur le client en utilisant une machine virtuelle (programmes écrits en Java ou en Telescript), soit compilé à la volée et exécuté chez le client (comme pour le langage Oasis).

5. Conclusion

Les bénéfices attendus lors de la mise en oeuvre de systèmes client-serveur ouverts ont, dans une large mesure, été obtenus. D'un point de vue économique, le coût des matériels et des systèmes d'exploitation a chuté considérablement. A l'inverse, les coûts liés au choix, à l'achat, à la mise en place et à la maintenance de ces systèmes se sont considérablement accrus. Les logiciels sont désormais constitués d'une architecture à base de composants réutilisables mobiles et/ou fixes qui coopèrent à travers un espace mémoire, un réseau local, voire un réseau mondial. En ce qui concerne les applications hétérogènes et monolithiques, on peut toujours les faire coopérer en utilisant des bibliothèques de communication portable telles que PVM [Geist & al. 93] ou MPI [MPIF 94] (dont une étude détaillée est proposée en Annexe E).

Les inconvénients de cette frénésie du domaine du génie logiciel sont nombreux. D'abord, les annonces de technologies qui n'existent pas (*vaporware*), qui sont souvent abandonnées très rapidement submergent les ingénieurs. Ensuite, les versions des logiciels livrées ne sont pas toujours très stables et exemptes de bugs. Enfin, si les systèmes d'exploitation et les environnement de constructions d'applications sont de plus en plus ouverts, il n'en est pas de même des technologies médiateurs qui se sont en quelque sorte re-propriétarisées. L'exemple de CORBA est de ce point de vue saisissant. Il aura fallu attendre les spécifications de la version 2, pour que des ORB de constructeurs différents puissent coopérer (et encore, du travail reste à faire sur ce point). Néanmoins CORBA semble devenir l'infrastructure standard de communication entre objets répartis. La preuve en est que des travaux actuels tentent d'intégrer des services MOM dans CORBA (pour gérer les communications asynchrones et faciliter l'intégration avec des application non objets) et des moniteurs transactionnels (via un nouveau service CORBA appelé *Object Transaction Services*).

La garantie d'indépendance tant recherchée par le client n'est donc pas totale. Elle a évolué d'une dépendance monoproduit et monofournisseur (logiciel IBM sur site central IBM), vers un partenariat multi-produit monofournisseur (logiciel Microsoft Visual Basic accédant à une base de données Microsoft SQL server) ou multi-fournisseur (logiciel Serveur Internet Netscape accédant à des programmes Java sur une plate-forme logicielle Sun Microsystems).

Conclusion dans le contexte de notre étude

Face à toutes ces possibilités, le développeur ne sait plus quelle technologie choisir et surtout n'a aucun guide pour trouver ou connaître la granularité et le placement des composants logiciels qu'il crée ou utilise. Ainsi, entre le monde applicatif (axe langage) et le monde des systèmes d'exploitation (axe système), il existe un fossé qu'il convient de combler. Notre méthode MEDEVER a été conçue pour combler ce fossé et passe par la description de l'architecture logicielle d'une application.

De cette étude sur les architectures logicielles d'application client-serveur, nous déduisons que toute application peut être vue comme un ensemble de composants logiciels et comme un ensemble d'interactions entre ses composants. Cette vision architecturale est à deux niveaux. On distingue alors l'architecture de définition de l'architecture d'exécution. Cette distinction est rendue quasi naturelle avec des langages de description d'architecture, par l'utilisation conjointe de gabarits de conception et de squelettes d'implémentation.

L'autonomie et la mobilité des composants logiciels, ainsi que les nouveaux schémas d'interaction qu'il faudra mettre en oeuvre vont entraîner une utilisation accrue et un élargissement du rôle des médiateurs. Ces médiateurs étant la plupart du temps des mises en oeuvre de facettes techniques, s'appuyant sur un noyau minimal (implémenté sous la forme de micro-noyau, de machine virtuelle ou de plate-forme d'intégration). Le seul élément de choix prépondérant étant alors la granularité des composants logiciels gérés.

Ainsi, que l'on adopte un point de vue langage ou un point de vue système, seuls comptent dorénavant la granularité et le schéma d'interaction des composants logiciels (et les informations qu'ils manipulent). Il est vrai que la construction d'applications basées sur des composants à gros grain et à durée de vie longue est plutôt réalisée en adoptant un point de vue système (plate-forme d'intégration ou systèmes ouverts et répartis) et en utilisant des technologies médiateurs pour gérer les services de répartition et de communication. Par contre, dans le cas d'applications manipulant des composants logiciels de grains fins (des dizaines d'objets répartis par exemple), on adopte plus facilement un point de vue langage (car les objets créés sont liés au langage) et des bus logiciels conforme à CORBA 2.0 comme médiateurs.

Notre approche a été au sein de notre méthodologie MEDEVER de prendre en compte les nouveaux besoins en terme de construction d'application. Nous avons donc créé un langage spécifique dont le but est :

- de décrire statiquement des architectures logicielles d'applications basées sur des composants à grain variable et comportant des schémas d'interaction dépassant le simple arrangement client-serveur ;
- de décrire les évolutions dynamiques de la granularité et des schémas d'interaction des composants logiciels lors de leur exécution et de leur déplacement.

Ce langage supporte les deux axes d'intérêt :

- système : il donne des informations au système et au médiateur sur le contenu de l'application pour qu'il alloue au mieux les ressources requises et que les performances de l'application soient maximales.
- langage : il remonte des informations du système vers les applications et le développeur, en transmettant des informations de contrôle et/ou des erreurs de conception ou d'exécution conduisant à des pertes de performances ou à des pannes logicielles.

Ce langage pivot de notre méthode MEDEVER, appelé VODEL, est détaillé dans le chapitre suivant.

6. Références bibliographiques

[Accetta & al. 86] M. Acceta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian & M. Young, «Mach: A New Kernel Foundation For Unix Development», In Proc. of the Usenix 1986 Summer Conference, June 1986.

[AFNOR 88] AFNOR, Interconnexion de Services Ouverts (OSI) : architecture, Systèmes et protocoles», 1988, 421 pages.

[Agha 86a] G. Agha, «Actors: A model of concurrent computation in distributed systems», MIT Press, Cambridge, 1986.

[Agha 86b] G. Agha, «An overview of Actor languages», Sigplan Notices, Vol. 21 (10), October 1986, pp. 58-67.

[Agha & al. 87] G. Agha & C. Hewitt, «Concurrent Programming Using Actors», A. Yonezawa Eds., Object-Oriented Concurrent Programming, The MIT Press, 1987.

[Agha 90] G. Agha, «Concurrent Object-Oriented Programming», Communications of the ACM, Vol 33 (9), pp. 125-141, September 1990.

[Agha & al. 92a] G. Agha, S. Frolund, R. Panwar & D. Sturman, «Linguistic Framework for Dynamic Composition of Dependability Protocols», In Proc. of the Conference on Dependable Computing for Critical Applications, IFIP, Sicily, 1992.

[Agha & al. 92b] G. Agha & C. Houk, «HAL: A High Level Actor Languages and its Distributed Implementation», 21st International Conference on Parallel Processing ICCP'92, VOL II, 1992.

[Alexander 79] C. Alexander, «The Timeless Way of Building», Oxford University Press Eds., 1979.

[Allen & al. 94] R. Allen and D. Garlan, «Formalizing Architectural Connexion», Proc. of the 16th International Conference on Software Engineering, pp. 71-80, Sorrento, Italy, May 1994.

[Almes & al. 85] G.T. Almes, A.P. Black, E.D. Lazowska & J.D. Noe, «The Eden System: A Technical Review», IEEE Transactions on Software Engineering, Vol. SE-11 (1), pp. 43-59.

[Amza & al. 96] C. Amza, A.L. Cox, S. Dwarkadas, P. Keheler, R. Rajamony, W. Yu & W. Zwaenepoel, «TreadMarks: Shared Memory Computing on Networks of Workstations», IEEE Computer, Vol. 29 (2), pp. 18-28, 1996.

[Arcangeli & al. 94] J.P. Arcangeli, A. Marcoux, C. Maurel & P. Sallé, «La programmation concurrente par acteurs : PLASMAII», Calculateurs Parallèles, «les langages à objets», No 22, Juin 1994, pp 99-121.

[Arrouye 96] Y. Arrouye, «Environnements de visualisation pour l'évaluation des performances des systèmes parallèles : étude, conception et réalisation», Thèse de doctorat de l'INPG, 1996.

[Banatre & al. 91] Banatre & al., «Les systèmes distribués, Concepts et Expériences du Projet Gothic», Interedition Eds., 1991.

[Bauer & al. 93] M.A. Bauer, G. Strom, N. Coburn, D.L. Erickson, P.J. Finnigan, J.W. Hong, P. A. Larson & J. Slonim, «Issues in Distributed Architectures: A Comparaison of Two Paradigms», In Proc. of the International Conference on Open Distributed Processing, Berlin, Germany, September 1993, pp. 78-86.

[Beck & al. 94] K. Beck & R. Johnson, «Patterns Generate Architecture», Proc. of the European Conference on Object-Oriented Programming, Springer Verlag Eds., Bologna, Italy, pp. 139-149, July 1994.

[Berners-Lee & al. 94] T. Berners-Lee, R. Cailliau, A. Luotonen, H.-K. Nielsen & A. Secret, «The World Wide Web», Communications of the ACM, Vol. 37 (8), pp. 76-82, August 94.

[Bernstein 90] P.A. Bernstein, «Transaction Processing Monitors», Communications of the ACM, Vol 33 (11), November 1990, pp. 75-86.

[Bernstein 96] P.A. Bernstein, «Middleware: a Model for Distributed System Services», Communications of the ACM, Vol 39 (2), February 1996, pp. 86-98.

[Bershad & al. 88a] B.N. Bershad & al., «PRESTO: A System for Object-Oriented Parallel Programming», Software Practice and Experience, vol. 18 (8), Aug. 1988, pp. 713-732.

[Bershad & al. 88b] B.N. Bershad & al., «An Open Environment for Building Parallel Programming Language», In Proc. of the ACM SIGPLAN, July 1988.

[Birrell & al. 84] A.D. Birrell & B.J. Nelson, «Implementing Remote Procedure Call», ACM Transactions on Computer Systems, Vol. 2 (1), 1984, pp. 39-59.

[Boasson & al. 95] M. Boasson & H. Signaalapparaten, «The Artistry of Software Architecture», IEEE Software, pp. 13-16, November 1995.

[Bonnaire & al. 96] X. Bonnaire & D. Prun, «Flexible Distributed Replay in a Message Server Network Environment», In Proc. of the ISCA 9th International Conference on Parallel And Distributed Computing Systems, pp 88-93, Dijon, FRANCE, September 25-27, 1996

[Briot 89] J.P. Briot, «Actalk: a testbed for classifying and designing Actor languages in the Smalltalk-80 Environment», In Proc. of the European Conference on Object-Oriented Programming, Cambridge University Press, 1989, pp. 109-129.

[Briot & al. 96] J.-P. Briot & R. Guerraoui, «Objets pour la programmation parallèle et répartie : intérêts, évolutions et tendances», Techniques et Sciences Informatiques, Vol 15 (6), pp. 765-800, 1996.

[Brown 94] D.K. Brown, «Practical Issues Involved in Architectural Evolution from IN to TINA», 3rd International Conference on Intelligence in Networks (ICIN'94), October 1994.

[Budinsky & al. 96] F.J. Budinski, M.A. Finnie, J.M. Vlissides & P.S. Yu, «Automatic Code Generation from Design Patterns», IBM systems Journal, Vol. 35 (2), pp. 151-171, 1996.

[Buhr & al. 92a] P.A. Buhr & al., «μ C++: Concurrency in the Object-Oriented Language C++», Software Practice and Experience, Vol. 22 (2), Feb. 1992, pp.137.

[Buhr & al. 92b] P.A. Buhr & G. Ditchfield, «Adding concurrency to a programming language», In Proc. of the USENIX C++ Technical Conference, USENIX Association, berkeley, 1992, pp. 207-223.

[Burns & al. 90] A. Burns & A. Wellings, «Real-time Systems and their Programming Language», Addison Wesley Eds., 1990.

[Buschmann & al. 96] T. Buschmann, R. Meunier, H. Rohnert, P. Sommerland & M. Stal, «Pattern Oriented Software Architecture: A system of Pattern», Wiley and Sons Eds., 1996.

[Cagan 90] M. Cagan, «The HP-Softbench Environment: An Architecture for a New Generation of Software Tools», Hewlett Packard Journal, Vol. 41 (3), juin 1990.

[Campbell & al. 95] R.H. Campbell & S-M. Tan, «μChoices: An Object-Oriented Multimedia Operating System», Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems, Orcas Island, Washington, May 1995.

[Card & al. 94] R. Card, H. Le Van Gong & P. G. Raverdy, «Design of the Masix Distributed Operating System on top of the Mach Microkernel», In Proc. of the IFIP Int. Conf. on Applications in parallel and distributed Computing, November 1994.

[Chase & al. 89] J. Chase, F. Amador, E. Lazowska, H. Levy & R. Littlefield, «The Amber system: Parallel programming on a network of multiprocessors», In Proc. of the 12th ACM Symposium on Operating System Principles, New York, 1989, in ACM Operating System Review, Vol. 23 (5), pp. 147-158.

[Cheriton 88] D.R. Cheriton, «The V Distributed System», Communciations of the ACM, Vol. 31(3), 1988, pp. 314-333.

[Clark 95] D. Clark, «Buy it by the slice: The Microsoft vision of distributed computing», IEEE Parallel & Distributed Technology, Winter 95, pp. 8-11.

[Coad 92] P. Coad, «Object-Oriented Pattern», Communications of the ACM Vol. 35 (9), pp 152-159, 1992.

[Coad & al. 95] P. Coad, D. North & M. Mayfield, «Objects Models: Strategies, Patterns and Applications», Englewood Cliffs, Prentice Hall Eds., 1995.

[Coplien 92] J.O. Coplien, «Advanced C++ Programming Styles and Idioms», Addison-Wesley Eds., 1992

[Coplien & al. 95] J.O. Coplien & D.C. Schmidt, «Pattern Languages Of Program Design», J.O. Coplien & D.C. Schmidt Eds., Addison Wesley, 1995, 562 pages.

[Coulouris & al. 94] G.F. Coulouris & J. Dollimore, «Distributed Systems : Concept and Design», 2nd Edition, Addison Wesley, 1994.

[Coutaz & al. 96] J. Coutaz, G. Calvary, L. Nigay & D. Salber, «Styles et motifs en architecture logicielle : Illustration avec la famille PAC», Document de travail GT-Scoop.

[Dasgupta & al. 90] Dasgupta & al., «The Design and Implementation of the Clouds Distributed Operating System», Computing System, Vol. 3 (1), 1990, pp. 11-45.

[Decouchant 86] D. Decouchant, «Design of a distributed object manager for the Smalltalk-80 system», In Proc. of OOPSLA'86, ACM / SIGPLAN, 1986, pp 444-452.

[Demeure & al. 94] I. Demeure & J. Farhat, «Système de processus légers : concepts et exemples», Techniques et Science Informatiques, Vol. 13 (6), 1994, pp 765-795.

[Deutsch 89] L.P. Deutsch, «Design Reuse and Frameworks in the Smalltalk-80 Collection Classes», T.J. Biggerstaff and A.J. Perlis Eds., Software Reusability, Volume II: Applications and Experiences, pp. 57-71, Addison Wesley, 1989.

[Ferber 83] J. Ferber, «Mering IV: Un langage d'acteurs pour la représentation et la manipulation des connaissances», Thèse de l'université Pierre et Marie Curie, Paris, 1983.

[Finkel & al. 89] R. Finkel, M.L. Scott, W.K. Kalsow, Y. Artsy & H.-Y. Chang, «Experience with charlotte: simplicity and function in a distributed operating system», IEEE Transactions on software engineering, Vol. 15 (6), 1989, pp. 676-685.

[Ford & al. 93] B. Ford & J. Lepreau, «Evolving Mach 3.0 to Use Migrating Threads», Technical Report, UUCS-93-022, University of UTAH, November 1993.

[Foughali 94] K. Foughali, «Conception et réalisation d'une plate-forme d'intégration distribuée. Application à l'atelier de génie logiciel AMI», Thèse de doctorat de l'Université Pierre et Marie Curie, rapport IBP Th94-09, 1994, 199 pages.

[Frankel 91] R. Frankel, «The Tooltalk Service», Sunsoft White Paper, June 1991.

[Gabriel 94] R. Gabriel, «Pattern Language», Journal of Object-Oriented Programming, Vol. 6 (2), p 14, January 1994.

[Gamma & al. 93] E. Gamma, R. Helm, R. Johnson & J. Vlissides, «Design Patterns - Abstraction and Reuse of Object-Oriented Designs», in Proc. of the European Conference on Object-Oriented Programming, O. Nierstrasz Eds., Springer Verlag, 1993.

[Gamma & al. 94] E. Gamma, R. Helm, R. Johnson & J. Vlissides, «Design Patterns : Elements of Reusable Object-Oriented Software», Addison Wesley, 1994, 395 pages.

[Gardarin 96] G. & O. Gardarin, «Le Client-Serveur», Eyrolles Eds., 1996.

[Garlan & al. 93] D. Garlan & M. Shaw, «An Introduction to Software Architecture», Advances in Software Engineering and Knowledge Engineering», Vol. 1, V. Ambriola and G. Totora Eds, World Scientific Publishing Company, New Jersey, 1993, pp. 1-39.

[Garlan & al. 94] D. Garlan, R. Allen & J. Ockerbloom, «Exploiting Style in Architectural Design Environment», Proc. of the ACM SIGSOFT'94, New Orleans, LA, December 1994.

[Garlan & al. 95] D. Garlan, R. Allen & J. Ockerbloom, «Architectural Mismatch or why it's hard to build systems out of existing parts», Proc. of the 17th International Conference on Software Engineering, Seattle, WA, April 1995.

[Gaudiot & al. 92] J.L. Gaudiot & D.K. Yoon, «A comparative study of Parallel Programming Languages: The Salishan Problem», J.T. Feo Eds., Elsevier Science, 1992, pp 217-262.

[Gehani & al. 88] N.H. Gehani & W.D. Roome, «Concurrent C++: Concurrent programming with classes», Software Practice and Experience, Vol. 16 (12), Dec. 1988.

[Geib & al. 91] J.M. Geib & J.F. Colin, «Eiffel classes for concurrent programming», In Proc. of TOOLS4 Conference, Prentice Hall, 1991, pp. 23-34.

[Geist & al. 93] A. Geist, J. Dongarra & R. Manchek, «PVM3 User's Guide and Reference Manual», Oak Ridge National Laboratory and University of Tenessee, May 1993.

[Ghernaouti 94] Ghernaouti-Hélie, «Client / serveur : les outils du traitement coopératif», Masson Eds., Chapitre 7, pp 95-111.

[Goldberg & al. 83] A.J. Goldberg & A.D. Robson, «SMALLTALK-80: The language and its implementation», Addison Wesley, 1983.

[Goscinski 91] A. Goscinski, «Distributed Operating Systems: the logical System», Addison Wesley Eds., 1991, 913 pages.

[Guerraoui 95] R. Guerraoui, «Les Langages Concurrents à Objets», Techniques et Sciences Informatiques, Vol 14 (8), pp. 945-971, 1995.

[Helm & al. 90] R. Helm, I. M. Holland & D. Gangopadhyay, «Contracts: Specifying Behavioral Compositions in Object-Oriented systems», Proc. of the OOPSLA'90, pp 169-180, October 1990.

[Helm 95] R. Helm, «Design Patterns Keynote Address», Proceedings of OOPSLA'95, 1995.

[Hewitt & al. 73] C. Hewitt, P. Bishop & R. Steiger, «A universal modular actor formalism for artificial intelligence», Proceedings of the IJCAI'73, Stanford University, 1973.

[Hewitt 77] C. Hewitt, «Viewing control structures as patterns of passing messages», Journal of artificial Intelligence, No 8, 1977.

[Holland 92] I.M. Holland, «Specifying Reusable Components Using Contracts», Proc. of the 6th European Conference on Object-Oriented Programming (ECOOP'92), pp. 287-308, 1992 - aussi dans le LNCS No 615.

[Hürsch & al. 95] Walter Hürsch & Cristina Videira Lopes, «Separation of Concerns», Northeastern University Technical Report, NU-CCS-95-03, Boston, February 1995.

[Jacquemot 94] C. Jacquemot, «Chorus / COOL V2 Reference Manual», Technical Report, CS / TR-94-16, Chorus System, 1994.

[Johnson & al. 88] R. Johnson & B. Foote, «Designing Reusable Classes», Journal of Object-Oriented Programming, Vol. 1 (2), pp. 22-35, June-July 1988.

[Johnson 92] R. Johnson, «Documenting Frameworks Using Patterns», in the Proc. of OOPSLA'92, SIGPLAN Notices, Vol. 27 (10), pp. 63-76, October 1992.

[Jones & al. 86] M.B. Jones & R.F Rashid, «Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems», In Proc. of the OOPSA'86, Portland (Oregon), 1986, pp. 66-77.

[Kafura & al. 89] D.G. Kafura & K.H. Lee, «Inheritance in Actor Based Concurrent Object-oriented Languages», Proceedings of ECOOP'89, pp. 131-145, Cambridge University Press, 1989.

[Kafura & al. 90] D.G. Kafura & K.H. Lee, «ACT++ : Building a Concurrent C++ with actors», Journal of Object-Oriented Programming, Vol. 3 (1), 1990.

[Karaorman & al. 93] M. Karaorman & J. Bruno, «Introducing concurrency to a sequential language», Communications of the ACM, Vol. 36 (9), September 1993, pp 103-116.

[Kiczales & al. 97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Cristina Videira Lopes, Chris Maeda, Jean-Marc Loingtier & John Irwin, «Aspect-Oriented Programming», Proceedings of European Conference on Object-Oriented Programming (ECOOP'97), Springer-Verlag Lecture Notes in Computer Science, Finland, June 1997.

[Kobayashi & al. 94] N. Kobayashi & A. Yonezawa, «Asynchronous communication model based on linear logic», Formal Aspects of Computing, Vol. 3, pp. 1-37, 1994.

[Konstantas 96] D. Konstantas, «Chapter 3 : Interoperation of Object-Oriented Applications», Object-Oriented Software Composition, O. Nierstrasz and T. Tsichritzis Eds., Prentice Hall, 1996.

[Kruchten 95] P.B. Kruchten, «The 4+1 View Model of Architecture», IEEE Software, pp. 42-50, November 1995.

[Kupfer 93] M. D. Kupfer, «Sprite On Mach», In Proc. of the Third Usenix Mach Symposium, April 1993.

[Lepreau & al. 93] J. Lepreau, M. Hibler, B. Ford & J. law, «In-kernels servers on Mach 3.0: Implementation and Performance», In Proc. of the 3rd Usenix Mach Symposium, April 1993, pp. 39-55.

[Lieberherr & al. 95] K.J. Lieberherr, I. Silva-Lepe & C. Xiao, «Adaptative Object-Oriented Programming Using Graph-based Customization», Communications of the ACM, Vol. 37(5), pp. 94-101, May 1995.

[Lieberman 87] H. Lieberman, «Concurent Object-Oriented Programming in Act1», Object-Oriented Concurrent Programming, A. Yonezawa and M. Tokore Eds., The MIT Press, 1987, pp. 9-36.

[Liskov 85] B.H. Liskov, «The Argus Language and System, Distributed Systems : methods and tools for specification», Lecture Notes in Computer Science, Springer Verlag, Vol. 190, 1985, pp. 343-430.

[Liskov & al. 87] B.H. Liskov, D. Curtis, P. Johnson & R. Scheifler, «Implementation of Argus», In Proc. of the 12th ACM Symposium on Operating System Principles, 1987, pp. 111-122.

[Liskov 88] B. Liskov, «Distributed Programming in Argus», Communications of the ACM, 31(3), March 1988, pp. 300-312.

[Lonchamps & al. 92] J. Lonchamps, C. Godard & J.-C. Derniame, «Les environnements Intégrés de Production de Logiciels», Techniques et Sciences Informatiques, Vol 11 (1), 1992.

[Long & al. 93] F. Long & E. Morris, «An Overview of PCTE: A Basis for a Portable Common Tool Environment», Carnegie Melon University Technical Report, CMU/SEI-93-TR-1 et ESC-TR-93-175, March 1993.

[Magee & al. 89] J. Magee, N. Dulay & J. Kramer, «Constructing Distributed Systems in Conic», IEEE Transactions on Software Engineering, SE-15 (6), 1989, pp. 663-675.

[Mahmoud 97] Q. Mahmoud, «Implementing Design Patterns in Java», Java Developer's Journal, Vol. 2 (5), pp. 8-14, 1997.

[Matsuoka & al. 93] S. Matsuoka & Y. Yonezawa, «Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages», Research Directions in Concurrent Object-Oriented Programming, G. Agha, P. Wegner, A. Yonezawa Eds., The MIT Press, 1993.

[Mercurio & al. 90] V.J. Mercurio, B.F. Meyers, A.M. Nisbet & G. Radin, «AD/Cycle Strategy and Architecture», IBM Systems Journal, No 29, 1990, pp. 170-188.

[Meseguer 93] J. Meseguer, «Solving the Inheritance Anomaly in Concurrent Objet-oriented Programming», Proceedings of the 7th Intrenational Conference ECOOP'93, O.S. Nierstrasz Ed., Lecture Notes in Computer Science, No 707, pp. 220-246, 1993.

[Mével & al. 96] F. Mével, & J. Simon, «Distributed Communication Services in the Masix System», Int. Phoenix Conf. on Computers and Communications (IPCCC'96), Phoenix (USA), March 27-29, 1996, Site : <http://www-masi.ibp.fr/masix>.

[Meyer 88] B. Meyer, «Object-Oriented Software Construction», Prentice Hall International, 1988.

[Meyer 92a] B. Meyer, «Eiffel: The Language», Prentice Hall, Englewoods Cliffs, 1992.

[Meyer 92b] B. Meyer, «Applying Design by contract», IEEE Computer, Vol 25(10), pp. 40-51, October 1992.

[Meyer 93] B. Meyer, «Systematic Object-Oriented Programming», Communications of the ACM, Vol. 36 (9), September 1993, pp. 56-80.

[Milner 89] R. Milner, «Communication and Concurrency», Prentice Hall Ed., 1989.

[Moreau 95] R. Moreau, «L'approche objet : concepts et techniques», Masson Ed., collection MIPS, 1995.

[Mowbray & al. 97] T. J. Mowbray & R. C. Malveau, «CORBA Design Pattern», Wiley and Sons Eds., 334 pages, 1997, ISBN 0-471-15882-8.

[MPIF 94] MPI Forum, «MPI: A Message Passing Interface Standard», Technical Report CS-94-230, University of Tennessee, April 1994. Est paru dans International Journal of Supercomputer Applications, Vol 8 (3/4), 1994.

[Mullender 93] S. Mullender, «Distributed Systems», 2nd edition, Addison-Wesley, 1993, 595 pages.

[Müllhaüser & al. 93] M. Müllhaüser, W. Gerteis & L. Heuser, «DOCASE: a methodic approach to distributed programming», Communications of the ACM, Vol. 36 (9), Sept. 1993, pp. 127-138.

[Nierstrasz 87] O.M. Nierstrasz, «Active Objects in Hybrid», ACM International Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'87, October 1987.

[Nierstrasz & al. 96] O. Nierstrasz & L. Dami, «Chapter 1 : Component Oriented Software Technology», Object-Oriented Software Composition, O. Nierstrasz and T. Tsichritzis Eds., Prentice Hall, 1996.

[OMG 95] Object Management Group, «CORBA 2.0 / Interoperability, Universal Networked Objects», OMG document No 95.3.10, Framingham, USA, Mars 1995.

[Ozanne 96] C. Ozanne, «Conception d'applications client-serveur : modèles d'architecture fonctionnelle et opérationnelle», Thèse de doctorat de l'Université Pierre et Marie Curie, Institut Blaise Pascal, Mars 1996.

[Papathomas 96] M. Papathomas, «Chapter 2 : Concurrency in Object-Oriented Programming Languages», Object-Oriented Software Composition, O. Nierstrasz and T. Tsichritzis Eds., Prentice Hall, 1996.

[Perry & al. 92] E. D. Perry & A.L. Wolf, «Foundations for the study of software architecture», ACM SIGSOFT Software Engineering Notes, Vol. 17 (4), pp. 40-52, October 1992.

[Popek & al. 85] G. Popek & B.J. Walker, «The Locus Distributed System Architecture, MIT Press, 1985.

[Pree 95] W. Pree, «Metapatterns», Design Pattern for Object-Oriented Software Development, Addison-Wesley Eds, pp. 105-171, 1995.

[Purtilo & al. 91] J.M. Purtilo & J.A. Atlee, «Module Reuse by Interface Adaptation», Software Practice and Experience, Vol. 21 (6), June 1991.

[Purtilo 94] J.M. Purtilo, «The Polylith Software Bus», ACM Transactions on Programming Languages and Systems, Vol. 16 (1), pp. 151-174, January 1994.

[Quasar 94] Quasar Knowledge System White Paper, «Smalltalk Agent Technical Overview and Design Goals», 1994.

[Rashid 86a] R.F. Rashid, «Experiences With The Accent Network Operating System», Lecture Notes in Computer Science, No 248, pp. 259-269.

[Rashid 86b] R.F. Rashid, «From RIG to Accent to Mach: the Evolution of a Network Operating System», In Proc. of the ACM/IEEE Computer Society Fall Joint Conference, ACM, November 1986.

[Rumbaugh & al. 95] J. Rumbaugh, M. Blaha, F. Eddy, W. Premerlani & W. Lorenson, «OMT - Modélisation et conception orientée objets», Masson Eds., 1995.

[Rymer 96] J.R. Rymer, «The Muddle in the Middle», Byte, Vol 21 (4), April 96, pp. 67-70, 1996.

[Schefstrom & al. 93] D. Schefstrom & G.Van Der Broek, «Tool Integration: Environments and Frameworks», Schefstrom and Van Der Broek Eds., Wiley Series in Software Based Systems, 1993.

[Shapiro 86] M. Shapiro, «Structure and encapsulation in distributed systems: the Proxy principles», Proc. of 6th Conf. on Distributed Computing Systems, Cambridge, May 1986, pp 198-204.

[Shapiro 89] M. Shapiro, «Prototyping a Distributed Object-oriented Operating System on Unix», Workshop on experiences with building distributed and multiprocessor systems (WEBDMS), Ft. Lauderdale (USA), October 1989.

[Shaw & al. 95] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young & G. Zelesnik, «Abstraction for Software Architecture and Tools to Support Them», IEEE Transactions on Software Engineering, Vol. 21 (4), pp. 314-335, April 1995.

[Shaw & al. 96] M. Shaw & D. Garlan, «Software Architecture: Perspective on an Emerging Discipline», Prentice Hall Eds., 1996.

[Smith & al. 95] R.B. Smith & D. ungar, «Programming as an experience: The inspiration of SELF», ECCOP'95 Conference Proceedings, Aarhus, Denmark, August 1995, <http://www.sun.com.>

[SSI 87] Swedish Standard Institute, Data Processing-Programming Languages-SIMULA, Svensk Standard SS 636114, May 20, 1987.

[Steiner & al. 88] J.G. Steiner, C. Neuman & J.I. Schiller, «Kerberos: An authentification Service for Open Network Systems», In Proc. of the Usenix Winter Conference, 1988, pp. 191-201.

[Stephani 95] J.B. Stephani, «Open Distributed Processing - An Architectural Basis for Information Networks», Computer Communications, Vol. 18 (11), pp. 849-862.

[Strom 86] R.E. Strom, «A comparaison of the Object-Oriented and Process Paradigm», SIGPLAN Notices, Vol. 21 (10), October 1986.

[Stroustrup 86] B. Stroustrup, «The C++ Programming Language», Addison Wesley, 1986.

[Tan & al. 95] S-M. Tan, D. Raila & R;H. Campbell, «An Object-Oriented Nano-Kernel for Operating System Hardware Support», Proceedings of the 4th international IEEE Workshop on Object-Orientation in Operating Systems, Lund, Sweden, August 1995.

[Tanenbaum & al. 85] A.S. Tanenbaum & V. Renesse, «Distributed Operating Systems», Computing Surveys, Vol. 17 (4), December 1985.

[Theriaut 82] D. Theriaut, «A primer for the Act-1 Language», AI Memo 672, MIT Artificial Intelligence Laboratory, April 1982.

[Theriaut 83] D. Theriaut, «Issues in the design and Implementations of Act2», Technical Report 728, MIT Artificial Intelligence Laboratory, June 1983.

[TSE 95] IEEE transactions on software engineering, «Special issue on software architecture», Vol. 21 (4), April 1995.

[Tsichritzis 89] D. Tsichritzis, «Object-Oriented Development for Open Systems», Proc. of the Information Processing (IFIP'89), North-Holland, Sans Francisco, pp. 1033-1040, 1989.

[Ungar & al. 87] D. Ungar & R.B. Smith, «SELF: the power of simplicity», Proceedings of the OOPSLA'87, Orlando, pp. 227-241, October 1987.

[Van Renesse & al. 89] R. Van Renesse, H. Van Staveren & A.S. Tanenbaum, «The Performance of the Amoeba Distributed Operating System», Software Practice And Experience, Vol. 19 (3), 1989, pp. 223-234.

[Vlissides & al. 96] J.M. Vlissides, J.O. Coplien & N.L. Kerth, «Pattern Languages of Program Design 2», Addison Wesley, 600 pages, 1996.

[Wasserman 90] A. Wasserman, «Tool Integration in Software Engineering Environments», Software Engineering Environments, Springer Verlag, LNCS, No 467, pp. 138-150, 1990.

[Wayner 94] P. Wayner, «Object on the March», Byte, pp. 139-150, January 1994.

[Wegner 90] P. Wegner, «Concepts and Paradigms of Object Oriented Programming», ACM OOP Messenger, Vol. 1 (1), August 1990.

[Wild 96] F. Wild, «Instantiating Code Patterns: Patterns applied to software development», Dr Dobb's Journal, No 248, pp. 72-76, Juin 1996.

[Wileden & al. 91] J.C. Wileden, A.L. Wolf, W.R. Rosenblatt & P.L. Tarr, «Specification Level Interoperability», Communications of the ACM, Vol. 34 (5), May 1991.

[WWW-Pattern 97] Site : <http://st-www.cs.uiuc.edu/users/patterns/patterns.html>.

[Wyatt & al. 92] B.B. Wyatt, K. Kavi & S. Hufnagel, «Parallelism in Object-Oriented Languages: A Survey», IEEE Software, pp. 56-66, November 1992.

[Yellin & al. 94] D.M. Yellin & R.E. Strom, «Interfaces, protocols and semi-automatic construction of software adaptors», Proceedings of the OOPSLA'94, October 1994.