Chapitre XX

Architectures de Composants Répartis

Frédéric Peschanski et Jean-Pierre Briot
Laboratoire d'Informatique de Paris 6

Résumé

Dans ce chapitre, nous discutons du problème de la répartition dans les architectures de composants logiciels. Pour cela, nous effectuons tout d'abord un tour d'horizon des spécificités des systèmes répartis à large-échelle, notamment les aspects d'hétérogénéité, de concurrence, de fiabilité et de sécurité. Les plateformes intergicielles « classiques » offrent une vision répartie de la conception et de la programmation orientée objet. Nous discutons de certaines limites importantes des modèles d'objets répartis ; limites qui conduisent naturellement aux modèles de composants. Avant de décrire les principes de base des modèles de composant répartis, nous discutons des architectures applicatives qu'ils ciblent en priorité. Nous passons du classique et quelque peu démodé client-serveur aux architectures émergentes comme les grilles de calcul ou le pair à pair, sans oublier les très stratégiques architectures 3-tiers. Ces dernières représentent de fait le domaine applicatif privilégié - et même quasiment exclusif - des plateformes industrielles disponibles aujourd'hui et notamment les Enterprise Java Beans et les Corba Component Model. Le principe d'inversion du contrôle forme, selon nous, le fil conducteur des différentes architectures proposées dans les modèles de composants répartis. De façon non exhaustive, nous présentons les approches à conteneurs, les plus courantes, les modèles hiérarchiques très prometteurs et certains modèles plus « exotiques » réflexifs et/ou génératifs. La complexité des communications réparties imposent d'analyser de façon poussée les principes d'interaction entre composants : interfaçage et communication. Le concept architecturel de connecteur est également discuté. Finalement, nous effectuons un « plongeon » à l'intérieur des infrastructures pour en décrire certains aspects fondamentaux dans le cadre de la répartition ; nous insistons notamment sur les modèles de contrôle et le déploiement logiciel.

Introduction

L'ingénierie du logiciel a connu ces dernières années un développement très intense, avec notamment l'avènement des technologies objet et, plus récemment, l'essor des approches orientées modèle. Les méthodes de développement modernes permettent de faire le lien, de la façon la plus confortable possible, entre les problématiques de haut niveau des applicatifs - *la logique métier* - et les aspects techniques de plus bas niveau - *la logique système*. Les plateformes intergicielles telles que CORBA, RMI/Java et DCOM/.NET ont été développées pour répondre aux besoins spécifiques des architectures réparties. Cependant, et malgré de nombreux efforts investis dans ces technologies intergicielles, les propositions actuelles se révèlent encore limitées face à la problématique très complexe de répartition à large-échelle. Les architectures de composants répartis jouent ici un rôle prépondérant. Ils sont perçus par beaucoup comme le point d'articulation privilégié entre les problématiques métiers et les aspects purement informatiques dont la complexité est décuplée du fait de la nature répartie de l'environnement d'exécution. Sur le front des composants logiciels répartis, il n'est donc pas étonnant de retrouver les plus grands acteurs du logiciel, notamment l'OMG avec les modèles de composants CORBA (CCM) et SUN avec les Enterprise Java Beans (EJB).

Principes des systèmes répartis

Les systèmes répartis sont, dans leur définition la plus simpliste, des logiciels dont l'exécution se déroule sur un ensemble d'ordinateurs distribués géographiquement et reliés entre eux par des interconnexions réseau [1] [2].

Figure 1 : Exemple de système réparti

Erreur ! Liaison incorrecte.

Les applications disponibles sur Internet forment une catégorie hautement stratégique de systèmes répartis. La figure 1 ci-dessus représente une architecture typique de service Internet, mettant en jeu un serveur métier (ex. : serveur Web), un serveur de bases de données et un certain nombre de clients répartis géographiquement. Le principe de fonctionnement d'une telle application est relativement simple : les clients effectuent des requêtes vers le serveur (ex. : afficher une page HTML) qui en retour répond en accédant potentiellement à des informations gérées par le serveur de données.

On le voit sur cet exemple, un système peut être réparti pour des raisons « physiques », comme ici la séparation géographique naturelle entre clients et serveurs. De nombreux systèmes sont ainsi naturellement répartis : systèmes bancaires dont les agences sont réparties géographiquement, système de contrôle de processus industriels répartis sur plusieurs sites, etc. La miniaturisation des ordinateurs et l'émergence des technologies de réseau sans fil à haut débit introduit un nombre sans cesse croissant d'applications de nature intrinsèquement répartie.

D'autres considérations, de nature plus « logique », s'ajoutent à la problématique de répartition géographique. Le choix de séparer sur notre exemple le serveur métier du serveur de données peut notamment s'expliquer par un souci d'augmentation des *performances* du système. Deux machines (ordinateurs complets ou simples processeurs) physiques fonctionnent généralement plus vite qu'une seule machine comparable, c'est le postulat de base du *parallélisme*.

Si la machine serveur tombe en panne, l'intégrité du serveur de données ne sera pas compromise, seules les transactions en cours devront être annulées. La répartition permet donc d'augmenter la *fiabilité* d'un système.

Du point de vue de l'organisation fournissant le service (figure 1), le serveur métier est en contact avec le monde extérieur. Il est donc soumis aux éventuelles attaques de « clients » malintentionnés. Le serveur de données, en revanche est en quelque sorte « protégé » par le serveur métier, il n'est pas directement en contact avec l'extérieur. La séparation, ici d'ordre physique, permet de profiter de matériels spécifiques pour la sécurisation du système ; la protection matérielle étant en générale plus efficace qu'une protection purement logicielle. La répartition est donc également un outil important pour renforcer la *sécurité* des systèmes.

Large échelle

On distingue en général deux catégories de systèmes répartis : les systèmes limités en taille et les systèmes à large échelle. Les développements actuels se concentrent, dans leur écrasante majorité, sur les problèmes spécifiques de la répartition à large échelle. L'Internet, qui relie des millions de machines entre elles par l'intermédiaire de réseaux de très grande complexité, représente l'archétype des systèmes à large-échelle.

Hétérogénéité

Les systèmes répartis, à part dans des cas très spécifiques, ne sont pas l'apanage d'un constructeur donné. Ce sont des systèmes hautement hétérogènes, mettant en œuvre des plateformes matérielles et logicielles variées et souvent incompatibles entre elles. Gérer cette hétérogénéité passe par le développement de technologies standards et ouvertes permettant l'*intéropérabilité* entre systèmes différents.

Communications asynchrones

Déplacer de l'information prend du temps. Ce phénomène bien connu en sciences physiques prend en informatique tout son sens dans le cadre des systèmes géographiquement répartis, et notamment lorsque les distances entre les différents éléments du système sont importantes. Lorsque les temps de communication entre éléments répartis du système ne peuvent être prévus de façon précise et absolue, on dit alors que le système fonctionne de façon *asynchrone*. Il s'agit d'une différence fondamentale avec les environnements centralisés dont le fonctionnement *synchrone* est orchestré par une horloge globale unique.

Contrôle concurrent

Chaque machine ou processus d'un système réparti possède, de par son fonctionnement asynchrone, un contrôle autonome dit *concurrent*. Par exemple, sur la figure 1, les clients du système accèdent en même temps, et potentiellement en très grand nombre, aux services proposés par le serveur métier. Ce dernier doit donc « entremêler » les requêtes des clients en mettant en œuvre une politique de *synchronisation* pour gérer les éventuels conflits entre requêtes concurrentes. Les techniques de contrôle de concurrence sont parmi les aspects les plus sensibles et les plus complexes dans la conception des applications réparties. Les moniteurs transactionnels forment une catégorie hautement stratégique d'outils pour le contrôle de concurrence.

Pannes partielles

Les systèmes informatiques, comme probablement tous les systèmes physiques, peuvent être victimes de défaillances. Ainsi un ordinateur peut cesser de fonctionner subitement, et avec lui toutes les applications qu'il exécutait (ex. panne d'une machine cliente sur la figure 1). Un processus logiciel peut également tomber en panne, par exemple à cause d'un bogue ou par manque de ressources. Les réseaux de communication ont également leurs pannes spécifiques : pannes de canaux de communication et pannes de message (disparition, réordonnancement ou pire, altération de messages communiqués). Ces *pannes* dites *partielles* sont omniprésentes dans les systèmes répartis. Théoriquement, le problème de décider si un processus est en panne ou alors simplement ralenti est indécidable. L'on peut au mieux « soupçonner » une machine d'être en panne quand elle est inactive pendant un temps assez long. Pour la plupart des systèmes en ligne, la disponibilité des services est essentielle, toute panne doit être « réparée » de façon la plus rapide et la plus transparente possible.

Sécurité

La sécurisation d'un système centralisé est relativement simple. En effet, le bon déroulement des applications peut être supervisé assez facilement puisque la vision instantanée et complète de l'ensemble des ressources employées par le système est possible. Dans un système réparti de grande taille, la sécurisation est une problématique cruciale et autrement plus difficile. Il est par exemple nécessaire de protéger certaines communications sensibles, généralement par cryptage des informations transmises. Dans l'exemple de la figure 1, nous pouvons supposer que des clients transmettent au serveur certaines informations secrètes comme des numéros de cartes bancaires. Il faut aussi s'assurer des accès aux domaines protégés des systèmes. Dans les services Web, il est par exemple d'usage de distinguer les clients invités (ou anonymes) accédant à la version minimale des services, et les clients authentifiés possédant eux des droits d'accès étendus. De par leur *complexité* et leur *ouverture* au monde extérieur, les systèmes répartis déployés sur l'Internet ne peuvent être totalement infaillibles. Tout est question d'équilibre entre la robustesse des protections proposées face à l'ingéniosité (sans limites) des attaquants potentiels.

Des objets aux composants répartis

Les questions spécifiques - et difficiles - de l'informatique réparties, notamment celles évoquées dans la section précédente, s'ajoutent aux problèmes également très complexes de l'ingénierie logicielle des domaines métier. La confrontation de ces deux mondes, aux intérêts très souvent antagonistes, a conduit à

l'émergence d'une nouvelle catégorie de logiciel, couches intermédiaires entre les applications et le système d'exploitation que l'on nomme plateformes intergicielles (en anglais *middleware*).

Les plateformes intergicielles

Les technologies intergicielles ont pour but de supporter un modèle de développement de haut-niveau, focalisé sur les questions métiers dans le cadre d'une exécution des applications en environnement réparti à large-échelle. Le modèle de développement de la conception et de la programmation orientée objet occupe une place écrasante dans le domaine du génie logiciel. Les plateformes intergicielles proposées par l'industrie sont donc, sans surprise, focalisées sur le support des méthodes de développement à base d'objets en environnement réparti. Dans un premier temps, les efforts se sont concentrés sur les problématiques de *portabilité* du code applicatif et d'*interopérabilité* entre applications réparties. Disposer d'un code portable permet de s'affranchir de l'hétérogénéité des infrastructures matérielles et logicielles sur lesquelles il doit s'exécuter. Il existe principalement deux catégories de technologies pour la portabilité du code:

1. Les approches de type machine virtuelle
2. Les approches de type génération de code

Les technologies de type machines virtuelles standardisent un unique code « machine » (appelé code-octet ou *bytecode*) et l'infrastructure permettant d'interpréter ce code objet. Ce sont notamment les solutions proposées par Sun avec Java et Microsoft avec .NET. En revanche, la norme Corba se veut indépendante du constructeur, la portabilité du code y est proposée via génération de code. Le code (non-portable) d'implémentation est ainsi généré à partir de la description portable du logiciel généralement sous forme d'interfaces. Ces technologies concurrentes ont chacune leurs avantages et leurs inconvénients. Les approches de type machine virtuelle ont clairement l'avantage de la simplicité. En revanche, elles sont confinées à des constructeurs spécifiques, voire à des langages de programmation spécifiques dans le cas de Sun avec Java. La norme CORBA est par principe une norme ouverte, indépendante de constructeurs spécifiques. La génération de code est également propice aux performances puisqu'elle permet de profiter des environnements de compilation natifs des différentes plateformes. Notons cependant que les techniques de compilation au vol permettent une augmentation sensible des performances des machines virtuelles.

Un autre aspect important des plateformes intergicielles concerne les possibilités d'interopérabilité entre applications développées indépendamment. Ceci passe par la standardisation de la représentation des données échangées entre applications, ainsi que des protocoles supportant ces échanges. Le protocole IIOP (*Internet Interoperable Object Protocol*) de CORBA est probablement la norme la plus aboutie dans ce domaine. La plupart des plateformes permettent l'interopérabilité via IIOP.

Les objets répartis

Les technologies intergicielles proposées dans l'industrie proposent toutes le « portage » des fondements de la programmation orientée objet en environnement réparti. Parmi ces fondements, nous pouvons citer entre autre l'encapsulation du code et des données, la modularisation basée sur la notion de classe, la spécialisation par héritage, le polymorphisme, etc. Les objets « de base » deviennent ainsi *objets répartis*.

Figure 2 : Principes de bus à objets répartis CORBA

Erreur ! Liaison incorrecte.

Pour assurer les services de base de gestion des objets répartis, les intergiciels introduisent un certain nombre d'outils. Parmi ces outils se trouve l'élément fondamental de toute plateforme intergicielle objet : le *bus d'objet réparti* (ou *ORB* en anglais pour *Object Request Broker*). Ce logiciel dédié, dont l'architecture typique est représentée sur la figure 2, représente la clef de voûte de toute l'architecture intergicielle. L'*ORB* permet à des objets implémentés dans des environnements variés et localisés sur des machines réparties géographiquement d'interagir entre eux. Il implémente les protocoles fondamentaux pour le bon fonctionnement des objets répartis, en particulier la gestion du cycle de vie des objets (déploiement sur des sites distants, localisation d'instances, etc.) ainsi que les invocations distantes de méthodes. Le critère fondamental pour un bus d'objet réparti est de proposer ce genre de service de

manière *transparente,* en s'abstrayant dans les plus larges mesures possibles des contraintes liées à la nature répartie et hétérogène de l'infrastructure sous-jacente.

L'OMG, à travers la norme CORBA, a démocratisé la séparation systématique entre interfaces et implémentations des objets répartis. Les interfaces sont décrites séparément dans un langage standard et portable nommé IDL (*Interface Definition Language*). Y sont décrites les classes d'objets, notamment les types et noms des attributs, les signatures des méthodes ainsi que les relations d'héritage. Comme indiqué précédemment, des générateurs de code permettent de proposer des squelettes d'implémentations, sur lesquels les développeurs se basent pour implémenter les objets tout en garantissant la satisfaction des interfaces initiales.

Pour aborder les questions spécifiques des systèmes répartis, les plateformes intergicielles proposent généralement des solutions sous la forme de *services communs*. Ces services couvrent les aspects importants de contrôle de concurrence, de gestion de transactions, de sécurité, etc. Le plus souvent, ces services sont spécifiés par des interfaces normalisées. Ceci permet à différents vendeurs de proposer des implémentations alternatives, à condition bien sûr de respecter le cahier des charges des spécifications associées. L'ensemble de ces services, associé au bus d'objet réparti et aux outils de génération de code, permet une approche souple et hautement structurée de la conception de logiciel réparti.

Limites des objets répartis

Malgré l'indéniable succès des technologies industrielles dans certains domaines clés de l'informatique d'entreprise, on peut affirmer aujourd'hui que les objectifs initiaux - et très ambitieux - d'embrasser l'ensemble de la problématique de conception du logiciel réparti ne sont pas encore atteints. Des limites tant sur le plan métier que sur le plan de la répartition ont été découvertes ; limites qui expliquent en grande partie l'émergence de la notion même de composant logiciel. Pour notre étude focalisée sur les problématiques liée à la répartition, nous évoquons ci-dessous deux problèmes remettant en cause de façon importante le modèle des objets répartis.

Séparation des préoccupations

Une tension forte existe entre les aspects métiers et les aspects systèmes au niveau de la conception et de la programmation orientées objet. La vision idéaliste des standards objets, au début des années 90, envisageait un monde futur purement objet. Y seraient parfaitement modularisés et interfacés objets purement applicatifs, objets métiers (banques, domaine médical, aéronautique, etc) ainsi que services purement système (contrôle de concurrence, persistance, etc.). Pourtant, on le sait aujourd'hui, cette décomposition « parfaite », selon les préceptes de la *séparation des préoccupations* (*separation of concerns*) se heurte en pratique à de nombreux obstacles. De fait, les interactions fines entre les différentes couches applicatives métier et système considérées sont difficiles à modulariser. Des aspects systèmes sensibles comme la gestion des activités concurrentes, les politiques de synchronisation et les aspects liés à la fiabilité ou la sécurisation des systèmes ne s'abstraient qu'imparfaitement dans la métaphore objet. En pratique, le code des objets applicatifs contient de nombreux aspects métiers, entremêlés de façon inextricable avec des aspects systèmes, ce qui rend le tout fort peu réutilisable et modulaire [3].

Communications synchrones et couplage structurel

Le médium de communication occupe une place décisive dans l'infrastructure répartie. Ce medium met en jeu, du fait de la répartition, des échanges naturellement *asynchrones*. En revanche, le mode de communication incontournable des objets, via invocations de méthodes, est essentiellement *synchrone*. De plus, dans une invocation de méthode, le client de l'invocation « connaît » le serveur qui implémente la méthode ; il en possède une référence explicite. Cette connaissance représente une forme de *couplage structurel* (ou *spatial*) qui fige l'architecture dans le code des objets. Une invocation de méthode représente donc une forme de couplage entre la source et la cible de l'invocation, tant en termes structurels que du point de vue du contrôle. Cette relative inadéquation, que l'on pourrait penser insignifiante, se révèle particulièrement limitante en matière de performance et, plus encore sensiblement, lorsqu'il s'agit de passage à l'échelle.

Vers les composants logiciels

Il est important de noter que les limites évoquées précédemment ne remettent aucunement en question la notion même d'objet réparti, qui est désormais acquise. La mise en œuvre de motifs de conception (en anglais *design patterns*) [27] dédiés permet de répondre assez précisément aux limites évoquées ci-dessus. L'inconvénient de ces motifs de conception concerne leur complexité : ils ne peuvent être mis en œuvre que par des experts en matière d'architectures logicielles réparties. De plus, parmi les nombreux motifs proposés, certains sont complémentaires et d'autres sont « en compétition ». Il est difficile de faire le tri comme il est parfois difficile de faire le tri entre plusieurs implémentations d'un même algorithme.

L'émergence du concept de composant réparti semble ainsi répondre à un besoin d'uniformisation et de systématisation face à cette profusion de recettes d'experts face aux problèmes posés. À ceci s'ajoute également une volonté évidente d'extraire certains concepts comme réellement fondamentaux, aussi fondamentaux que la notion d'objet elle-même. Il est notable que les plateformes intergicielles mettant en œuvre les composants répartis sont généralement bâties sur des technologies intergicielles à objets. C'est en particulier le cas pour l'Object Management Group qui se base sur CORBA pour proposer le modèle des composants CCM. Mais c'est également vrai pour SUN avec les EJB qui reposent beaucoup sur les services du bus d'objets Java RMI. Le cas de Microsoft est un peu à part puisque l'objet « ubiquitaire » n'est apparu que récemment avec le langage C# et la boîte-à-outils .NET (rappelons que les technologies plus anciennes comme COM+ ne sont pas des technologies purement objet). On le voit donc, les plateformes intergicielles à base de composants représentent essentiellement une évolution des plateformes à objets de la génération précédente.

Architectures applicatives

Avant d'aborder les principes architecturaux des plateformes à base de composants répartis, nous devons nous intéresser à ceux des applications une fois déployées. Contrairement au cas centralisé ou la décomposition du logiciel peut se faire de façon purement logique, il est nécessaire de réfléchir longuement à la problématique de décomposition architecturale dans le cadre réparti. Des critères tant matériels que logiciels entrent en ligne de compte. Des architectures spécifiques à la répartition ont été proposées et largement mises en œuvre, d'autres sont encore à l'étude ou restent encore à inventer.

Le client/serveur

Les architectures de type client/serveur ont longtemps été considérées comme intrinsèques des systèmes répartis, en particuliers dans le milieu industriel. En effet, il est un peu réducteur mais assez consensuel de schématiser le rôle d'une entreprise par la fourniture de produits ou de services au plus grand nombre possible de clients. Ces clients sont de plus généralement physiquement éloignés des ressources informatiques du fournisseur de services, qui forment la partie serveur du système. Depuis les premiers systèmes d'informations industriels ou organisationnels (comme les précurseurs de l'Internet, ArpaNet) et jusqu'aux principaux services Internet déployés à l'heure actuelle (ex. : le courriel ou le Web), la métaphore client/serveur est quasiment incontournable. Cette décomposition minimale a pour principal intérêt de se concentrer sur la partie la plus stratégique du modèle à l'aune de la répartition : le serveur. Il ne s'agit pourtant pas de la seule décomposition possible pour les applications réparties, loin s'en faut.

Les architectures 3-tiers, N-tiers

On désigne aujourd'hui le client/serveur comme une architecture de type 2-tiers, composée d'un tiers client et d'un tiers serveur. Le tiers serveur, sur lequel les plus grands efforts de développement sont aujourd'hui concentrés, peut évidemment être décomposé à son tour.

Figure 3 : Du client-serveur aux architectures 3-tiers

Erreur ! Liaison incorrecte.

Comme indiqué sur la figure 3, la décomposition structurelle du serveur en un tiers métier et un tiers s'occupant exclusivement des aspects systèmes est très motivante pour la séparation des préoccupations. Nous le verrons, cette décomposition n'est pas toujours possible en termes purement architecturaux. Elle

est en revanche possible pour une catégorie importante de logiciel : les applications d'entreprise orientées données. En effet, les serveurs de bases de données sont aujourd'hui des produits « clé en main », sortes de boîtes noires de haute technicité avec lesquelles on communique essentiellement par des requêtes dans des langages de haut niveau (tel SQL), dans le cadre de transactions. Il est donc fort intéressant d'extraire cet aspect système stratégique du reste du serveur. Ces architectures 3-tiers orientées données sont aujourd'hui très populaires, ciblées de façon prioritaire par les technologies componentielles. Cependant, la plupart des acteurs du marché sont bien conscients que la décomposition côté serveur ne s'arrête pas en si bon chemin. Les architectures N-tiers généralisent ainsi le principe de décomposition de la partie serveur des systèmes d'entreprise. Dans les technologies Microsoft, le tiers *proxy* occupe également une place autonome et importante. En effet, les outils d'interaction client/serveur sont disponibles en grand nombre : messages (MQS), invocations de méthodes (DCOM), scripts et services WEB (ASP, SOAP), etc. Le tiers *proxy* introduit en tant qu'entité indépendante permet l'intégration souple de ces technologies variées et souvent très différentes, voire incompatibles entre elles, au sein d'un serveur d'entreprise. Une des problématiques majeures des fournisseurs de solutions pour l'informatique d'entreprise concerne l'interopérabilité avec les systèmes d'information pré-existants souvent propriétaires et parfois archaïques mais incontournables ; c'est le fameux tiers *legacy*. D'autres raffinements N-tiers sont à l'étude à l'OMG, chez Sun, Microsoft, IBM [28] et de nombreux autres acteurs des infrastructures réparties d'entreprise.

Les architectures nouvelles

Les architectures de type N-tiers couvrent essentiellement les besoins de systèmes d'information d'entreprise relativement « classiques ». Mais ces architectures centrées serveur semblent parfois quelque peu démunies face à l'évolution incessante des technologies informatiques : vitesse, complexité et dynamicité des réseaux, puissance et multiplicité des points d'accès, etc.

Le P2P ou la « fin » du tiers serveur

Le point de vue de la décomposition - réputée incontournable - en 2-tiers « minimaux » client et serveur est sensiblement remise en question dans les approches de type pair à pair (en anglais *peer to peer* P2P) [4]. D'un point de vue conceptuel, la principale innovation du P2P est de considérer chaque « acteur » du système sur un pied d'égalité : chacun peut tour à tour occuper le rôle d'un client ou d'un serveur dans la métaphore traditionnelle des systèmes répartis, et qui perd au passage quelque peu de son sens. En pratique, il s'agit d'une remise en question de nombreux aspects sous-jacents des technologies de répartition, notamment sur les aspects de routage réseau (principes d'acheminement des informations communiquées sur le réseau). Les systèmes P2P sont aujourd'hui parmi les services Internet les plus employés ; et on leur prête une gamme d'applications de plus en plus variée : outils collaboratifs, partage de contenu multimédia, jeux en ligne, etc.

Les grilles de calcul

Dans un registre différent mais tout aussi « chaud », le projet de lancement du plus grand accélérateur de particules au monde, le célèbre LHC du CERN à Genève (là même où Tim Berners Lee inventa le Web) propose un défi de taille aux acteurs de la communauté informatique. Jamais une quantité d'information aussi importante, et produite à un rythme si rapide, n'aura été générée par un système numérique. Ce sont des milliers d'ordinateurs, répartis sur la planète entière, qui seront nécessaires pour « digérer » cette énorme quantité d'information. Des systèmes spécifiques sont en cours d'élaboration, on les appelle les *grilles de calcul* (les anglo-saxons parlent de *grid computing*) [5]. Là également l'ensemble de l'infrastructure est à revoir, mais cette fois-ci la tendance est inversée. Le « client » (les scientifiques) est paradoxalement l'élément le plus « coûteux » du système. A partir d'un flot unique d'information, des millions de nœuds du système doivent être fournis en informations correctement formatées et découpées.

Les approches centrées réseau

Dans le logiciel « traditionnel », l'accent est mis sur les entités qui entrent en interaction (procédures, objets) plus encore que sur leurs modes d'interaction (envois de message, invocations de méthodes, etc.). Les technologies centrées sur le réseau, notamment SUN avec Jini [6] ainsi que les spécifications ouvertes de l'alliance OSGi [24], prennent de l'importance notamment en raison de l'essor des technologies sans fil. Ces approches opèrent ainsi un autre « retournement » en se focalisant sur la couche

de communication plutôt que sur les entités réalisant les calculs proprement dits (sans pour autant les oublier totalement !). Les réseaux possèdent ainsi parfois des capacités de traitement propre (réseaux intelligent et actifs) et proposent également une dynamique structurelle (réseaux mobiles) qui ont un impact sensible sur les technologies logicielles.

Autres architectures

Au-delà des trois exemples proposés ci-dessus, la famille des architectures de systèmes répartis ne cesse de s'agrandir. On peut par exemple citer la diffusion de flux multimédias en ligne, les réseaux de capteurs actifs, les systèmes multi-agents ou encore les espaces actifs et autres cités numériques, etc. Il est intéressant de noter, en outre, que la notion de composant occupe souvent une place de choix dans les approches dédiées à ces différents domaines émergents.

Modèles de composants logiciels répartis

Les plateformes intergicielles basées sur la notion de composant proposent des solutions aux limites des approches objets de la génération précédente. Au nombre de ces limites, la *séparation des préoccupations* entre code fonctionnel, ou logique métier, et code non-fonctionnel, ou logique système, se trouve réellement au cœur des développements centrés sur la notion de composant logiciel. Dans ce but, les modèles de composants répartis sont généralement architecturés selon le principe de l'*inversion du contrôle*, également appelé « principe d'Hollywood ». L'idée est de placer le contrôle applicatif, et notamment tout ce qui concerne les interactions entre couches logicielles métier et système, sous l'autorité d'entités extérieures, gérées elles au sein de la plateforme intergicielle. Dans le cadre des composants répartis, plusieurs approches ont été proposées pour réaliser cette inversion du contrôle : les approches à conteneurs, les approches hiérarchiques ainsi que des approches plus expérimentales comme les modèles réflexifs et/ou génératifs, notamment certains travaux issus de la programmation par aspects. Il est intéressant de noter que ces différentes approches ne sont pas incompatibles entre elles ; elles correspondent plutôt à des points de vue différents, et parfois conciliables, sur la réalisation du principe d'inversion du contrôle.

Modèles à conteneurs

Dans les plateformes industrielles EJB et CCM, qui ciblent essentiellement les systèmes d'information d'entreprise et les architectures 3-tiers, les composants logiciels sont avant tout considérés comme des entités métier. La notion complémentaire de composant système commence également à émerger. Mais dans tous les cas de figures, les composants sont introduits comme entités spécialisées, qui doivent interagir avec d'autres concepts indépendants. En particulier, l'articulation entre code métier (implémenté par les composants) et code système (implémentés par les services du tiers système) est placé sous l'autorité d'entités indépendantes que l'on nomme *conteneurs*.

Figure 4 : Architectures à conteneurs

Erreur ! Liaison incorrecte.

Comme le montre la figure 4 ci-dessus, les conteneurs se situent dans les architectures 3-tiers à la croisée des trois logiques client, métier et système. Leur rôle primordial est de contrôler les composants métiers, en fonction à la fois des requêtes client et des ressources système sous-jacentes. Pour cela, la mise en œuvre des conteneurs repose sur deux principes fondamentaux :

1. Mécanismes d'interception
2. Mécanismes de gestion contextuelle

Les interactions entre les clients et le serveur reposent sur des modèles de communication variés (cf. suite du chapitre), elles sont conceptualisées par la notion de *proxy*. En plus de gérer la communication, les *proxys* sont également employés pour distinguer les requêtes de nature métier (ex. : achat d'un article en ligne) des requêtes système (ex. : ouverture d'une transaction), on parle dans ce cas de *rôle d'interception*. Une partie de la gestion de l'identité et du cycle de vie des composants métiers est parfois déléguée à des entités spécifiques nommées *maisons* (*homes* en anglais). Le conteneur est également responsable des interactions entre les composants métier et la logique système sous-jacente. Pour les EJB,

un unique type de conteneur, dit conteneur EJB, englobe l'ensemble des fonctionnalités système supportées par la plateforme. Dans les approches les plus récentes, et notamment CCM, des composants de service sont de plus introduits en tant qu'intermédiaires supplémentaires dans la couche système. Les *contextes*, pour leur part, permettent de conserver au sein du conteneur des informations reflétant, en termes de contrôle, l'état des composants métiers. Une partie de cet état contextuel est visible par le code métier, le reste étant encapsulé de façon opaque dans la logique système.

Modèles hiérarchiques

Dans la métaphore componentielle, la notion de *composition hiérarchique* éveille de façon grandissante l'intérêt des architectes logiciels. La possibilité de créer des composants de haut niveau - dits *composites* - par composition de composants de niveaux d'abstraction inférieurs représente un principe de construction logicielle à la fois naturel et expressif. Si cette notion fait relativement lentement (mais sûrement) son apparition dans les approches industrielles, les modèles hiérarchiques issus du monde de la recherche arrivent à maturité. Parmi les nombreuses approches proposées, nous pouvons distinguer les spécifications *Fractal* qui ont l'avantage d'être à la fois portables et extensibles [7]. Ces spécifications sont développées au sein du consortium ObjectWeb. Dans ce modèle sont distinguées trois catégories complémentaires de composants :

1. *Composants basiques* : ce sont des composants minimaux avec lesquels on communique uniquement par appels de méthodes sur leur interface. Cette couche est avant tout spécifiée pour permettre l'interopérabilité avec les environnements objet traditionnels.
2. *Composant primaires* : ils ajoutent aux composants basiques un *contrôleur* chargé d'un certain nombre de fonctionnalités liées au contrôle du composant.
3. *Composants composites* : ces composants ajoutent aux composant primaires un *contenu*, sous la forme d'une interconnexion de sous-composants, eux-mêmes basiques, primaires ou composites.

La grande nouveauté du modèle concerne donc la troisième catégorie qui donne corps à la notion de composition hiérarchique. Un composant composite *Fractal* possède la structure représentée sur la figure 5 ci-dessous.

Figure 5 : Modèle de composant hiérarchique Fractal

Erreur ! Liaison incorrecte.

Un composant composite est subdivisé en deux parties distinctes : la partie *contrôleur* et la partie *contenu*. Un contrôleur fractal ressemble à une généralisation du principe de conteneur des architectures EJB et CCM, réalisant tout comme ce dernier une inversion du contrôle, propice à la séparation des préoccupations. Les interfaces standards pour les contrôleurs sont cependant de plus bas niveau dans le cadre de fractal, elles concernent les propriétés non-fonctionnelles suivantes :

1. L'introspection des interfaces de composant : noms et contenus des interfaces.
2. La gestion du cycle de vie : démarrage et arrêt de composants et sous-composants.
3. Gestion des liaisons extérieures : relations entre interfaces requises et fournies.
4. Gestion des attributs : lecture/écriture de propriétés.
5. Gestion du contenu pour les composants composites : ajout/retrait de sous-composants.

Ces interfaces standards sont loin de couvrir l'ensemble des besoins en matière de contrôle, notamment dans le cadre des systèmes répartis. Mais les spécifications *Fractal* sont ouvertes et extensibles, ce qui permet de les implémenter sous des formes variées (il existe notamment des implémentations en C et en Java) et d'introduire des interfaces de contrôles spécialisées à différents domaines d'application. Ainsi, la plateforme ProActive ajoute un ensemble d'interfaces de contrôles pour les applications de grilles de calcul. Ces interfaces spécialisées concernent notamment la concurrence, la communication de groupe ou encore la mobilité [8].

Modèles réflexifs et génératifs

L'inversion du contrôle, c'est-à-dire l'externalisation du contrôle sur les composants applicatifs, est répétons-le encore, au cœur des différentes plateformes intergicielles à base de composants répartis. Dans les plateformes « traditionnelles », les interfaces de contrôle sont fournies par des entités (conteneurs ou autres contrôleurs) dont l'implémentation est généralement assez *opaque* ; elle n'est pas - ou peu - accessible au développeur applicatif. Dans les modèles *réflexifs* et *génératifs*, en revanche, les implémentations des principes d'inversion du contrôle sont ouvertes. Les modèles réflexifs sont basés sur une séparation entre un niveau base, fonctionnel, et un niveau méta, pour la gestion de propriétés non-fonctionnelles. Les liens entre ces deux couches reposent sur des principes d'*introspection* qui permet au niveau méta de « découvrir » les fonctionnalités disponibles au niveau base. En complément, on introduit le principe d'*intercession* qui permet la prise de contrôle par le niveau méta à certains moments clé de l'exécution des applications. Il existe de multiples façons de structurer les niveaux base et méta des architectures réflexives. Dans le cadre des plateformes componentielles, l'idée est généralement de fournir les principes de composition (éventuellement hiérarchique) à la fois au niveau base et au niveau méta (cf. figure 6). La nature des composants méta est variable d'une implémentation à l'autre. Par exemple, dans la plateforme OpenORB développée à l'Université de Lancaster en Grande-Bretagne, on trouve des méta-composants essentiellement dédiés aux applications multimédia (streaming, qualité de service, etc.) [9].

Figure 6 : Exemple de modèle réflexif

Erreur ! Liaison incorrecte.

Les approches génératives proposent, tout comme les approches réflexives, de réifier certains aspects de l'implémentation de fonctionnalités de contrôle pour les rendre accessibles au niveau applicatif. En revanche, les technologies sous-jacentes reposent essentiellement sur des principes de génération de code et/ou de manipulation au vol de code-octet (*bytecode*). Le domaine très en vogue de la *programmation par aspects* repose principalement sur de telles techniques génératives. Parmi les aspects de contrôle les plus fréquemment réifiés, citons entre autre l'interception des invocations de méthodes permettant d'encadrer ces invocations par des pré et/ou post-traitements. Un intérêt supplémentaire des techniques génératives est de permettre l'introduction des principes d'intercession en tant que primitives de langage de programmation (points de coupure, conseils, etc.). Récemment, certains principes de la programmation par aspects ont été introduits pour « ouvrir » l'implémentation de conteneurs EJB industriels, notamment dans les conteneurs JBoss [10] et Spring [11].

Principes d'interfaçage et modes de communication

Nous avons décrit les principales approches pour donner corps à la notion de composant logiciel réparti. Les architectures proposées sont principalement focalisées sur la séparation des préoccupations entre logique système et logique métier, et reposent essentiellement sur l'introduction de formes variées d'inversion du contrôle. Dans le domaine de la répartition, il est important de considérer les modèles d'interactions entre composants avec au moins autant d'attention que les modèles de composant eux-mêmes. Tout comme pour les objets répartis, il est important, au préalable, d'établir une distinction précise entre le niveau conceptuel des interactions et leur réalisation pratique, qui considère plus précisément la nature répartie du système et l'omniprésence des réseaux de télécommunication.

Interfaces de composants

Tout comme pour les objets répartis, les plateformes intergicielles à base de composants introduisent invariablement une notion plus ou moins complexe d'*interface*. Les interfaces de composants logiciels sont généralement plus riches que les interfaces classiques d'objets. En premier lieu, les interfaces décrivant les principes d'interaction avec les composants sont définies à l'extérieur des descriptions des composants eux-mêmes. De plus, les modèles de composants introduisent généralement une séparation claire entre *interfaces fournies* et *interfaces requises*. Les interfaces fournies décrivent classiquement les services implémentés par les composants logiciels. La nouveauté concerne l'adjonction d'interfaces requises pour décrire les services dont les composants ont besoin pour fonctionner. On trouve donc des mécanismes de liaison spécifiques consistant à mettre en correspondance des interface requises par

certains composants à des interfaces compatibles fournies par d'autres composants. Pour effectuer cette liaison, les interfaces sont réifiées au statut d'entités de première classe, que l'on peut manipuler de façon autonome.

Considérons un service de calcul proposant une méthode permettant l'addition de deux nombre, dont l'interface est la suivante :

```
Interface AddService {
  Number Add(Number a, Number b);
}
```

Cette méthode, de nom `Add`, prend deux nombres `a` et `b` (de type `Number`) et retourne le nombre résultat de l'addition. Supposons un composant de calcul, par exemple dans le domaine de la conception assisté par ordinateur, ayant besoin du service d'addition. Ce dernier exposera donc l'interface `AddService` comme une interface requise. Ce service pourra être fourni par un composant de calcul de plus bas niveau qui lui explicitera `AddService` comme une interface fournie. Pour faire le lien entre ces deux composants et donc le lien entre les interfaces requises et fournies, une possibilité est d'identifier chacune de ces interfaces par un nom unique. Ainsi, on pourra utiliser une construction de la forme :

```
Bind(nom de l'interface requise, nom de l'interface fournie)
```

Pour obtenir ce nom unique d'interface, on peut considérer le nom (ou le nom de type) des composants eux-mêmes. Si les composants se nomment respectivement `C1` (client de `AddService`) et `C2` (fournisseur de `AddService`), on pourra effectuer le lien entre les deux interfaces par une construction de la forme :

```
Bind(C1.AddService, C2.AddService)
```

Ainsi, chaque invocation au service d'addition sur `C1` pourra être automatiquement délégué au service d'addition proposé par `C2`.

Invocation de méthodes distantes

Les invocations de méthodes distantes représentent l'évolution des « anciens » appels de procédures distantes ou RPC (*Remote Procedure Calls* en anglais) vers les technologies d'objets répartis. L'interface d'une méthode est composée, comme le montre l'exemple précédent, au minimum du nom de la méthode, du type de chacun de ses paramètres, et finalement, du type de sa valeur de retour. En termes techniques, la mise en œuvre des invocations de méthodes distantes est complexe, notamment du fait de l'hétérogénéité des systèmes. Pour illustrer les différents mécanismes mis en jeu, considérons les détails d'une invocation à la méthode `Add` de l'interface `AddService` décrite précédemment, tels que décrits sur la figure 7.

Figure 7 : Le mécanisme d'invocation de méthodes distantes

Erreur ! Liaison incorrecte.

Du point de vue de l'invocation, il est naturel de distinguer le composant client qui réalise l'invocation, du composant serveur qui implémente la méthode. Contrairement aux objets répartis, cette distinction est ici dynamique et ne correspond pas forcément à une séparation client/serveur en terme d'architecture. Ainsi, on peut imaginer que le composant « serveur » de la figure 7 puisse ultérieurement invoquer une méthode du « client » (il échangeront donc leur rôle respectif à cette occasion). Pour qu'un client initie une invocation, il doit au préalable « posséder » le fil de contrôle ; on parle encore de *contrôle actif*. Ce contrôle est suspendu au moment de l'invocation, en raison de la nature synchrone du principe d'invocation. Cette nature synchrone est due en partie à l'attente éventuelle d'une valeur de retour de l'invocation ou, plus subtilement, au fait que deux invocations successives (même sans valeur de retour) sont souvent « pensées » comme s'exécutant en séquence. Ce contrôle synchrone correspond, du point de vue de l'environnement réparti, à un couplage fort, pendant toute la durée de l'invocation, entre le client et le serveur, ou plus précisément entre la souche du client et le squelette d'implémentation du serveur, qui gèrent le code système. D'un point de vue système, ce couplage peut poser problème. Il est possible, par exemple, que le destinataire mette un temps très long, voire infini, pour répondre. Pendant toute cette période, le composant client (et éventuellement lui-même serveur pour d'autres composants) restera en attente passive. L'encapsulation ne couvre donc pas le phénomène du contrôle. D'un point de vue plus pratique, l'invocation de nature synchrone et fiable n'est pas très efficace en terme de système réparti. Pour la mettre en œuvre, il faut échanger de nombreux messages de synchronisation et autres subtilités dont on aimerait parfois se passer.

Passage de messages

Les infrastructures réseaux privilégient, du moins à large échelle, les échanges asynchrones. Le mode de communication dit du passage de message fut le premier à être mis en œuvre dans les technologies logicielles pour les systèmes répartis. Sa proximité avec la structure physique du réseau sous-jacent n'y est bien sûr pas étrangère. Les interfaces pour le passage de message se décomposent plus simplement que les méthodes en parties requises et parties fournies. Ainsi un fournisseur de service de messagerie pourra exposer une méthode `Receive` retournant les éventuels messages reçus, comme ci-dessous :

```
Interface ProvideMessageService {
  Message Receive() ;
}
```

Du côté des composants utilisateurs du service, l'interface requise sera essentiellement caractérisée par une méthode paramétrée par le message à émettre et son destinataire :

```
Interface RequireMessageService {
  Send(Message m, Component dest) ;
}
```

Figure 8 : passage de messages

Erreur ! Liaison incorrecte.

Comme le montre la figure 8, le principal intérêt du passage de message est de permettre un échange non-bloquant, ou asynchrone. Au début du scénario, la source est dans l'activité d'émission d'un message vers la destination alors que cette dernière est en attente (passive) de réception d'un nouveau message. On pourra également considérer le cas d'un destinataire actif ; le message ne pourrait être délivré que lorsque le destinataire se retrouve en état d'attente. Ce qui réclame la mise en œuvre de files d'attentes de messages, ou boîtes aux lettres. Après avoir émis son message, la source peut naturellement et instantanément continuer son travail. La destination pour sa part s'active pour traiter ce nouveau message reçu. Ce type d'interaction est efficace en terme de système réparti. Le fait que la source puisse continuer à fonctionner après l'émission de son message, indépendamment du fonctionnement de la destination, reflète naturellement le fonctionnement concurrent des différents nœuds d'un système réparti.

Incontournables dans les couches basses des systèmes répartis (les protocoles de communication de bas niveau sont tous basés sur les échanges asynchrones de messages), et contrairement à ce que l'on pourrait penser, le passage de message est également intéressant à un haut niveau d'abstraction, notamment dans le cadre des intergiciels orientés messages MOM (*Message Oriented Middleware*) [12]. Ces infrastructures permettent la mise en place de systèmes répartis dont les différentes composantes sont fortement découplées entre elles, à la fois dans le temps et dans l'espace. Les MOM modernes comme MSMQ (*Microsoft Message Queuing*) de Microsoft [13], WebSphere MQ d'IBM [14] ou JMS (*Java Messaging Service*) de SUN [15] offrent toute une panoplie de services. Ils permettent par exemple à des messages non livrables instantanément d'être stockés temporairement (parfois plusieurs jours), via le « stocker et faire suivre » (en anglais, *store and forward*). Dans les systèmes embarqués, ce mode d'interaction est particulièrement adapté à la gestion des déconnexions intempestives. Les MOM sont également largement employés pour permettre l'interopérabilité avec les systèmes d'informations d'entreprises « *lecacy* » [16].

Modes de communications émergents

Les deux précédents modes de communication identifiaient, de façon explicite, une source/client et un destinataire/serveur pour la communication. Il s'agit de modes de communication point-à-point inspirés pour une bonne part par le modèle téléphonique. En complément, de nombreuses variantes jouent sur le nombre d'émetteurs et de récepteurs d'information ; on parle de *cardinalité* du modèle de communication. Dans la métaphore du courrier électronique, le mode de diffusion multipoint (ou *multicast*) occupe une place stratégique au sein de l'Internet. Ces modes de communication généralisent le passage de message de cardinalité 1-1 (1 émetteur associé à 1 récepteur) à la diffusion de message de cardinalité 1-N (1 émetteur et N récepteurs). La communication de groupe (utilisée notamment pour la résistance au pannes), les architectures de *streaming* (ex. : télévision sur internet, diffusion multimédia, etc.) et les grilles de calcul reposent largement sur cette idée en terme d'infrastructure de communication. En termes d'interfaces, la principale différence avec le passage de message concerne l'absence de destinataire explicite pendant l'envoi. On pourra par exemple passer un identifiant de groupe explicite :

```
Interface RequireGroupMessageService {
  Send(Message m, ComponentGroup destgroup) ;
}
```

La cardinalité 1-N se généralise encore aux modes de communication de cardinalité M-N où l'on trouve un nombre non prédéfini d'émetteurs d'informations associé à un nombre également non prédéfini de récepteurs. Les communications par événements typés et les protocoles de type publication/souscription (*publish/subscribe*) font partie de cette catégorie. Ces modes de communication sont particulièrement adaptés aux réseaux ubiquitaires et à l'informatique nomade. Cette fois-ci, c'est la nature des informations contenues dans l'événement (message typé) qui décide de l'acheminement. L'information la plus largement utilisée concerne le type de l'événement, on parle alors de *publish/subscribe* basé sur les types ; il existe également des services de *publish/subscribe* basés sur les « topiques » et/ou le contenu [17]. Du côté du composant publiant l'événement, on dispose ainsi d'une interface de la forme suivante :

```
Interface PublisherService {
  Publish(Event e) ;
}
```

Les souscripteurs, pour leur part, possèdent une interface double permettant à la fois de souscrire à des catégories d'événements (types, topiques et/ou descripteurs de contenu) et également de recevoir les événements comme dans le cadre du passage de message « standard » :

```
Interface SubscriberService {
  Subscribe(EventType t);
  Event Receive();
}
```

De façon intéressante, les CCM introduisent des concepts de sources et de puits d'événements pour prendre en compte, au niveau du modèle abstrait lui-même, un mode de communication M-N. L'avantage de la communication événementielle asynchrone est que le découplage devient total, en termes aussi bien temporels (asynchronisme) que spatiaux (pas de références explicites de sources et de destinataires).

Concernant le passage à l'échelle, à l'exception des architectures particulières du publish/subscribe, les modes de communications introduits jusqu'à présent possèdent un facteur limitant incarné par le routage déterministe qu'ils sous-entendent. Or toutes ces propositions peuvent être envisagées sous l'angle du non-déterminisme. Par exemple, un objet désirant utiliser un service, par exemple par invocation de méthode ou passage de message, identifie statiquement (parfois dynamiquement par l'intermédiaire de motifs de conception adequats) un destinataire. Ce destinataire explicite dénote une route précise pour le réseau sous-jacent, une route potentiellement surchargée, peu fiable, etc. Une autre possibilité consiste en un choix non-déterministe d'un destinataire capable de rendre ce service, suivant une route la plus efficace possible. Il s'agit probablement du grand apport des technologies pair-à-pair, pour l'instant essentiellement représentée pas les services d'échanges de fichiers mais qui auront sans aucun doute une incidence sur les problématiques du monde industriel [4].

Langages de description d'architectures et connecteurs

La liste des modes de communication disponibles sur le réseau ne cesse de grandir. Ainsi nous n'avons pas évoqué le cas des protocoles de haut-niveau comme SOAP ou XML-RPC. Il existe également des protocoles spécifiques au monde de l'embarqué et de la téléphonie, au satellitaire, etc. La plupart des approches du marché se contentent d'intégrer ces différentes technologies au fur et à mesure des besoins. Une approche fédératrice permettant de spécifier ces différents modes de communication au sein d'un même environnement semble de plus en plus nécessaire. Des chercheurs et industriels proposent dans ce but les langages de description d'architectures (ADL) et les modèles de connecteurs pour spécifier, de manière portable et réutilisable, les principes d'interfaçages, protocoles de communication et technologies sous-jacentes. Ces connecteurs disposent d'un type ainsi que d'un ensemble de propriétés [18]. Par exemple, les propriétés de base de connecteurs pour les modes de communication décrits dans ce chapitre sont résumées sur le tableau 1. Le critère de scalabilité (ou de capacité à passer à l'échelle) est bien sûr assez subjectif.

Tableau 1 : Type de connecteurs et propriétés

Type de connecteur	**Contrôle**	**Cardinalité**	**Couplage Temporel/Spatial**	**Routage**	**Scalabilité**
Invocations de méthodes	Sync	1-1	Oui/Oui	Dét	Faible
Passage de message	Async	1-1	Non/Oui	Dét	Moyenne
Multipoint	Async	1-N	Non /Oui	Dét	Moyenne
Pub/sub	Async	N-M	Non/Non	Dét	Bonne
Pair à pair (P2P)	Tout	Toute	-/-	Non-dét	Excellente

En pratique, de nombreuses autres propriétés sont à prendre en compte comme le format des données échangées, les possibilités de persistance, etc. Une fois réalisées, de telles spécifications représentent une aide inestimable pour les experts architectes, qui peuvent ainsi choisir de manière confortable les infrastructures de communication correspondant le mieux au cahier des charges des applications. Le concept de connecteur est également très intéressant pour la conception et l'implémentation même des plateformes intergicielles. Récemment, ce concept a été employé avec succès pour concevoir et implémenter des couches génériques pour la communication entre les composants répartis et des services Web ou des systèmes d'information d'entreprise anciens et/ou propriétaires. Il s'agit notamment du J2EE Connector Architecture de Sun [19].

Problématique de contrôle

L'infrastructure de contrôle concentre une bonne partie des problèmes difficiles liés au déploiement des applications en environnement réparti à large échelle. De ce point de vue, nous l'avons vu, la principale

rupture avec le monde objet concerne la séparation la plus claire possible, dans les fondements mêmes des modèles de composant, entre la logique métier, exprimée par les composants eux-mêmes et la logique de contrôle (ou logique système), mise en œuvre par des concepts et mécanismes dédiés. Il nous reste à décrire les différents services de contrôle proposés dans les platerformes componentielles majeures.

Contrôle de concurrence

La concurrence est un problème fondamental posé par le partage de ressources réparties au sein d'un système ; ces ressources pouvant donc être accédées par plusieurs activités/composants simultanément, depuis plusieurs sites répartis du système (cf. figure 9).

Figure 9 : Composants accédant en concurrence à une même ressource

Erreur ! Liaison incorrecte.

Le contrôle de concurrence est un problème difficile - parmi les plus difficiles - de l'informatique répartie. Trois tensions fondamentales orchestrent la synchronisation d'activités concurrentes. Tout d'abord, il s'agit d'optimiser le degré de concurrence (ou de parallélisme) du système, c'est la propriété d'*efficacité*. Il existe un objectif non moins important de *sûreté* qui interdit aux applications concurrentes de se retrouver dans un état inconsistant ou bloqué. Par exemple, si deux activités devant se dérouler en exclusion mutuelle se produisent simultanément, alors on enfreint la propriété de sûreté. Les malencontreusement célèbres *deadlocks* (« verrous mortels ») qui bloquent le système avant terme sont également du domaine de la sûreté. Enfin, il est important de garantir que le système progresse dans son exécution. On parle alors d'en assurer la *vivacité*. Un *livelock* (« verrou vivant ») dans lequel le système semble avancer (i.e. il change d'état) mais ne produit en fait rien d'intéressant est un problème typique de *vivacité*. Toute la difficulté du contrôle de concurrence réside dans l'équilibre entre ces trois propriétés, souvent opposées. En effet, la tension entre vivacité et sûreté est forte, et les algorithmes de contrôle de concurrence sont complexes et consommateurs de ressources.

Synchronisation par verrous

Au plus bas niveau des infrastructures, des services système sont proposer pour décrire les politiques de synchronisation entre activités concurrentes. Les modèles de synchronisation par *verrou* (*lock*) forment une catégorie importante pour la synchronisation de grain fin ; il s'agit en l'occurrence du modèle spécifié dans la norme CORBA [20]. L'idée est de permettre de bloquer l'accès en concurrence par plusieurs activités simultanées à une ressource partagée par la « pose » d'un *verrou* sur cette ressource. On trouve des types de verrous différents pour le verrouillage en lecture (qui autorise d'autres lectures) et en écriture (qui interdit toute autre lecture ou écriture). Un moyen de systématiser la pose et le retrait des verrous est de mettre en place deux phases distinctes dans le cadre du 2PL (*2-Phase Locking* en anglais). Les synchronisations se déroulent donc par « vagues » avec dans un premier temps l'acquisition des ressources puis au final leur relâchement. Les modèles sont encore raffinés en structurant hiérarchiquement les verrous. Les services de contrôle de concurrence proposé dans les plateformes intergicielles sont conçus pour faire reposer la manipulation des verrous sur l'implémentation des composants et non de leurs clients. En revanche, la séparation du code métier et du code de verrouillage au sein des composants est un problème difficile ; ce qui limite les possibilités de réutilisation de ce code, par exemple par héritage [29]. Des approches expérimentales issues du monde de la recherche ont cependant montré qu'une certaine modularisation était possible à ce niveau [8] [9]. Les modèles de composants hiérarchiques, réflexifs et/ou génératifs offrent tous trois des possibilités pour séparer code métier et code concurrent. Mais ces travaux n'ont pas encore réellement franchit les portes des laboratoires de recherche [3][9].

Synchronisation par communication

Une autre catégorie de contrôle de concurrence passe par la communication de messages spécifiques de synchronisation. Ce modèle est particulièrement adapté à la nature répartie des applications puisque la communication ne peut de toute façon se faire que par envoi de messages. Encore une fois, ces modèles de bas niveau sont généralement manipulés dans le code fonctionnel des composants, qui en devient peu réutilisable, difficile à comprendre et déboguer. Mais des évolutions notables sont en cours également dans ce domaine, notamment par les techniques de modélisation de processus industriels qui s'inspirent des modèles de communication. Les normes émergentes du BPELWS (*Business Process Execution Language for Web Services*) [21] et BPML (*Business Process Modeling Language*) [22] introduisent, en termes métiers, des modèles de gestion de concurrence basés sur la communication à un niveau d'abstraction assez élevé. Les processus échangent des messages pour se mettre d'accord sur qui fait quoi, et dans quel ordre. Si l'objectif principal est avant tout du domaine du génie logiciel pour les systèmes d'information d'entreprise, il est clair qu'en cas de succès de ces modèles de processus métier, de gros efforts devront être fournis pour les supporter dans les plateformes intergicielles.

Transactions

Dans certains domaines spécifiques, tout particulièrement celui de la gestion des données, l'expression séparée du code concurrent et du code métier est rendue au moins partiellement possible, par exemple

dans le cadre des transactions. Chaque requête de client est représentée par une transaction qui est une suite de lectures et d'écritures de données qu'il faut réaliser en séquence. Les transactions sont délimitées par des ordres initiaux d'ouverture et des ordres finaux de validation (les célèbres *commit*). Le problème du serveur revient donc à interpréter le plus grand nombre possible de transaction dans les temps les plus brefs possibles. Le choix très sûr d'interpréter chaque transaction à tour de rôle est hautement inefficace. Le moniteur transactionnel se charge donc d'entremêler de multiples transactions de façon savamment orchestrée afin de préserver les propriétés fondamentales des transactions, à savoir les propriétés ACID :

1. *Atomicité* : si une transaction est interrompue (par exemple à cause d'une panne, une déconnexion, etc.), alors l'ensemble des opérations effectuées jusqu'au moment de l'interruption sont annulées. Tout doit se passer comme si la transaction n'avait jamais existé.
2. *Cohérence* : chaque base de données dispose de règles strictes concernant les données qui peuvent être stockées. La propriété de cohérence indique qu'une transaction ne peut pas circonvenir à ces règles.
3. *Isolation* : si pour le moniteur les transactions sont entremêlées, pour les clients tout doit se passer comme si les transactions sont totalement indépendantes les unes des autres.
4. *Durabilité* : une fois validée, les effets liés à l'interprétation d'une transaction sont permanents, on ne peut plus revenir en arrière.

Les moniteurs transactionnels représentent - dans le domaine clairement délimité mais hautement stratégique de la gestion des données - une des rares intégrations réussies, et hautement efficaces, de la gestion « métier » des serveurs de données, de leur accès en concurrence par de nombreux clients et même de la mise en œuvre politiques de résistance aux pannes. Il s'agit d'un élément hautement stratégique pour les infrastructures d'entreprise et le pilier de certaines approches basées sur les composants, notamment les Enterprise Java Beans et Corba CCM.

Sécurité

Dans les systèmes répartis ouverts, comme Internet, la sécurisation des applications est un problème incontournable. Les plateformes intergicielles proposent, sous la forme d'un modèle de sécurité implanté par des services du tiers système, une couche d'abstration permettant de résoudre les problèmes de sécurisation au niveau des composants. En premier lieu, les services de sécurité mettent généralement en place une politique de contrôle d'accès aux composants. Il s'agit schématiquement de décider de qui (ou quoi), par exemple un composant externe, peut accéder à tel ou tel composant dans le système. Le contrôle d'accès peut s'opérer au niveau :

1. des interfaces de composants
2. des implémentations

Le contrôle au niveau des interfaces suffit en général à implémenter le contrôle statique d'accès, qui ne change pas au cours du temps. En revanche, les aspects dynamiques (comme le changement de niveau de protection d'une ressource) reposent très souvent sur le code d'implémentation, qui doit donc également fournir certaines informations en terme d'accès.

Dans les approches orientées serveur, la session d'accès au système, qui correspond en gros à la réalisation d'un processus métier donné (par exemple l'achat d'un produit dans un service de vente en ligne) représente le niveau de granularité privilégié pour traiter des problèmes de sécurité. Pour reprendre une terminologie usuelle dans les services en ligne, une session d'accès peut être authentifiée ou invitée. Une session *authentifiée* permet d'accéder à des ressources protégées, non-accessibles par les sessions *invitées*. À chaque session authentifiée est associé un certain nombre d'information permettant de décider de l'ensemble des ressources accessibles pour cette session particulière. Cette décision repose à la fois sur la description par la session de ses capacités d'accès (et donc de façon transitive des capacités de l'utilisateur du système), et également sur la description précise des propriétés d'accès pour les ressources protégées manipulées par le système. Pour la plupart des approches à base de composants modernes, comme les EJB, CCM ou .NET, la politique de contrôle d'accès peut être largement décrite indépendamment du code fonctionnel des composants. Dans les architectures à conteneur, le principe d'inversion mis en place permet d'implémenter les politiques de contrôle d'accès en amont des composants métier.

La cohérence de la politique d'accès mise en place repose sur deux principes :

1. La viabilité du modèle de sécurité et la robustesse de son implémentation
2. L'application correcte de la politique d'accès

Le deuxième point fait partie du cahier des charges du développeur d'applications. En général, des experts en sécurité sont nécessaires pour résoudre ce problème que l'on peut dès lors considérer comme essentiellement métier. Le premier point est lui du domaine de la logique système et est donc essentiellement du ressort de la plateforme sous-jacente et de son implémenteur. Les plateformes modernes sont généralement considérées comme « assez robustes » mais non infaillibles (ce que la licence d'utilisation du logiciel explicite clairement en général). L'expérience montre que seuls le déploiement à large-échelle et le test en grandeur nature - en production - des plateformes ainsi que l'occurrence (inévitable) d'événements malencontreux suivie de l'analyse précise de la faille exploitée conduisent à des niveaux de robustesse vraiment acceptables. Cette phase de « solidification » de la plateforme prend en général plusieurs années.

Résistance aux pannes

Malgré les nombreux efforts dans ce sens, les services de résistance aux pannes réellement exploitables ne sont pas encore légions dans le domaine des composants répartis. Ceci peut en partie s'expliquer par la grande complexité des problèmes posés par les pannes partielles des systèmes répartis. Les services de résistance aux pannes en cours de développement, par exemple dans le cadre du serveur d'entreprise K2 [23], se concentrent sur deux types de pannes de granularité assez importante :

1. Les pannes de processus serveur : occasionnées par exemple par des bogues dans le logiciel, ou encore résultant d'une attaque ou d'une manipulation malencontreuse.
2. Les ruptures de connexion entre clients et serveurs.
3. Les pannes matérielles totales : qui occasionnent l'arrêt de tous les processus serveurs tournant sur un ordinateur spécifique

Dans le cadre des applications de type 3-tiers « classiques », les serveurs de données couplés aux moniteurs transactionnels apportent une aide précieuse pour la fiabilisation des systèmes. En effet, dans le domaine de la gestion des données, des technologies avancées offrant un haut niveau de résistance aux pannes sont disponibles. Les serveurs de données peuvent être ainsi répliqués sur plusieurs machines physiques. Si l'une de ces machines tombe en panne, l'exécution peut continuer sur l'une des répliques (moyennant de complexes protocoles de synchronisation et de *rollback* ou retour en arrière). La journalisation permet également de rejouer des transactions anciennes, précédemment validées mais dont les effets ont été perdus lors de la panne. Les spécifications CORBA supportent la résistance aux pannes dans les objets répartis (et par voie de transitivité, dans les composants CCM) depuis la version 2.5. L'idée est de mettre en œuvre une réplication par objet offrant une propriété de cohérence forte. Il s'agit de garantir que l'objet serveur et toutes ses répliques, au sein de ce que l'on appelle un *groupe d'objets*, proposent au même « moment » exactement le même état. Plus précisément, c'est ce qu'il s'agit de faire « croire » au client puisque les délais de communication ne permette pas d'obtenir ce cas idéal en pratique. Malgré l'intérêt évident de telles approches, il est important de comprendre que la synchronisation forte nécessaire pour maintenir les objets serveurs et leur répliques à jour est très coûteuse ; elle n'est envisageable que dans le cadre de systèmes de taille réduite, essentiellement en réseau local.

Dans le monde du P2P, la réplication est un phénomène naturel. Lorsqu'un service est réclamé sur une infrastructure P2P, on ne se préoccupe pas vraiment de qui répond, le choix du fournisseur étant dépendant du protocole de routage non-déterministe. Ce découplage est très intéressant en matière de résistance aux pannes dans le cadre d'applications à large échelle. Il s'agit d'un domaine de recherche très actif mais dont les applications concrètes restent encore peu nombreuses.

Autres aspects

D'autres aspects de contrôle, qui peuvent prendre une importance capitale dans certains domaines métiers, comme le temps réel ou la mobilité, sont également à l'étude par la communauté scientifique. Certains travaux prototypes sont d'ores et déjà proposés mais le passage au monde industriel « à large-échelle » reste ici également à effectuer.

Le déploiement logiciel

Le déploiement applicatif concerne le passage du logiciel de sa forme descriptive (code source/objet, ressources statiques, etc.) à sa forme opérationnelle (exécution du code, communications, etc.). Dans le cadre réparti, cette phase de déploiement est autrement plus complexe que dans le cadre centralisé. Il s'agit au minimum de :

1. Dresser la carte des sites d'exécution possibles.
2. Installer les implémentations sur les sites choisis.
3. Créer les instances de composants.
4. Configurer les composants et interconnexions entre composants.
5. Arrêter et détruire les composants et les ressources associées.

Pour installer les implémentations des composants, une représentation binaire standard doit être définie ; il s'agit des *paquetages logiciels*. Les paquetages contiennent le code applicatif, les ressources statiques nécessaires au fonctionnement de ce code (ex. : des images, du texte, etc.) ainsi que des descripteurs de déploiement qui décrivent en quelque sorte le contenu et le mode d'emploi du paquetage. Le descripteur de déploiement est également l'outil privilégié pour décrire les politiques de sécurité associées au bon usage du ou des composants encapsulés dans le paquetage.

Le déploiement du logiciel est également concerné par le retrait de composants, ou le remplacement de composants par d'autres composants, ou encore par la modification des interconnexions entre composants. Ces aspects souvent considérés comme mineurs sont en fait aussi important que la phase initiale de déploiement.

Gestion du cycle de vie

La terminologie objet sépare clairement le pendant conceptuel des objets - leur classe - et leurs représentants opérationnels - les instances. Les modèles de composants, en revanche, parlent de composants à la fois en termes conceptuels et opérationnels. Mais il est tout de même important de faire une distinction entre la description des composants, on parle parfois de définition ou de type de composant, et leur opérationnalisation sous forme d'*instances de composants*. Le cycle de vie des composants concerne la gestion opérationnelle des instances de composants. Dans les plateformes intergicielles, cette gestion du cycle de vie est complexifiée en raison de la nature répartie du déploiement. Le cycle de vie d'un composant peut-être plus ou moins long. On distingue souvent au moins deux catégories principales à ce niveau :

1. Les *composants de session* qui possèdent une durée de vie limitée.
2. Les *composants entité* dont la durée de vie n'est pas précisément délimitée.

Cette terminologie est avant tout spécifique aux architectures 3-tiers ; les composants de session y implémentant les processus métiers dans une session client et les entités servant d'entrées persistantes dans les bases de données. D'autres types de composants sont à l'étude comme les composants système, les composants orientés messages, etc. Il est cependant possible de résumer les phases « essentielles » du cycle de vie des composants logiciels répartis, quels que soient leur type, selon le schéma de la figure 10.

Figure 10 : Gestion du cycle de vie des composants répartis

Erreur ! Liaison incorrecte.

L'*instanciation* des composants correspond à la création d'instances de composants à partir de leur description sous forme de paquetage. Les instances se divisent généralement en deux partie, l'instance proprement dite, à vocation métier, et son *contexte* système. Le principe d'inversion du contrôle discuté précédemment explique cette subdivision. Après création des instances et contextes de composants, ces derniers doivent être configurés (cf. section suivante). La résolution de l'identité du composant permet de le rendre disponible depuis l'extérieur. Il peut s'agit de la mise en œuvre de services simples de nommage, ou de services plus évolués de répertoires partagés ou de certificats, si l'on désire protéger l'accès au composant. Le composant devient alors disponible et est activé lorsque l'on interagit avec lui. Les composants sessions sont en général actifs pendant toute leur durée de vie mais des variations du

modèle sont apparues récemment, comme par exemple les message Beans dans les EJB, qui eux ne sont actifs que lorsqu'ils reçoivent un message, et passifs le reste du temps. Les composants entités alternent également du mode passif au mode actif en raison de leur nature persistante. De nombreuses plateformes raffinent ce schéma minimal en ajoutant de nouveaux états ou de nouvelles transitions entre les états. Par exemple, les spécifications *OSGi Service Platform* [24], spécialisées dans les applications réseau-centriques, prévoient le fonctionnement en mode déconnecté des composants. Un cycle est donc possible au niveau de la résolution du composant, ce dernier pouvant être à un moment accessible depuis l'extérieur et à un autre moment déconnecté du réseau.

Configuration et assemblage

Dans la section précédente, la transition du cycle de vie qui fait passer un composant déployé d'un mode *créé* à un mode *configuré* occupe une place importante. La configuration d'un composant concerne la modification des paramètres - généralement nommés *propriétés* - qui en régissent le fonctionnement. Certaines propriétés sont statiques, décidées au moment de la définition même du composant, ou plutôt au moment de sa mise sous forme de paquetage logiciel. Au-delà de cette *configuration statique*, les composants possèdent en général des propriétés que l'on peut configurer, voir modifier, au moment de l'exécution. Ce concept de *configuration dynamique* de propriété, popularisé par le modèle assez avant-gardiste des JavaBeans, offre un surcroît d'ouverture aux composants répartis. On peut classer les mécanismes de configuration dynamique en deux catégories :

1. Mécanismes d'introspection pour décrire quels sont les propriétés configurables
2. Mécanismes de modification des valeurs de propriétés

Le deuxième point passe généralement par certaines conventions au niveau des interfaces de méthodes de composant, comme par exemple l'utilisation standardisée de préfixes `get` et `set` suivis des noms de propriétés configurables. Pour des propriétés configurables par des données simples comme des entiers ou des chaînes de caractères, ces conventions de nommage sont suffisantes mais des mécanismes plus évolués d'introspection sont nécessaire dans le cadre général. Ces mécanismes reposent généralement sur des descriptions de haut-niveau, sous la forme de méta-données, associées aux descripteurs de déploiement.

Au-delà de la configuration isolée des composants, les plateformes évoluent à l'heure actuelle vers des mécanismes plus généraux de configuration d'*assemblages de composants*. Il s'agit ici de configurer les interconnexions entre les composants. Encore une fois, cette configuration repose à la fois sur des principes statiques, comme par exemple la mise en relation d'interfaces requises avec les interfaces fournies correspondantes, et également, et de plus en plus, sur des principes dynamiques. La configuration dynamique des interconnexions de composants prend généralement de l'importance dans le cadre des modèles de communication découplés dans l'espace comme par exemple les modèles de publication/souscription événementiels. Ainsi, dans les composants CCM, la mise en correspondance entre sources et puits d'événements est essentiellement réalisée au moment de l'exécution des composants. Les architectures qui considèrent les assemblages de composants comme des composants (composites) à part entière, comme notamment les spécifications Fractal, disposent semble-t-il d'un avantage sur les modèles plus « plats » en matière de configuration dynamique.

Vers l'adaptation logicielle

Dans ce qui précède, nous avons décrits les principes de déploiement « classiques » des applications réparties, reposant sur un enchaînement purement séquentiel des activités du déploiement. Ce cycle, conduisant du déploiement du code jusqu'à la configuration de l'application avant son démarrage, est largement remis en question dans le cadre de l'adaptation logicielle [25]. L'idée est de pouvoir intervenir sur le fonctionnement du système après la première phase de déploiement. Ainsi, l'on veut pouvoir enrichir le système en déployant de nouveaux composants, sans pour autant interrompre le fonctionnement actuel du système. Le remplacement « à chaud » d'un composant par un autre est également une opération intéressante. Par exemple, cela permet de remplacer un composant défectueux par une version corrigée. La reconfiguration « au vol » du système permet de modifier les paramètres de fonctionnement du système. Enfin, l'adaptation logicielle s'intéresse à la possibilité de modifier dynamiquement les interconnexions entre composants. Si les applications pratiques de ces principes, par ailleurs largement discutés dans le monde de la recherche scientifique, sont encore peu nombreuses, les

grands acteurs des composants répartis s'y intéressent de plus en plus. Nous pouvons par exemple signaler le projet *autonomic computing* d'IBM [26]. Les approches basées sur la composition dynamique et hiérarchique semblent particulièrement attractives pour supporter certaines fonctionnalités adaptatives comme le réassemblage dynamique des applications. Mais les expérimentations « en grandeur nature » de ces principes adaptatifs semblent encore rares, voire inexistants.

Conclusion

Les architectures de composants logiciels ont pour ambition de devenir la clé de voûte technologique des applications réparties à large échelle de demain. Pour cela, nous l'avons vu dans ce chapitre, de nombreuses questions encore en suspend aujourd'hui devront être discutées. Les modèles actuels de gestion de concurrence et de fiabilisation sont encore assez pauvres, voir pratiquement inexistants, dans les plateformes industrielles. Ces plateformes, notamment les composants EJB de SUN, les composants CCM de CORBA et les technologies .NET de Microsoft se concentrent sur la problématique spécifique des serveurs d'entreprise et des architectures N-tiers, à l'heure actuelle la « killer application » des technologies componentielles. Mais les domaines émergents de l'Internet, notamment les technologies P2P ou centrées réseaux montrent que le concept de « killer application » est assez changeant. Avec l'introduction ininterrompue de nouveautés, comme par exemple les Message Beans pour Sun ou le développement de SOAP chez Microsoft, il est cependant clair que ces technologies couvrent des domaines applicatifs de plus en plus vastes. Cette évolution constante se veut plus maîtrisée, plus concentrée sur le long terme, à l'OMG. Les composants CCM intègrent par exemple à l'origine beaucoup plus de fonctionnalités que leurs équivalents chez Sun et Microsoft. Ils s'articulent de plus dans une infrastructure très ambitieuse et de très grande envergure, couvrant l'ensemble des besoins du développement informatique, des aspects métiers des architectures orientées modèles jusqu'aux technologies CORBA de bas niveau. L'expérience des intergiciels objets, qui est loin d'être un échec mais dont le déploiement s'est fait en moindre nombre que prévu, montre cependant que la vision à long terme est un exercice délicat en informatique industrielle.

Bibliographie

[1] George Coulouris et al., Distributed Systems Concepts and Design, Prentice-Hall, 2001.

[2] IEEE Distributed Systems Online, http://dsonline.computer.org

[3] Jean-Pierre Briot, Rachid Guerraoui et Klaus-peter Löhr, Concurrency and Distribution in Object-Oriented Programming, ACM Computing Surveys, 1998

[4] Andy Oram et al. Peer-to-peer, Harnessing the Power of Disruptive Technologies, O'Reilly & associates, 2001.

[5] Fran Berman et al. Grid Computing: Making the Global Infrastructure a Reality, Wiley, 2003.

[6] Sun Microsystems, Jini Network Technology, http://wwws.sun.com/software/jini

[7] Consortium ObjectWeb, The Fractal Project, http://fractal.objectweb.org

[8] Consortium ObjectWeb, ProActive, http://www-sop.inria.fr/oasis/ProActive

[9] Gordon S. Blair et al. The Design and Implementation of Open ORB 2, IEEE DS Online, Vol. 2, N°6, 2001

[10] JBoss Inc., JBoss Application Server, http://www.jboss.org/product/jbossas

[11] Spring, Java/J2EE Application Framework, http://www.springframework.org

[12] Guruduth Banavar, et al. A Case for Message Oriented Middleware. Lecture Notes in Computer Science 1693, 1999

[13] Microsoft, Microsoft Message Queuing (MSMQ), http://www.microsoft.com/msmq

[14] IBM, WebSphere MQ, http://www-306.ibm.com/software/integration/wmq

[15] Sun Microsystems, Java Messaging Service (JMS), http://java.sun.com/products/jms

[16] ScalAgent Distributed Technologies, Mediation Suite, http://www.scalagent.com

[17] Patrick Eugster et al. The Many Faces of Publish/Subscribe, ACM Computing Surveys, Vol. 35 N°2, Juin 2003

[18] David Garlan et Mary Shaw, Software Architecture: Perspectives on an Emerging Discipline, Prentice-Hall, 1996

[19] Sun Microsystems, J2EE Connector Architecture, http://java.sun.com/j2ee/connector

[20] Object Management Group, CORBA Concurrency Service 1.0, http://www.omg.org/technology/documents/formal/concurrency_service.htm

[21] IBM et al. Business Process Execution Language for Web Services (BPELWS) 1.1, http://www-106.ibm.com/developerworks/library/ws-bpel

[22] Business Process Management Initiative, Business Process Modelling Language (BPML) 1.0, http://www.bpmi.org/specifications.esp

[23] ICMG, K2 Component Server, http://www.icmgworld.com/corp/k2/k2.overview.asp

[24] OSGi Alliance, OSGi Service Platform Release 3, http://www.osgi.org

[25] Michel Riveill editeur, Systèmes à composants adaptables et extensibles, Technique et Science Informatique, Hermes Science, Vol. 23 N°2, 2004

[26] IBM, Projet Autonomic Computing, http://www.research.ibm.com/autonomic

[27] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design Patterns, Addison-Wesley, 1995

[28] Holger Wunderlich, Building Multi-Tier Scenarios for WebSphere Enterprise Applications, IBM Redbooks, 2004

[29] Giuseppe Milicia et Vladimiro Sassone, The inheritance anomaly: ten years after, Symposium on Applied Computing, ACM Press, 2004.