Chapitre 1

Introduction à l'intergiciel

Ce chapitre présente les fonctions de l'intergiciel, les besoins auxquels elles répondent, et les principales classes d'intergiciel. L'analyse d'un exemple simple, celui de l'appel de procédure à distance, permet d'introduire les notions de base relatives aux systèmes intergiciels et les problèmes que pose leur conception. Le chapitre se termine par une note historique résumant les grandes étapes de l'évolution de l'intergiciel.

1.1 Pourquoi l'intergiciel ?

Faire passer la production de logiciel à un stade industriel grâce au développement de composants réutilisables a été un des premiers objectifs identifiés du génie logiciel. Transformer l'accès à l'information et aux ressources informatiques en un service omniprésent, sur le modèle de l'accès à l'énergie électrique ou aux télécommunications, a été un rêve des créateurs de l'Internet. Bien que des progrès significatifs aient été enregistrés, ces objectifs font encore partie aujourd'hui des défis pour le long terme.

Dans leurs efforts pour relever ces défis, les concepteurs et les réalisateurs d'applications informatiques rencontrent des problèmes plus concrets dans leur pratique quotidienne. Les brèves études de cas qui suivent illustrent quelques situations typiques, et mettent en évidence les principaux problèmes et les solutions proposées.

Exemple 1 : réutiliser le logiciel patrimonial. Les entreprises et les organisations construisent maintenant des systèmes d'information globaux intégrant des applications jusqu'ici indépendantes, ainsi que des développements nouveaux. Ce processus d'intégration doit tenir compte des applications dites *patrimoniales* (en anglais : *legacy*), qui ont été développées avant l'avénement des standards ouverts actuels, utilisent des outils « propriétaires », et nécessitent des environnements spécifiques.

Une application patrimoniale ne peut être utilisée qu'à travers une interface spécifiée, et ne peut être modifiée. La plupart du temps, l'application doit être reprise telle quelle car le coût de sa réécriture serait prohibitif.

Le principe des solutions actuelles est d'adopter une norme commune, non liée à un langage particulier, pour interconnecter différentes applications. La norme spécifie des interfaces et des protocoles d'échange pour la communication entre applications. Les protocoles sont réalisés par une couche logicielle qui fonctionne comme un bus d'échanges entre applications, également appelé courtier (en anglais *broker*). Pour intégrer une application patrimoniale, il faut développer une *enveloppe* (en anglais *wrapper*), c'est-à-dire une couche logicielle qui fait le pont entre l'interface originelle de l'application et une nouvelle interface conforme à la norme choisie.

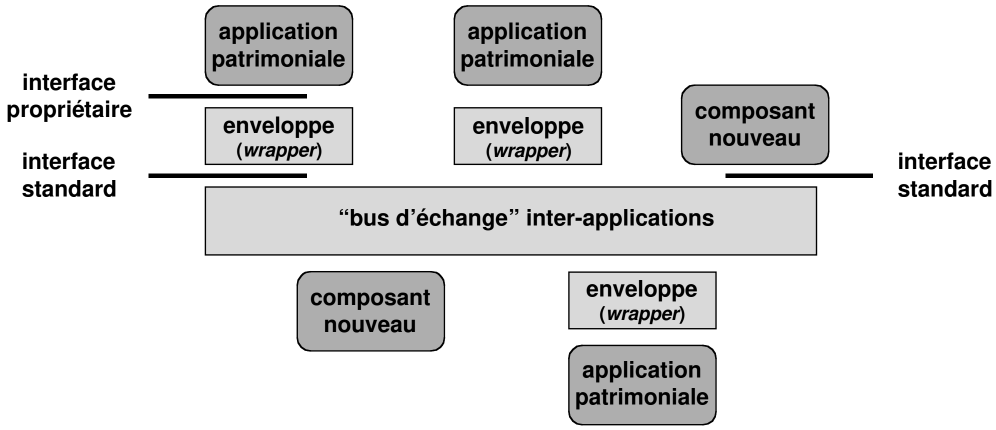

Figure 1.1 – Intégration d'applications patrimoniales

Une application patrimoniale ainsi « enveloppée » peut maintenant être intégrée avec d'autres applications du même type et avec des composants nouveaux, en utilisant les protocoles normalisés du courtier. Des exemples de courtiers sont CORBA, les files de messages, les systèmes à publication et abonnement (en anglais *publish-subscribe*). On en trouvera des exemples dans ce livre.

Exemple 2 : systèmes de médiation. Un nombre croissant de systèmes prend la forme d'une collection d'équipements divers (capteurs ou effecteurs) connectés entre eux par un réseau. Chacun de ces dispositifs remplit une fonction locale d'interaction avec le monde extérieur, et interagit à distance avec d'autres capteurs ou effecteurs. Des applications construites sur ce modèle se rencontrent dans divers domaines : réseaux d'ordinateurs, systèmes de télécommunications, équipements d'alimentation électrique permanente, systèmes décentralisés de production.

La gestion de tels systèmes comporte des tâches telles que la mesure continue des performances, l'élaboration de statistiques d'utilisation, l'enregistrement d'un journal, le traitement des signaux d'alarme, la collecte d'informations pour la facturation, la maintenance à distance, le téléchargement et l'installation de nouveaux services. L'exécution de ces tâches nécessite l'accès à distance au matériel, la collecte et l'agrégation de données, et la réaction à des événements critiques. Les systèmes réalisant ces tâches s'appellent *systèmes de médiation*[1].

[1]Ce terme a en fait un sens plus général dans le domaine de la gestion de données ; nous l'utilisons ici

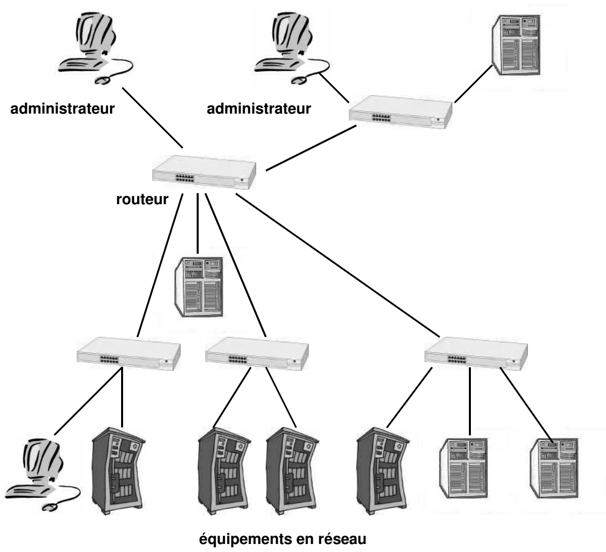

Figure 1.2 – Surveillance et commande d'équipements en réseau

L'infrastructure interne de communication d'un système de médiation doit réaliser la collecte de données et leur acheminement depuis ou vers les capteurs et effecteurs. La communication est souvent déclenchée par un événement externe, comme la détection d'un signal d'alarme ou le franchissement d'un seuil critique par une grandeur observée.

Un système de communication adapté à ces exigences est un *bus à messages*, c'est-à-dire un canal commun auquel sont raccordées les diverses entités (Figure 1.2). La communication est asynchrone et peut mettre en jeu un nombre variable de participants. Les destinataires d'un message peuvent être les membres d'un groupe prédéfini, ou les « abonnés » à un sujet spécifié.

Exemple 3 : architectures à composants. Le développement d'applications par composition de « briques »logicielles s'est révélé une tâche beaucoup plus ardue qu'on ne l'imaginait initialement [McIlroy 1968]. Les architectures actuelles à base de composants logiciels reposent sur la séparation des fonctions et sur des interfaces standard bien définies. Une organisation répandue pour les applications d'entreprise est l'architecture à trois étages (3-*tier*[2]), dans laquelle une application est composée de trois couches : entre l'étage de présentation, responsable de l'interface avec le client, et l'étage de gestion de bases de données, responsable de l'accès aux données permanentes, se trouve un étage de traitement (*business logic*) qui réalise les fonctions spécifiques de l'application. Dans cet étage intermédiaire, les aspects propres à l'application sont organisés comme un ensemble de « composants », unités de construction pouvant être déployées indépendamment.

Cette architecture repose sur une infrastructure fournissant un environnement de déploiement et d'exécution pour les composants ainsi qu'un ensemble de services communs tels que la gestion de transactions et la sécurité. En outre, une application pour un domaine

dans le sens restreint indiqué

[2] La traduction correcte de *tier* est « étage » ou « niveau ».

particulier (par exemple télécommunications, finance, avionique, etc.) peut utiliser une bibliothèque de composants développés pour ce domaine.

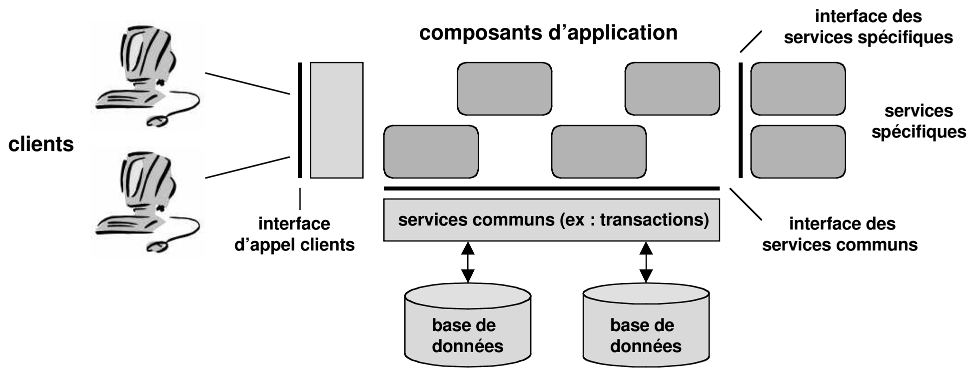

Figure 1.3 – Un environnement pour applications à base de composants

Cette organisation a les avantages suivants :

– elle permet aux équipes de développement de se concentrer sur les problèmes propres à l'application, les aspects génériques et l'intendance étant pris en charge par les services communs ;
– elle améliore la portabilité et l'interopérabilité en définissant des interfaces standard ; ainsi on peut réutiliser un composant sur tout système fournissant les interfaces appropriées, et on peut intégrer du code patrimonial dans des « enveloppes » qui exportent ces interfaces ;
– elle facilite le passage à grande échelle, en séparant la couche d'application de la couche de gestion de bases de données ; les capacités de traitement de ces deux couches peuvent être séparément augmentées pour faire face à un accroissement de la charge.

Des exemples de spécifications pour de tels environnements sont les *Enterprise JavaBeans* (EJB) et la plate-forme .NET, examinés dans la suite de ce livre.

Exemple 4 : adaptation de clients par des mandataires. Les utilisateurs accèdent aux applications sur l'Internet via des équipements dont les caractéristiques et les performances couvrent un spectre de plus en plus large. Entre un PC de haut de gamme, un téléphone portable et un assistant personnel, les écarts de bande passante, de capacités locales de traitement, de capacités de visualisation, sont très importants. On ne peut pas attendre d'un serveur qu'il adapte les services qu'il fournit aux capacités locales de chaque point d'accès. On ne peut pas non plus imposer un format uniforme aux clients, car cela reviendrait à les aligner sur le niveau de service le plus bas (texte seul, écran noir et blanc, etc.).

La solution préférée est d'interposer une couche d'adaptation, appelée *mandataire* (en anglais *proxy*) entre les clients et les serveurs. Un mandataire différent peut être réalisé pour chaque classe de point d'accès côté client (téléphone, assistant, etc.). La fonction du mandataire est d'adapter les caractéristiques du flot de communication depuis ou vers le client aux capacités de son point d'accès et aux conditions courantes du réseau. À cet effet, le mandataire utilise ses propres ressources de stockage et de traitement. Les mandataires peuvent être hébergés sur des équipements dédiés (Figure 1.4), ou sur des serveurs communs.

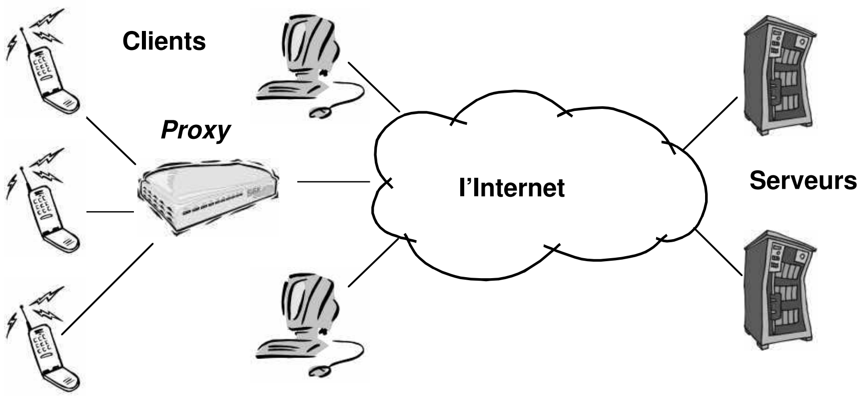

Figure 1.4 – Adaptation des communications aux ressources des clients par des mandataires

Des exemples d'adaptation sont la compression des données pour réagir à des variations de bande passante du réseau, la réduction de la qualité des images pour prendre en compte des capacités réduites d'affichage, le passage de la couleur aux niveaux de gris. Un exemple de mise en œuvre de l'adaptation par mandataires est décrit dans [Fox et al. 1998].

Les quatre études de cas qui précèdent ont une caractéristique commune : dans chacun des systèmes présentés, des applications utilisent des logiciels de niveau intermédiaire, installés au-dessus des systèmes d'exploitation et des protocoles de communication, qui réalisent les fonctions suivantes :

1. cacher la *répartition*, c'est-à-dire le fait qu'une application est constituée de parties interconnectées s'exécutant à des emplacements géographiquement répartis ;
2. cacher l'*hétérogeneité* des composants matériels, des systèmes d'exploitation et des protocoles de communication utilisés par les différentes parties d'une application ;
3. fournir des *interfaces* uniformes, normalisées, et de haut niveau aux équipes de développement et d'intégration, pour faciliter la construction, la réutilisation, le portage et l'interopérabilité des applications ;
4. fournir un ensemble de *services* communs réalisant des fonctions d'intérêt général, pour éviter la duplication des efforts et faciliter la coopération entre applications.

Cette couche intermédiaire de logiciel, schématisée sur la Figure 1.5, est désignée par le terme générique d'*intergiciel* (en anglais *middleware*). Un intergiciel peut être à usage général ou dédié à une classe particulière d'applications.

L'utilisation de l'intergiciel présente plusieurs avantages dont la plupart résultent de la capacité d'abstraction qu'il procure : cacher les détails des mécanismes de bas niveau, assurer l'indépendance vis-à-vis des langages et des plates-formes, permettre de réutiliser l'expérience et parfois le code, faciliter l'évolution des applications. En conséquence, on

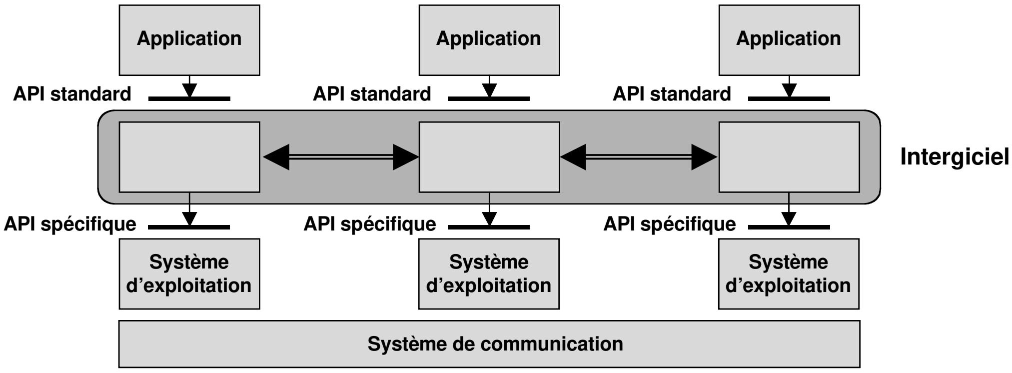

Figure 1.5 – Organisation de l'intergiciel

peut espérer réduire le coût et la durée de développement des applications (l'effort étant concentré sur les problèmes spécifiques et non sur l'intendance), et améliorer leur portabilité et leur interopérabilité.

Un inconvénient potentiel est la perte de performances liée à la traversée de couches supplémentaires de logiciel. L'utilisation de techniques intergicielles implique par ailleurs de prévoir la formation des équipes de développement.

1.2 Quelques classes d'intergiciel

Les systèmes intergiciels peuvent être classés selon différents critères, prenant en compte les propriétés de l'infrastructure de communication, l'architecture d'ensemble des applications, les interfaces fournies.

Caractéristiques de la communication. L'infrastructure de communication sous-jacente à un intergiciel est caractérisée par plusieurs propriétés qui permettent une première classification.

1. *Topologie fixe ou variable.* Dans un système de communication fixe, les entités communicantes résident à des emplacements fixes, et la configuration du réseau ne change pas (ou bien ces changements sont des opérations peu fréquentes, prévues à l'avance). Dans un système de communication mobile, tout ou partie des entités communicantes peuvent changer de place, et se connecter ou se déconnecter dynamiquement, y compris pendant le déroulement des applications.
2. *Caractéristiques prévisibles ou imprévisibles.* Dans certains systèmes de communication, on peut assurer des bornes pour des facteurs de performance tels que la gigue ou la latence. Néanmoins, dans beaucoup de situations pratiques, ces bornes ne peuvent être garanties, car les facteurs de performance dépendent de la charge de dispositifs partagés tels que les routeurs ou les voies de transmission. Un système de communication est dit *synchrone* si on peut garantir une borne supérieure pour le temps

de transmission d'un message ; si cette borne ne peut être établie, le système est dit *asynchrone*[3].

Ces caractéristiques se combinent comme suit.

– Fixe, imprévisible. C'est le cas le plus courant, aussi bien pour les réseaux locaux que pour les réseaux à grande distance (comme l'Internet). Bien que l'on puisse souvent estimer une moyenne pour la durée de transmission, il n'est pas possible de lui garantir une borne supérieure.
– Fixe, prévisible. Cette combinaison s'applique aux environnements spécialement développés pour des applications ayant des contraintes particulières, comme les applications critiques en temps réel. Le protocole de communication garantit alors une borne supérieure pour le temps de transfert, en utilisant la réservation préalable de ressources.
– Variable, imprévisible. C'est le cas de systèmes de communication qui comprennent des appareils mobiles (dits aussi nomades) tels que les téléphones mobiles ou les assistants personnels. Ces appareils utilisent la communication sans fil, qui est sujette à des variations imprévisibles de performances. Les environnements dits *ubiquitaires*, ou *omniprésents* [Weiser 1993], permettant la connexion et la déconnexion dynamique de dispositifs très variés, appartiennent à cette catégorie.

Avec les techniques actuelles de communication, la classe (variable, prévisible) est vide.

L'imprévisibilité des performances du système de communication rend difficile la tâche de garantir aux applications des niveaux de qualité spécifiés L'adaptabilité, c'est-à-dire la capacité à réagir à des variations des performances des communications, est la qualité principale requise de l'intergiciel dans de telles situations.

Architecture et interfaces. Plusieurs critères permettent de caractériser l'architecture d'ensemble d'un système intergiciel.

1. *Unités de décomposition.* Les applications gérées par les systèmes intergiciel sont habituellement décomposées en parties. Celles-ci peuvent prendre différentes formes, qui se distinguent par leur définition, leurs propriétés, et leur mode d'interaction. Des exemples, illustrés dans la suite de ce livre, en sont les objets, les composants et les agents.
2. *Rôles.* Un système intergiciel gère l'activité d'un ensemble de participants (utilisateurs ou systèmes autonomes). Ces participants peuvent avoir des rôles prédéfinis tels que *client* (demandeur de service) et *serveur* (fournisseur de service), ou *diffuseur* (émetteur d'information) et *abonné* (récepteur d'information). Les participants peuvent aussi se trouver au même niveau, chacun pouvant indifféremment assumer un rôle différent, selon les besoins ; une telle organisation est dite *pair à pair* (en anglais *peer to peer*, ou P2P).
3. *Interfaces de fourniture de service.* Les primitives de communication fournies par un système intergiciel peuvent suivre un schéma synchrone or asynchrone (ces termes sont malheureusement surchargés, et ont ici un sens différent de celui vu

[3]Dans un système réparti, le terme *asynchrone* indique généralement en outre qu'on ne peut fixer de borne supérieure au rapport des vitesses de calcul sur les différents sites (cela est une conséquence du caractère imprévisible de la charge sur les processeurs partagés).

précédemment pour le système de communication de base). Dans le schéma synchrone, un processus client envoie un message de requête à un serveur distant et se bloque en attendant la réponse. Le serveur reçoit la requête, exécute l'opération demandée, et renvoie un message de réponse au client. À la réception de la réponse, le client reprend son exécution. Dans le schéma asynchrone, l'envoi de requête n'est pas bloquant, et il peut ou non y avoir une réponse. L'appel de procédure ou de méthode à distance est un exemple de communication synchrone ; les files de messages ou les systèmes de publication-abonnement (*publish-subscribe*) sont des exemples de communication asynchrone.

Les diverses combinaisons des propriétés ci-dessus produisent des systèmes variés, qui diffèrent selon leur structure ou leurs interfaces ; des exemples en sont présentés dans la suite. En dépit de cette diversité, nous souhaitons mettre en évidence quelques principes architecturaux communs à tous ces systèmes.

1.3 Un exemple simple d'intergiciel : l'appel de procédure à distance

Nous présentons maintenant un exemple simple d'intergiciel : l'appel de procédure à distance (en anglais, *Remote Procedure Call*, ou RPC). Nous ne visons pas à couvrir tous les détails de ce mécanisme, que l'on peut trouver dans tous les manuels traitant de systèmes répartis ; l'objectif est de mettre en évidence quelques problèmes de conception des systèmes intergiciels, et quelques constructions communes, ou patrons, visant à résoudre ces problèmes.

1.3.1 Motivations et besoins

L'abstraction procédurale est un concept fondamental de la programmation. Une procédure, dans un langage impératif, peut être vue comme une « boîte noire » qui remplit une tâche spécifiée en exécutant une séquence de code encapsulée, le corps de la procédure. L'encapsulation traduit le fait que la procédure ne peut être appelée qu'à travers une interface qui spécifie ses paramètres et ses résultats sous la forme d'un ensemble de conteneurs typés (les paramètres formels). Un processus qui appelle une procédure fournit les paramètres effectifs associés aux conteneurs, exécute l'appel, et reçoit les résultats lors du retour, c'est-à-dire à la fin de l'exécution du corps de la procédure.

L'appel de procédure à distance peut être spécifié comme suit. Soit sur un site A un processus p qui appelle une procédure locale P (Figure 1.6a). Il s'agit de concevoir un mécanisme permettant à p de faire exécuter P sur un site distant B (Figure 1.6b), en préservant la forme et la sémantique de l'appel (c'est-à-dire son effet global dans l'environnement de p). A et B sont respectivement appelés site client et site serveur, car l'appel de procédure à distance suit le schéma client-serveur de communication synchrone par requête et réponse.

L'invariance de la sémantique entre appel local et distant maintient l'abstraction procédurale. Une application qui utilise le RPC est facilement portable d'un environnement à un autre, car elle est indépendante des protocoles de communication sous-jacents.

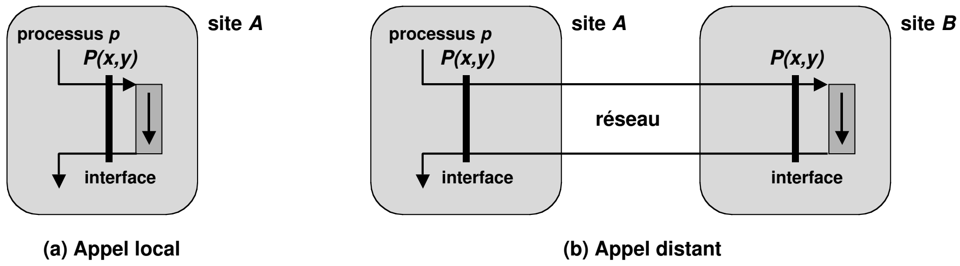

Figure 1.6 – Schéma de l'appel de procédure à distance

Elle peut également être portée aisément, sans modifications, d'un système centralisé vers un réseau.

Néanmoins, le maintien de la sémantique est une tâche délicate, pour deux raisons principales :

- les modes de défaillance sont différents dans les cas centralisé et réparti ; dans cette dernière situation, le site client, le site serveur et le réseau sont sujets à des défaillances indépendantes ;
- même en l'absence de pannes, la sémantique du passage des paramètres est différente (par exemple, on ne peut pas passer un pointeur en paramètre dans un environnement réparti car le processus appelant et la procédure appelée s'exécutent dans des espaces d'adressage différents).

Le problème du passage des paramètres est généralement résolu en utilisant le passage par valeur pour les types simples. Cette solution s'étend aux paramètres composés de taille fixe (tableaux ou structures). Dans le cas général, l'appel par référence n'est pas possible ; on peut néanmoins construire des routines spécifiques d'emballage et de déballage des paramètres pour des structures à base de pointeurs, graphes ou listes. Les aspects techniques de la transmission des paramètres sont examinés en 1.3.2.

La spécification du comportement du RPC en présence de défaillances soulève deux difficultés, lorsque le système de communication utilisé est asynchrone. D'une part, on ne connaît en général pas de borne supérieure pour le temps de transmission d'un message ; un mécanisme de détection de perte de message à base de délais de garde risque donc des fausses alarmes. D'autre part, il est difficile de différencier l'effet de la perte d'un message de celui de la panne d'un site distant. En conséquence, une action de reprise peut conduire à une décision erronée, comme de réexécuter une procédure déjà exécutée. Les aspects relatifs à la tolérance aux fautes sont examinés en 1.3.2.

1.3.2 Principes de réalisation

La réalisation standard du RPC [Birrell and Nelson 1984] repose sur deux modules logiciels (Figure 1.7), la souche, ou talon, client (*client stub*) et la souche serveur (*server stub*). La souche client agit comme un représentant local du serveur sur le site client ; la souche serveur joue un rôle symétrique. Ainsi, aussi bien le processus appelant, côté client, que la procédure appelée, côté serveur, conservent la même interface que dans le cas centralisé. Les souches client et serveur échangent des messages via le système de communication. En outre, ils utilisent un service de désignation pour aider le client à localiser le serveur (ce point est développé en 1.3.3).

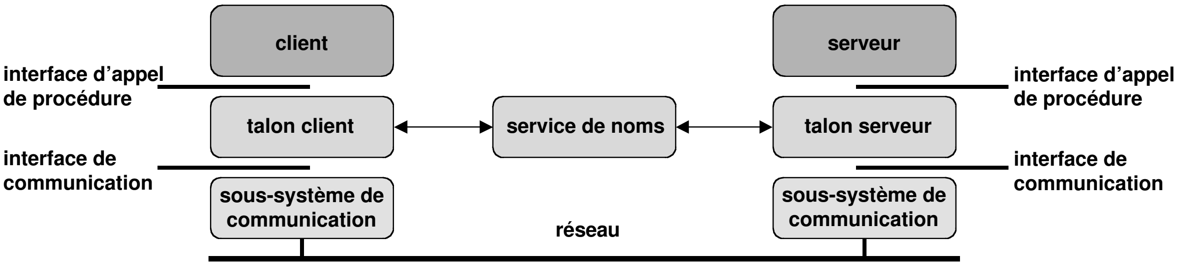

Figure 1.7 – Composants principaux de l'appel de procédure à distance

Les fonctions des souches sont résumées ci-après.

Gestion des processus et synchronisation. Du côté client, le processus[4] appelant doit être bloqué en attendant le retour de la procédure.

Du côté serveur, le problème principal est celui de la gestion du parallélisme. L'exécution de la procédure est une opération séquentielle, mais le serveur peut être utilisé par plusieurs clients. Le multiplexage des ressources du serveur (notamment si le support du serveur est un multiprocesseur ou une grappe) nécessite d'organiser l'exécution parallèle des appels. On peut utiliser pour cela des processus ordinaires ou des processus légers (*threads*), ce dernier choix (illustré sur la figure 1.8) étant le plus fréquent. Un *thread* veilleur (en anglais *daemon*) attend les requêtes sur un port spécifié. Dans la solution séquentielle (Figure 1.8a), le veilleur lui-même exécute la procédure ; il n'y a pas d'exécution parallèle sur le serveur. Dans le schéma illustré (Figure 1.8b), un nouveau *thread*) est créé pour effectuer l'appel, le veilleur revenant attendre l'appel suivant ; l'exécutant disparaît à la fin de l'appel. Pour éviter le coût de la création, une solution consiste à gérer une réserve de *threads* créés à l'avance (Figure 1.8c). Ces *threads* communiquent avec le veilleur à travers un tampon partagé, selon le schéma producteur-consommateur. Chaque *thread* attend qu'une tâche d'exécution soit proposée ; quand il a terminé, il retourne dans la réserve et se remet en attente. Un appel qui arrive alors que tous les *threads* sont occupés est retardé jusqu'à ce que l'un d'eux se libère.

Voir [Lea 1999] pour une discussion de la gestion de l'exécution côté serveur, illustrée par des *threads* Java.

Toutes ces opérations de synchronisation et de gestion des processus sont réalisées dans les souches et sont invisibles aux programmes principaux du client et du serveur.

Emballage et déballage des paramètres. Les paramètres et les résultats doivent être transmis sur le réseau. Ils doivent donc être mis sous une forme sérialisée permettant cette transmission. Pour assurer la portabilité, cette forme doit être conforme à un standard connu, et indépendante des protocoles de communication utilisés et des conventions

[4]selon l'organisation adoptée, il peut s'agir d'un processus ordinaire ou d'un processus léger (*thread*).

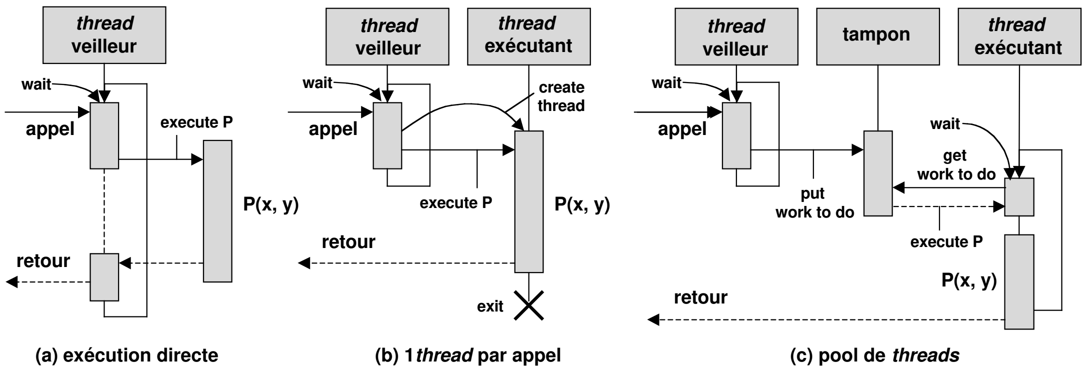

Figure 1.8 – Appel de procédure à distance : gestion de l'exécution côté serveur

locales de représentation des données sur les sites client et serveur, comme l'ordre des octets. La conversion des données depuis leur représentation locale vers une forme standard transmissible est appelée emballage (en anglais *marshalling*) ; la conversion en sens inverse est appelée déballage (en anglais *unmarshalling*).

Un emballeur (*marshaller*) est un jeu de routines, une par type de données (par exemple `writeInt`, `writeString`, etc.), qui écrivent des données du type spécifié dans un flot de données séquentiel. Un déballeur (*unmarshaller*) remplit la fonction inverse et fournit des routines (par exemple `readInt`, `readString`, etc.) qui extraient des données d'un type spécifié à partir d'un flot séquentiel. Ces routines sont appelées par les souches quand une conversion est nécessaire. L'interface et l'organisation des emballeurs et déballeurs dépend du langage utilisé, qui spécifie les types de données, et du format choisi pour la représentation standard.

Réaction aux défaillances. Comme indiqué plus haut, les défaillances (ou pannes) peuvent survenir sur le site client, sur le site serveur, ou sur le réseau. La prise en compte des défaillances nécessite de formuler des hypothèses de défaillances, de détecter les défaillances, et enfin de réagir à cette détection.

Les hypothèses habituelles sont celle de la panne franche (*fail-stop*) pour les sites (ou bien un site fonctionne correctement, ou bien il est arrêté), et celle de la perte de message pour le réseau (ou bien un message arrive sans erreur, ou bien il n'arrive pas, ce qui suppose que les erreurs de transmission sont détectées et éventuellement corrigées à un niveau plus bas du système de communication). Les mécanismes de détection de pannes reposent sur des délais de garde. À l'envoi d'un message, une horloge de garde est armée, avec une valeur estimée de la borne supérieure du délai de réception de la réponse. Le dépassement du délai de garde déclenche une action de reprise.

Les horloges de garde sont placées côté client, à l'envoi du message d'appel, et côté serveur, à l'envoi du message de réponse. Dans les deux cas, l'action de reprise consiste à renvoyer le message. Le problème vient du fait qu'il est difficile d'estimer les délais de garde : un message d'appel peut être renvoyé alors que l'appel a déjà eu lieu, et la procédure peut ainsi être exécutée plusieurs fois.

Le resultat net est qu'il est généralement impossible de garantir la sémantique d'appel dite « exactement une fois » (après réparation de toutes les pannes, l'appel a été exécuté une fois et une seule). La plupart des systèmes assurent la sémantique « au plus une fois » (l'appel a été exécuté une fois ou pas du tout, ce qui exclut les cas d'exécution partielle ou multiple). La sémantique « au moins une fois », qui autorise les exécutions multiples, est acceptable si l'appel est idempotent, c'est-à-dire si l'effet de deux appels successifs est identique à celui d'un seul appel.

L'organisation d'ensemble du RPC, en dehors de la gestion des pannes, est schématisée sur la figure 1.9.

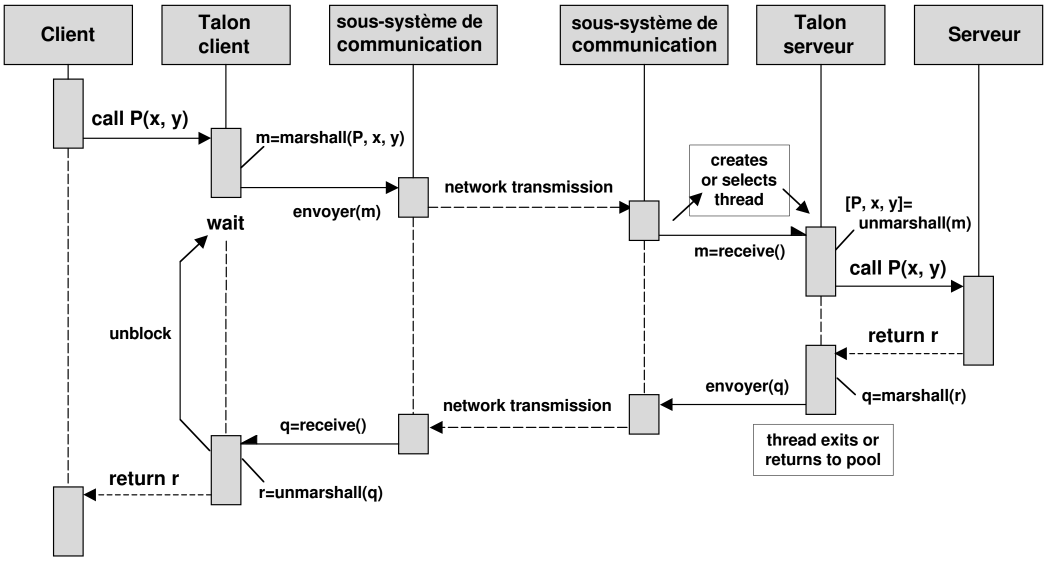

Figure 1.9 – Flot d'exécution dans un appel de procédure à distance

1.3.3 Développement d'applications avec le RPC

Pour utiliser le RPC dans des applications, il faut régler plusieurs problèmes pratiques : liaison entre client et serveur, création des souches, déploiement et démarrage de l'application. Ces aspects sont examinés ci-après.

Liaison client-serveur. Le client doit connaître l'adresse du serveur et le numéro de port de destination de l'appel. Ces informations peuvent être connues à l'avance et inscrites « en dur » dans le programme. Néanmoins, pour assurer l'indépendance logique entre client et serveur, permettre une gestion souple des ressources, et augmenter la disponibilité, il est préférable de permettre une liaison tardive du client au serveur. Le client doit localiser le serveur au plus tard au moment de l'appel.

Cette fonction est assurée par le service de désignation, qui fonctionne comme un annuaire associant les noms (et éventuellement les numéros de version) des procédures avec les adresses et numéros de port des serveurs correspondants. Un serveur enregistre un nom de procédure, associé à son adresse IP et au numéro de port auquel le veilleur attend les appels. Un client (plus précisément, la souche client) consulte l'annuaire pour obtenir ces informations de localisation. Le service de désignation est généralement réalisé par un serveur spécialisé, dont l'adresse et le numéro de port sont connus de tous les sites participant à l'application[5].

La figure 1.10 montre le schéma général d'utilisation du service de désignation.

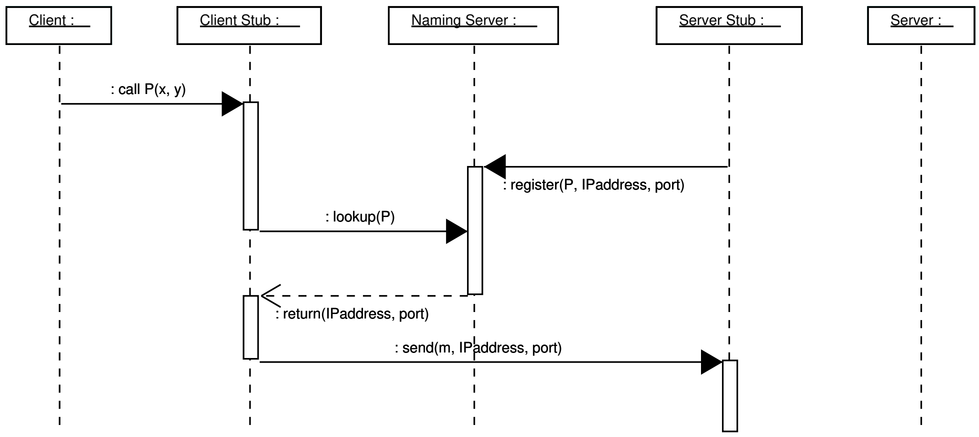

Figure 1.10 – Localisation du serveur dans l'appel de procédure à distance.

La liaison inverse (du serveur vers le client) est établie en incluant l'adresse IP et le numéro de port du client dans le message d'appel.

Génération des souches. Comme nous l'avons vu en 1.3.2, les souches remplissent des fonctions bien définies, dont une partie est générique (comme la gestion des processus) et une autre est propre à chaque appel (comme l'emballage et le déballage des paramètres). Ce schéma permet la génération automatique des souches.

Les paramètres d'un appel nécessaires pour la génération de souches sont spécifiés dans une notation appelée langage de définition d'interface (en anglais, *Interface Definition Language*, ou IDL). Une description d'interface écrite en IDL contient toute l'information qui définit l'interface pour un appel, et fonctionne comme un contrat entre l'appelant et l'appelé. Pour chaque paramètre, la description spécifie son type et son mode de passage (valeur, copie-restauration, etc.). Des informations supplémentaires telles que le numéro de version et le mode d'activation du serveur peuvent être spécifiées.

Plusieurs IDL ont été définis (par exemple Sun XDR, OSF DCE). Le générateur de souches utilise un format commun de représentation de données défini par l'IDL ; il insère les routines de conversion fournies par les emballeurs et déballeurs correspondants. Le cycle complet de développement d'une application utilisant le RPC est schématisé sur la figure 1.11 (la notation est celle de Sun RPC).

[5]Si on connaît le site du serveur, on peut utiliser sur ce site un service local d'annuaire (*portmapper*) accessible sur un port prédéfini (n°111).

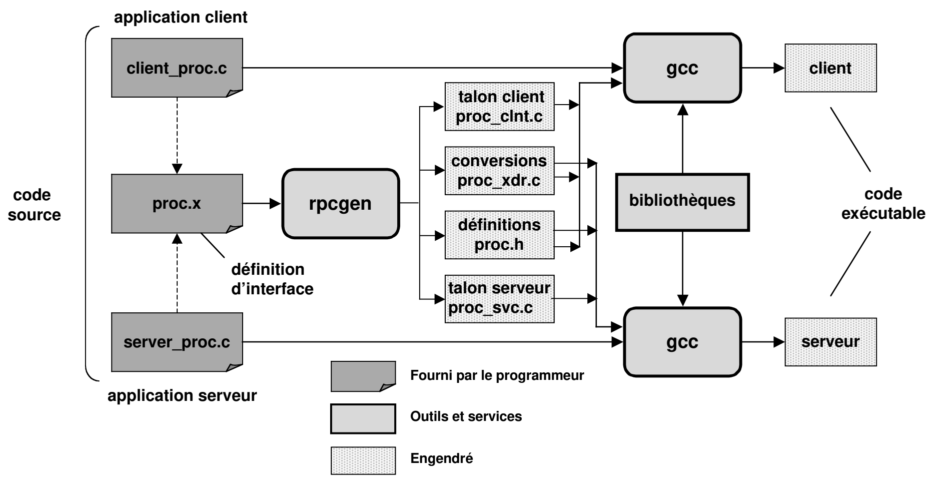

Figure 1.11 – Infrastructure de développement pour l'appel de procédure à distance.

Déploiement d'une application. Le déploiement d'une application répartie est l'installation, sur les différents sites, des parties de programme qui constituent l'application, et le lancement de leur exécution. Dans le cas du RPC, l'installation est généralement réalisée par des scripts prédéfinis qui appellent les outils représentés sur la figure 1.11, en utilisant éventuellement un système réparti de fichiers pour retrouver les programmes source. Les contraintes de l'activation sont que le serveur doit être lancé avant le premier appel d'un client et que le service de désignation doit être lancé avant le serveur pour permettre à ce dernier de s'enregistrer. Les activations peuvent être réalisées par un script lancé par l'administrateur du système ou éventuellement par un client (dans ce cas, celui-ci doit disposer des autorisations nécessaires).

1.3.4 Résumé et conclusions

Plusieurs enseignements utiles peuvent être tirés de cette étude de cas simple.

1. La transparence complète (c'est-à-dire la préservation du comportement d'une application entre environnements centralisé et réparti) ne peut pas être réalisée. Bien que cet objectif ait été initialement fixé aux premiers temps du développement de l'intergiciel, l'expérience a montré que la transparence avait des limites, et qu'il valait mieux explicitement prendre en compte le caractère réparti des applications, au moins pour les aspects où la distinction est pertinente, tels que la tolérance aux fautes et les performances. Voir [Waldo et al. 1997] pour une discussion détaillée.
2. Plusieurs schémas utiles, ou patrons, ont été mis en évidence. L'usage de représentants locaux pour gérer la communication entre entités distantes est l'un des patrons les plus courants de l'intergiciel (le patron de conception Proxy, voir chapitre 2, section 2.3.1). Un autre patron universel est la liaison client-serveur via un service de désignation (le patron d'architecture Broker). D'autres patrons, peut-être moins immédiatement visibles, sont l'organisation de l'activité du serveur par création dynamique ou gestion d'une réserve de *threads* (voir chapitre 2, section

2.3.3), et le schéma détection-réaction pour le traitement des défaillances.

3. Le développement d'une application répartie, même avec un schéma d'exécution conceptuellement aussi simple que le RPC, met en jeu une infrastructure logistique importante : IDL, générateurs de souches, représentation commune des données, emballeurs et déballeurs, mécanisme de réaction aux défaillances, service de désignation, outils de déploiement. La conception de cette infrastructure, avec l'objectif de simplifier la tâche des développeurs d'applications, est un thème récurrent de cet ouvrage.

En ce qui concerne l'utilité du RPC comme outil pour la structuration des applications réparties, on peut noter quelques limitations.

– La structure de l'application est statique ; rien n'est prévu pour la création dynamique de serveurs ou pour la restructuration d'une application.
– La communication est restreinte à un schéma synchrone : rien n'est prévu pour la communication asynchrone, dirigée par les événements.
– Les données gérées par le client et le serveur ne sont pas persistantes, c'est-à-dire qu'elles disparaissent avec les processus qui les ont créées. Ces données peuvent bien sûr être rangées dans des fichiers, mais leur sauvegarde et leur restauration doivent être explicitement programmées ; rien n'est prévu pour assurer la persistance automatique.

Dans la suite de cet ouvrage, nous présentons d'autres infrastructures, qui échappent à ces limitations.

1.4 Problèmes et défis de la conception de l'intergiciel

Dans cette section, nous examinons les problèmes posés par la conception des systèmes intergiciels, et quelques principes d'architecture permettant d'y répondre. Nous concluons par une brève revue de quelques défis actuels posés par l'intergiciel, qui orientent les recherches dans ce domaine.

1.4.1 Problèmes de conception

La fonction de l'intergiciel est l'intermédiation entre les parties d'une application, ou entre applications. Les considérations *architecturales* tiennent donc une place centrale dans la conception de l'intergiciel. L'architecture couvre l'organisation, la structure d'ensemble, et les schémas de communication, aussi bien pour les applications que pour l'intergiciel lui-même.

Outre les aspects architecturaux, les principaux problèmes de la conception de l'intergiciel sont ceux résultant de la répartition. Nous les résumons ci-après.

La désignation et la liaison sont des notions centrales de la conception de l'intergiciel, puisque celui-ci peut être défini comme le logiciel assurant la liaison entre les différentes parties d'une application répartie, et facilitant leur interaction.

Dans tout système intergiciel, cette interaction repose sur une couche de communication. En outre, la communication est une des fonctions que fournit l'intergiciel aux applications. Les entités communicantes peuvent assumer des rôles divers tels que client-serveur ou pair à pair. À un niveau plus élevé, l'intergiciel permet différents modes d'interaction (appel synchrone, communication asynchrone par messages, coordination via des objets partagés).

L'architecture logicielle définit l'organisation d'un système réalisé comme assemblage de parties. Les notions liées à la composition et aux composants logiciels sont maintenant un élément central de la conception des systèmes intergiciels, aussi bien pour leur propre structure que pour celle des applications qui les utilisent.

La gestion de données soulève le problème de la persistance (conservation à long terme et procédures d'accès) et des transactions (maintien de la cohérence pour l'accès concurrent aux données en présence de défaillances éventuelles).

La qualité de service comprend diverses propriétés d'une application qui ne sont pas explicitement spécifiées dans ses interfaces fonctionnelles, mais qui sont très importantes pour ses utilisateurs. Des exemples de ces propriétés sont la fiabilité et la disponibilité, les performances (spécialement pour les applications en temps réel), et la sécurité.

L'administration est une étape du cycle de vie des applications qui prend une importance croissante. Elle comprend des fonctions telles que la configuration et le déploiement, la surveillance (*monitoring*), la réaction aux événements indésirables (surcharge, défaillances) et la reconfiguration. La complexité croissante des tâches d'administration conduit à envisager de les automatiser (*autonomic computing*).

Les différents aspects ci-dessus apparaissent dans les études de cas qui constituent la suite de cet ouvrage.

1.4.2 Quelques principes d'architecture

Les principes développés ci-après ne sont pas propres aux systèmes intergiciels, mais prennent une importance particulière dans ce domaine car les qualités et défauts de l'intergiciel conditionnent ceux des applications, en raison de son double rôle de médiateur et de fournisseur de services communs.

Modèles et spécifications

Un *modèle* est une représentation simplifiée de tout ou partie du monde réel. Un objet particulier peut être représenté par différents modèles, selon le domaine d'intérêt choisi et le degré de détail de la représentation. Les modèles servent à mieux comprendre l'objet représenté, en formulant explicitement les hypothèses pertinentes et en en déduisant des propriétés utiles. Comme aucun modèle n'est une représentation parfaite de la réalité, la prudence est requise lorsqu'on transpose au monde réel des résultats obtenus à partir d'un modèle. [Schneider 1993] discute l'usage et l'utilité des modèles pour les systèmes répartis.

Des modèles formels ou semi-formels sont utilisés pour différents aspects de l'intergiciel, notamment la désignation, la composition, la tolérance aux fautes et la sécurité.

Les modèles sont utiles pour la formulation de spécifications rigoureuses. Les spécifications du comportement d'un système sont classées en deux catégories :

- Sûreté (*safety*). De manière informelle : un événement ou une situation indésirable ne se produira jamais.
- Vivacité (*liveness*). De manière informelle : un événement souhaité finira par se produire.

Prenons l'exemple d'un système de communication. Une propriété de sûreté est que le contenu d'un message délivré est identique à celui du même message émis. Une propriété de vivacité est qu'un message émis finira par être délivré à son ou ses destinataire(s). La vivacité est souvent plus difficile à garantir que la sûreté. [Weihl 1993] examine les spécifications des systèmes parallèles et répartis.

Séparation des préoccupations

En génie logiciel, la séparation des préoccupations (*separation of concerns*) est une démarche de conception qui consiste à isoler, dans un système, des aspects indépendants ou faiblement couplés, et à traiter séparément chacun de ces aspects.

Les avantages attendus sont de permettre au concepteur et au développeur de se concentrer sur un problème à la fois, d'éliminer les interactions artificielles entre des aspects orthogonaux, et de permettre l'évolution indépendante des besoins et des contraintes associés à chaque aspect. La séparation des préoccupations a une incidence profonde aussi bien sur l'architecture de l'intergiciel que sur la définition des rôles pour la répartition des tâches de conception et de développement.

La séparation des préoccupations peut être vue comme un « méta-principe », qui peut prendre diverses formes spécifiques dont nous donnons quatre exemples ci-après.

– Le principe d'encapsulation (voir chapitre 2, section 2.1.3) dissocie les préoccupations de l'utilisateur d'un composant logiciel de celles de son réalisateur, en les plaçant de part et d'autre d'une définition commune d'interface.
– Le principe d'abstraction permet de décomposer un système complexe en niveaux (voir chapitre 2, section 2.2.1), dont chacun fournit une vue qui cache les détails non pertinents, qui sont pris en compte aux niveaux inférieurs.
– La séparation entre politiques et mécanismes [Levin et al. 1975] est un principe largement applicable, notamment dans le domaine de la gestion et de la protection de ressources. Cette séparation donne de la souplesse au concepteur de politiques, tout en évitant de « sur-spécifier » des mécanismes. Il doit par exemple être possible de modifier une politique sans avoir à réimplémenter les mécanismes qui la mettent en œuvre.
– Le principe de la persistance orthogonale sépare la définition de la durée de vie des données d'autres aspects tels que le type des données ou les propriétés de leurs fonctions d'accès.

Ces points sont développés dans le chapitre 2.

Dans un sens plus restreint, la séparation des préoccupations vise à traiter des aspects dont la mise en œuvre (au moins dans l'état de l'art courant) est dispersée entre différentes parties d'un système logiciel, et étroitement imbriquée avec celle d'autres aspects. L'objectif est de permettre une expression séparée des aspects que l'on considère comme indépendants et, en dernier ressort, d'automatiser la tâche de production du code qui traite chacun des aspects. Des exemples d'aspects ainsi traités sont ceux liés aux propriétés « extra-fonctionnelles » (voir chapitre 2, section 2.1.2) telles que la disponibilité, la sécurité, la persistance, ainsi que ceux liés aux fonctions courantes que sont la journalisation, la mise au point, l'observation, la gestion de transactions. Tous ces aspects sont typiquement réalisés par des morceaux de code dispersés dans différentes parties d'une application.

La séparation des préoccupations facilite également l'identification des rôles spécialisés au sein des équipes de conception et de développement, tant pour l'intergiciel proprement dit que pour les applications. En se concentrant sur un aspect particulier, la personne ou le groupe qui remplit un rôle peut mieux appliquer ses compétences et travailler plus efficacement. Des exemples de rôles associés aux différents aspects des applications à base de composants sont donnés dans le chapitre 5.

Évolution et adaptation

Les systèmes logiciels doivent s'accommoder du changement. En effet, les spécifications évoluent à mesure de la prise en compte des nouveaux besoins des utilisateurs, et la diversité des systèmes et organes de communication entraîne des conditions variables d'exécution. S'y ajoutent des événements imprévisibles comme les variations importantes de la charge et les divers types de défaillances. La conception des applications comme celle de l'intergiciel doit tenir compte de ces conditions changeantes : l'évolution des programmes répond à celle des besoins, leur adaptation dynamique répond aux variations des conditions d'exécution.

Pour faciliter l'évolution d'un système, sa structure interne doit être rendue accessible. Il y a une contradiction apparente entre cette exigence et le principe d'encapsulation, qui vise à cacher les détails de la réalisation.

Ce problème peut être abordé par plusieurs voies. Des techniques pragmatiques, reposant souvent sur l'interception (voir chapitre 2, section 2.3.5), sont largement utilisées dans les intergiciels commerciaux. Une approche plus systématique utilise la réflexivité. Un système est dit *réflexif* [Smith 1982, Maes 1987] quand il fournit une représentation de lui-même permettant de l'observer et de l'adapter, c'est-à-dire de modifier son comportement. Pour être cohérente, cette représentation doit être *causalement connectée* au système : toute modification apportée au système doit se traduire par une modification homologue de sa représentation, et vice versa. Les méta-objets fournissent une telle représentation explicite des mécanismes de base d'un système, ainsi que des protocoles pour examiner et modifier cette représentation. La programmation par aspects, une technique destinée à assurer la séparation des préoccupations, est également utile pour réaliser l'évolution dynamique d'un système. Ces techniques et leur utilisation dans les systèmes intergiciels sont examinées dans le chapitre 2.

1.4.3 Défis de l'intergiciel

Les concepteurs des futurs systèmes intergiciels sont confrontés à plusieurs défis.

- Performances. Les systèmes intergiciels reposent sur des mécanismes d'interception et d'indirection, qui induisent des pertes de performances. L'adaptabilité introduit des indirections supplémentaires, qui aggravent encore la situation. Ce défaut peut être compensé par diverses méthodes d'optimisation, qui visent à éliminer les coûts supplémentaires inutiles en utilisant l'injection[6] directe du code de l'intergiciel dans celui des applications. La souplesse d'utilisation doit être préservée, en permettant d'annuler, en cas de besoin, l'effet de ces optimisations.

[6] Cette technique s'apparente à l'optimisation des appels de procédure par insertion de leur code à l'endroit de l'appel (*inlining*).

- Passage à grande échelle. À mesure que les applications deviennent de plus en plus étroitement interconnectées et interdépendantes, le nombre d'objets, d'usagers et d'appareils divers composant ces applications tend à augmenter. Cela pose le problème de la capacité de croissance (*scalability*) pour la communication et pour les algorithmes de gestion d'objets, et accroît la complexité de l'administration (ainsi, on peut s'interroger sur la possibilité d'observer l'état d'un très grand système réparti, et sur le sens même que peut avoir une telle notion). Le passage à grande échelle rend également plus complexe la préservation des différentes formes de qualité de service.
- Ubiquité. L'informatique ubiquitaire (ou omniprésente) est une vision du futur proche, dans laquelle un nombre croissant d'appareils (capteurs, processeurs, actionneurs) inclus dans divers objets physiques participent à un réseau d'information global. La mobilité et la reconfiguration dynamique seront des traits dominants de ces systèmes, imposant une adaptation permanente des applications. Les principes d'architecture applicables aux systèmes d'informatique ubiquitaire restent encore largement à élaborer.
- Administration. L'administration de grandes applications hétérogènes, largement réparties et en évolution permanente, soulève de nombreuses questions telles que l'observation cohérente, la sécurité, l'équilibre entre autonomie et interdépendance pour les différents sous-systèmes, la définition et la réalisation des politiques d'allocation de ressources, etc.

1.4.4 Plan de l'ouvrage

Le chapitre 2 couvre les aspects architecturaux des systèmes intergiciels. Outre une présentation des principes géneraux d'organisation de ces systèmes, ce chapitre décrit un ensemble de patrons (en anglais *patterns*) et de canevas (en anglais *frameworks*) récurrents dans la conception de tous les types d'intergiciels et des applications qui les utilisent.

Les chapitres suivants sont des études de cas de plusieurs familles d'intergiciel, qui sont représentatives de l'état actuel du domaine, tant en ce qui concerne les produits utilisés que les normes proposées ou en cours d'élaboration.

Le chapitre 3 présente un modèle de composants appelé Fractal, dont les caractéristiques sont : (a) une grande généralité (indépendance par rapport au langage de programmation, définition explicite des dépendances, composition hiérarchique avec possibilité de partage) ; (b) la présence d'une interface d'administration extensible, distincte de l'interface fonctionnelle.

Le chapitre 4 est une introduction aux services Web. Ce terme recouvre un ensemble de normes et de systèmes permettant de créer, d'intégrer et de composer des applications réparties en utilisant les protocoles et les infrastructures du World Wide Web.

Le chapitre 5 décrit les principes et techniques de la plate-forme J2EE, spécifiée par Sun Microsystems, qui rassemble divers intergiciels pour la construction et l'intégration d'applications. J2EE a pour base le langage Java.

Le chapitre 6 décrit la plate-forme .NET de Microsoft, plate-forme intergicielle pour la construction d'applications réparties et de services Web.

Le chapitre 7 présente une autre démarche pour la construction modulaire de systèmes. Il s'agit d'une architecture à base de composants, définie par le consortium OSGi, et constituée de deux éléments : une plate-forme pour la fourniture de services et un environnement de déploiement, organisés autour d'un modèle simple de composants.

L'ouvrage comprend des annexes qui présentent la mise en œuvre pratique de certaines des plates-formes ci-dessus (annexes A : Les services Web en pratique, B : .NET en pratique, C : OSGI en pratique), ainsi que des compléments divers.

1.5 Note historique

Le terme *middleware* semble être apparu vers 1990[7], mais des systèmes intergiciels existaient bien avant cette date. Des logiciels commerciaux de communication par messages étaient disponibles à la fin des années 1970. La référence classique sur l'appel de procédure à distance est [Birrell and Nelson 1984], mais des constructions analogues, liées à des langages particuliers, existaient déjà auparavant (la notion d'appel de procédure à distance apparaît dans [White 1976][8] et une première réalisation est proposée dans [Brinch Hansen 1978]).

Vers le milieu des années 1980, plusieurs projets développent des infrastructures intergicielles pour objets répartis, et élaborent les principaux concepts qui vont influencer les normes et les produits futurs. Les précurseurs sont Cronus [Schantz et al. 1986] et Eden [Almes et al. 1985], suivis par Amoeba [Mullender et al. 1990], ANSAware [ANSA], Arjuna [Parrington et al. 1995], Argus [Liskov 1988], Chorus/COOL [Lea et al. 1993], Clouds [Dasgupta et al. 1989], Comandos [Cahill et al. 1994], Emerald [Jul et al. 1988], Gothic [Banâtre and Banâtre 1991], Guide [Balter et al. 1991], Network Objects [Birrell et al. 1995], SOS [Shapiro et al. 1989], et Spring [Mitchell et al. 1994].

L'*Open Software Foundation* (OSF), qui deviendra plus tard l'*Open Group* [Open Group], est créée en 1988 dans le but d'unifier les diverses versions du système d'exploitation Unix. Cet objectif ne fut jamais atteint, mais l'OSF devait spécifier une plate-forme intergicielle, le *Distributed Computing Environment* (DCE) [Lendenmann 1996], qui comportait notamment un service d'appel de procédure à distance, un système réparti de gestion de fichiers, un serveur de temps, et un service de sécurité.

L'*Object Management Group* (OMG) [OMG] est créé en 1989 pour définir des normes pour l'intergiciel à objets répartis. Sa première proposition (1991) fut la spécification de CORBA 1.0 (la version courante, en 2005, est CORBA 3). Ses propositions ultérieures sont des normes pour la modélisation (UML, MOF) et les composants (CCM). L'*Object Database Management Group* (ODMG) [ODMG] definit des normes pour les bases de données à objets, qui visent à unifier la programmation par objets et la gestion de données persistantes.

Le modèle de référence de l'*Open Distributed Processing* (RM-ODP) [ODP 1995a], [ODP 1995b] a été conjointement défini par deux organismes de normalisation, l'ISO et l'ITU-T. Il propose un ensemble de concepts définissant un cadre générique pour le calcul réparti ouvert, plutôt que des normes spécifiques.

La définition du langage Java par Sun Microsystems en 1995 ouvre la voie à plusieurs intergiciels, dont *Java Remote Method Invocation* (RMI) [Wollrath et al. 1996] et les *En-*

[7]Le terme français *intergiciel* est apparu plus récemment (autour de 1999-2000).

[8]version étendue du RFC Internet 707.

terprise JavaBeans (EJB) [Monson-Haefel 2002]. Ces systèmes, et d'autres, sont intégrés dans une plate-forme commune, J2EE [J2EE 2005].

Microsoft développe à la même époque le *Distributed Component Object Model* (DCOM) [Grimes 1997], un intergiciel définissant des objets répartis composables, dont une version améliorée est COM+ [Platt 1999]. Son offre courante est .NET [.NET], plate-forme intergicielle pour la construction d'applications réparties et de services pour le Web.

La première conférence scientifique entièrement consacrée à l'intergiciel a lieu en 1998 [Middleware 1998]. Parmi les thèmes actuels de la recherche sur l'intergiciel, on peut noter les techniques d'adaptation (réflexivité, aspects), l'administration et notamment les travaux sur les systèmes autonomes (pouvant réagir à des surcharges ou à des défaillances), et l'intergiciel pour appareils mobiles et environnements ubiquitaires.

Bibliographie

[Almes et al. 1985] Almes, G. T., Black, A. P., Lazowska, E. D., and Noe, J. D. (1985). The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–59.

[ANSA] ANSA. Advanced networked systems architecture. http://www.ansa.co.uk/.

[Balter et al. 1991] Balter, R., Bernadat, J., Decouchant, D., Duda, A., Freyssinet, A., Krakowiak, S., Meysembourg, M., Le Dot, P., Nguyen Van, H., Paire, E., Riveill, M., Roisin, C., Rousset de Pina, X., Scioville, R., and Vandôme, G. (1991). Architecture and implementation of Guide, an object-oriented distributed system. *Computing Systems*, 4(1):31–67.

[Banâtre and Banâtre 1991] Banâtre, J.-P. and Banâtre, M., editors (1991). *Les systèmes distribués : expérience du projet Gothic*. InterÉditions.

[Birrell and Nelson 1984] Birrell, A. D. and Nelson, B. J. (1984). Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59.

[Birrell et al. 1995] Birrell, A. D., Nelson, G., Owicki, S., and Wobber, E. (1995). Network objects. *Software–Practice and Experience*, 25(S4):87–130.

[Brinch Hansen 1978] Brinch Hansen, P. (1978). Distributed Processes: a concurrent programming concept. *Communications of the ACM*, 21(11):934–941.

[Cahill et al. 1994] Cahill, V., Balter, R., Harris, N., and Rousset de Pina, X., editors (1994). *The COMANDOS Distributed Application Platform*. ESPRIT Research Reports. Springer-Verlag. 312 pp.

[Dasgupta et al. 1989] Dasgupta, P., Chen, R. C., Menon, S., Pearson, M. P., Ananthanarayanan, R., Ramachandran, U., Ahamad, M., LeBlanc, R. J., Appelbe, W. F., Bernabéu-Aubán, J. M., Hutto, P. W., Khalidi, M. Y. A., and Wilkenloh, C. J. (1989). The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1):11–46.

[Fox et al. 1998] Fox, A., Gribble, S. D., Chawathe, Y., and Brewer, E. A. (1998). Adapting to network and client variation using infrastructural proxies: Lessons and perspectives. *IEEE Personal Communications*, pages 10–19.

[Grimes 1997] Grimes, R. (1997). *Professional DCOM Programming*. Wrox Press. 592 pp.

[J2EE 2005] J2EE (2005). Java 2 Enterprise Edition. http://java.sun.com/products/j2ee.

[Jul et al. 1988] Jul, E., Levy, H., Hutchinson, N., and Black, A. (1988). Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133.

[Lea 1999] Lea, D. (1999). *Concurrent Programming in Java.* The Java Series. Addison-Wesley, 2nd edition. 412 pp.

[Lea et al. 1993] Lea, R., Jacquemot, C., and Pillevesse, E. (1993). COOL: System Support for Distributed Object-oriented Programming. *Communications of the ACM (special issue, Concurrent Object-Oriented Programming, B. Meyer, editor)*, 36(9):37–47.

[Lendenmann 1996] Lendenmann, R. (1996). *Understanding OSF DCE 1.1 for AIX and OS/2.* Prentice Hall. 312 pp.

[Levin et al. 1975] Levin, R., Cohen, E. S., Corwin, W. M., Pollack, F. J., and Wulf, W. A. (1975). Policy/mechanism separation in hydra. In *Proceedings of the Fifth ACM Symposium on Operating Systems Principles*, pages 132–140.

[Liskov 1988] Liskov, B. (1988). Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312.

[Maes 1987] Maes, P. (1987). Concepts and Experiments in Computational Reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, pages 147–155, Orlando, Florida, USA.

[McIlroy 1968] McIlroy, M. (1968). Mass produced software components. In Naur, P. and Randell, B., editors, *Software Engineering: A Report On a Conference Sponsored by the NATO Science Committee*, pages 138–155, Garmisch, Germany.

[Middleware 1998] Middleware (1998). IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing. September 15-18 1998, The Lake District, England.

[Mitchell et al. 1994] Mitchell, J. G., Gibbons, J., Hamilton, G., Kessler, P. B., Khalidi, Y. Y. A., Kougiouris, P., Madany, P., Nelson, M. N., Powell, M. L., and Radia, S. R. (1994). An overview of the Spring system. In *Proceedings of COMPCON*, pages 122–131.

[Monson-Haefel 2002] Monson-Haefel, R. (2002). *Enterprise JavaBeans.* O'Reilly & Associates, Inc., 3rd edition. 550 pp.

[Mullender et al. 1990] Mullender, S. J., van Rossum, G., Tanenbaum, A. S., van Renesse, R., and van Staveren, H. (1990). Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53.

[.NET] .NET. Microsoft Corp. http://www.microsoft.com/net.

[ODMG] ODMG. The Object Data Management Group. http://www.odmg.org.

[ODP 1995a] ODP (1995a). ITU-T & ISO/IEC, Recommendation X.902 & International Standard 10746-2: "ODP Reference Model: Foundations".
http://archive.dstc.edu.au/AU/research_news/odp/ref_model/standards.html.

[ODP 1995b] ODP (1995b). ITU-T & ISO/IEC, Recommendation X.903 & International Standard 10746-3: "ODP Reference Model: Architecture".
http://archive.dstc.edu.au/AU/research_news/odp/ref_model/standards.html.

[OMG] OMG. The Object Management Group. http://www.omg.org.

[Open Group] Open Group. http://www.opengroup.org/.

[Parrington et al. 1995] Parrington, G. D., Shrivastava, S. K., Wheater, S. M., and Little, M. C. (1995). The design and implementation of Arjuna. *Computing Systems*, 8(2):255–308.

[Platt 1999] Platt, D. S. (1999). *Understanding COM+.* Microsoft Press. 256 pp.

[Schantz et al. 1986] Schantz, R., Thomas, R., and Bono, G. (1986). The architecture of the Cronus distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 250–259. IEEE.

[Schneider 1993] Schneider, F. B. (1993). What Good are Models and What Models are Good? In Mullender, S., editor, *Distributed Systems*, chapter 2, pages 17–26. ACM Press Books, Addison-Wesley.

[Shapiro et al. 1989] Shapiro, M., Gourhant, Y., Habert, S., Mosseri, L., Ruffin, M., and Valot, C. (1989). SOS: An object-oriented operating system - assessment and perspectives. *Computing Systems*, 2(4):287–337.

[Smith 1982] Smith, B. C. (1982). *Reflection And Semantics In A Procedural Language*. PhD thesis, Massachusetts Institute of Technology. MIT/LCS/TR-272.

[Waldo et al. 1997] Waldo, J., Wyant, G., Wollrath, A., and Kendall, S. (1997). A Note on Distributed Computing. In Vitek, J. and Tschudin, C., editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *LNCS*, pages 49–64. Springer-Verlag.

[Weihl 1993] Weihl, W. E. (1993). Specifications of Concurrent and Distributed Systems. In Mullender, S., editor, *Distributed Systems*, chapter 3, pages 27–53. ACM Press Books, Addison-Wesley.

[Weiser 1993] Weiser, M. (1993). Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–84.

[White 1976] White, J. E. (1976). A high-level framework for network-based resource sharing. In *National Computer Conference*, pages 561–570.

[Wollrath et al. 1996] Wollrath, A., Riggs, R., and Waldo, J. (1996). A Distributed Object Model for the Java System. *Computing Systems*, 9(4):265–290.