CHAPITRE 1. Des systèmes centralisés aux objets distribués

Objectif de ce chapitre

En introduction de cet ouvrage, nous allons en premier lieu retracer, à travers un très bref historique, l'évolution des architectures de systèmes informatisés depuis les systèmes centralisés jusqu'aux premiers systèmes répartis, ayant donné naissance au concept de *client-serveur*. Ceci nous conduira naturellement à présenter les différents cas d'interaction client-serveur consacrés par l'usage, ainsi que les technologies associées.

Nous présenterons ensuite l'approche par *objets distribués*, qui constitue l'évolution naturelle du client-serveur dans le contexte de la programmation par objets, et étudierons à cette occasion plus en détail la spécification *CORBA* (*Common Object Request Broker Architecture*) ainsi que l'invocation de méthode à distance avec *RMI* (*Remote Method Invocation*), liée au langage Java.

Nous aborderons enfin, avec la montée en puissance d'Internet, les *architectures 3-tiers* et *N-tiers* actuellement très utilisées dans la pratique des projets, lesquelles nous permettrons d'introduire les thèmes abordés dans les deux chapitres suivants

1.1. Introduction : systèmes centralisés et architectures réparties

1.1.1. Architecture centralisée et architectures réparties

Les premières générations de systèmes informatisés s'appuyaient sur un modèle centralisé, reposant sur un ordinateur unique, avec d'abord, historiquement, les *mainframes*, puis les matériels issus de la *mini-infomatique*. Logiquement, les constructeurs de machines développaient alors naturellement les logiciels adaptés aux matériels commercialisés, ce qui comprenait bien sûr les logiciels de base, dont le système d'exploitation et les gestionnaires de périphériques. Mais ces logiciels comprenaient aussi tout un ensemble de logiciels applicatifs destinés aux usages d'entreprise, sachant que les besoins portaient alors sur des aspects relativement génériques, essentiellement issus du monde de la gestion. Pour des usages plus spécifiques, soit de nature scientifique ou technique, soit liés à un système d'information, les organisations étaient invitées à développer leurs propres programmes dans des langages qui étaient essentiellement, au départ, Fortran (pour le calcul scientifique) et Cobol (pour les applications de gestion), complétés ensuite par de nombreux autres langages. Une fois ces logiciels conçus et développés, leur mise en œuvre ne posait pas de problème particulier d'intégration ou de déploiement puisque tout était conçu pour fonctionner dans le cadre d'un environnement unique, bien connu et stable.

L'arrivée des matériels issus de ce qui était alors appelé la *micro-informatique*, et celle des ordinateurs personnels et des stations de travail graphiques, a considérément modifié cette vision des choses. Dans la mesure où l'on disposait maintenant de matériels de coût bien inférieur, voire même très bon marché, bien plus faciles à déployer, dotés de systèmes d'exploitation standards, et susceptibles d'être connectés en réseaux locaux, tout un horizon potentiel d'applications nouvelles apparaissait alors.

Mais la contrepartie d'un tel bénéfice, certes considérable, était que le système informatisé apparaissait alors comme un ensemble de logiciels et de matériels potentiellement hétérogènes qu'il fallait interconnecter et faire coopérer, requérant donc la définition d'*architectures* matérielle et logicielle adaptées au problème à résoudre. Le rôle d'*architecte logiciel*, c'est-à-dire celui d'acteur en charge de la conception du système, responsable de l'organisation du logiciel et du choix des éléments à y intégrer, devenait alors absolument essentiel. On passait ainsi d'une notion relativement simple d'*application à développer* à celle, plus complexe, de *système à architecturer*. Nous parlerons ici, pour qualifier une telle situation, de logiciels ou de systèmes *distribués*, ou encore *répartis*[1].

La figure ci-dessous illustre ce changement de paradigme dans le développement de systèmes informatisés, avec, d'un côté, un ordinateur central exécutant des applications, accédées via de simples terminaux passifs, et, de l'autre, un ensemble de machines unies en réseau, et exécutant des entités logicielles réparties sur celles-ci.

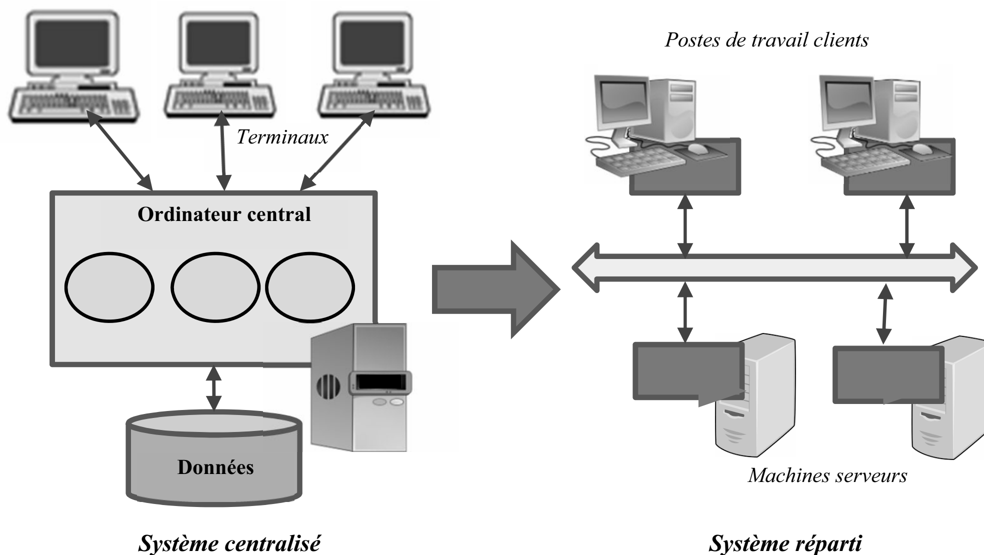

Figure 1. Système centralisé et système réparti.

Dans une *approche répartie*, nous aurons ainsi différentes machines, parmi lesquelles des machines dites *clientes*, au contact direct des utilisateurs, et responsables en particulier des interactions avec ces derniers, et des machines plus puissantes, dites *serveurs*, en charge du stockage des données et/ou de l'exécution de traitements communs.

On remarquera que cette approche de *système réparti* est toujours l'approche actuelle, maintenant transposée à une plus grande échelle, celle d'Internet. La répartition est ainsi

[1] Nous considérerons ici ces deux termes comme synonymes. Si la langue anglaise ne connait qu'un seul terme, à savoir *distributed*, les professionnels de l'informatique en France emploient volontiers les deux termes de *distribué* et de *réparti*. Il a un temps été proposé de préférer le terme de *distribué* pour qualifier la localisation des traitements sur différents matériels distants, et celui de *réparti* pour parler de la distribution des données. L'usage semble considérer à l'heure actuelle ces deux termes comme synonymes, ce que nous ferons ici.

devenue à ce jour l'approche naturelle pour la conception d'un nouveau système informatisé de quelque importance. Cette répartition peut se concevoir à l'échelle d'un réseau local, mais aussi, comme très souvent maintenant, à une échelle beaucoup plus grande, avec des machines réparties géographiquement sur plusieurs sites physiques.

1.1.2. Les architectures réparties : intérêts et difficultés

De manière générale, la conception d'un système implique donc de définir une certaine répartition des traitements et des données sur différentes machines physiques distantes. Les composants logiciels présents sur ces machines vont coopérer entre eux, via des mécanismes logiciels particuliers, très souvent basés, comme nous le verrons dans la suite, sur une relation *asymétrique* du type *client à fournisseur de service*, ou bien encore, parfois, sur une relation *émetteur-récepteur* d'information, dans un lien cette fois d'*égal à égal*.

Outre les avantages déjà indiqués, liés aux coûts et à la *déployabilité*, la mise en œuvre d'une solution répartie présente de *nombreux intérêts potentiels*. L'un de ses intérêts est la possibilité d'introduire des *redondances* dans un système, que ce soit une redondance des *données* (gestion des mêmes données sur plusieurs serveurs, voire sur plusieurs sites géographiques) et/ou des *traitements*, de manière à se prémunir contre des pannes, augmenter la *disponibilité* du système et ainsi la continuité du service fourni. Un autre intérêt du principe de répartition réside dans une *extensibilité*[1] *a priori plus facile* du système. Ainsi, si un système est convenablement architecturé à cette fin, il pourra permettre une certaine adaptation (dans une certaine limite), via l'ajout de machines supplémentaires, aux nouvelles dimensions du problème posé, qu'il s'agisse du volume des données à gérer, du débit de flux entrant, de la charge à supporter, etc. Enfin, l'approche des systèmes répartis permet d'envisager l'*intégration d'applications issues de différentes générations technologiques*, lesquelles, on le sait, doivent fréquemment cohabiter au sein d'une même organisation. Ainsi, une application *patrimoniale*[2] pourra-t-elle continuer à être exploitée sans modification sur le matériel pour lequel elle a été conçue, tandis que des applications nouvellement développées tourneront sur d'autres machines plus récentes, et s'interfaceront néanmoins avec celle-ci via des *couches logicielles de médiation* conçues à cet effet. Bien sûr, tous ces intérêts sont des intérêts *potentiels*, en ce sens qu'ils ouvrent des possibilités, sans bien sûr apporter de réponse directe toute prête à chaque cas.

Le principe d'un système réparti permet d'envisager de nouvelles manières, beaucoup plus simples, de résoudre un certain nombre de problèmes. Encore faudra-t-il faire inter opérer concrètement les éléments logiciels répartis sur les différentes machines. De manière générale, pour construire un système réparti, nous aurons donc besoin de technologies logicielles convenables, permettant de *mettre en œuvre les interopérations indispensables entre applications distantes*. Sur cette base, il faudra concevoir les applications ainsi réparties sur la base d'une organisation architecturale adaptée, c'est-à-dire qui réponde bien au problème posé. Il nous faudra bien sûr aussi choisir les approches et les technologies logicielles d'interopération adaptées à chaque cas.

Au-delà des avantages de la répartition, il convient de bien noter ici le fait que, lorsqu'on développe un système réparti, des difficultés de conception particulières, spécifiquement liées à la répartition, apparaissent. Une première source de difficultés vient bien sûr des

---

[1] Nous employons ici ce terme d'*extensibilité* dans le sens du mot anglais *scalability*.

[2] On emploie ce terme d'application *patrimoniale*, ou encore *legacy* pour désigner une application basée sur une technologie ancienne, que l'on continue, par choix, à opérer en l'état. On parlera aussi d'application « léguée ».

réseaux de communications eux-mêmes. Certains composants logiciels n'interagissent plus directement entre eux, mais via divers réseaux de communication, qui vont de simples réseaux locaux jusqu'à la mise en œuvre d'échanges à travers Internet. Ces réseaux introduisent inévitablement une plus ou moins grande *latence* et un *temps de transmission* dont il faut tenir compte dans le comportement global du système, en particulier pour ce qui est de l'atteinte des performances attendues. Par ailleurs, la *fiabilité* d'un réseau ne peut être absolue, et une limite dans les *débits de transmission* apparaît. Enfin, les communications à travers un réseau à grande échelle posent des problèmes particuliers de *sécurité*, qui n'apparaissaient nullement dans le cas d'une application locale, liée géographiquement à un site unique.

Une autre source de difficulté vient de la *dynamicité intrinsèque à tout système distribué*, constitué par nature de machines indépendantes. A tout instant, une machine peut être arrêtée du fait d'une panne, voire d'une opération de maintenance. Il convient donc de prendre en compte cette possibilité dans les interactions logicielles en traitant convenablement les erreurs. Enfin, si nous avons mentionné l'intérêt de pouvoir ajouter facilement de nouvelles machines (ouvrant la voie à des caractéristiques d'*extensibilité*), cet intérêt nécessite bien sûr, pour être concrétisé, une mise en œuvre de mécanismes logiciels *ad hoc*, sans lesquels on ne pourra espérer la moindre adaptabilité du logiciel. Ceci n'est pas toujours simple.

Tous ces éléments font que les interactions entre composants logiciels à travers les réseaux de communication posent des problèmes tout à fait spécifiques. Les solutions apportées et les modes d'interactions ne pourront être une simple transposition de mécanismes logiciels locaux, c'est-à-dire mis en œuvre au sein d'un logiciel tournant sur une machine unique. De manière générale, la résolution de ces problèmes appelle ainsi de nouvelles approches, avec, pour mettre en œuvre celles-ci, des logiciels de liaison particuliers, que nous appellerons des *connecteurs*. A un certain niveau de complexité, on a coutume de désigner de tels logiciels par le terme d'*intergiciel*[1].

Ce sont ces approches et leurs mises en œuvre que nous nous proposons d'étudier ici.

1.1.3. Objectif de cet ouvrage

Dans les systèmes répartis, les technologies logicielles permettant les interactions entre composants distants constituent le nécessaire ciment liant les applications distantes entre elles. Elles jouent donc un rôle tout à fait essentiel. Ces technologies sont de différents types, et constituent les éléments de base mis à disposition de l'architecte logiciel pour concevoir l'architecture globale du système réparti. L'architecte doit donc parfaitement maîtriser les principes sous-jacents aux différentes technologies, ainsi que les approches architecturales associées, afin de faire les choix d'architecture pertinents. Mais plus que les technologies elles-mêmes, ce sont bien en fait les *principes* sur lesquels celles-ci sont basées qui sont importants.

L'objectif que nous nous fixons ici est avant tout de présenter quelques-unes de ces approches parmi les plus importantes, et étudier la manière dont elles sont mises en œuvre dans les technologies logicielles présentes sur le marché et fixées dans les standards.

[1] Traduction du terme anglais *middleware*.

Nous examinerons donc les principales approches autorisant les interactions entre entités logicielles distantes sur un réseau de machines, avec des exemples de mises en œuvre techniques. Nous avons fait ici le choix, pour nos exposés, de baser les exemples sur le monde du langage Java, pour lequel il existe quantité d'implémentations *open source*. Bien sûr, ceci constitue un choix de présentation arbitraire. Les technologies décrites dans la suite ne le sont qu'à titre d'exemples de mises en œuvre, sachant qu'il existe également, quasiment pour chaque aspect traité, d'autres approches technologiques, dont des implémentations « propriétaires », à peu près équivalentes en termes de fonctionnalités, *a priori* tout aussi intéressantes sur le plan technique. Ce choix ne reflète donc aucune espèce de préférence ni jugement de valeur.

Dans le présent chapitre, nous examinerons en premier lieu le modèle de base proposé pour les interactions entre applications distantes, à savoir le client-serveur simple dit « à deux niveaux ». Nous verrons comment ce principe peut se décliner selon plusieurs variantes, et servir ainsi de base à différentes familles technologiques. Nous verrons ensuite des évolutions naturelles de ce modèle dans le monde objet, avec, d'une part le modèle d'architecture proposé par *CORBA* (*Common Object Request Broker Architecture*), puis le modèle d'interaction spécifique du monde Java, très utilisé, à savoir *RMI* (*Remote Method Invocation*).

Enfin, avec l'arrivée d'Internet dans les applications informatiques, et à titre de préambule aux chapitres qui suivent, nous présenterons l'évolution ayant mené aux architectures dites *N-tiers*, base des *applications orientées Web*.

1.2. Le client-serveur de base, ou à deux niveaux

Un modèle de base pour l'interaction entre entités logicielles distantes est le *modèle client-serveur.* Ce modèle est basé sur la séparation physique entre, d'une part, des *entités logicielles clientes*, requérant pour s'exécuter la fourniture d'un certain *service* (par exemple délivrer une information, ou encore exécuter un traitement) et, d'autre part, une entité logicielle dite *serveur*, en charge de fournir ce service au bénéfice des dits clients.

1.2.1. Le principe du client-serveur

On peut ainsi définir le modèle client-serveur comme une *approche architecturale* qui vise à *distribuer une application logicielle sur un ensemble de machines physiques distantes*, et où des *entités logicielles clientes et serveur interagissent via des protocoles de communication standards, sur la base de la notion de service*.

Le modèle client-serveur définit donc *deux rôles*, celui de *client*, joué par autant d'entités logicielles que nécessaire, et celui de *serveur*, joué par *une* entité particulière. Dans le modèle d'interaction client-serveur, le client adresse une *requête* au serveur puis reçoit un résultat en réponse à celle-ci, comme symbolisé sur la figure ci-après. Dans le modèle de base, cette interaction est dite *synchrone*, c'est-à-dire que l'entité cliente suspend son exécution en attente de la réponse de l'entité serveur, et la reprend une fois ce résultat obtenu.

L'intérêt fondamental du client-serveur est de mettre en commun un ensemble de traitements et/ou de centraliser l'accès à des ressources communes, tout en répartissant la charge entre les machines clientes et la machine serveur.

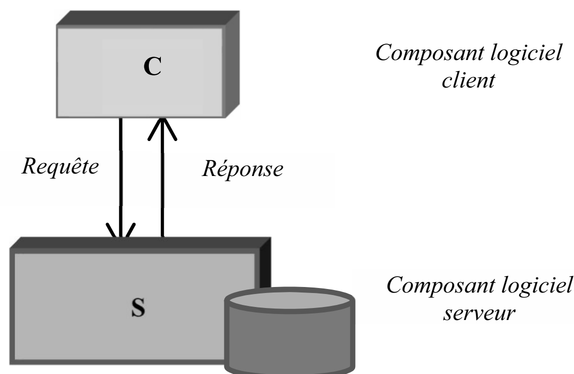

Figure 2. Interaction client-serveur.

Un tel découpage entre code client et code serveur obéit donc à une double logique : celle liée à une décomposition de l'application en entités logicielles bien découplées, et celle liée à une répartition de la charge de travail nécessaire sur différentes machines physiques. Bien sûr, outre l'exécution des traitements dont il a en propre la charge, l'entité serveur devra, en outre, savoir gérer les accès multiples concurrents aux mêmes ressources.

La figure ci-dessous symbolise une telle organisation :

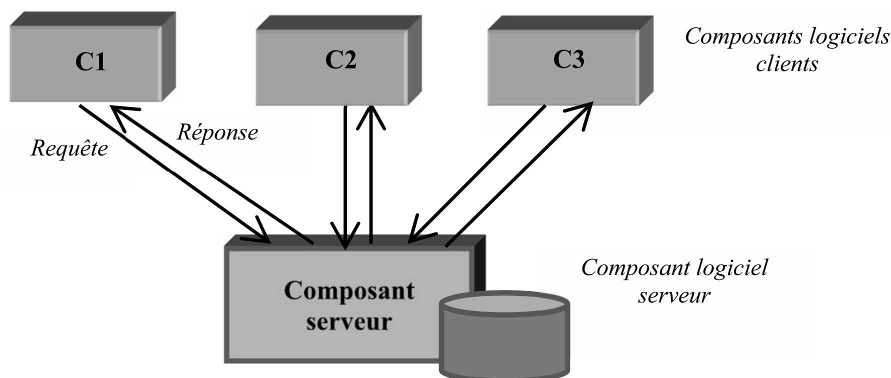

Figure 3. Illustration du modèle client-serveur.

Bien sûr, rien n'empêche l'architecte d'utiliser plusieurs relations client-serveur au sein de la même architecture, avec donc plusieurs logiciels serveurs assurant différents services.

Les caractéristiques définissant le modèle *client-serveur* sont ainsi les suivantes :

- Il est basé sur la notion de *service*, selon un mode d'interaction de type *requête-réponse*.
- Il respecte un principe d'*encapsulation*, la manière dont un service est rendu restant invisible au client. Le client reçoit un résultat sans savoir comment ce résultat a été produit et sans avoir aucunement à s'en préoccuper.

- Il assure le *partage des données et des ressources communes entre les clients*, dans le respect de l'intégrité de ces dernières. Le serveur doit ainsi gérer convenablement les accès concurrents aux mêmes ressources afin de garantir à tout instant cette intégrité. Ainsi, chaque client peut-il interagir avec le serveur comme s'il était le seul client.
- Il assure la *transparence vis-à-vis des protocoles de transmission de données* sous-jacents, dont le client n'a pas à se préoccuper.
- Il assure une *transparence vis-à-vis de l'emplacement physique du serveur*. Une fois la connexion au serveur convenablement effectuée, le client interagit avec celui-ci comme avec une entité locale dont il n'a plus besoin de connaître l'emplacement physique.
- Il assure enfin la *transparence vis à vis des matériels et des systèmes d'exploitation*, en masquant l'hétérogénéité de ceux-ci.

On notera que, dans le modèle de base client-serveur, les communications requièrent un mode *connecté*, avec présence d'une notion de *session*.

Du fait des caractéristiques ci-dessus, le modèle client-serveur apporte une certaine capacité d'*extensibilité*. Ainsi, une évolution du système visant, par exemple, à faire face à une plus grande charge de travail, pourra-t-elle se traduire selon deux axes possibles et non exclusifs. Ces deux axes concernent d'une part 1) le niveau des clients, avec l'ajout de machines et de composants logiciels clients associés (forme d'*extensibilité* dite *horizontale*), et d'autre part, 2) le niveau du serveur, par remplacement de la machine serveur par une machine plus puissante (*extensibilité* dite *verticale*).

Il convient ici de bien distinguer le client-serveur en tant que *principe d'interaction* entre entités logicielles, dont il vient d'être question, et le client-serveur vu cette fois en tant que *style architectural*, organisation d'un système réparti où les composants logiciels interagissent précisément entre eux selon ce *mode client-serveur*. On parle plus volontiers dans ce dernier cas de *style client-serveur à deux niveaux*. Les composants sont alors soit clients, soit serveurs, de manière exclusive, et l'architecture distingue donc de fait deux niveaux. Plus généralement, on considère des styles *client-serveur à N niveaux*, certains composants jouant alors en même temps des rôles de client et de serveur dans différentes relations client-serveur. Dans la première situation, l'architecture apparaîtra structurée sur deux niveaux, à savoir le niveau des clients et celui des serveurs. Dans la seconde situation, l'architecture présentera cette fois non pas deux, mais N niveaux. Un composant d'un niveau intermédiaire pourra alors jouer en même temps le rôle de serveur pour le niveau supérieur, et celui de client vis-à-vis du niveau inférieur. En d'autres termes, dans une architecture à N niveaux, un composant serveur peut donc s'appuyer sur un autre composant pour rendre le service qui lui est demandé.

1.2.2. Découpage des responsabilités entre client et serveur

Nous avons vu que le client-serveur sous-entendait une *interaction de type requête-réponse*, c'est-à-dire une interaction basée sur des requêtes adressées par un client à un serveur, lequel rendait un résultat, et ce de manière *synchrone*.

Mais le modèle, dans son principe, ne présuppose rien quant à la répartition des responsabilités entre le client et le serveur. Si certaines tâches, tels les accès aux données partagées, ou encore les contrôles d'intégrité de celles-ci, ont leur place naturelle côté

serveur, il n'en va pas de même des traitements applicatifs, dont on peut imaginer qu'ils soient exécutés côté client ou bien côté serveur, ou bien encore pour partie côté client et pour partie côté serveur. Le choix d'une répartition particulière à adopter résulte d'une multitude de facteurs, parmi lesquels, par exemple, la minimisation du couplage, la réduction de la quantité de données à faire transiter entre les machines, ou bien encore le souci d'équilibrer la charge sur la base des puissances respectives des machines clientes et serveur. Il s'agit donc toujours d'un certain *compromis*, efficace dans un certain contexte de besoins.

Le positionnement pertinent de la « frontière », répartissant les responsabilités respectives entre clients et serveur, constitue donc un problème à part entière. Ce problème est résolu de diverses manières, comme nous le verrons dans la suite, avec une multitude de solutions, très variables selon le contexte et la technologie considérée. La figure ci-dessous illustre ce problème de séparation entre entités clientes et serveur, qui vise à déterminer les tâches respectives associées à l'un et l'autre des composants en interaction.

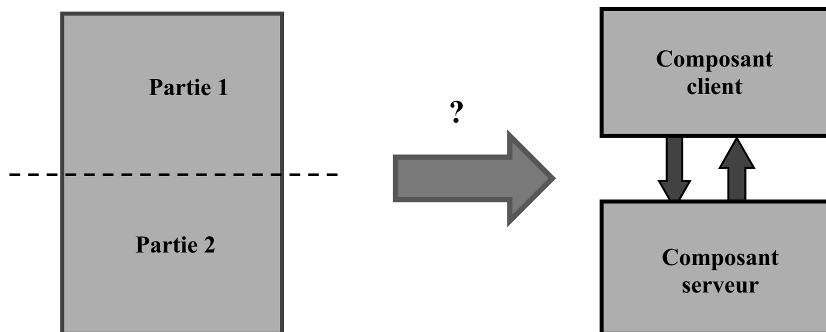

Figure 4. Illustration du choix de frontière entre client et serveur.

Il est évident qu'en fonction du choix opéré quant au positionnement de cette séparation, ce ne sont pas du tout les mêmes données qui seront échangées entre le client et le serveur, et qui donc circuleront sur le réseau de communication. Il y a là clairement un compromis à opérer dans chaque cas entre les aspects afférant aux charges respectives des machines, à la réduction de la quantité de données transitant sur le réseau (que l'on cherche à rendre la plus faible possible), et enfin à la minimisation du couplage entre composants (visant à faciliter l'évolutivité). Nous allons examiner ici quelques solutions typiques apportées à ce problème, telles que mises en œuvre dans quelques approches technologiques classiques.

Afin d'éclairer au préalable cette problématique, nous considérerons ici que le code de tout système logiciel[1] assure naturellement plusieurs responsabilités, et que l'on peut donc, en principe, trouver dans le code plusieurs grandes parties afférant à celles-ci. Ainsi, on trouvera une certaine partie du code en charge de l'interaction avec les usagers. C'est la partie dite *Interface Homme-Machine,* ou en abrégé I.H.M. Elle sera en charge de la *présentation*, c'est-à-dire de la gestion des éléments d'interaction présentés à l'écran, mais aussi du *contrôle du dialogue*, c'est-à-dire de l'implémentation des règles présidant à la

[1] Nous ne ferons pas ici de différence entre la notion de *système logiciel* et celle d'*application*, et emploierons ici les deux termes de manière indifférente.