Architecture Client/Serveur

***Chapitre 1***

***Introduction à la Répartition***

***Thierry Desprats / Patrice Torguet***

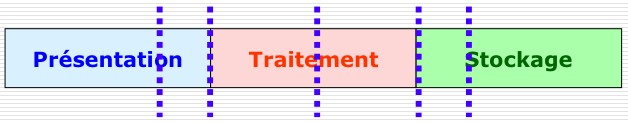
# Introduction

Un système réparti est un ensemble composé d’éléments reliés par un canal de communication et partageant au moins une partie de l’état global du système.

Une application est dite répartie dès qu’au moins deux processus différents s’exécutant sur des sites distants doivent communiquer pour accomplir une tâche globale demandée par un utilisateur.

Le schéma suivant représente une application typique composée de 3 parties :

* la partie présentation : Interface Homme Machine qui permet d’interagir avec l’utilisateur et qui peut être graphique ou textuelle ;
* la partie traitement : qui représente tout le code « métier » de l’application et qui traite les informations données par l’utilisateur et celles précédemment stockées pour produire des résultats intéressants à afficher et à stocker ;
* la partie stockage : qui permet de rendre persistante (entre deux exécutions de l’application) une partie des données de l’application (dans des fichiers, dans une base de données…).



Pour une application répartie on peut découper ces différentes parties à plusieurs endroits comme indiqués ci-dessus.

Par exemple :

* la partie présentation peut s’exécuter sur un poste client alors que tout le reste s’exécute sur un serveur distant. C’est le modèle de ce qu’on appelle généralement le client léger.
* La partie stockage peut s’exécuter sur un serveur distant alors que le reste de l’application s’exécute sur le poste client. C’est le modèle des applications utilisant des Systèmes de Gestion de Bases de Données en réseau.

Dans le cadre de ce cours on s’intéressera plus aux cas où c’est la partie traitement qui est répartie et donc on coupe l’application au milieu sur le schéma.

Cette introduction est basée sur les sources suivantes :

* le cours d’applications réparties de la MIAGE de Toulouse, de Thierry Desprats et Emmanuel Lavinal, Equipe SIERA, IRIT, Université de Toulouse.
* Le livre « Distributed Systems », édité par Sape Mullender, Addison-Wesley, ISBN : 0-201-62427-3.
* Le livre « Distributed Systems – Concepts and Design » de George Coulouris, Jean Dollimore et Tim Kindberg, Addison-Wesley, ISBN : 0-201-62433-8.
* Le livre « CORBA – Des concepts à la pratique » de Jean-Marc Geib, Christophe Gransart et Philippe Merle, 2ème édition, Dunod, ISBN : 2-10-004806-6.

# Intérêts de la répartition

Historiquement le premier intérêt de la répartition a été de partager des ressources. Ces ressources peuvent être matérielles, comme un ordinateur, une imprimante, un lecteur de bandes (ancêtres des disques durs pour stocker des informations) ou immatérielles comme des fichiers, un ensemble de données (base de données), des vidéos, une interface graphique, une messagerie...

Ainsi on a rapidement connecté plusieurs terminaux sur un même ordinateur pour le rendre accessible à plusieurs personnes. De même les premiers réseaux locaux servaient à partager une imprimante ou des serveurs de fichiers entre plusieurs ordinateurs personnels.

Assez rapidement on s’est aussi rendu compte qu’on pouvait répartir les traitements d’une application pour accélérer son exécution. Ainsi, les premiers logiciels répartis permettaient de réaliser du calcul numérique sur plusieurs machines. On pouvait, par exemple, faire des opérations matricielles réparties pour gérer des matrices de très grande taille.

Un autre intérêt des systèmes répartis est lié à la sûreté opérationnelle. En effet, étant donné qu’un système informatique peut avoir des pannes, il est intéressant de dupliquer des traitements ou du stockage sur plusieurs machines pour conserver une système utilisable même si un des ordinateurs tombe en panne ou si un processus de l’application répartie s’arrête anormalement.

Enfin, plusieurs avantages sont liés à la composabilité des applications réparties. C’est à dire que l’application est découpée en plusieurs composants qui peuvent s’exécuter sur plusieurs machines. Ainsi, dans certains cas, cette composabilité est intrinsèque à l’application. Si on prend l’exemple d’une application spatiale avec un segment embarqué (dans un satellite par exemple) et un segment sol, il est clair que l’application ne peut pas s’exécuter sur la même machine. C’est aussi le cas de la majorité des applications multi-utilisateur si ces utilisateurs sont géographiquement dispersés. La composabilité peut aussi servir à intégrer plusieurs applications existantes qui s’exécutent sur des machines différentes en un tout applicatif. La composabilité peut aussi avoir des avantages au niveau génie logiciel en rendant une application modulaire et plus évolutive (on peut redévelopper uniquement un seul module).

# Classification d’applications réparties

## Coordination d’activités

Un premier type d’application répartie est celui qui vise à répartir les diverses fonctionnalités d’une application monolithique (c’est à dire non répartie) et à **coordonner** les activités de ces diverses taches. On parlera dans ce cas de traitements répartis.

On peut aussi avoir l’inverse : plusieurs applications que l’on intègre en une même méta application répartie. Un exemple de cela est EAI (Entreprise Applications Integration : Intégration des Applications d’Entreprise) qui intègre diverses applications visant à gérer une entreprise : comme les applications de gestion des stocks, des ressources humaines (paie notamment), des clients (CRM : customer relationship management).

## Collecticiels

Ici on va trouver toutes les applications impliquant **un ensemble d’utilisateurs**.

Il peut s’agir d’édition coopérative (google documents, agenda partagé…) ou d’ingénierie concurrente (gestion du code source, CAO…).

Mais aussi de toutes les applications de téléconférence (audio ou vidéo) ou de téléenseignement.

Enfin on citera aussi les jeux multi-participants (des plus simples jeux de cartes en ligne jusqu’au jeux massivement multi-joueurs comme World Of Warcraft).

## Temps réel

Ici on va trouver toutes les applications visant à contrôler ou superviser un ensemble de systèmes électroniques ou électromécaniques qui fonctionnent en temps-réel (usine robotisée par exemple).

On distingue en général les systèmes qui fonctionnent en :

* temps réel dur (hard) : systèmes embarqués critiques (échéances en quelques milli secondes) qui peuvent contrôler des avions ou des voitures ;
* temps réel souple (soft) : le reste : réseaux de capteurs, informatique ambiante, géolocalisation (échéances en quelques secondes ou quelques minutes).

Exemples illustrant l’informatique ubiquitaire/omniprésente :

* Services sur téléphones mobiles (réseaux sociaux…)
* Services sur terminaux spécifiques : technicien « gaz », « expertise assurance »,… (l’équipement mobile réalisant des mesures ou un diagnostic va communiquer avec un serveur de l’entreprise).

On en arrive au concept de **système ambiant**. Il s’agit d’un ensemble d’entités plus ou moins petites qu’on dote de capacités de calcul et qui interagissent entre elles. Cela utilise une nouvelle logique : le système s’adapte à l’utilisateur / à l’environnement, et non l’inverse comme c’est le cas classiquement.

Exemples :

- Domotique : capteurs température, lumière, présence,… adaptation du système de chauffage / refroidissement, contrôle des volets.

- Assistance des personnes défaillantes (handicaps) : guide d’itinéraire pour mal voyants.

## Virtualisation de ressources et d’activités

Il s’agit ici de rendre virtuels des objets ou des activités réelles pour les améliorer. On peut citer par exemple le commerce électronique, le partage d’informations (annuaires en ligne par exemple) mais aussi les systèmes répartis gérant les médias électroniques (e-books, musique dématérialisée, vidéo à la demande), les bibliothèques numériques, ou les musées virtuels…

# Exigences

Nous allons ici faire un panorama des exigences pour construire une application ou un système répartis.

Certaines sont spécifiques aux systèmes répartis (comme la désignation ou l’interopérabilité) d’autres sont plus générales comme l’administration, la sécurité ou la persistance.

## Organisation, Architecture

Organiser la façon doit on va concevoir notre application : entités distribuées, leurs rôles, leurs interactions. Les **relations** entres les **rôles** vont nous permettre de déduire des **interactions**.

A partir de là on va en déduire une architecture logicielle.

Il existe de nombreux modèles ou paradigmes utilisés pour décrire les principes régissant les interactions des éléments fonctionnels composant une application répartie et parfois la nature conceptuelle de ceux-ci.

Exemples :

* modèle client/serveur dans lequel une entité (le client) envoie un message appelé une requête à une autre entité (le serveur) qui va réaliser un traitement sur cette requête et renvoyer une réponse (résultat) à la première entité (et qui sera vu en détail dans la suite de ce cours) ;
* architecture n-tiers qui ajoute d’autres tiers à l’architecture client/serveur (ou 2 tiers) la plus utilisée étant l’architecture trois tiers (client / serveur de traitements / serveur de base de données) ;
* P2P (peer to peer ou pair à pair) qui mélange les rôles de client et de serveur ;
* producteur/consommateur dans lequel des entités produisent des informations qui sont ensuite consommées (pour traitement) par d’autres entités (on pourra distinguer le modèle prod/conso à flots discrets qui travaille avec des messages unitaires ou à flots continus : multimédia) ;
* Objets partagés dans lequel des entités travaillent sur un ou plusieurs objets partagés en prenant soin de la consistance des données contenues dans ces objets (en utilisant par exemple des verrous répartis) ;
* Lecteurs/Rédacteurs qui est un cas particulier des objets partagés pour lesquels les entités lecteurs peuvent travailler en parallèle alors que les rédacteurs sont mutuellement exclusifs ;
* Code mobile qui permet de faire migrer des taches ;
* Agents, sociétés (Systèmes Multi Agents) qui s’inspirent du vivant pour résoudre des problèmes complexes…

## Désignation/référence/liaison

La **désignation** ou l’**identification** consiste à associer un nom ou un identifiant à une entité (atomique ou non) selon des règles de nommage.

Ceci peut se réaliser d’une manière unitaire (un nom différent pour chaque entité) ou collective en utilisant par exemple des hiérarchies de nommage comme pour le DNS (Domain Name System qui régit les noms de machines dans l’Internet).

La **référence** est un identificateur qui permet l’accès physique à une entité. Elle en général composée de :

* l’adresse réseau de la machine sur laquelle se trouve la machine (par exemple une adresse IP) ;
* le numéro de port sur la machine ;
* une localisation interne (adresse mémoire par exemple).

La **liaison** (ou binding)consiste à associer la désignation d’une entité et la référence à sa localisation physique.

Ceci peut être fait dans :

* des référentiels ;
* des annuaires (pages blanches) ;
* des courtiers (pages jaunes) ;

Les courtiers permettent de trouver une entité en précisant les services fonctionnels qu’elle peut offrir ou encore une qualité de service spécifique (une entité offrant un service de type « best effort » alors qu’une autre peut offrir un service fiable).

## Interopérabilité

L’interopérabilité est la capacité que possèdent deux entités informatiques hétérogènes à fonctionner conjointement, grâce à l’utilisation de langages et de protocoles communs, et à donner accès à leurs ressources de façon réciproque.

L’interopérabilité est dite technique ou syntaxique si les entités sont capables de communiquer et d’échanger des données (elles utilisent un format commun comme par exemple le code ASCII pour les caractères). Cette interopérabilité est le préalable aux formes plus poussées. Elle se heurte aux différences d’architectures des machines (processeurs little-endian qui codent les mots de 16, 32 et 64 bits en commençant par l’octet de poids faible ou big-endian qui fait l’inverse), des langages de programmation (matrices lignes/colonnes en C et colonnes/lignes en Fortran), des systèmes d’exploitation (codage des retours à la ligne sous Windows – CRLF – et sous Linux – LF), des codecs multimédia...

L’interopérabilité est dite sémantique si l’entité qui reçoit les données est capable d’interpréter automatiquement les informations échangées de façon significative et avec précision afin de produire des résultats utiles comme défini par les utilisateurs finaux du système réparti.

En général l’interopérabilité est basée sur une représentation commune. C’est à dire qu’on adopte une syntaxe mutuellement compréhensible lors du transfert d’information entre entités.

C’est le rôle de la couche présentation du modèle OSI de l’ISO.

Sans cela on est obligé de réaliser des traducteurs 1 vers 1. Et donc si on a N entités qui communiquent on peut avoir à réaliser N\*(N-1) traducteurs.

L’utilisation d’un langage pivot est bien meilleure puisqu’on aura au plus 2\*N convertisseurs (vers/depuis ce langage pivot).

Dans ce cas on parle de sérialisation/désérialisation. Qui sont des opérations qui consistent à coder (et respectivement décoder) un flux d’informations représenté localement à l’aide d’une syntaxe propriétaire vers/depuis une syntaxe commune de transfert.

## Synchronisation/Coordination

Ici on s’intéresse au degré de couplage entre le fonctionnement des deux entités interagissant.

Ce couplage peut être :

* Synchrone : l’initiateur de l’interaction suspend son exécution pendant tout le temps du déroulement de celle-ci ; il ne la reprendra qu’après la réception d’un compte-rendu lui indiquant une terminaison.
* Asynchrone : l’initiateur de l’interaction poursuit son exécution indépendamment de celle de l’interaction.
* Semi-synchrone : l’initiateur poursuit son exécution mais récupère plus tard le compte rendu de l’interaction.

Attention : ici on ne s’intéresse qu’à la synchronisation lors des interactions. On peut aussi parler d’aspects plus complexes comme les problèmes d’inter-blocage, de synchronisation d’horloge, de workflow…

## Sécurité/sûreté

Ici on peut soit s’intéresser globalement à l’application qui pourra présenter des propriétés de :

* confidentialité (aucun accès illicite n’est possible) ;
* intégrité (aucune altération des données n’est possible) ;
* disponibilité (elle offre un accès continu et non dégradé au service qu’elle rend).

Pour cela, on va s’appuyer sur des services :

* d’authentification/identification (qui s’assure que les entités qui communiquent sont bien celles qui font partie du système et pas des entités externes espionnes par exemple) ;
* de confidentialité (qui s’assure que les messages qui circulent dans le systèmes ne soient pas compréhensibles par des entités externes) ;
* d’intégrité (qui s’assure que les messages ne puissent pas être modifiés) ;
* de non répudiation (qui s’assure qu’une entité qui a envoyé un message ne puisse pas faire croire que ce n’est pas elle qui l’a envoyé) ;
* de contrôle d’accès (qui empêche certaines entités d’accéder à certaines ressources ou données) ;
* de traçabilité (qui permet de conserver à chaque instant qui fait quoi dans le système).

Ces services utilisant des mécanismes comme :

* le chiffrement (codage d’un message à l’aide d’une clé de façon à ce que le décodage sans la clé soit très long ou très compliqué) ;
* la certification (ajout d’information à un message pour le contrôler lors de sa réception) ;
* le filtrage ;
* le contrôle d’accès ;
* la journalisation ;
* la duplication…

## Cohérence/persistance

On s’intéresse ici aux problèmes liés aux arrêts non désirés et autres indisponibilités des connexions réseaux et des matériels. Dans ce type de cas, il est important de s’assurer que la nature répartie de l’application n’engendre pas de problèmes d’incohérences ou de perte d’information. Par exemple, si on a une application qui transfère une somme d’argent d’un compte géré par une machine vers un autre géré par une deuxième machine, que se passe-t-il si l’une des deux tombe en panne ? Si on a d’abord enlevé la somme du premier compte et que la seconde machine pose problème on peut avoir une perte d’argent, à l’inverse si on commence par ajouter la somme sur le second compte et que la première machine s’arrête on peut dupliquer de l’argent.

Dans ce cadre on peut tout d’abord citer des propriétés nommées ACID :

* Atomicité : l’action (pouvant être répartie comme dans l’exemple ci-dessus) est réalisée entièrement ou ne l’est pas ;
* Cohérence : les données (qui peuvent être dupliquées) demeurent cohérentes ;
* Isolement : pas de répercussion en cas d’échec ;
* Durabilité : les actions entreprises sont durables dans le temps.

Pour obtenir ces propriétés on peut s’appuyer sur des mécanismes :

* CCR (Commitment, Concurency and Recovery) :
  + Engagement / Reprise : soit on a un engagement (commitment) sur une action qui n’a pas été interrompue et donc qui s’est exécutée jusqu’au bout ; soit on a un retour à l’état initial (rollback) ;
  + Concurrence : gestion des conflits d’accès simultanés à une ressource ;
  + Gestion de sessions, de transactions…
* Persistance : sauvegarde de l’état d’une ressource sur un support non volatile ;
* Redondance : duplication de ressources pour éviter qu’une ressource perturbe le système en cas d’arrêt non programmé (par exemple).

## Passage à l’échelle et Performances

On s’intéresse ici à la performance globale de l’application et à son comportement lorsqu’on essaie d’augmenter le nombre de clients, de requêtes, de traitements, de transactions…

Pour les performances on peut s’intéresser soit :

* à la performance brute c’est à dire au temps (calculé à toutes les étapes) que met le système pour traiter une information, une requête, une transaction…
* à la fiabilité du système c’est à dire à sa faculté à résister aux défaillances partielles.

Dans le premier cas on peut mettre en place des mécanismes d’équilibrage des charges (on s’assure que toutes les ressources de calcul sont utilisées à peu près pareil) ou de réplication (on va dupliquer les ressources qui sont le plus utilisées).

Dans le second cas on s’intéresse à la tolérance aux pannes qui est essentiellement basée sur la réplication d’un grand nombre de ressources de façon à ce que la panne d’une des ressources soit compensée par une autre ressource identique.

Ensuite, pour que la performance dure dans le temps on va s’intéresser au passage à l’échelle (scalability) qui correspond à la capacité de croissance d’un système.

Cette croissance peut concerner :

* les sites ou machines qui composent le système en s’intéressant à leur densité et à leur distribution géographique (en règle générale un système marche mieux si on a un grand nombre de ressources de calcul au même endroit que si on en a moins ou le même nombre mais réparties dans le monde entier) ;
* les utilisateurs ;
* le volume des flux d’information ;
* la fréquence des interactions.

Ici aussi les mécanismes utilisés sont essentiellement basés sur la réplication de ressources.

En général il est difficile de combiner une croissance importante sans dégradation des performances.

Exemple d’application qui a bien géré la montée en puissance : le DNS (Domain Name System) qui gère la traduction des noms de machines sur Internet en adresses IP.

## Dynamisme, Adaptabilité et Administrabilité

Ici on s’intéresse d’une part au dynamisme de l’application répartie. Est-elle une composition évolutive d’entités ? Exemple : partage de fichier en P2P avec ed2k/eMule pour lequel des milliers d’applicatifs se connectent et se déconnectent en permanence. Chaque applicatif apportant un certain nombre de fichiers partagés qui sont référencés sur des serveurs ou de façon complétement répartie (en utilisant KAD) ces fichiers pouvant être téléchargés par n’importe quel autre applicatif du système après une recherche elle même répartie.

Dans ces cas là on peut aussi avoir une hétérogénéité très forte dans l’application (plusieurs versions d’un même applicatif, plusieurs applicatifs compatibles, des machines très puissantes et des netbooks…).

D’autre part on peut considérer l’adaptabilité de l’application. Peut-elle gérer des conditions variables d’exécution (adaptation à la bande passante disponible, par exemple, en limitant les échanges lorsqu’un terminal est connecté via un réseau cellulaire et en augmentant ces échanges lorsqu’on se connecte à un réseau WiFi) ? Est-elle personnalisable (l’utilisateur pouvant par exemple choisir de faire fonctionner l’application en tâche de fond quand il n’utilise pas sa machine) ? Ceci nous rapprochant des applications mobiles et ubiquitaires.

Enfin on s’intéresse aussi à l’administrabilité de l’application. Est-il possible de la configurer pendant son exécution et éventuellement à distance ? Peut-on complètement la superviser en obtenant des journaux et/ou des statistiques précises de ce qui se passe dans l’application en temps-réel ? Est-il possible de faire de la maintenance préventive ou corrective pendant l’exécution de l’application (par exemple en ajoutant des serveurs lors d’une montée en charge) ?

# Conclusion

Nous espérons que cette introduction à la répartition vous en aura fait apprécié à la fois les avantages et la complexité. Cette complexité se ressent à la fois lors du développement, du déploiement et de l’administration de telles applications réparties.

Pour atténuer cette complexité il existe de nombreux outils, méthodes et modèles. Ils peuvent être applicable lors du développement :

* en conception (MDA, approche dirigée par les modèles, qui permet de simplifier la conception et la réalisation d’applications en les décomposant en modules plus simples par exemple) ;
* lors de la programmation (par exemple RMI, remote method invocation, présenté à la fin de ce cours, qui permet de simplifier la réalisation d’application réparties en Java) ;
* lors des tests (Apache JMeter permet par exemple de tester des applications web).

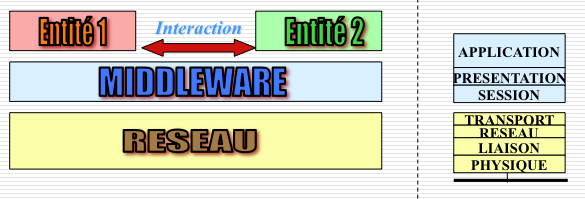
D’autres peuvent s’utiliser lors de l’exécution :

* pour déployer une application répartie (il existe, par exemple, des outils permettant de déployer des antivirus sur un ensemble de machines en vérifiant qu’ils sont toujours à jour) ;
* pour l’administrer (JMX, Java Management eXtensions, permet par exemple de rajouter dans une application distribuée des mécanismes permettant de tracer ce qui s’y passe et aussi de modifier à distance des paramètres des différentes entités grâce à une console d’administration).

Ces outils peuvent aussi aider à intégrer des évolutions technologiques comme le web (en rendant une application accessible via une interface web par exemple), le multimédia, le sans fil, les grilles de calcul, la géo-localisation, les réseaux de capteurs…

Ils peuvent aussi satisfaire de nouveaux besoins comme la mobilité, l’ubiquité, le temps-réel, la consommation énergétique…

Les intergiciels (middleware) sont de tels outils. On peut les définir comme des couches de logiciel (répartis) offrant aux applications un complément de services du réseau pour supporter leurs interactions tout en facilitant la mise en œuvre de la répartition. Dans le modèle OSI ils interviennent essentiellement aux niveaux session et présentation.



Ils ont pour objectif de :

* fournir des interfaces uniformes (standardisées) de haut niveau pour faciliter la programmation des applications distribuées ;
* masquer l’hétérogénéité des systèmes matériels et logiciels sous-jacents (et donc favoriser l’interopérabilité) ;
* rendre la répartition la plus « transparente » possible ;
* fournir des services communs supportant les exigences inhérentes à la répartition (synchronisation, transactions, sécurité, etc.).

Dans ce cours nous présenterons un intergiciel simple : RMI.