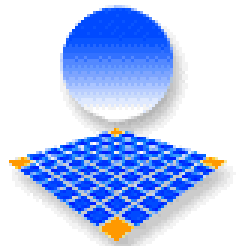

Architectures n-tiers

Intergiciels à objets et services web

Clémentine Nebut

Clementine.nebut@lirmm.fr



Introduction

- Applications distribuées (ou réparties)
 - Définition : une application distribuée est :
 - un ensemble de programmes,
 - distribués sur un réseau de communication,
 - qui collaborent pour assurer un service.
 - Exemples :
 - Une grappe de calculateurs
 - Une application de commerce en ligne
 - Un calendrier partagé
 - ...

Introduction

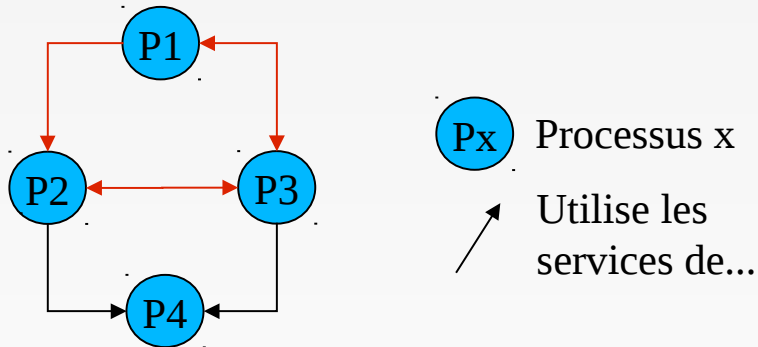
- Pourquoi des applications distribuées ?
 - Besoin intrinsèque de l'application
 - Les utilisateurs sont répartis (ex : site web)
 - Les données sont réparties (ex : stations météo)
 - Partage de données/informations (ex :P2P)
 - Mais aussi
 - Besoin de performances (ex : grappe de calcul)
 - Besoin de « disponibilité » (ex : redondance)
 - Besoin de modularité (ex : découplage gestion client / gestion personnel)
 - Utilisation de services externes
 - Et bien sûr toutes les combinaisons possibles

Architectures classiques

- Deux grands types d'architectures
 - Couplage fort
 - Ex : Architecture « peer-to-peer »
(tous les processus ont le même rôle)
 - Grappes de calculateurs
 - Couplage faible
 - Architecture client/serveur
 - Architecture N-tiers
 - Architecture 3-tiers
 - Couplage très faible
 - Architecture orientée services

Couplage fort

- Inter-dépendance entre les composants de l'application

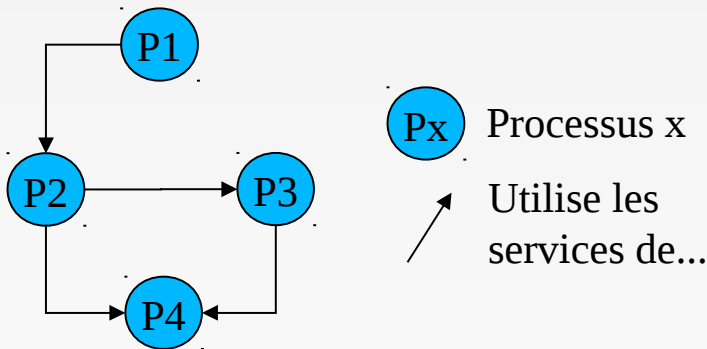


Il existe au moins un cycle dans le graphe de dépendances entre les composants de l'application

- Ce type d'architecture pose problème tant pour le développement que pour la maintenance.
A éviter autant que possible

Couplage faible

- Pas d'inter-dépendance entre les composants de l'application

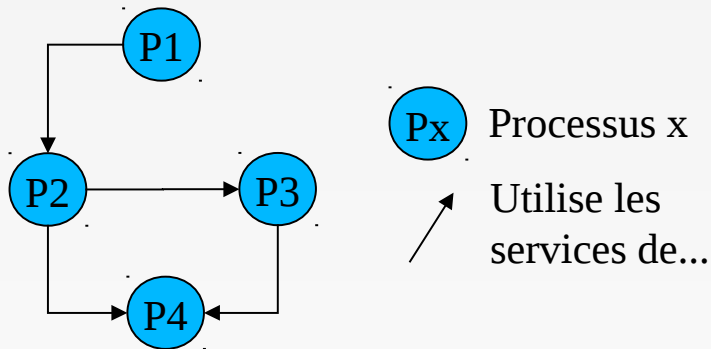


Pas de cycle dans le graphe de dépendances entre les composants de l'application

- Permet de maîtriser la complexité de l'architecture :
 - Pour le développement
 - Pour le test
 - Pour la maintenance

Couplage très faible

- Les composants sont remplaçables, conçus en indépendance, avec des technologies diverses



Un processus survit à la déconnexion d'un autre processus
Un processus est facilement remplaçable par un autre rendant les mêmes services

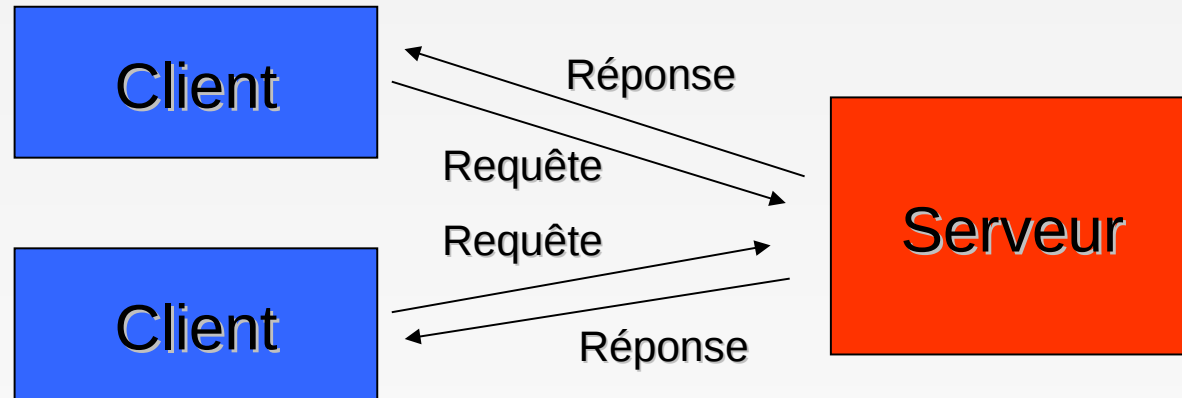
- Permet de faciliter la construction/maintenance :
 - Services sous-traités
 - Construction par assemblage

Couplage

Pressman R. S., Software Engineering: A Practitioner's Approach, 3rd Edition. McGraw-Hill. Ch. 10, 1992

- Sans couplage : pas d'échange d'information.
- Par données : échange par des méthodes avec arguments/paramètres de type simple.
- Par paquet : échange par des méthodes avec des arguments de type composé (structure, classe).
- Par contrôle : les composants se passent ou modifient leur contrôle par changement d'un drapeau (verrou).
- Externe : échange par un media externe (fichier, pipeline, lien de communication).
- Commun (global) : échange via un ensemble de données (variables) commun.
- Par contenu (interne) : échange par lecture/écriture directe dans les espaces de données (variables) respectifs des composants.

Architecture Client/Serveur

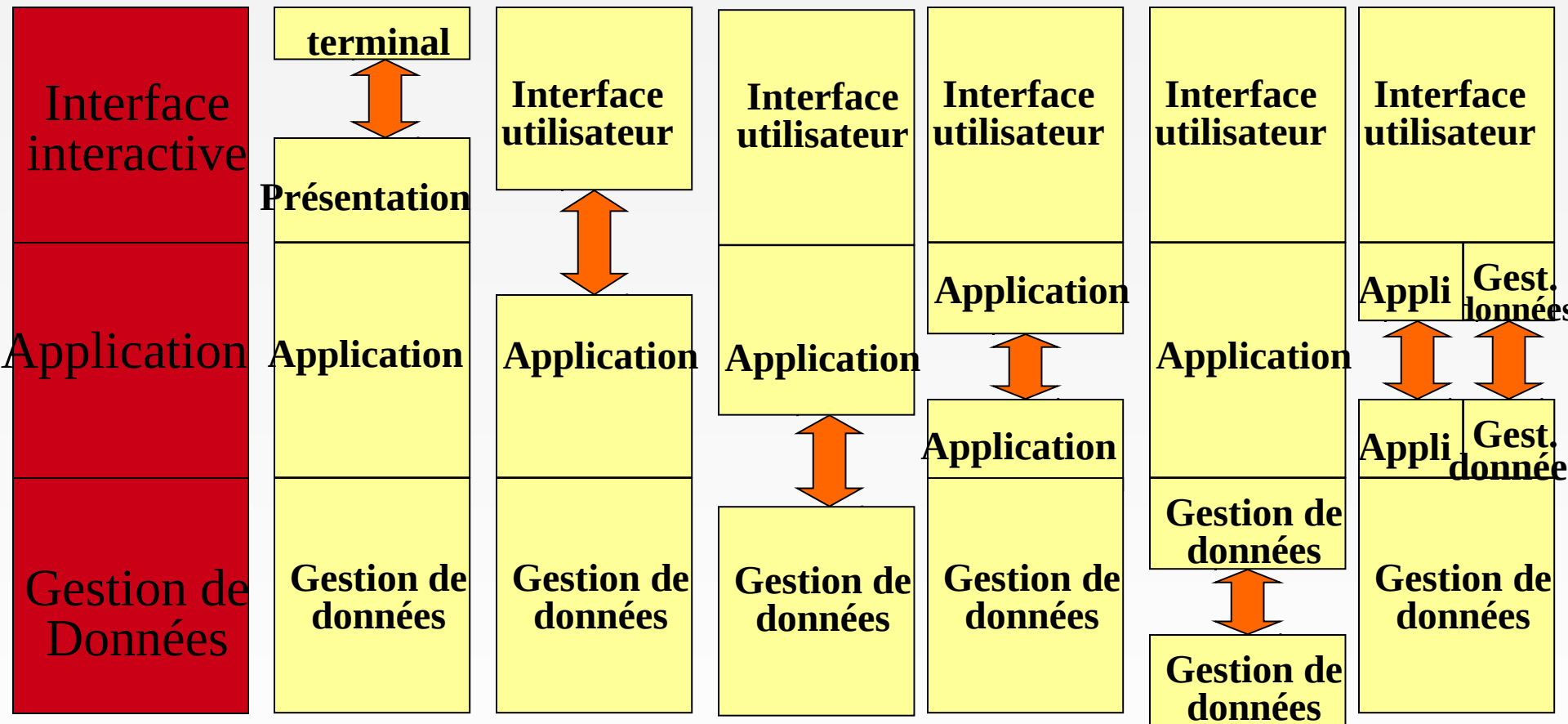


- Deux types de noeuds
 - Un serveur
 - Des clients

Architecture client-serveur

- Un client fait une requête
- Et reçoit une réponse du serveur
- Notion de session
 - ensemble des requêtes et réponses pour un même client
 - nécessite l'identification du client
 - ex: session yahoo, session telnet, session ftp...

Classification du Gartner roup



Architecture Client/Serveur

- Exemples :
 - Client FTP/Serveur FTP
 - Terminal X/Serveur d'exécution
 - Navigateur Web/Serveur de noms
 - Navigateur Web/Serveur Web
 - ...
- La majorité des architectures sont construites autour du modèle client serveur

Architecture Client/Serveur classique

- Points forts
 - Couplage assez faible (le serveur n'a pas besoin des clients)
 - Maintenance et administration du serveur simplifiées
 - Souplesse : possibilité d'ajouter/supprimer dynamiquement des clients sans perturber le fonctionnement de l'appli
 - Les ressources sont centralisées sur le serveur
 - Sécurisation simple des données : 1 seul point d'entrée
 - Pas de problème d'intégrité/cohérence des données
- Points faibles
 - Un maillon faible : le serveur
 - Coût élevé : le serveur doit être très performant pour honorer tous les clients

Architecture Client/Serveur

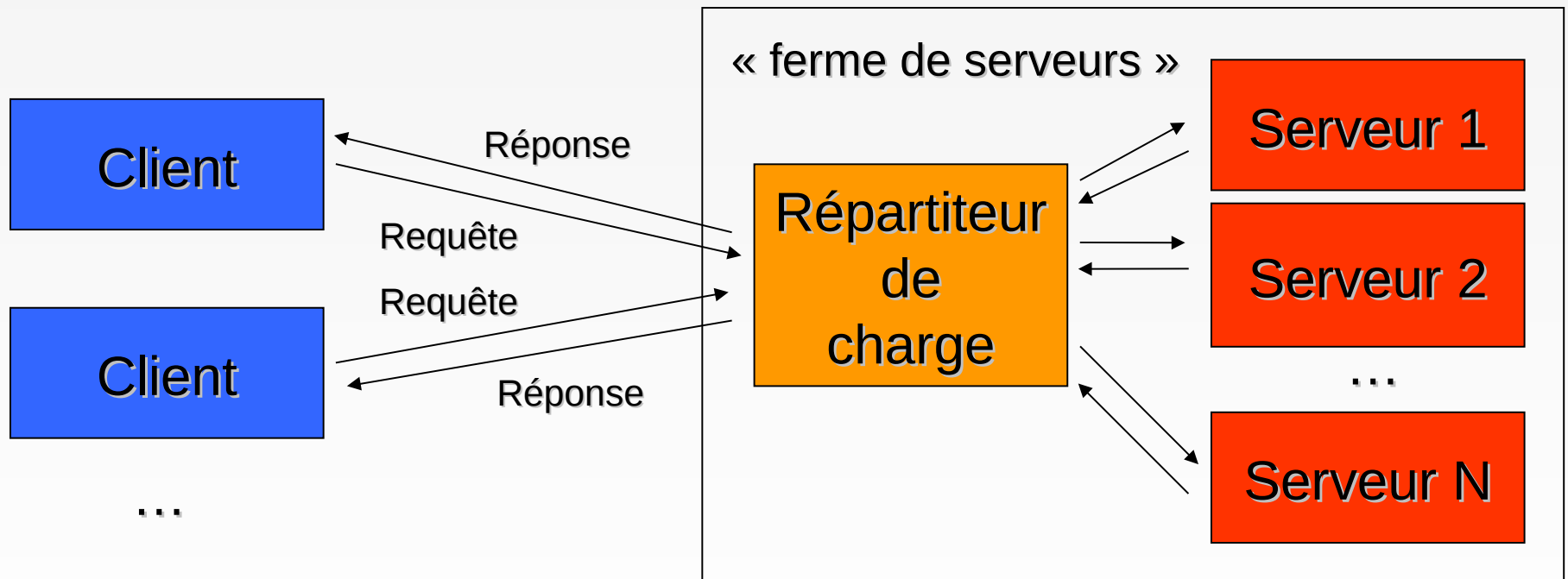
- Architectures de plus en plus complexes :
 - Application de commerce en ligne
 - Radio/télévision numérique interactive
 - Moteur de recherche...
- Dans le modèle client/serveur toute la complexité est concentrée dans le serveur
 - Problème de performance/disponibilité
 - Répartition de charge (load balancing)
 - Problème pour maîtriser la complexité
 - Architectures multicouches (n-Tiers)

Répartition de charge (load balancing)

- Un serveur traite simultanément les requêtes de plusieurs clients
- Les requêtes de deux clients sont indépendantes
- « load balancing » : paralléliser sur plusieurs serveurs identiques s'exécutant sur des machines différentes le traitement des requêtes concurrentes

Répartition de charge

Les requêtes des **clients** passent par un **répartiteur de charge** qui les répartit sur **N serveurs** identiques.



Le répartiteur de charge

- Rôle principal : diriger les requêtes des clients en fonction de la charge de chacun des serveurs
- Rôles annexes
 - Gérer les sessions des clients (2 solutions)
 - Toutes les requêtes d'un client sont dirigées vers un seul serveur
 - Les données de session sont transmises avec la requête
 - Ex: gestion des données concernant un client
 - Gérer les défaillances :
 - Ne plus diriger de requêtes sur un serveur « crashé »
 - Assurer le passage à l'échelle (scalabilité)
 - Permettre l'ajout et le retrait de serveurs sans interruption de service

Répartition de charge

- Points forts
 - Transparent pour les clients
 - Scalable : nb de serveurs adaptable à la demande
 - Tolérant aux défaillances : la défaillance d'un serveur n'interrompt pas le service
 - Plus besoin de machines très chères : en mettre +
- Point faible
 - Les données ne sont plus centralisées mais dupliquées
 - Le répartiteur de charge devient le « maillon faible »

Répartition de charge : Exemple

Le site web de yahoo est hébergé simultanément sur plusieurs serveurs web d'adresses différentes

```
$ host www.yahoo.com
www.yahoo.com is an alias for www.yahoo.akadns.net.
www.yahoo.akadns.net has address 216.109.118.70
www.yahoo.akadns.net has address 216.109.118.71
www.yahoo.akadns.net has address 216.109.118.76
www.yahoo.akadns.net has address 216.109.118.77
www.yahoo.akadns.net has address 216.109.118.78
www.yahoo.akadns.net has address 216.109.118.64
www.yahoo.akadns.net has address 216.109.118.66
www.yahoo.akadns.net has address 216.109.118.67
```

Le serveur de noms (DNS) joue le rôle de répartisseur de charge en traduisant « www.yahoo.com » par l'adresse de chacun des serveur web à tour de rôle

Maîtriser la complexité : les archis multicouches/n-tiers

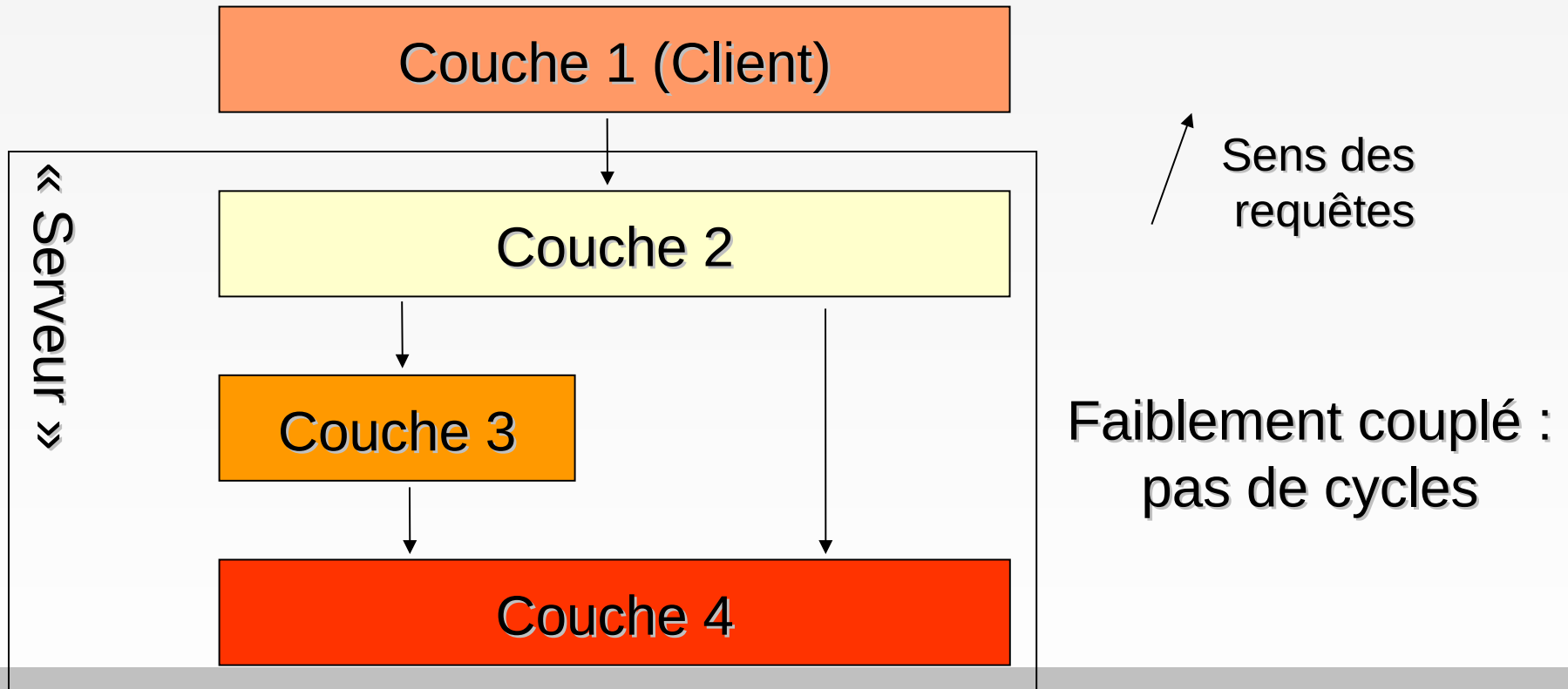
- L'application devient complexe :
 - Difficile à développer
 - Difficile à tester
 - Difficile à maintenir
 - Difficile à faire évoluer
- Solution : les architectures multicouches
 - Inspiré par le développement en couches des protocoles réseaux et des architectures à base de composants

Les architectures multicouches

- Principe
 - Découper l'application en un ensemble de composants (ou couches) fonctionnels distincts et faiblement couplés.
 - Chaque composant est ainsi plus simple et l'application distribuée reste faiblement couplée
 - Les composants communiquent entre eux sur le modèle client/serveur
 - Les composants de l'application peuvent être facilement répartis sur plusieurs machines

Les architectures multicouches

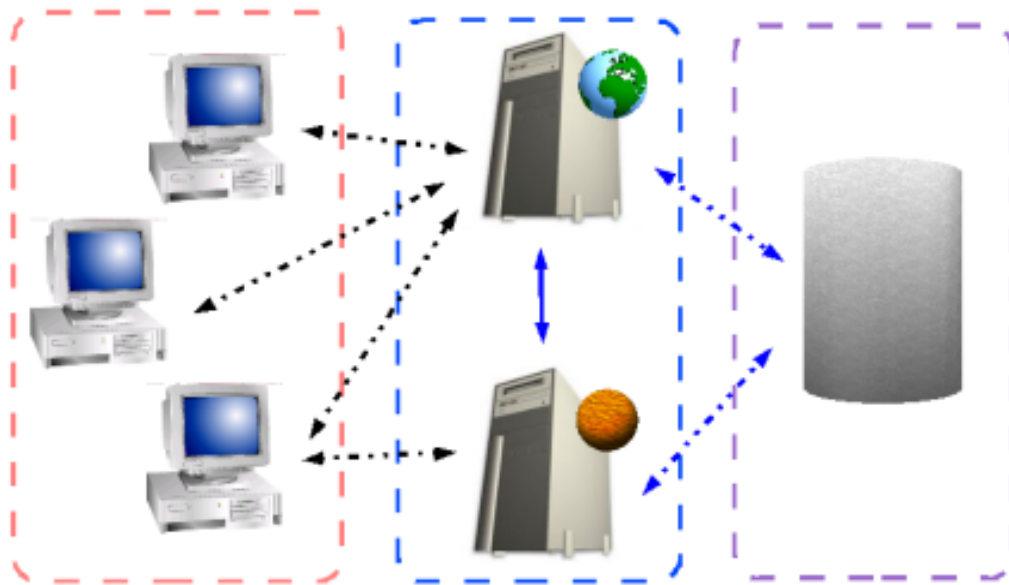
- Chaque couche d'une application multicouches est cliente de ses couches inférieures et serveur pour les couches supérieures.



Les architectures multicouches

- La plupart des applications développées se ressemblent et se composent :
 - De données
 - De traitements (sur ces données)
 - De présentation (des données et des résultats des traitements)

Architecture N-tiers



Tier 1 : Clients Tier 2 : Serveurs (Web – Applications) Tier 3 : Données

➔ On distingue généralement 3 grandes couches dans une application

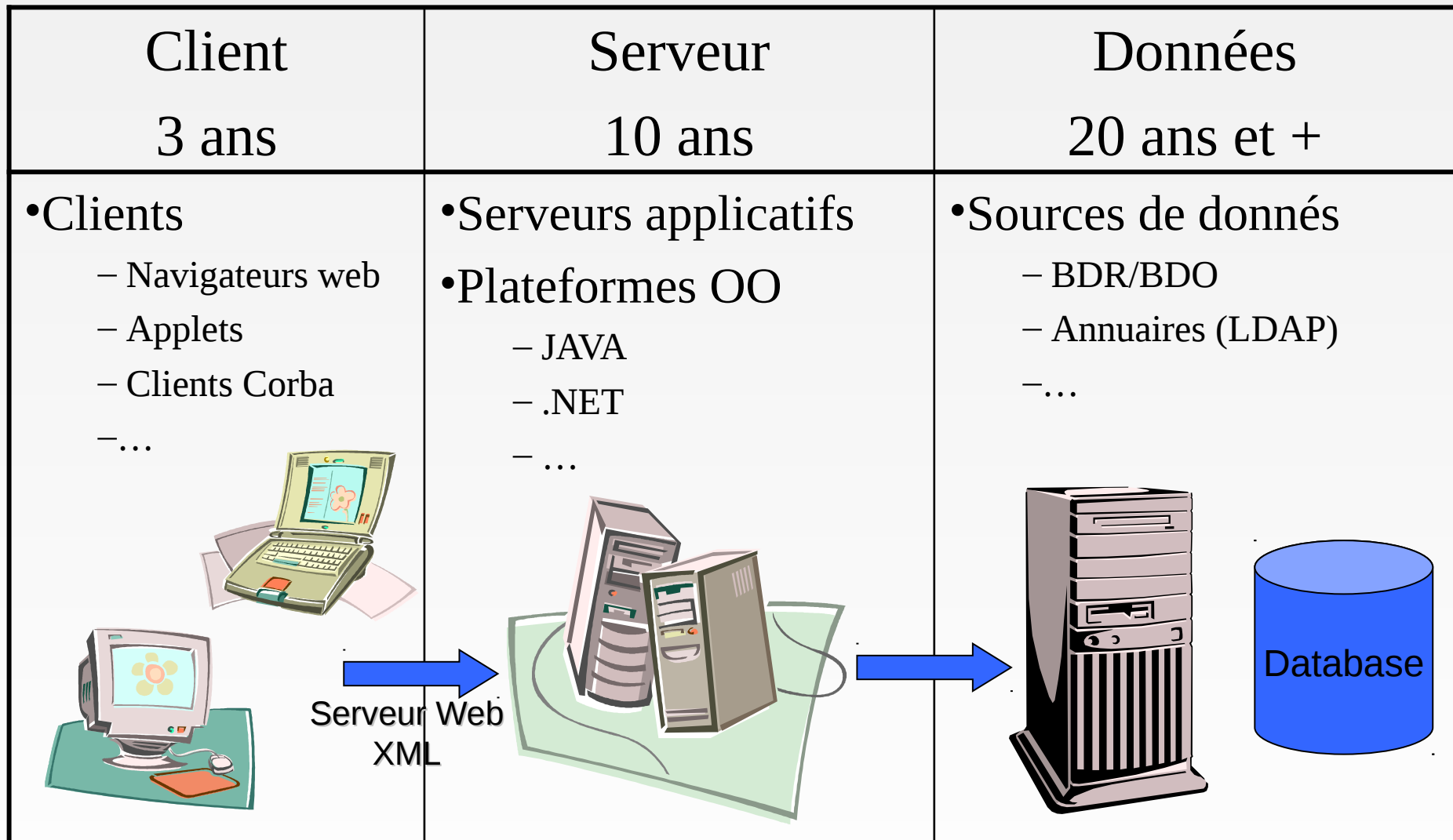
Remarque de vocabulaire

- « Tier »=couche \neq 1/3
- Tier=layer

Les architectures à 3 couches (3-tiers)

- Le cœur de l'application (modèle métier) :
 - modèle objet et traitements propres au domaine de l'application (Analyse et conception OO habituelle).
- Les données persistantes (couche d'accès aux données) :
 - Couche basse faisant le lien entre le modèle métier et stockage physique des données (système de fichier, SGBD...)
- L'interface utilisateur (couche de présentation)
 - Interface permettant à l'utilisateur d'agir sur le modèle métier (interface graphique, interface web...)

Les architectures à 3 couches principales : Exemples de technos



Les architectures à 3 couches

- Points forts
 - Découplage logique applicative/interface
 - Ce ne sont généralement pas les même équipes de développement.
 - Possibilité de changer ou d'avoir plusieurs UI sans toucher à l'application elle-même.
 - Découplage données/logique applicative
 - Possibilité d'utiliser des données existantes sans complexifier le modèle métier.
 - Possibilité de changer le mode de stockage des données sans modifier la logique métier.
 - Favorise la **réutilisation** puisque toute la logique propre à l'application est concentrée dans le modèle métier

4 tiers

- Présentation : clients légers et lourds
- Couche applicative : traitements des règles métier
- Objets métier : objets du domaine, entités persistantes de l'application
- Accès aux données : usines d'objets métiers, création indépendamment du mode de stockage

Architectures 5-tiers

- Présentation : apparence de l'application
- Coordination : équivalent des contrôleurs MVC. Interception des interactions utilisateurs.
- Service : réalisation de cas d'utilisation, exposition des services pour les applications clientes.
- Métier, domaine : objets métier
- Persistance : accès aux données

Multi-couches et Intergiciels

- Intergiciel (middleware)
 - logiciel servant d'intermédiaire de communication entre plusieurs applications, généralement complexes ou distribuées sur un réseau informatique.
- Intergiciels : une définition floue
 - Ensemble de fonctionnalités intégrées (persistance, répartition, ...)
 - Objectif principal : la répartition/distribution
 - Orientation message
 - Orientation RPC

Les architectures multicouches

- Problèmes génériques

- Comment échanger des objets entre différentes machines :

-  gestion de la distribution

- Comment stocker des objets :

-  la persistance

- Comment présenter des données :

-  interface utilisateur

- Comment sécuriser les données et les échanges :

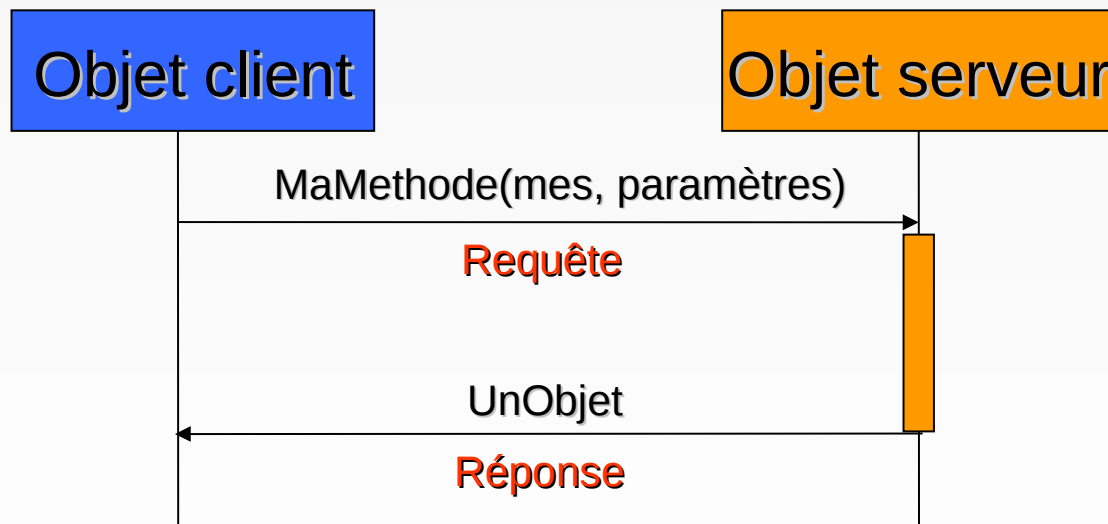
-  cryptage et authentification

Gestion de la distribution

- Communication entre les couches basses
 - Appel de méthode à distance (RPC)
 - Ex : RMI, CORBA, .NET Remoting, Service Web
 - Communication par messages asynchrones
 - Ex : JMS (Java Message System)
- Communication avec le client
 - Appel de méthode à distance (RPC)
 - Ex : Applet JAVA via RMI, Service Web via Soap
 - Par l'intermédiaire d'un serveur web
 - Utilisent toujours http
 - Ex : IIS (Internet Information Services - MS.net), Apache (PHP), Tomcat (Java), ...

Appel de méthodes distantes

- Dans le paradigme objet, les communications sont déjà sur le modèle client serveur :
 - Requête : Appel de méthode
 - Réponse : Valeur de retour



Appel de méthodes distantes

- Communication entre les différentes couches :
Appel de méthode sur des objets distants.

Couche cliente

Objet client

MaMethode(mes, paramètres)

Couche serveur

Objet serveur

Requête

Réseau

UnObjet

Réponse

Appel de méthodes distantes

- Avantages :
 - Les communications entre couches restent à un bon niveau d'abstraction
 - on manipule **toujours** des objets (si on le souhaite)
 - Le mode de communication entre les couches est le même qu'à l'intérieur d'une couche
 - Transparence de la distribution : théoriquement on ne s'en préoccupe pas à la conception mais au niveau du déploiement.
 - Il existe des technologies permettant l'appel de méthodes distantes très simplement
 - RMI (JAVA) : Remote Method Invocation
 - Corba
 - .NET remoting
 - ...

Appel de méthodes distantes

- Comment transférer des instances d'une couche à l'autre (ex: callback avec en retour un objet)
 - La couche qui reçoit un objet doit avoir connaissance de sa classe (2 solutions) :
 - La classe est transmise avec l'objet
 - Charge réseau importante : il faut transmettre tout l'arbre d'héritage
 - Toutes les couches ont une copie des classes qu'elles sont amenées à manipuler
 - Moins souple, mais plus économique pour le réseau

Appel de méthodes distantes

- Comment transférer des instances d'une couche à l'autre
 - Les paramètres (2 solutions)
 - Par valeur : on transmet une copie des objets (sérialisation en binaire)
 - Par référence : on transmet un pointeur sur l'objet.
 - C'est le mode classique des langages OO
 - Utilisation de proxy et stubs pour rendre le passage par référence transparent
 - `a:=d.toto(obj)` -> `a` est une instance du proxy accédant à l'objet distant. Le proxy spécifie l'interface
 - `a.coucou` -> le code de `coucou` est exécuté à distance

Communication par messages

- Communication par messages asynchrones
 - Émission d'une information à une couche inférieure sans besoin de réponse : asynchronisme
 - Exemple : notification d'évènement, déclenchement d'un traitement batch (ex: à minuit envoyer les emails)...
 - Technologies : JMS (Java Message System), ...

Communication par messages

- Message Passing
 - messages envoyés directement d'un processus à un autre (calculs répartis sur des machines parallèles/clusters/grilles)
- Message Queuing
 - messages stockés dans des files de message
 - où ils sont récupérés de façon asynchrone
 - exemple : JMS (Java Message Service), MSMQ
 - applications : en gestion principalement (EAI, Enterprise Application Integration)

Communications par Serveur Web

- Utilisation du protocole HTTP
 - passe partout : Permet de transmettre les données sur de longues distances : le protocole HTTP est toléré par tous les firewalls.
 - compris partout : Tout le monde a un client HTTP : le navigateur web.
 - La gestion de session « web » est prise en charge par le serveur web
- Permet de créer des interfaces utilisateur (HTML)
 - compris partout : Un navigateur web suffit pour utiliser l'application : pas de problème d'administration, de mise à jour des clients !
- Permet à des applications de communiquer (Services Web)
 - Permet de définir et utiliser des API tout en étant indépendant du langage utilisé.

Les architectures multicouches

- Problèmes génériques

- Comment échanger des objets entre différentes machines : gestion de la distribution
→
- Comment stocker des objets :
→ la persistance
- Comment présenter des données :
→ interface utilisateur
- Comment sécuriser les données et les échanges :
→ cryptage et authentication

La persistance

- Stocker de façon permanente les données manipulées par l'application
 - Permettre d'arrêter et redémarrer l'application sans perdre d'informations
 - Partager des données entre plusieurs applications indépendantes
- ➡ Besoin de stocker des instances du modèle métier :
Comment rendre persistant des objets ?

La persistance

- Par exemple :
 - Stockage sur disque des objets
 - Bases de données relationnelles
 - Système de fichiers, xml
 - Bases de données objets
 - ...
- La couche d'accès aux données doit rendre transparente la technologie utilisée pour la couche métier et assure l'intégrité des données

L'intégrité des données

- Assurer que les données stockées sont cohérentes
 - Exemple : Une appli gère des comptes bancaires
 - Les objets persistants « compte bancaire » comportent entre autres les méthodes « débiter(montant) » et « créditer (montant)».
 - La couche d'accès aux données permet de charger et enregistrer des objets « compte bancaire »
 - On souhaite implanter un service de virement bancaire :
 - Charger les deux comptes bancaires
 - Débiter le montant du compte source du virement
 - Créditer le compte destination
 - Enregistrer les comptes avec leur nouveau montant
 - Chacune des étapes peut échouer et conduire à un problème d'intégrité : Compte destination pas crédité, compte source pas débité, ...

➡ La persistance doit s'accompagner d'un mécanisme de transactions

Transaction





- Un mécanisme de transaction permet de rendre atomique un ensemble d'actions
 - Exemple :
 - Début de transaction
 - Charger les deux comptes bancaires
 - Débiter le montant du compte source du virement
 - Créditer le compte destination
 - Enregistrer les comptes avec leur nouveau montant
 - Si tout va bien : Valider la transaction (Commit)
 - Si il se produit une erreur : Annuler la transaction (Rollback)
 - Si il y a une erreur, toute les modifications réalisées dans le corps de la transaction sont annulées.
 - L'état du système reste ainsi cohérent même si une erreur se produit.

Serveur d'application

- Objectifs
 - Prendre en charge les problèmes récurrents rencontrés lors du développement d'applis (Persistance, Accès à distance, transaction, session,...)
- Exemple :
 - Plateforme J2EE
 - Plateforme .NET
- Point fort
 - Permet au développeur de se concentrer sur la logique métier.
- Point faible
 - Comme toute application qui propose de tout prendre en charge, les serveur d'application sont assez difficile à maîtriser...

Les architectures multicouches

- Problèmes génériques

- Comment échanger des objets entre différentes machines : gestion de la
 distribution
- Comment stocker des objets :
 la persistance
- Comment présenter des données :
 interface utilisateur
- Comment sécuriser les données et les échanges :
 cryptage et authentication

Interface utilisateur (couche présentation)

- Interface Web : peu de traitements locaux
 - Client léger (= le même client pour se connecter à tous les services- il n'y a rien à installer chez les clients - (navigateur web: Internet explorer, firefox, ...))
 - Communication via serveur Web
- Programme client dédié à l'application (traitements locaux)
 - Possibilité d'adapter le client à l'application (ex: calendar interne à un réseau local, Objectteering : interface trop complexe)
 - Mode de communications au choix
 - Serveur web (en utilisant des service web)
 - RPC (RMI, CORBA,...)
 - Déploiement et maintien plus difficile

La technologie doit être choisie en fonction du nombre et du statut des utilisateurs.

Les architectures multicouches

- Problèmes génériques

- Comment échanger des objets entre différentes machines :

 gestion de la distribution

- Comment stocker des objets :

 la persistance

- Comment présenter des données :

 interface utilisateur

- Comment sécuriser les données et les échanges :

 cryptage et authentication

Cryptage et Authentification

- Les applis sont généralement multi-utilisateurs et peuvent manipuler des données sensibles
 - Exemple : les numéros de carte de crédit pour les applications de commerce en ligne.
- Il est donc nécessaire de prendre en compte très tôt la sécurité des données et des échanges.
- Plusieurs problèmes
 - La personne qui m'envoie une requête est-elle bien la personne qu'elle prétend être ?
 - **Authentification**
 - Comment être sûr que personne ne va intercepter les messages échangés ?
 - **Cryptage**

Le rôle des intergiciels

- Cacher la répartition
- Cacher l'hétérogénéité des différentes parties impliquées
- Fournir des interfaces de haut niveau pour faciliter le développement et l'intégration
- Fournir des services communs

Avantages/ Inconvénients des intergiciels

- Cachent le bas niveau (ex : implémentation de la communication à base de sockets)
- Assurent l'indépendance : langages et/ou plateformes
- Perte de performance liée à la traversée des couches
- Architectures complexes, technologies complexes

Intergiciels : un peu d'histoire

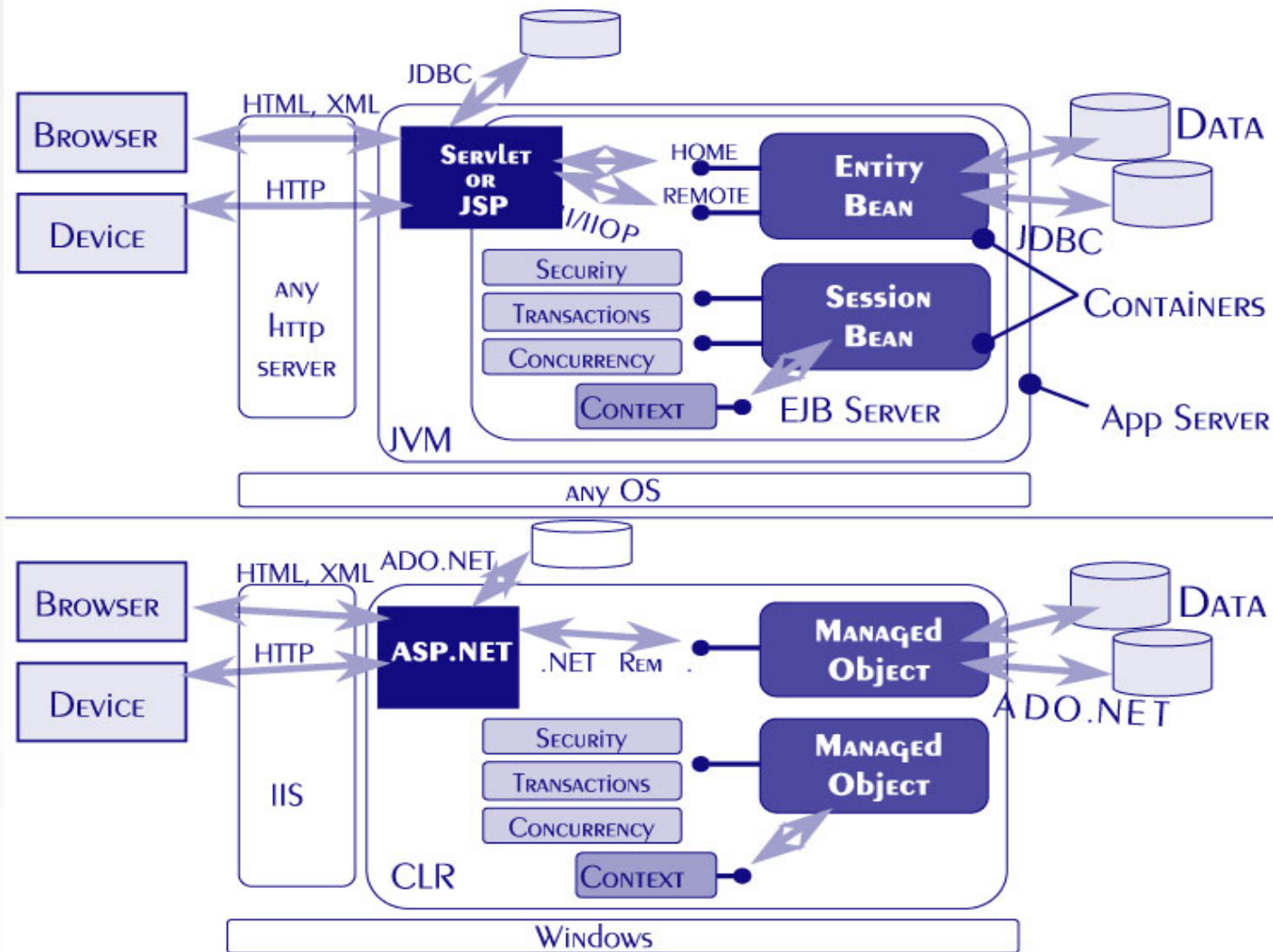
- Appel de procédure distante : 1984
- Terme middlewar : 1990
- 1988. OSF (Open Software Foundation)
maintenant Open Group
 - Devait unifier versions unix
 - Puis spécifier une plateforme intergicielle (Distributed Computing Environment)
- OMG (1989)
 - Corba (1991)

Intergiciels : un peu d'histoire

- Java (1995)
 - RMI (1996)
 - EJB (1997-1999)
 - J2EE (fin 1999)
- Microsoft
 - 1997 : DCOM (Distributed Component Object Model)
 - 1999 : COM+
 - 2002 : .net

J2EE et .NET

J2EE ET .NET: MÊME CONCEPT



Conclusion

- La construction d'une application n-tiers nécessite :
 - D'en élaborer l'architecture
 - Modèle métier
 - Qu'est ce qui est distribué ? Persistant ?
 - Comment les différentes composantes communiquent-elles ?
 - De définir des stratégies de sécurité
 - De choisir les technologies adéquates

Conclusion

- Dans ce cours
 - On s'intéresse essentiellement aux mécanismes de communication
 - Deux grandes catégories étudiées
 - Distribution d'objets (RMI, .NET remoting, CORBA)
 - Services web (avec JAVA/AXIS et .NET)
 - Des points communs
 - Notion d'interface
 - Notion de proxy
 - Masquage des couches basses