Université Larbi Ben M'Hidi **2025/2026**
Faculté des sciences exactes et des sciences de la nature et de la vie Dr. Kouah S.

Examen final - Architecture Distribuée (Master 2)

Questions de Compréhension (4 points)

1. Une application informatique se structure en trois niveaux d'abstraction. Citer ces niveaux.
2. Expliquer pourquoi l'absence d'horloge globale rend nécessaire l'usage de datations logiques.
3. Définir brièvement une **architecture 3-tiers** et préciser le rôle de chaque tiers.
4. Cocher la/les bonnes réponses :
   A. Quel est le rôle principal d'un **Middleware** ?
      a) Remplacer le système d'exploitation
      b) Masquer l'hétérogénéité et la complexité des communications
      c) Stocker les données applicatives
      d) Fournir des API de haut niveau aux applications distribuées
   B. **CORBA / RMI** :
      a) CORBA est multi-langages via IDL et interopérabilité via IIOP
      b) RMI est réservé aux objets Java et s'appuie sur une infrastructure de stubs/skeletons
      c) CORBA ne nécessite pas de mécanisme de nommage
      d) RMI et CORBA sont des exemples de middleware orientés objets distribués

Exercice 1 (8 points)

On souhaite concevoir une application de gestion des inscriptions universitaires accessible par un grand nombre d'étudiants via une application mobile. L'application doit permettre :

• La saisie des informations personnelles,
• La vérification des règles d'inscription (prérequis, capacité des groupes, …etc.),
• L'enregistrement sécurisé des données dans une base centralisée.

1. Identifier et justifier le type d'architecture distribuée le plus adapté à cette application.
2. En vous appuyant sur cette architecture, préciser le rôle de chaque composant de l'architecture.
3. Définir le middleware et expliquer son rôle pour maîtriser l'hétérogénéité.
4. Justifier le passage vers une architecture n-tiers.

Exercice 2 (8 points)

On souhaite concevoir un système distribué « **Gestion Comptes** » utilisé par des guichets et une application Web. Le système doit gérer des comptes bancaires et supporter un grand nombre de clients simultanés.

- Le **client** (guichet/Web) permet à un utilisateur de déclencher les opérations : crediter(idCompte, montant), debiter(idCompte, montant), consulterSolde(idCompte).
- La **logique métier** applique des règles (exemples : montant > 0, solde suffisant pour un débit, plafonds éventuels), puis sollicite la **couche d'accès aux données** pour lire/mettre à jour l'état des comptes.
- La **persistance** assure la gestion des comptes dans une base distante (lecture, création, mise à jour), via un service dédié exposant des opérations de type chargerCompte, mettreAJourCompte, etc.

Les services sont distribués : le client n'accède pas directement à la base, il **invoque** un **service distant**.

La localisation des services n'est pas **paramétrée statiquement dans le code** : le client obtient une référence vers le service de comptes via un **service de nommage** (annuaire), puis invoque les opérations sur la référence obtenue (*lookup* côté client ; *bind/rebind* côté serveur).

Hypothèse technique : les appels distants reposent sur un mécanisme de type **RMI/CORBA**, utilisant **Stub/Skeleton** et des opérations de **marshalling/unmarshalling**.

1. Expliquer l'intérêt de la séparation **interface / implémentation** pour l'évolution d'un système réparti.
2. Décrire le rôle du **Stub** (côté client) et du **Skeleton** (côté serveur) en mentionnant **marshalling/unmarshalling**.
3. Dessiner le **diagramme de composants UML** correspondant à l'architecture, en considérant notamment les éléments suivants :
   - un composant client d'interface utilisateur (ClientIHM) ;
   - un composant métier exposant les opérations de comptes (ServiceComptes) ;
   - un composant d'accès/persistance des comptes (PersistanceComptes) ;
   - un composant d'annuaire (ServiceNommage).

Le diagramme doit faire apparaître : **ports**, **interfaces fournies/requises** (ex. service de comptes, service de persistance, service de nommage), et **connecteurs** entre ports/interfaces.

Corrigé type : Examen final – Architecture Distribuée

Questions de Compréhension (4 points)

1) Trois niveaux d'abstraction d'une application (1Pt)

- **Présentation (IHM)** : interaction utilisateur, affichage, saisie.
- **Logique applicative / métier** : règles de gestion, traitements.
- **Données / persistance** : stockage, accès BD/fichiers, requêtes.

2) Absence d'horloge globale ⇒ datations logiques : Dans un système distribué, il n'existe pas de temps global parfaitement synchronisé entre machines. On ne peut donc pas ordonner de manière fiable des événements à partir d'horodatages physiques. Les **datations/horloges logiques** permettent d'assurer un **ordre cohérent** (ex. causalité) pour raisonner sur "qui a précédé quoi" (ordonnancement, cohérence, résolution de concurrence). **(1Pt)**

3) Architecture 3-tiers + rôle de chaque tiers (1Pt)

- **Tiers Présentation (client)** : IHM, collecte des données, envoi requêtes.
- **Tiers Logique applicative (serveur applicatif)** : règles métier, orchestration, sécurité applicative, services.
- **Tiers Données (serveur BD)** : persistance, requêtes, transactions/stockage.

4) QCM (1Pt)

A. Rôle principal d'un middleware :

b) Masquer l'hétérogénéité et la complexité des communications (Vrai)
d) Fournir des API de haut niveau aux applications distribuées (Vrai)

B. CORBA / RMI :

a) CORBA multi-langages via IDL + interopérabilité via IIOP (Vrai)
b) RMI réservé aux objets Java + stubs/skeletons (Vrai)
d) RMI et CORBA = middleware orientés objets distribués (Vrai)

Exercice 1 (8 points)

1) Architecture distribuée la plus adaptée : Architecture 3-tiers **(mobile → serveur d'application → BD).**

Justifications (au moins 2) : (2 Pts)

- **Sécurité** : la BD n'est pas exposée aux clients ; accès via serveur applicatif.
- **Maintenance/évolutivité** : règles d'inscription centralisées côté serveur.
- **Scalabilité** : serveur applicatif répliquable (montée en charge).

2) Rôle des composants (2 Pts)

- **Client mobile (présentation)** : formulaires, validation légère, envoi requêtes, affichage résultats.
- **Serveur applicatif (logique métier)** : vérifie prérequis/capacité, applique règles, gère sessions/API, contrôle d'accès, orchestre opérations.
- **Serveur de données (BD)** : stocke étudiants/inscriptions/groupes, exécute lectures/écritures, assure intégrité/persistance.

3) Middleware : définition + rôle pour maîtriser l'hétérogénéité (2 Pts)

Définition : couche logicielle intermédiaire fournissant des services et API facilitant la communication et l'intégration entre composants distribués.

Rôle :

- **Positionnement** : entre applications (client/serveur) et OS/réseau.
- **Utilité** : masque l'hétérogénéité (langages, plateformes), standardise l'échange (appel distant, sérialisation/marshalling), offre des services (naming, sécurité, etc.).

4) Justifier le passage vers n-tiers (2 Pts)

En 3-tiers, le **serveur applicatif** peut devenir un goulot d'étranglement (toutes les règles + trafic + validations + accès BD). Le passage en **n-tiers** permet de **distribuer la logique** en services spécialisés (ex. ServiceAuth, ServiceInscriptions, ServiceRègles, ServicePaiement, etc.), de **répliquer** les services critiques, et d'améliorer scalabilité, maintenabilité et disponibilité.

Exercice 2 (8 points) — Gestion Comptes

1) Intérêt séparation interface / implémentation : (2 Pts)

- **Découplage** : les clients dépendent d'une **interface stable**, pas du code interne.
- **Évolutivité** : on peut modifier/remplacer l'implémentation serveur (optimisation, refactorisation, migration BD) sans impacter les clients tant que le contrat ne change pas.
- **Interopérabilité/maintenance** : limite les dépendances, facilite versioning, déploiement indépendant.
- **Robustesse** : meilleure gestion des changements (substitution de composants, tolérance à l'évolution).

2) Rôle Stub/Skeleton + marshalling/unmarshalling : (3 pts)

Étape 1 — Appel côté client (transparence)

Le client invoque crediter(...) comme un appel "local" sur une référence.

Étape 2 — Stub (côté client)

Le **Stub** :

- construit la requête (identité objet/service + méthode + paramètres),
- réalise le **marshalling** (encodage/sérialisation des paramètres),
- envoie le message via le réseau.

Étape 3 — Skeleton (côté serveur)

Le **Skeleton** :

- reçoit la requête,
- fait l'**unmarshalling** (décodage),
- appelle la méthode réelle de l'implémentation (ServiceComptes).

Étape 4 — Retour résultat / exception

Le serveur renvoie résultat/exception : marshalling côté serveur, transmission, unmarshalling côté client, puis le client reçoit la valeur comme retour d'appel.

3) Diagramme de composants UML : (3 pts)

Composants :

1. ClientIHM (client)
2. ServiceComptes (métier)
3. PersistanceComptes (DAO/persistance)
4. ServiceNommage (annuaire)

Exemples de quelques interfaces :

- IComptes : crediter(), debiter(), consulterSolde()
- IPersistanceComptes : chargerCompte(), mettreAJourCompte()
- INommage : lookup(), bind(), rebind()

Exemples de Ports & interfaces fournies/requises :

- ClientIHM
  - **requiert** IComptes (port pComptesReq)
  - **requiert** INommage (port pNamingReq)
- ServiceComptes
  - **fournit** IComptes (port pComptesProv)
  - **requiert** IPersistanceComptes (port pPersistReq)
  - *(optionnel)* **requiert** INommage pour publier/renouveler sa référence
- PersistanceComptes
  - **fournit** IPersistanceComptes (port pPersistProv)
- ServiceNommage
  - **fournit** INommage (port pNamingProv)

Connecteurs :

- Connecteur 1 : ClientIHM.pNamingReq → ServiceNommage.pNamingProv
- Connecteur 2 : ClientIHM.pComptesReq → ServiceComptes.pComptesProv
- Connecteur 3 : ServiceComptes.pPersistReq → PersistanceComptes.pPersistProv
- *(optionnel)* Connecteur 4 : ServiceComptes → ServiceNommage (bind/rebind)