Projet systèmes concurrents/intergiciels

2AI

27 novembre 2016

Le projet a pour but d'expérimenter le développement d'applications concurrentes selon un modèle de mémoire partagée. Pour cela, il s'agira

- de réaliser un noyau d'exécution pour un espace partagé de données typées, dans un environnement centralisé, puis réparti ;
- de développer des applications concurrentes basées sur l'utilisation de cet espace, afin
  - d'une part d'évaluer les choix de conception possibles pour ces applications,
  - et d'autre part de tester et évaluer l'espace partagé réalisé.
- de développer des outils d'instrumentation, facilitant la supervision de l'exécution des applications

Cette approche est inspirée du modèle Linda (ou TSpaces). Dans ce modèle, les processus partagent un espace de *tuples* qu'ils peuvent manipuler à l'aide d'un jeu de primitives spécifiques.

1 Le modèle Linda

1.1 Qu'est-ce qu'un tuple ?

Tuple de valeurs : un tuple de valeurs est un n-uplet ordonné de valeurs. Les éléments sont de types quelconques (entier, booléen...ou même tuple). Par exemple, les données suivantes sont des tuples :
`[ 10 'A' true ], [ 1 2 3 ], [ "azerty" 1 ], [ 'A' [ 10000 false ] 23 ]`

Motif : un tuple motif (ou *template*) est un n-uplet dont certaines des composants sont des types, qui représentent « n'importe quelle valeur de ce type ». Par exemple `[ ?Integer true ]` est un motif et les tuples `[ 6 true ]` et `[ 2 true ]` correspondent au motif (« match »). Dans notre cas, un type est une classe Java. On peut utiliser `?Object` qui correspond à n'importe quelle valeur de n'importe quel type.

Par exemple le tuple `[ 'A' [ 10000 false ] 2 ]` correspond aux motifs `[ 'A' ?Tuple ?Integer ]`, `[ ?Character [ ?Integer false ] ?Object ]` et bien d'autres. La propriété « correspondre » est une propriété d'inclusion et forme un ordre partiel : on peut comparer des motifs (cf figure 1).

1.2 Les primitives du modèle Linda

- `write(tuple)` : dépose le tuple dans l'espace partagé ;
- `take(motif)` : extrait de l'espace partagé un tuple correspondant au motif précisé en paramètre ;
- `read(motif)` : recherche (sans l'extraire) dans l'espace partagé un tuple correspondant au motif fourni en paramètre ;
- `tryTake(motif)` : version non bloquante de `take` ;
- `tryRead(motif)` : version non bloquante de `read` ;
- `takeAll(motif)` : renvoie, en extrayant, tous les tuples correspondant au motif (vide si aucun ne correspond) ;

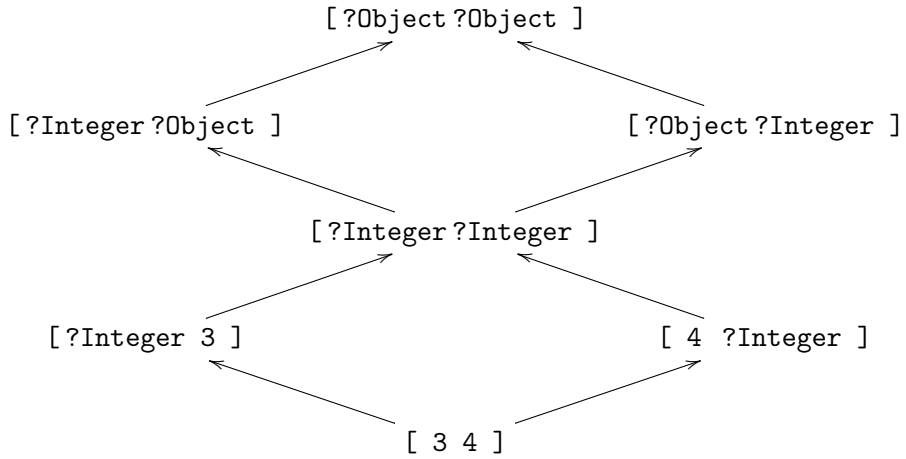

FIGURE 1 – Ordre partiel de la propriété `match`

— `readAll(motif)` : renvoie, sans extraire, tous les tuples correspondant au motif (vide si aucun ne correspond) ;
— `eventRegister(mode, timing, motif, callback)` : s'abonner à l'événement d'existence/de dépôt d'un tuple correspondant au motif.

L'opération `write` ajoute une copie du tuple déposé dans l'espace des tuples partagés. L'espace des tuples est un multi-ensemble : un même tuple peut être présent en plusieurs exemplaires. Noter aussi que l'on peut déposer aussi bien des tuples de valeur et des tuples motifs.

Les primitives `take` et `read` sont bloquantes : l'appelant reste bloqué tant qu'aucun tuple de l'espace partagé ne satisfait le motif demandé.

L'opération `take` consiste à rechercher un tuple présent dans l'espace des tuples partagés. La recherche s'effectue d'après un tuple motif. Lorsqu'un tuple satisfaisant le motif est finalement trouvé (après une attente éventuelle), celui-ci est extrait de l'espace des tuples partagés.

L'opération `read` recherche un tuple correspondant au motif fourni en paramètre, et retourne à l'appelant une copie du tuple trouvé. Autrement dit, le tuple trouvé reste dans l'espace des tuples partagés.

Les opérations `tryTake` et `tryRead` sont les versions non bloquantes des opérations `take` et `read`. Elles renvoient le tuple trouvé (une copie si consultation) ou une valeur nulle si aucun tuple ne correspond.

Les opérations `takeAll` et `readAll` renvoient la collection de *tous* les tuples qui correspondent au motif, avec ou sans extraction de l'espace des tuples. Si aucun tuple ne correspond, ces opérations renvoient une collection vide : elles ne bloquent jamais.

Enfin, l'opération `eventRegister` permet de s'abonner à l'occurrence d'un événement de d'existence/de dépôt (selon la valeur du timing : `eventTiming.IMMEDIATE/FUTURE`) d'un tuple correspondant au motif spécifié. Le `callback` est exécuté avec le tuple identifié. Ce tuple est retiré de l'espace de tuple (comme un `take`) si le mode est `eventMode.TAKE`, il est laissé dans l'espace si le mode est `eventMode.READ`. Le callback n'est déclenché qu'une fois ; s'il le souhaite, il peut se réenregistrer lui-même. Noter que le callback peut être déclenché immédiatement à son enregistrement si un tuple correspondant au motif existe déjà dans l'espace de tuple et que le timing est `eventTiming.IMMEDIATE`.

Spécification libérale – La spécification des opérations est volontairement assez libérale :
— quand plusieurs tuples correspondent, `take` en retourne un arbitraire ;
— quand plusieurs `take` sont en attente et qu'un dépôt peut en débloquer plusieurs, le choix est arbitraire (pas nécessairement le plus ancien) ;
— quand des `read` et un `take` sont en attente et qu'un dépôt peut les débloquer, il n'est pas

spécifié si, outre le `take`, tous les `read` doivent être débloqués, ou aucun, ou seulement certains ;
— quand il y a à la fois un `take` et un `callback` enregistré pour un même motif, le choix entre déblocage du `take` et déclenchement du `callback` n'est pas spécifié.

L'implantation pourra, au choix, définir précisément le comportement (par exemple FIFO) ou pas. La correction de l'exécution d'un exemple ne doit pas dépendre de ces choix.

1.3 Manipulation des tuples

Le type tuple est défini dans la classe `Tuple`. Il s'agit d'une liste d'objets (sérialisables à cause de la suite du projet).

Outre les méthodes de l'interface `List` (telles `add`, `get`...), la classe fournit des méthodes pour créer un tuple, pour tester si un tuple correspond à un motif ainsi que pour lire ou écrire des tuples.

2 Mise en œuvre du noyau Linda

Le noyau Linda sera développé en plusieurs étapes, correspondant à un enrichissement progressif des possibilités d'exécution, d'une JVM à un environnement réparti. Chaque étape donnera lieu à la production d'une version distincte.

2.1 Version en mémoire partagée

Il s'agit de réaliser une implantation de l'interface `Linda` qui tourne directement dans la même machine virtuelle que les codes utilisateurs.

Contrat : les interfaces et classes `Linda`, `Tuple`, `Callback`, `AsynchronousCallback` sont figées et ne doivent pas être modifiées. L'implantation doit être dans la classe `linda.shm.CentralizedLinda` avec un constructeur sans paramètre. Les exemples fournis doivent compiler et s'exécuter correctement sans qu'il soit nécessaire d'y faire le moindre changement.

Conseil : faire une première version sans se préoccuper d'`eventRegister`.

Attention : les exemples fournis ne sont pas des tests suffisants, loin de là !

2.2 Version multiactivités en mémoire partagée

Il s'agit d'améliorer la version précédente en réalisant une implémentation multiactivités des opérations de l'interface `Linda`. Le gain de performances devra être évalué.

2.3 Version client / mono-serveur

Il s'agit de réaliser une implantation de l'interface `Linda` qui accède à un serveur distant qui centralise l'espace de tuples.

Contrat : La classe `linda.server.LindaClient` (à écrire) doit être une implantation de l'interface `Linda`. Le constructeur doit prendre un unique paramètre chaîne qui est l'URI du serveur Linda à utiliser (par exemple `//localhost:4000/LindaServer`).

Conseil : l'implantation d'`eventRegister` est le morceau acrobatique de cette partie.

Bonus si l'implantation du serveur réutilise en interne la version Linda en mémoire partagée sans y toucher une virgule (et sans tricher en y rétroportant des aspects client/serveur).

Cette version monoserveur peut s'appuyer sur la version monoactivité en mémoire partagée, en cas de difficultés avec la version multiactivités en mémoire partagée.

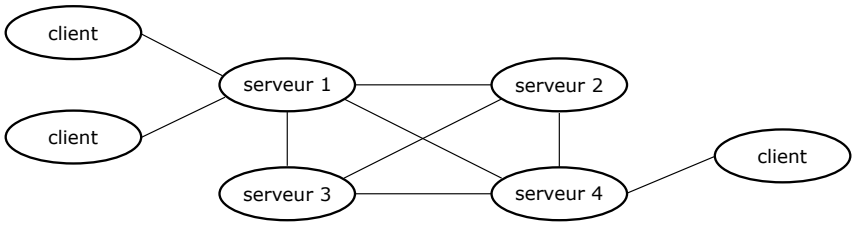

FIGURE 2 – Linda multi serveurs

2.4 Version multi-serveurs

On a n (> 1) serveurs qui gèrent chacun une partie de l'espace des tuples. Un client est connecté à un unique serveur, et les serveurs sont tous interconnectés entre eux (cf figure 2). Le client utilise la même classe `linda.server.LindaClient` que précédemment, seule l'implantation du serveur change. Quand un client dépose un tuple, le dépôt se fait dans l'espace du serveur auquel il est connecté. Quand un client cherche un tuple (`take`, `read`), la recherche se fait sur le serveur auquel il est connecté et en cas d'échec sur les autres serveurs. Comme le client n'est connecté qu'à un seul serveur, c'est ce serveur qui doit propager la requête.

Cette version répartie prendra en compte et proposera des solutions pour (au moins) l'une des problématiques caractéristiques de la répartition : désignation, équilbrage de charge, tolérance aux pannes, réplication...

2.5 Outils de supervision

Les applications tournant sur le noyau Linda, et le noyau Linda lui-même, devront être testés et évalués de manière suffisamment complète. A cette fin, il sera utile de développer un certain nombre d'outils *simples*, comme :

- un interpréteur de commandes, permettant
  - de jouer des scénarios interactifs ou prédéfinis ;
  - d'évaluer (sommairement) les performances des applications exécutées ;
- de consulter et de fixer des paramètres relatifs aux ressources utilisées comme la taille (maximale ou courante) de l'espace de tuples, le nombre de tuples, la taille des files d'attente, le nombre de processus actifs ;
- ...

Notez que cette liste n'est qu'indicative : elle n'est ni exhaustive, ni prescriptive. Cependant, il est attendu qu'un certain nombre de ces outils soient développés et utilisés.

3 Applications

Le noyau servira de support à l'exécution de différentes applications concurrentes. L'objectif sera d'une part d'évaluer différents choix de conception et d'architecture pour les applications concurrentes et d'autre part de permettre de comparer et valider les différentes versions du noyau lui-même.

3.1 Tests élémentaires

La validation du noyau s'appuiera sur des tests et scénarios de base afin d'en évaluer les aspects fonctionnels et les performances. Ces tests mettront en jeu des threads réalisant différentes configurations d'accès, sans que ces accès aient forcément une signification particulière.

Les tests fonctionnels reposeront sur la méthodologie et les outils de test – comme Junit – vus en première année, dans le cadre des enseignements de programmation, ainsi que sur la définition d'un ensemble de scénarios suffisamment représentatif de situations d'utilisation mettant en jeu

des activités concurrentes. L'objectif de ces scénarios est de valider la cohérence et la vivacité des opérations sur l'espace de tuples.

Les tests de performances devront permettre de comparer les différentes versions développées et d'évaluer leurs limites en termes (quantitatifs) de volumes de données traitées ou conservées, de débit d'opérations concurrentes, de nombre de processus concurrents possibles, de temps d'exécution... Ces tests reposeront sur un jeu d'applications de référence *simples*, qui pourront être inspirées de ce qui a été fait dans le cadre des TPs sur les threads ou la concurrence.

Pour cette phase, le développement d'outils *simples* de supervision ou d'instrumentation paraît particulièrement approprié.

3.2 Schémas et problèmes de base

Une bonne manière de se familiariser avec le modèle de programmation Linda consiste à réaliser dans ce modèle des exercices simples et classiques traités précédemment avec d'autres outils. On pourra ainsi :

- réaliser en Linda des mécanismes/objets d'interaction élémentaires comme
  - un sémaphore ;
  - une barrière ;
  - un canal de communication ;
- développer en Linda des solutions pour les problèmes de coordination classiques :
  - lecteurs-rédacteurs
  - philosophes et spaghetti
  - ...

  Le but de ces exercices n'est pas de développer une palette complète de solutions pour un ensemble de variantes/politiques connues. Il s'agit avant tout de pointer ce qui est facilement exprimable en Linda (auquel cas, la solution devrait être fournie), et ce qui est moins naturel/direct à spécifier dans ce cadre (auquel cas, la description de la difficulté et de sa cause suffisent).

3.3 Calcul concurrent

L'archive fournie comporte un exemple (simple) caractéristique d'une classe d'applications où l'utilisation du modèle Linda est pertinente : il s'agit d'un « tableau blanc » interactif, partagé par un ensemble d'utilisateurs. Le contenu du tableau blanc est conservé dans l'espace de tuples. Chaque utilisateur rafraîchit régulièrement sa copie du tableau blanc à partir de l'espace de tuples et peut modifier le contenu du tableau blanc en accédant à l'espace de tuples.

Les applications que vous devrez développer se focalisent sur une autre catégorie d'applications pour laquelle le modèle Linda est a priori adapté : les calculs intensifs sur des masses de données.

3.3.1 Calcul des nombres premiers inférieurs à K

Cet exemple se focalise sur la conception d'un algorithme parallèle.

Il s'agit de produire l'ensemble des nombres premiers inférieurs à une borne donnée en paramètre, au lancement de l'application. Il faudra tout d'abord développer une version séquentielle (très simple) basée sur la technique du crible d'Eratosthène, puis envisager, réaliser différentes formes de parallélisation de cet algorithmes. Cette étude se conclura par une évaluation et une comparaison de ces différentes versions.

3.3.2 Une application de calcul intensif : comparaison de séquences génomiques

Note préalable : cette application est proposée comme étude de cas « réaliste » mettant en jeu des calculs concurrents intensifs sur des données (relativement) volumineuse. Il est possible de proposer une autre application présentant les mêmes caractéristiques **à condition qu'elle soit validée dès le départ par l'équipe enseignante**.

La comparaison de séquences protéiques est une activité routinière en biologie. En se limitant aux aspects de la programmation concurrente [1], le problème qu'il est demandé de traiter peut être présenté comme suit :

- on dispose d'une base de données contenant un ensemble de séquences protéiques ;
- une séquence protéique est représentée comme une suite de caractères d'un alphabet fixé. Les séquences contenues dans la base ont des tailles variables (de quelques centaines à plusieurs milliers de caractères)
- le but de l'application est de déterminer quelles sont les séquences de la base qui sont les plus proches d'une séquence cible, donnée en paramètre au lancement de l'application.
- pour cela, on dispose d'une fonction $similarit(s_1, s_2)$ qui renvoie un entier correspondant à la mesure de l'écart entre les séquences s_1 et s_2).
- il s'agit donc essentiellement de comparer chacune des séquences de la base à la séquence cible et de retenir la ou les plus proches.
- la fonction $similarit(s_1, s_2)$ est basée sur la construction d'un tableau dont les colonnes correspondent à la séquence s_1 et les lignes correspondent à la séquence s_2. L'algorithme (séquentiel) de construction de ce tableau est connu et fourni.

Il s'agira de réaliser une application concurrente permettant d'extraire la ou les séquences de la base les plus proches d'une séquence cible donnée. Différentes modalités de parallélisation sont possibles et **devront** être envisagées :

- les évaluations de similarité peuvent être menées en parallèle. Sur cet aspect, deux points doivent être pris en considération :
  - l'ordonnancement et l'équilibrage de charge des tâches d'évaluation ;
  - le fait que les données de la base ne peuvent tenir intégralement en mémoire vive. Il faudra donc gérer le fait que l'espace de tuples ne peut à tout moment contenir qu'un nombre limité de séquences.
- la parallélisation du calcul de similarité lui-même. A ce niveau, les points essentiels sont l'équilibrage de charge (encore) et le grain des données traitées par un ouvrier.

4 Modalités pratiques

4.1 Structure du projet

Le projet comporte 3 étapes. Les étapes sont décomposées de la manière suivante :

Etape 1

partie 1.1 version en mémoire partagée (cf section 2.1), instrumentation (2.5)

partie 1.2 tests élémentaires (3.1), outils de supervision (2.5), schémas et problèmes de base (3.2), calcul des nombres premiers (3.3.1)

Etape 2

partie 2.1 version multi-activités (cf section 2.2), version mono-serveur (2.3)

partie 2.2 tests élémentaires (3.1), calcul concurrent (3.3.2)

Etape 3 version multi-serveurs (cf section 2.4)

Lorsque qu'une étape est décomposée en parties, ces parties sont indépendantes.

1. Vous pourrez trouver quelques éléments de contexte, hors informatique, à l'url `https://interstices.info/jcms/c_10593/alignement-optimal-et-comparaison-de-sequences-genomiques-et-proteiques`

4.2 Constitution des groupes

Projet réalisé en binôme (ou trinôme si l'organisation le nécessite, voir plus bas). Les binômes sont constitués au sein d'un même groupe de TD. Dans le cas où le groupe de TD comporte un nombre impair d'étudiants, un trinôme est constitué. Chaque groupe de TD doit avoir un nombre pair de binômes/trinôme. Dans le cas où ce nombre est impair, un binôme est réparti pour constituer deux trinômes. Les binômes/trinômes d'un groupe de TD sont répartis en un nombre égal de binômes/trinômes "A" et de binômes/trinômes "B".

Les "A" et les "B" sont enfin appariés en groupes. Un groupe est constitué par un "A" et un "B", avec la contrainte que l'effectif du groupe ne peut être 6.

La constitution des binômes/trinômes, des groupes et le choix des lettres sont libres, sous réserve du respect des contraintes précédentes. Les cas de blocage ou l'absence de choix seront réglés par tirage au sort.

Les "A" réaliseront les parties 1.1 et 2.2, et évalueront les parties1.2 et 2.1.

Les "B" évalueront les parties 1.1 et 2.2, et réaliseront les parties1.2 et 2.1.

L'étape 3 sera réalisée par le groupe complet.

4.3 Déroulement des étapes 1 et 2

Chacune des étapes 1 et 2 se déroule en trois temps :

- chaque groupe dispose d'une dizaine de jours pour réaliser la partie qui lui est affectée. A l'issue de ce travail, il remet une archive contenant
  - un rapport provisoire succinct présentant l'architecture, les algorithmes des opérations essentielles, une explication claire des points délicats et de leur résolution.
  - le code complet de la partie réalisée.
- Ce travail n'est pas noté mais il est transmis, pour évaluation et validation, à l'autre binôme/trinôme du groupe. Cette étape d'évaluation consiste à
  - valider le code fourni et en évaluer les performances par des tests élémentaires (cf 3.1)
  - réaliser une revue critique du code et des choix de conception effectués
  - proposer des améliorations, ou non. Les propositions d'amélioration, ou la validation du code et des choix existants doivent être étayés dans tous les cas.

  Cette évaluation aboutira à un rapport qui sera remis et transmis à l'autre binôme/trinôme du groupe. Ce rapport est noté.
- Enfin, chaque binôme dispose d'une semaine pour intégrer (ou non) les remarques et propositions issues de l'évaluation. A l'issue de ce travail, il remet une archive contenant
  - un rapport succinct présentant l'architecture, les algorithmes des opérations essentielles, une explication claire des points délicats et de leur résolution.
  - le code complet de la partie réalisée.

  Ce travail est noté.

4.4 Echéancier

28/11 constitution des groupes, binômes...

8/12 remise des rapports et livrables provisoires de l'étape 1

14/12 remise des évaluations de l'étape 1

semaine du 3/1 remise des rapports et livrables finaux de l'étape 1 et présentation par le groupe du travail réalisé

9/1 remise des rapports et livrables provisoires de l'étape 2

12/1 remise des évaluations de l'étape 2

19/1 remise des rapports et livrables finaux des l'étapes 2 et 3 ; présentation par le groupe du travail réalisé sur ces étapes

Les dates ci-dessus sont des dates fermes, car le travail des binômes est interdépendant.

Note : il est possible (et même recommandé) de démarrer l'étape $i + 1$ dès lors que les évaluations de l'étape i ont été réalisées.

4.5 Notation

Le poids respectif des étapes 1, 2, et 3 sera de 40%, 30%, 30%.

Les notes des étapes 1 et 2 seront attribuées par binôme/trinôme, et la note de l'étape 3 sera attribuée au groupe, sauf dysfonctionnement avéré [2].

Pour ce qui est des parties de réalisation, la notation portera sur le contenu du rapport définitif (dont les éléments ont été présentés plus haut), et sur nos propres évaluations, lectures et tests du code fourni. A ce propos, il est impératif que l'API qui vous est fournie soit strictement respectée, afin que nos tests puissent être opérationnels. Ainsi, le code des exemples qui vous sont fournis doit compiler et s'exécuter correctement *sans que vous y touchiez le moindre caractère*. Ces tests valident votre respect de l'API.

Pour ce qui est des parties d'évaluation, les deux éléments pris en compte seront

— le caractère complet de la démarche de validation du code, suivant les lignes présentées en 3.1

— la pertinence de la revue critique et des propositions d'amélioration présentées, par rapport à notre propre évaluation du rapport provisoire, dont nous disposerons : si les rapports provisoires ne sont pas notés en eux-mêmes, ils servent de base pour noter les évaluations.

L'étape 3, réalisée en groupe sera évaluée sur le code et les éléments de validation présentés.

4.6 Fournitures

Les différents documents et fournitures utiles sont (ou seront) mis en ligne sur la page `Moodle` de l'enseignement, au fur et à mesure de la progression du projet, et notamment :

— une archive contenant
  — l'arborescence de fichiers dans laquelle devront se trouver les livrables ;
  — le code source des interfaces, classes de base, tests et exemples élémentaires ;
  — une archive contenant le bytecode d'une version centralisée du noyau Linda, destinée à permettre l'élaboration des tests et des applications, et à servir de point de référence pour les évaluations de performance ;

— pour l'application de comparaison de séquences génomiques, la description de l'algorithme séquentiel, et un paquetage fournissant les opérations de base sur les séquences, et les opérations d'accès à la base de données, base de données.

— ...

4.7 Organisation des codes sources, et contenu de l'archive fournie

package `linda` : `Tuple`, `TupleFormatException`, `Linda`, `Callback`, `AsynchronousCallback` (intégralement fourni, classes et interfaces pour les codes utilisateurs, **ne pas modifier**) ;

package `linda.shm` : classe `CentralizedLinda` (implantation de `linda.Linda` en mémoire partagée, à écrire sous ce nom). Vous pouvez ajouter ici d'autres classes/interfaces, mais elles seront cachées aux codes utilisateurs ;

package `linda.tshm` **(archive** `clinda.jar`**)** : version exécutable du noyau centralisé ;

packages `monoserver` **et** `multiserver` : versions mono-serveur et multi-serveurs du noyau ;

package `linda.server` : classe `LindaClient` (implantation de `linda.Linda` pour accéder (indifféremment) à un (mono/multi)-serveur distant). Vous pouvez ajouter ici d'autres classes/interfaces, mais elles ne seront pas visibles par lescodes utilisateurs ;

package `linda.test` : nos tests élémentaires ;
la classe `BasicTestX` utilise le paquetage `linda.tshm` fourni par l'archive `clinda.jar`.

package `linda.outils` : pour les outils de supervision ;

package `linda.applications` : pour les applications. Contient l'exemple de tableau blanc (qui est écrit pour la version mono-serveur).

package `linda.`*`autre`* : ce que vous voulez.

2. En cas (exceptionnel) d'anomalies importantes constatées dans le fonctionnement/travail du groupe ou du binôme/trinôme, les notes seront individualisées.