Université Joseph Fourier
Promotion 2001

TéléCabriJava : édition coopérative d'une figure géométrique sur Internet

DEA ISC Informatique,

Systèmes et Communication

soutenu par Fabien Rigaux

le 18 Juin 2001

Travaux effectués au sein de

l'équipe SIRAC de l'INRIA Rhône-Alpes

Tuteur : Gilles.Kuntz@inrialpes.fr

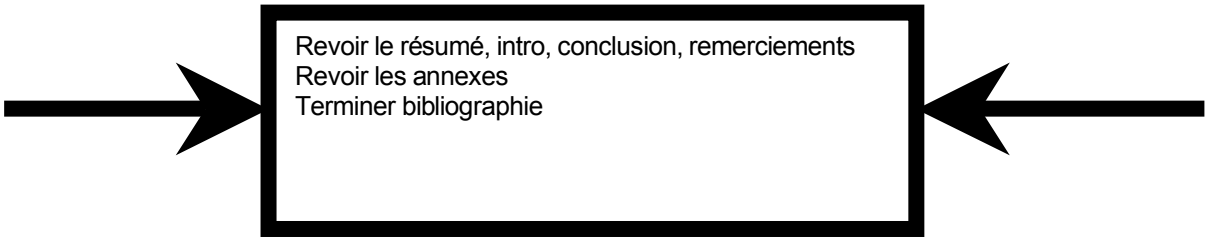
Résumé

Dans le contexte actuel de connexions Internet bon marché, de machines de plus en plus puissantes, conviviales et orientées multimédia, le télé-enseignement prend une part grandissante au sein des établissements publics de recherche. L'idée de TéléCabriJava est de permettre le tutorat distant autour d'une feuille de géométrie partagée.

J'ai démarré le DEA avec CabriJava : un clone en Java de Cabri Géomètre II. J'ai étudié les différents outils qui s'offrent à nous pour répartir CabriJava afin de réaliser l'application coopérative TéléCabriJava. J'ai donc procédé à une évaluation des Remote Procedure Call RMI et CORBA, du Message Oriented Middleware JORAM et des protocoles TCP et UDP. Il s'est révélé que TéléCabriJava est adapté à une communication asynchrone par messages, ce qui exclut les Remote Procedure Call. JORAM aurait été la solution idéale avec son concept de *Topic*, malheureusement, il est bien trop lent pour une application gourmande en ressources réseaux comme TéléCabriJava. TCP s'est donc raisonnablement imposé face à un protocole UDP de trop bas niveau pour réaliser notre application.

J'ai commencé l'implantation de TéléCabriJava avec TCP et tout ce qui concerne une session est plus ou moins terminé. Les premiers tests avec deux machines interconnectées par une liaison grand public (modem 56Kbits) sont tout à fait concluants avec une synchronisation largement acceptable des figures. Si l'aspect performance est enthousiasmant, la gestion des problèmes réseaux a été traitée de façon plutôt radicale et il serait bon de poursuivre le développement et les expérimentations avec plus de machines et plus de connexions de mauvaise qualité. En outre, tout ce qui concerne le serveur de sessions reste à implanter.

Mots clés : application coopérative, middleware, RMI, JORAM, TCP, UDP.



Remerciements

En premier lieu, je remercie mon tuteur Gilles Kuntz qui m'a orienté initialement dans mes recherches et qui m'a donné quelques coups de pouces dans la réalisation de TéléCabriJava. Je le remercie un peu moins en ce qui concerne sa passion des MAC, car j'ai du développer et tester TéléCabriJava sur un couple hétérogène de machines : un PC IBM qui avait des problèmes de Scheduler avec son OS WIN NT4 et un MAC G4 Cube avec MAC OS X. *Je n'aime pas les MAC, mais le dites pas à Gilles …*

Je remercie l'équipe de JORAM qui m'a aidé à installer ce MOM prometteur qu'est JORAM (ce qui lui a permis de découvrir un petit bug dans un script sh de configuration). Je tiens aussi à exprimer ma gratitude envers Gérard Vandôme sans lequel je n'aurais jamais soupçonné l'existence de JORAM.

Grand merci aussi à Christophe Rippert pour ses conseils sur RMI et JAVA ainsi qu'à Eric Bruneton pour ses explications sur son Remote Procedure Call Javapod.

Table des matières

1. Introduction

Le télé-enseignement fait partie depuis plusieurs années des axes de recherche prioritaires des établissements publics de recherche. Aujourd'hui, les technologies disponibles sur Internet et le développement des réseaux permettent de concevoir des applications partagées coopératives à partir de liaisons et de logiciels standards. Dans ce cadre, le projet TéléCabriJava construit autour des principes de la géométrie dynamique a pour but de permettre du tutorat distant autour d'une feuille de géométrie partagée et d'une zone de dialogue textuelle; l'élève étant aidé à la fois par le tuteur et par l'assistance de démonstrateurs répartis sur le Web.

Disposant initialement de CabriJava qui est un clone de Cabri-Géomètre II en Java, l'objectif de mon DEA fut donc de recenser et d'étudier les outils de communication afin de répartir CabriJava pour créer TéléCabriJava. Mon étude s'est donc portée sur les protocoles TCP et UDP ainsi que le MOM (Message Oriented Middleware) JORAM et les RPC (Remote Procedure Call) RMI et CORBA. Pour de multiples raisons, TCP s'est imposé à nous comme la meilleure solution pour réaliser TéléCabriJava.

Ce rapport est composé de quatre chapitres :

- le premier présente le fameux CAhier de BRouillon Interactif (CABRI-Géomètre II) qui permet l'édition et l'animation dynamique de figures géométrique et qui est à l'origine de CabriJava,
- le second présente le cahier des charges que j'ai établi pour TéléCabriJava,
- on poursuit ensuite avec une exploration des différents outils de communication possibles pour la réalisation de TéléCabriJava,
- enfin, la dernière partie explique pourquoi notre choix s'est porté sur TCP et je précise quelques décisions importantes concernant la réalisation.

En outre, le lecteur pourra trouver en annexes des détails concernant la réalisation en Java de l'interface graphique (en swing) et du package gérant les communications réseaux avec TCP. Il y figure aussi une sommaire modélisation en UML ainsi qu'une description approfondie du protocole TCP pour celui ou ceux qui reprendront le flambeau.

2. Cabri-Géomètre : origine de CabriJava

Cabri-Géomètre II est à l'origine de [CabriJava] que l'on propose de répartir avec TéléCabriJava pour permettre l'édition coopérative d'une figure géométrique. C'est un outil de géométrie dynamique développé par l'équipe EIAH.

2.1. L'équipe de Cabri Géomètre

L'équipe EIAH (Environnement Informatique d'Apprentissage Humain) a pour but d'étudier les problèmes de la conception et de la réalisation d'environnements didactiques et les conditions de leur mise en œuvre dans la perspective d'un apprentissage des Mathématiques. Ces problèmes mélangent Informatique, Mathématiques et didactique des Mathématiques. Les travaux de l'équipe s'organisent autour du développement du projet Cabri (CAhier de BRouillon Interactif) dont la réalisation de référence est le micro monde Cabri-Géomètre.

Les problématiques informatiques et didactiques interagissent d'une façon particulière dans le cadre de travaux portant sur l'informatisation des processus didactiques. Cet axe de recherche est développé dans la perspective d'une machine partenaire de l'enseignant autant que de l'élève. Initialement abordée en pensant à l'interaction entre micro monde et tuteur intelligent, cette problématique est aujourd'hui développée autour d'une architecture distribuée sur des réseaux associant micro mondes, assistants mathématiques, précepteurs artificiels, apprenants et enseignants Humains. Les mots clés caractérisant cet axe sont : coopération et interaction entre agents, scriptabilité, Internet, Multimédia, didactique computationnelle, c'est dans cette optique qu'ont été créés les projets CabriJava et TéléCabriJava.

2.2. La géométrie dynamique

Aujourd'hui, le développement de la géométrie dynamique est étroitement lié à celui de la manipulation directe par ordinateur. Les origines de la géométrie dynamique remontent aux Mathématiciens dont Clairault (XVIII ième Siècle) et d'autres acteurs plus récents ont développé l'idée de déplacer les éléments d'une figure dans le but d'illustrer des phénomènes (géométriques) et de prouver des théorèmes.

Géométrie dynamique et manipulation directe sont souvent associées dans les environnements informatiques proposés à l'utilisateur, enseignant ou apprenant comme moyen de développer des activités intellectuelles basées sur leurs connaissances géométriques. En d'autres termes, l'utilisateur est placé dans un micro monde. Au contraire de Logo, dans un micro monde bâti sur la géométrie, comme dans Cabri Géomètre ou Geometer's Sketchpad, les bases mathématiques à enseigner jouent un rôle primordial. Au travers de la manipulation directe, l'étudiant peut construire ses propres constructions complexes, faisant appel pour cela à des connaissances géométriques plus ou moins avancées : il peut ainsi interagir directement sur les représentations d'objets géométriques théoriques.

Les caractéristiques clés de tels environnements de géométrie dynamique sont :

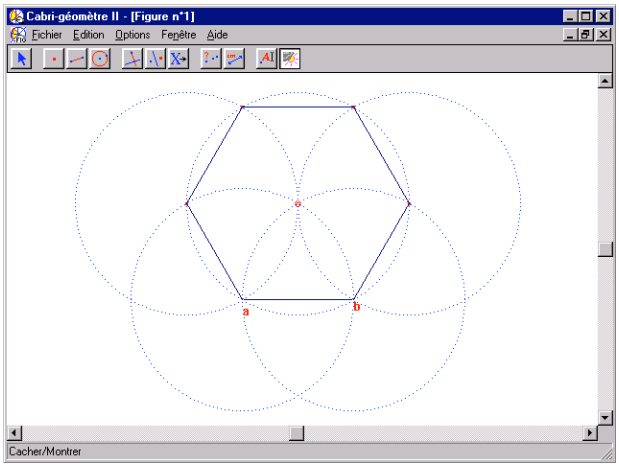
- une base mathématique solide de la géométrie à représenter,
- une interface utilisateur conviviale autorisant une prise en main immédiate d'une part importante des outils,
- un rapide retour d'interaction du système permettant à l'utilisateur de contrôler le comportement des objets qu'il a construits.

Depuis plusieurs années, de nombreux travaux ont montré l'importance de l'engagement direct de l'apprenant à travers l'usage de logiciels de construction et de manipulation d'objets géométriques.

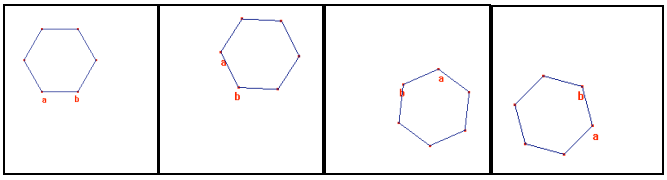
2.3. Cabri Géomètre

Les recherches menées autour du logiciel Cabri Géomètre soulignent le rôle prépondérant de l'investissement personnel de l'élève, qui, affranchi des difficultés de construction d'objets géométriques souvent difficiles à dessiner, peut analyser librement les problèmes qui lui sont posés d'une manière dynamique et active. Ce logiciel multilingue disponible sur plate-forme Macintosh ou PC, mais aussi sur calculatrices de poche a aujourd'hui dépassé le million d'exemplaires vendus dans le monde et a suscité la création de nombreux groupes collaboratifs appelés « clubs cabri ». Son usage en classe, mais aussi en auto-apprentissage hors du temps scolaire s'est largement développé au point de réunir des centaines d'utilisateurs lors d'Universités d'été organisées autour de ce logiciel. Cet outil de géométrie dynamique est également utilisé de manière fructueuse dans d'autres domaines où la géométrie sert de support mathématique comme l'optique, l'électronique, la mécanique, l'astronomie, …

2.3.1. Exemple de construction



Il s'agit de construire un hexagone à partir du segment [ab] à l'aide de l'outil cercle et segment. En pointillé, les constructions qui ont permis d'aboutir à l'hexagone. CabriGéomètre, outil basé sur la géométrie dynamique nous permet ensuite de vérifier l'exactitude de la construction en déplaçant les points de bases ***a*** et ***b***.



Voici une illustration des capacités d'animation dynamique d'une figure de Cabri en déplaçant le point de base ***a*** de la construction.

3. Cahier des charges de TéléCabriJava

3.1. Description de CabriJava

[CabriJava] était initialement une applet permettant la consultation de figures géométriques construites à partir de Cabri Géomètre II. Actuellement, CabriJava est aussi décliné sous forme d'application et fournit tous les services d'édition, consultation et animation de figures de l'outil Cabri Géomètre II. CabriJava comme son nom l'indique est écrit en Java et par conséquent portable à l'opposé de Cabri Géomètre qui n'est compatible qu'avec Windows ou MacOS.

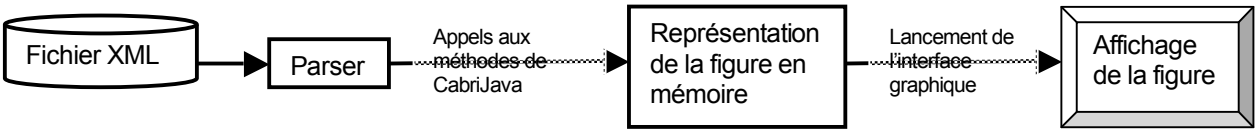
Les figures éditables avec CabriJava seront stockées au format XML pour assurer une portabilité maximale. CabriJava n'implante pas d'API, toutefois on proposera à l'avenir des interfaces pour permettre son extension.

TéléCabriJava sera une couche au-dessus de CabriJava qui permettra l'édition coopérative de figure géométrique.

Pour cela, CabriJava fournira des méthodes à l'usage spécifique de TéléCabriJava :

- TéléCabriJava doit pouvoir récupérer les actions menées par l'utilisateur dans CabriJava pour les envoyer aux autres participants d'une session TéléCabriJava,
- et inversement, TéléCabriJava doit pouvoir mettre à jour la figure.

3.1.1. Chargement d'une figure avec CabriJava



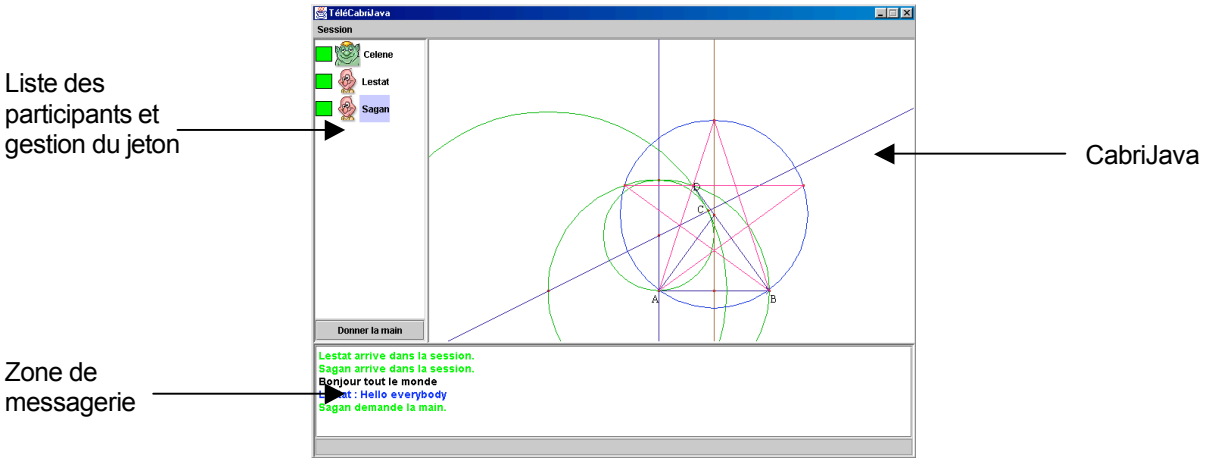
3.2. TéléCabriJava

3.2.1. Répartir CabriJava

TéléCabriJava répartira CabriJava sur Internet. Il offrira la possibilité à un ou plusieurs élèves et professeurs dispersés sur Internet d'éditer de manière coopérative une figure. A l'intérieure d'une session TéléCabriJava, les participants disposeront de la même figure et la synchronisation des figures affichées sera donc une priorité. Le langage Java sera adopté pour réaliser TéléCabriJava et n'imposera donc aucun choix matériel particulier.

TéléCabriJava constitue donc une couche au-dessus de CabriJava. Il sera indépendant autant que possible de CabriJava et sera indifférent à une refonte éventuelle de ce dernier. TéléCabriJava devra s'occuper uniquement de l'aspect réparti de CabriJava.

3.2.2. Interface graphique de TéléCabriJava



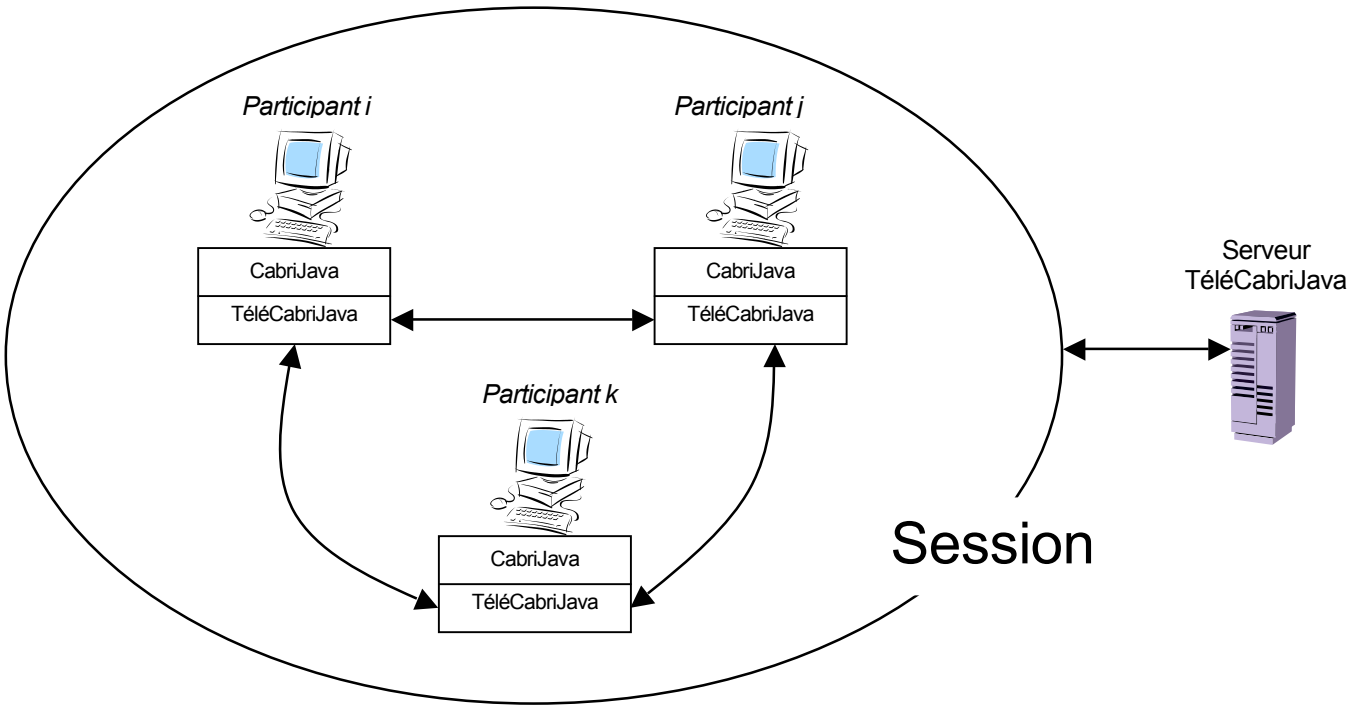
CabriJava sera localisé dans un panneau de la fenêtre de TéléCabriJava. Il y aura une panneau de messagerie pour pouvoir soumettre un commentaire à toute la session ou à l'un des participants. Pour se faire, on disposera de la liste des participants.

Comme on le verra en détail par la suite, seul un des participants d'une session peut modifier simultanément la figure géométrique pour des raisons évidentes de coopération. On dira que c'est le possesseur du jeton. Il y aura donc un panneau permettant de gérer ce jeton. L'utilisateur pourra y trouver le nom du possesseur actuel du jeton, il pourra aussi en faire une demande et enfin s'il le possède, il pourra le donner à un des participants de la session.

La barre de menu contiendra différentes rubriques permettant notamment d'ouvrir, fermer, sauvegarder une figure. On y trouvera aussi des commandes pour rejouer la construction d'une figure. L'utilisateur pourra aussi lancer un outil pour connaître la qualité de connexion des participants de la connexion et il pourra bien évidemment quitter la session.

3.3. Une session TéléCabriJava

Le serveur TéléCabriJava permet d'administrer les sessions TéléCabriJava, toutefois, il ne devra en aucun cas servir de relais aux messages que pourraient s'échanger les différents participants, tant destinés à un dialogue textuel, que liés aux actions du possesseur du jeton. Dans la mesure du possible, suivant que l'on utilisera un protocole de bas niveau (TCP, UDP, …), un RPC (Remote Procedure Call comme RMI, CORBA, …) ou un MOM (Messages Oriented Middleware comme JMS), on tentera de faire communiquer directement les différents participants d'une connexion en évitant des détours coûteux ou des goulots d'étranglement.



3.3.1. Description d'une session TéléCabriJava

- Dans une session, un seul des participants peut modifier la figure simultanément. On dit que « ce participant possède le jeton ».
- Le créateur d'une session possède par défaut le jeton.
- Le possesseur du jeton peut donner le jeton a un participant.
- Le possesseur du jeton peut s'en défaire et le rendre disponible.
- Les participants d'une session peuvent réclamer le jeton. Dés qu'il est disponible, il est allouer au premier qui l'a réclamé.
- Si un professeur réclame le jeton, il lui est alloué dès que possible.
- Si le possesseur du jeton se déconnecte, le jeton devient disponible.
- Tous les participants peuvent envoyer des messages qu'ils aient ou non le jeton. Les messages sont envoyés à tous les participants,

- Le possesseur du jeton peut :
  - créer une nouvelle figure,
  - modifier une figure,
  - ouvrir une figure qu'il possède localement et l'envoyer à tous les participants.
- Une modification de la figure par le possesseur du jeton ne se traduit pas par un envoi de la figure dans son intégralité aux autres participants de la session, seule la méthode de CabriJava qui a été utilisée pour modifier la figure est envoyée.
- Les participants doivent pouvoir suivre les actions du possesseur du jeton. Pour cela, chacun d'entre eux pourra suivre les déplacements du curseur sur la machine du possesseur de jeton. Le curseur sera modélisé sous forme d'une main par exemple et laissera le contrôle de la machine aux participants qui pourront déplacer leur curseur tout en observant le déplacement de celui du possesseur du jeton.
- Lors de l'arrivée d'un nouveau participant, le possesseur du jeton lui envoie la figure courante dans son intégralité. Nous n'envisageons pas de lui envoyer toutes les méthodes de CabriJava qui ont amené à sa construction actuelle. Une figure peut être envoyée sous deux formes :
  - soit au format XML ⇔ compact mais la machine destinataire devra reconstruire la figure,
  - soit en sérialisant sa représentation mémoire ⇔ plus volumineux que le format XML mais l'objet figure n'est pas à reconstruire.
- Un nouveau participant pourra obtenir un historique des messages et des constructions qui ont abouti à la figure qu'il vient de récupérer en joignant la session.

3.3.2. Travail en mode hors connexion

Il est tout à fait envisageable qu'un élève ou un professeur désire construire une figure hors connexion pour la soumettre lors d'une session future. De même, en cas de rupture involontaire de la session, le possesseur du jeton ne désire pas recommencer la figure à zéro mais préfère la continuer à partir du dernier état perçu par les participants. Dans ce cadre, un participant peut ouvrir une figure stockée localement et la soumettre aux autres participants s'il est possesseur du jeton. On peut aussi lui offrir la possibilité de rejouer automatiquement en vitesse accélérée la construction de la figure jusqu'à l'état dans lequel elle a été sauvegardée.

3.4. Le serveur TéléCabriJava

3.4.1. Administrer des sessions TéléCabriJava

Les sessions sont dynamiques, les utilisateurs peuvent les joindre, les créer ou les quitter. Un serveur TéléCabriJava permettra d'administrer les sessions. Il permettra en outre de récupérer des informations sur les sessions comme la liste des participants, la présence ou non d'un enseignant, un résumé de l'objectif de la session. Il pourra aussi indiquer la qualité des connexions des différents participants qui seront probablement connectés à Internet avec le câble ou l'ADSL. Chacun des participants d'une session aura l'application TéléCabriJava ainsi que la sous-application CabriJava qui tourneront localement sur sa machine.

3.4.2. Description des services offerts

Le serveur administre les sessions TéléCabriJava, il permet de :
- récupérer la liste des sessions en cours,
- joindre une session en cours,
- créer une nouvelle session.

Lors d'une demande de connexion au serveur, il y aura plusieurs cas :
- l'utilisateur n'est pas inscrit,
- l'utilisateur est un professeur,
- l'utilisateur est un élève.

L'identification du nouveau connecté est importante : un participant ne devra jamais pouvoir se faire passer pour quelqu'un d'autre.

Une fois connecté au serveur, l'utilisateur pourra récupérer :
- la liste des professeurs connectés,

- la liste des sessions que l'on pourra trier par localisation, par qualité de connexion, par le sujet de la session, …

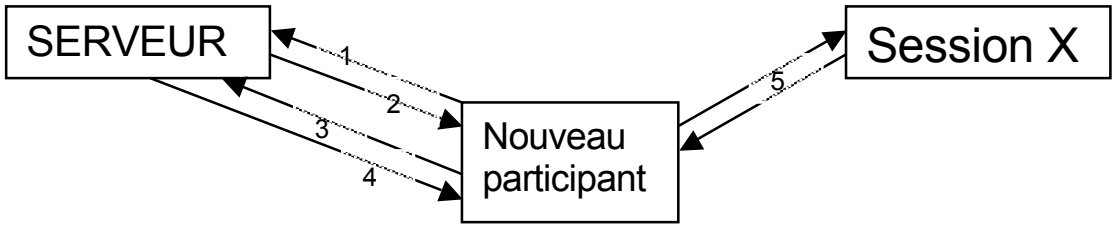
L'utilisateur pourra donc joindre une session de plusieurs manières :

- soit en joignant explicitement une session,
- soit en rejoignant la session d'un professeur.

Pour chaque session, il serait intéressant de récupérer les informations suivantes :

- son nom,
- son intitulé,
- l'URL de la page web qui lui est éventuellement rattachée,
- le nom de son initiateur,
- le nom des participants,
- le nombre de participants ainsi que sa capacité maximale,
- la langue utilisée pour communiquer par les participants,
- un indice de la rapidité de connexion des participants.

3.4.3. Connexion d'un participant à une session



1 ⇔ requête au serveur pour l'obtention de la liste des sessions en cours,
2 ⇔ récupération de la liste des sessions en cours,
3 ⇔ choix de la session X,
4 ⇔ récupération des adresses IP des participants de l'équipe,
5 ⇔ le nouvel arrivant est intégré dans la session.

3.4.4. Déconnexion d'un participant

Il y a deux types de déconnexions :

- un participant quitte la session,
- un participant est déconnecté involontairement de la session.

Dans le premier cas, il signale au serveur et aux autres participants qu'il quitte la session. Dans le second, le participant doit informer le serveur qu'il n'est plus dans la session. S'il n'est pas en état de le faire, le serveur détectera une rupture de la connexion qui le reliait à ce dernier et mettra automatiquement à jour sa liste de session. Il est impératif que le serveur connaisse à tout moment l'état exact des sessions. Enfin, il faut discerner si le participant déconnecté involontairement possédait le jeton et si c'est le cas, le donner à l'un des participants restants de la session.

3.4.5. Interface de communication avec le serveur TéléCabriJava

<table>
<tr><td rowspan="2">Fenêtre de discussion</td><td colspan="2">Nom de la session (en cours ou nouvelle)</td></tr>
<tr><td colspan="2">Session en cours | Professeurs connectés

Fenêtre contenant soit :
• la liste des sessions en cours avec leur descriptif
• la liste des professeurs connectés</td></tr>
<tr><td>Message pour la fenêtre de</td><td>Joindre</td><td>Créer</td></tr>
</table>

L'utilisateur peut soit :

- joindre une session en cours en la sélectionnant dans la liste (la description ainsi que le nom de la session sélectionnée s'affichent),
- joindre une session en cours en sélectionnant un professeur dans la liste (le nom de la session où se trouve le professeur sélectionné s'affiche dans le champ en haut de la fenêtre),
- créer une session : pour cela il commence par taper le nom de la nouvelle session dans le champ supérieur et il fournit sa description dans la fenêtre centrale.

3.5. Ressources réseaux pour l'exécution de TéléCabriJava

Lors de la réalisation de TéléCabriJava, il faut prendre en compte que les utilisateurs seront connectés à Internet via l'ADSL, le câble voire même des modems 56Kb. Ces accès à Internet sont de qualité très médiocre avec un RTT (Round Trip Time) très mauvais, des coupures de connexions ainsi que de faibles débits. Pour aggraver encore ce constat, j'ai pu observer des pauses de plusieurs secondes voir une minute au cours desquels aucun datagramme IP n'est échangé. Ce problème provient très probablement des fournisseurs d'accès à Internet qui offrent un service de mauvaise qualité.

Au cours de mon étude d'une éventuelle réalisation de TéléCabriJava avec le protocole TCP, j'ai procédé au test d'une connexion TCP avec deux modems 56Kbits. Il en ressort qu'un RTT (Round Trip Time ou Ping-Pong) nécessite 300ms en moyenne. Enfin, il a été établit qu'il est possible d'envoyer une centaine de segments de petite taille par seconde via une connexion TCP. Il apparaît aussi que par mon expérience de joueur sur Internet, le câble, l'ADSL ou un modem 56Kbits se comporte de la même façon et que les résultats de mon test d'une connexion TCP avec ces différents supports réseaux auraient été très similaires. La seule différence entre modem 56Kbits, câble et ADSL réside dans la bande passante qui est de 56Kbits pour un modem et de 512Kbits pour le câble et l'ADSL. Comme beaucoup de surfeur déçus par le câble ou l'ADSL, tout le monde sait maintenant qu'on a beau avoir une superbe bande passante, l'accès sera quand même mauvais si le fournisseur d'accès n'assure pas son service.

La suite du cahier des charges devra tenir compte de la piètre qualité d'accès à Internet des futurs utilisateurs de TéléCabriJava. TéléCabriJava sera développé à l'INRIA sur un LAN Ethernet 100Mbits, néanmoins, il sera testé régulièrement sur des connexions Internet standard afin de valider sa réalisation.

3.6. Estimation du nombre de messages envoyés par le possesseur du jeton

Rappelons que le possesseur du jeton est le participant d'une session TéléCabriJava qui a le droit de modifier la figure géométrique. De fait, c'est lui qui doit informer les autres participants des actions qu'il mène à l'intérieur de CabriJava.

On considère qu'il doit non seulement envoyer les modifications de la figure mais aussi le déplacement du curseur de sa souris ainsi qu'éventuellement les ouvertures de menus et différentes actions qu'il pourrait mener. L'intérêt de communiquer toutes ces informations aux participants est didactique ; cela permet de leur faire découvrir le fonctionnement de CabriJava mais aussi le déroulement exact de la construction de la figure.

On considère que le possesseur du jeton doit faire parvenir un minimum de 10 messages concernant ses actions par seconde à chacun des autres participants afin de maintenir une animation à peu près fluide chez ces derniers. Cela induit une certaine gourmandise en ressources réseaux de la part de TéléCabriJava. Ce nombre n'inclut pas les éventuels message nécessaires au bon fonctionnement de l'application comme la réalisation de ping-pong pour vérifier l'état des connexions.

3.7. Détails sur les messages envoyés par le possesseur du jeton

Le possesseur du jeton devra informer les autres participants des actions qu'il mènera dans CabriJava. Pour cela, dans le cas d'une modification de la figure ou du déplacement du curseur de la souris, il doit construire un message contenant la méthode à appeler sur les machines distantes ainsi que le ou les objets géométriques qui entrent en jeu dans cette modification.

L'ensemble des objets d'une figure Cabri est contenu dans un vecteur. Chaque objet est ainsi numéroté et on pourra le référencer par son numéro lors de l'appel d'une méthode distante.

Si on décide de réaliser TéléCabriJava sans passer par un Remote Procedure Call, il faudra trouver une représentation pour chacune des méthodes distantes à appeler par le possesseur du jeton.

3.8. Evaluation du nombre de participants d'une session TéléCabriJava

On envisage pour l'instant TéléCabriJava comme un outil pour dispenser des cours particuliers de géométrie. Il y aura donc à l'intérieure d'une session un professeur et un élève. Toutefois, le nombre d'élèves pourra éventuellement être supérieur et l'on considèrera que certains des élèves sont spectateurs. Néanmoins, il est déraisonnable que ce nombre puisse croître à l'infini et on le limitera donc en fonction des ressources réseaux disponibles. Cette limite qui sera donc dynamique permettra d'éviter que les participants ne voient la qualité de leur session s'effondrer suite à l'arrivée de participants indésirables. D'autre part, l'initiateur de la session pourra spécifier un nombre maximal de participants. Nous pensons que, raisonnablement, ce nombre ne doit pas dépasser cinq pour des raisons didactiques.

3.9. Contraintes de synchronisation de la messagerie et de l'édition de la figure

Seul le possesseur du jeton peut modifier la figure, toutefois tous les participants peuvent envoyer des messages textuels. Il faut donc veiller à ce que l'arrivée de ces messages et l'état courant de la figure soient synchronisés chez tous les participants. En ce qui concerne le possesseur du jeton, étant donné que tous les middlewares sont basés sur le protocole TCP qui garantit l'ordre d'arrivée des segments, on est certain que les actions et les éventuels messages textuels qu'il enverra, arriveront de manière synchronisée chez les autres participants.

Le problème se pose donc chez les participants qui n'ont pas le jeton. Si l'un d'eux envoie un message textuel et que celui ci n'est pas délivré immédiatement aux autres participants de la session suite à un incident réseau, ce message arrivera probablement avec un décalage intolérable par rapport à l'état courant de la figure. Pour pallier ce problème, il faudra dater les messages pour déterminer s'ils sont hors contexte.

3.10. Gestion des problèmes réseaux

J'ai effectué au cours de mon DEA des tests « ping-pong » et « envoi de segments » d'une connexion TCP entre deux modems 56Kb. Il en ressort que la durée moyenne d'un ping-pong se situe autour des 300ms avec des minima à 200ms et des pics atteignant souvent les 500ms voir plusieurs secondes pour certains d'entre eux. Il apparaît aussi que l'on peut faire transiter une centaine de petits segments par secondes via une connexion TCP quand il n'y a pas de problème réseau.

3.10.1. Mise en place d'un tampon

Il semble donc évident qu'il y aura des interruptions dans le transit des actions menées par le possesseur du jeton vers les autres participants. On peut imaginer de mettre en place une sorte de tampon chez les participants qui recevraient ces actions. Ce tampon stockerait l'équivalent de 500ms d'actions, ce qui permettrait de rendre transparente de courtes congestions réseau. Bien sûr la mise en place d'un tampon augmente la latence entre le moment où le possesseur du jeton modifie la figure et celui où les participants reçoivent cette action. On peut raisonnablement tabler sur une latence réseau de 150ms, ce qui nous fait 650ms au total avec le tampon. C'est une durée tout à fait raisonnable qui permet de garder une bonne synchronisation entre les actions du possesseur du jeton et les messages éventuels des autres participants.

3.10.2. Cas de débordement du tampon

Pour quelqu'un qui joue sur Internet, il n'est pas difficile d'imaginer des « pauses » réseaux de l'ordre de 10s. Dans ce cas, le tampon se vide et l'utilisateur ne peut plus suivre les actions opérées par le possesseur du jeton. On pourra indiquer à l'utilisateur que le réseau est congestionné mais se pose le problème de la reprise quand les ressources réseaux seront rétablies. En effet, l'utilisateur va recevoir une masse de segments et il doit rattraper le cours actuel de la session. On va donc lui afficher les messages et les actions opérées sur la figure en vitesse accélérée jusqu'au moment où le tampon sera de nouveau approvisionné et que tout sera rentré dans l'ordre.

Si c'est le possesseur du jeton qui subit des congestions réseaux, on va lui interdire de continuer à modifier la figure pour éviter qu'il ne sature les autres participants de la session lorsqu'il pourra à nouveau communiquer avec eux. Dans ce cas, il faudra par ailleurs signaler aux participants que le possesseur du jeton est momentanément coupé du réseau.

4. Etude des outils existants pour la réalisation de TéléCabriJava

J'ai procédé à une petite expertise des outils de communication disponibles pour la réalisation d'une [application répartie] comme TéléCabriJava. Voici les catégories que j'ai étudiées :

- les protocoles réseaux de la couche transport du modèle OSI,
- les MOM (Message Oriented Middleware) qui nous proposent une communication par messages de façon asynchrone,
- les RPC (Remote Procedure Call) qui permettent d'appeler des méthodes distantes de façon synchrone.

J'entends par synchrone le fait que l'application lors d'un appel de méthode distante est bloquée jusqu'au retour du résultat de l'exécution de cette méthode. Lors de communications asynchrones, l'application au contraire continue son exécution après avoir envoyé son message et n'attend pas de réponse immédiate de l'application distante.

J'ai ainsi étudié les protocoles TCP et UDP. J'ai découvert le MOM JORAM développé par l'équipe SIRAC. Parmi les RPC, j'ai passé en revue les célèbres RMI et CORBA. Je n'exposerai que RMI dans ce rapport car CORBA est très similaire. Bien sûr, je fais totalement abstraction des services offerts par CORBA (persistance, fiabilité, …) inexistants chez son petit frère. Ces services ne sont de toute façon pas intéressants pour notre application comme on le verra par la suite.

4.1. Réalisation avec le protocole TCP

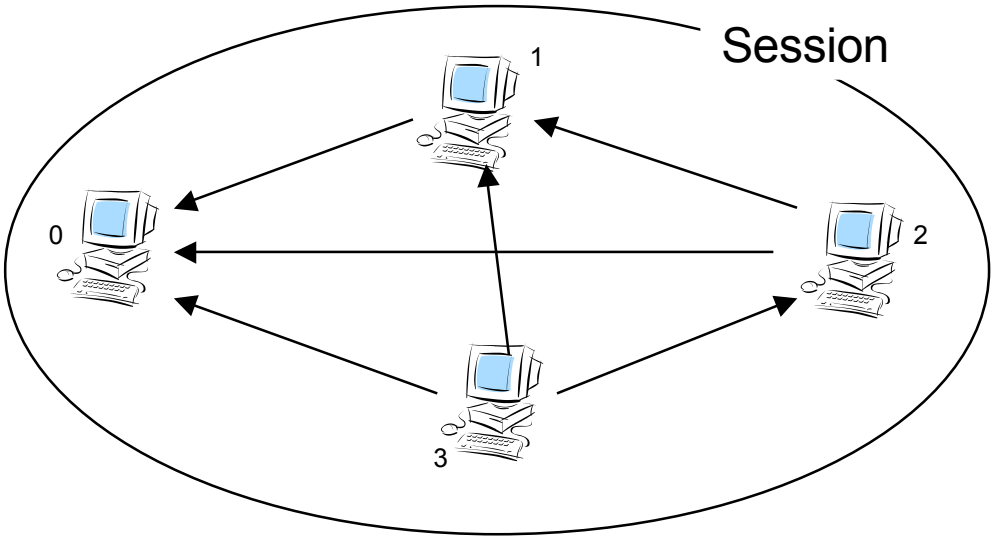
4.1.1. Description rapide du protocole TCP

- TCP est un [protocole] de la couche transport du modèle OSI, il est orienté connexion, il fournit une communication point à point asynchrone,
- TCP garantit que les données émises arriveront au destinataire,
- TCP garantit que la réception des données se fait dans le même ordre que leur émission,
- Les sockets utilisés par TCP sont full duplex. On peut donc émettre et recevoir simultanément sur une connexion TCP,
- TCP n'offre aucun autre service et notamment pas les services de persistance ou de résistance aux pannes réseaux de CORBA.

Cette description de TCP est vraiment succincte et vous pourrez trouver en annexes des détails sur tout ce qui se cache derrière ce protocole.

4.1.2. Modélisation d'une session TéléCabriJava en TCP

4.1.2.1. Première solution : interconnexion de tous les participants



Les machines sont numérotées suivant leur ordre d'arrivée dans la session. Le sens des flèches indique quelle est la machine qui établi la connexion : les machines déjà présentes dans la session attendent les connexions des nouveaux arrivants. Les *sockets* sont full duplex et permettent l'attente et la réception de messages simultanément, un seul *socket* est donc nécessaire pour connecter deux machines. A chaque *socket*, on associe un processus léger qui boucle en attente de segments.

Evaluation

Pour la suite de ce chapitre, on définit des variables afin d'évaluer le coût des différentes réalisations :

- *n ⇔ le nombre de participants d'une session,*
- *x ⇔ le nombre de messages par seconde nécessaires au possesseur du jeton pour informer les participants des actions qu'il mène dans CabriJava,*
- *k ⇔ le nombre de messages textuels envoyés par un participant par seconde.*

Nombre de sockets nécessaires pour n machines :

0+1+2+…+(n-1)

Or, (0+(n-1))+(1+(n-2))+…+((n-1)+0)=n(n-1)

D'où, 0+1+2+…+(n-1)=n(n-1)/2, soit de l'ordre de n_/2.

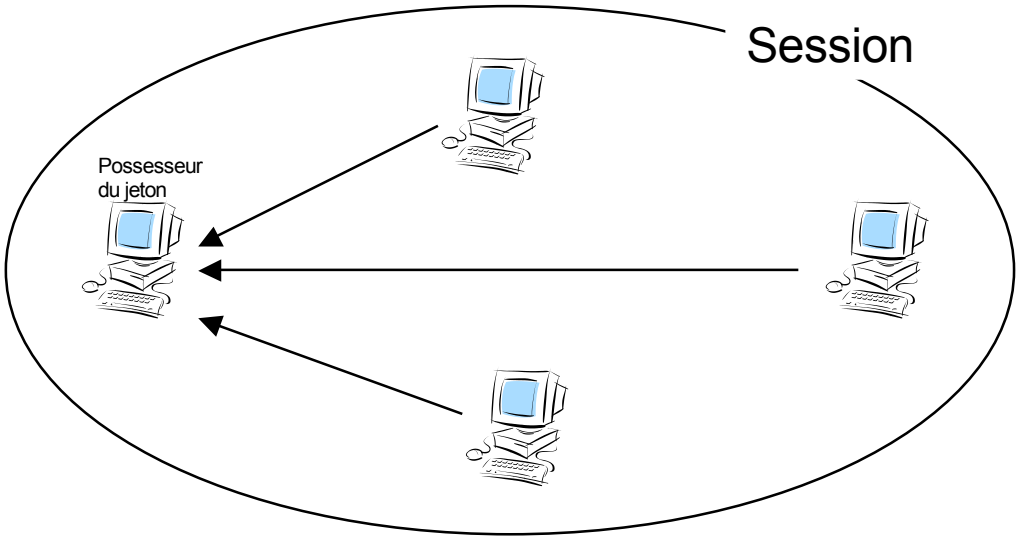
Nombre de segments à envoyer par secondes par les participants d'une session:

- par le possesseur du jeton :

- x(n-1) pour informer les participants de ses actions,
- k(n-1) pour les messages textuels,

- par les autres participants :
  - k(n-1) pour les messages textuels.

4.1.2.2. Deuxième solution : interconnexion du possesseur du jeton aux autres participants



On peut faire les mêmes remarques que pour la solution précédente :

- le sens des flèches indique quelle est la machine qui établit la connexion,
- un *socket* nécessaire pour connecter deux machines en *full duplex*,
- on associe un processus léger qui boucle en attente de messages à chaque *socket*.

Evaluation

Nombre de sockets nécessaires pour n machines :

n-1

Nombre de segments à envoyer par secondes par les participants d'une session:

- par le possesseur du jeton :
  - x(n-1) pour informer les participants de ses actions,
  - k(n-1) pour les messages textuels,
  - k(n-1)(n-2) pour la re-direction des messages des participants,
- par les autres participants :
  - k pour les messages textuels.

Comparaison avec la solution précédente

Cette solution offre l'avantage de nécessiter beaucoup moins de sockets. Toutefois, elle est plus lourde que la précédente pour le possesseur du jeton qui doit se charger de rediriger les messages des participants. D'autre part, si le possesseur du jeton change, toutes les connexions TCP sont à refaire. On préfèrera donc la solution précédente.

4.1.3. Benchmark de TCP et évaluation sur la solution « interconnexion totale »

J'ai procédé aux tests « ping-pong » et « envoi de message » d'une connexion TCP avec deux modems 56 Kbits. Un ping-pong s'effectue en une durée moyenne de 300ms et le débit atteint environ une centaine de petits segments à la seconde. Si on se réfère à la première solution qui interconnecte tous les participants et que l'on fixe ***x*** à 15 messages par seconde, ***n*** le nombre de participants peut atteindre 7 ce qui est plus qu'honorable.

4.2. Réalisation avec le protocole de bas niveau UDP

4.2.1. Description rapide du protocole UDP

- UDP est un [protocole] de la couche transport du modèle OSI, il n'est pas orienté connexion, il permet d'envoyer des paquets de données indépendants appelés *datagrammes*,
- UDP permet la diffusion d'un *datagramme* à un groupe de machine,
- UDP contrairement à TCP ne garantit pas l'arrivée des *datagrammes*, ni l'ordre d'arrivée des *datagrammes*,
- enfin, UDP comme TCP ne fournit aucun des services de CORBA (persistance, résistance aux pannes réseaux, …).

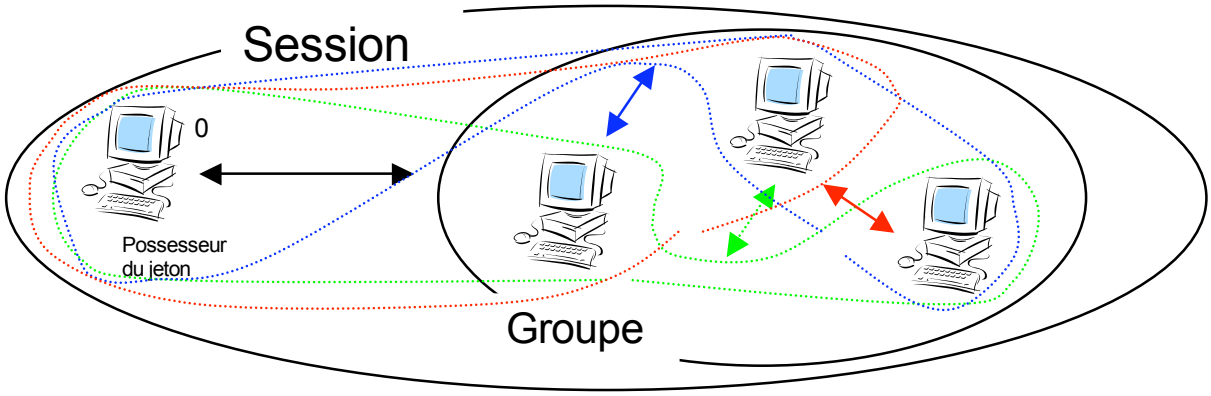
On s'aperçoit immédiatement de la complexité de mise en œuvre de TéléCabriJava avec UDP. Il faudrait commencer par ré-implanter tous les services de TCP, à savoir :

- numéroter les *datagrammes*,
- bufferiser les *datagrammes* dont le numéro ne correspond pas au numéro attendu,
- s'il s'avère qu'un *datagramme* est perdu, demander sa retransmission,
- vérifier de temps en temps que la pseudo-connexion est encore opérationnelle en envoyant un *datagramme* de test,
- gérer l'ouverture et la fermeture de la pseudo-connexion.

4.2.2. Modélisation d'une session TéléCabriJava en UDP

Les deux solutions proposées avec le protocole TCP sont envisageables en UDP mais totalement inintéressantes car elles ne profitent pas de l'atout majeur d'UDP à savoir le *multicast*. Le *multicast* permet la diffusion d'un message par un serveur à un groupe de clients pour un coût équivalent à celui d'un envoi d'un message à une seule machine. Les clients du groupe peuvent par ailleurs envoyer des messages au serveur mais pas aux autres membres du groupe.

4.2.2.1. Première solution : « interconnexion de tous les participants »



On a modélisé ici les n-1 groupes possibles dans une session de n participants. Le possesseur du jeton peut envoyer les actions qu'il mène dans CabriJava à tous les participants simultanément. De même, les participants du groupe peuvent envoyer des messages textuels à tous les autres participants simultanément. A chaque `MulticastSocket`, on associe un processus léger qui boucle en attente de messages.

Les serveurs qui envoient des datagrammes à un groupe utilisent un `DatagramSocket`. Les `MulticastSocket` sont utilisés du côté des clients pour recevoir des paquets diffusés au groupe par un serveur.

Evaluation

Pour la suite de ce chapitre, on définit des variables afin d'évaluer le coût des différentes réalisations :

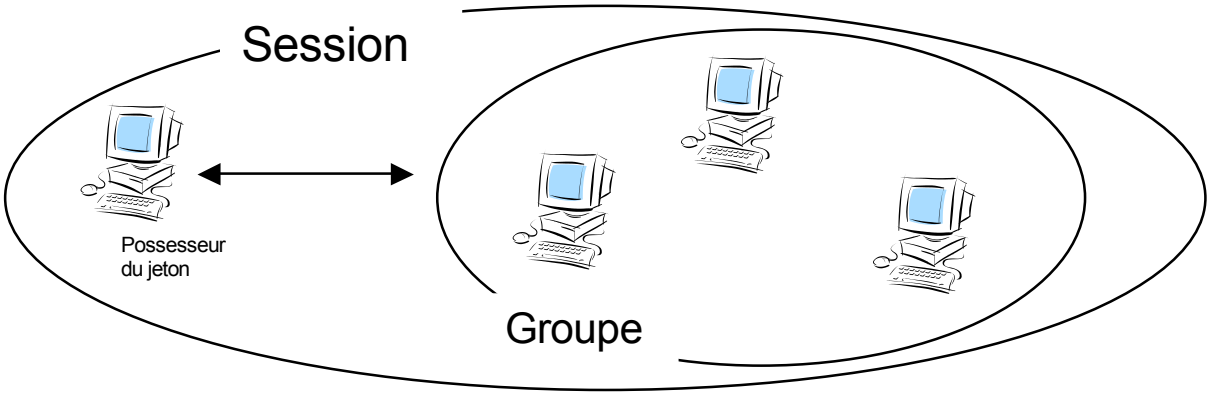
- *n ⇔ le nombre de participants d'une session,*
- *x ⇔ le nombre de messages par seconde nécessaires au possesseur du jeton pour informer les participants des actions qu'il mène dans CabriJava,*

- *k ⇔ le nombre de messages textuels envoyés par un participant par seconde,*
- *p ⇔ le pourcentage de datagrammes perdus par participant par seconde (on peut l'estimer à 1 pour 1000 soit 0.1%).*

Nombre de sockets nécessaires pour n machines :
pour la diffusion :
il y a n groupes de n-1 clients
n `DatagramSocket` + n(n-1) `MulticastSocket`
soit un total de n_

Nombre de datagrammes à envoyer par seconde par les participants d'une session:
- par le possesseur du jeton :
  - x pour informer les participants de ses actions,
  - k pour les messages textuels,
  - p(x+k)(n-1) pour la ré-émission en cas de perte de datagrammes,
- par les autres participants :
  - k pour les messages textuels,
  - p(x+k(n-1)) pour la demande de ré-émission en cas de perte de datagrammes.
  - pk(n-1) pour la ré-émission des messages textuels perdus.

4.2.2.2. Deuxième solution : « connexion du possesseur du jeton aux autres participants »



Comme dans la seconde solution proposée avec le protocole TCP :
- le possesseur du jeton peut communiquer directement avec les autres participants,
- les participants doivent passer par le serveur pour s'envoyer des messages,
- si le possesseur du jeton change, il faut reconfigurer la session,
- cette solution est économique en *sockets*,
- on associe un processus léger qui boucle en attente de messages à chaque *socket*.

Evaluation

Nombre de sockets nécessaires pour n machines :
1 `DatagramSocket` pour le serveur + n-1 `MulticastSocket` pour les clients du groupe

Nombre de datagrammes à envoyer par secondes par les participants d'une session:
- par le possesseur du jeton :
  - x pour informer les participants de ses actions,
  - k pour les messages textuels,
  - p(x+k)(n-1) pour la ré-émission en cas de perte de datagrammes,
  - k(n-1) pour la re-direction des messages des participants,
  - pk(n-1)_ pour la ré-émission des messages textuels perdus par les participants,
- par les autres participants :
  - k pour les messages textuels,
  - p(x+k(n-1)) pour la demande de ré-émission en cas de perte de datagrammes,
  - pk(n-1) pour la demande de ré-émission des messages textuels perdus.

4.2.3. Conclusion concernant UDP et la multi-diffusion

L'entête des datagrammes d'UDP est minimal, ce qui permet à UDP d'être plus performant que TCP. Toutefois, il faut émettre des réserves car hautes performances ne rime pas ici avec qualité de service : UDP n'offre pas la garantie d'arrivée, ni l'ordre d'arrivée de ses datagrammes. Ces deux services étant indispensables pour la réalisation de TéléCabriJava, il faudrait les ré-implanter avec UDP alors qu'ils sont fournis avec TCP. De plus, cela réduira sensiblement les performances attrayantes d'UDP.

Un des autres points forts d'UDP est la multi-diffusion. Son intérêt serait majeur pour TéléCabriJava où le possesseur du jeton doit envoyer les actions qu'il mène dans CabriJava à tous les autres participants. Malheureusement, il s'avère que la diffusion à un groupe est impossible car il n'est pas possible d'allouer dynamiquement une adresse IP à un nouveau groupe TéléCabriJava sur Internet. Tout l'attrait d'UDP s'effondre alors face à TCP.

La multi-diffusion utilise les adresses de réseau de classe D. Les quatre bits de poids forts de ces adresses sont 1, 1, 1 et 0, ce qui correspond aux valeurs décimales allant de 224 à 239.

L'IANA (Internet Assigned Numbers Authority) maintient une liste d'adresses IP pour la diffusion à des groupes. Les groupes les plus connus sont les suivants :

Groupe	Description	Net/3 constante
224.0.0.0	Réservé	INADDR_UNSPEC_GROUP
224.0.0.1	tous les systèmes sur ce subnet	INADDR_ALLHOSTS_GROUP
224.0.0.2	tous les routeurs sur ce subnet	
224.0.0.3	non assigné	
224.0.0.4	routeurs DVMRP	
224.0.0.255	Non assigné	INADDR_MAX_LOCAL_GROUP
224.0.1.1	NTP (Network Time Protocol)	
224.0.1.2	SGI-Dogfight	

Enfin, d'un point de vue personnel, je trouve qu'il est déraisonnable d'utiliser un protocole comme UDP qui ne fait aucun contrôle de congestion du réseau et qui a forte tendance à s'accaparer toute la bande passante face à des protocoles comme TCP qui tentent de réguler leur débit. Pour ces multiples raisons, le protocole TCP s'impose face à UDP.

4.3. Réalisation avec RMI

L'idée dissimulée derrière [RMI] est de réunir une famille d'objets qui collaboreraient tous entre eux et qui pourrait être disséminés sur Internet. Ces objets sont naturellement censés communiquer entre eux au moyen de protocoles standards et par l'intermédiaire d'un réseau.

RMI est un service fournit par Java et ses objets doivent impérativement être écrits en Java à la différence de CORBA qui permet d'implanter ses interfaces décrites en IDL (pseudo C) avec un large choix de langages.

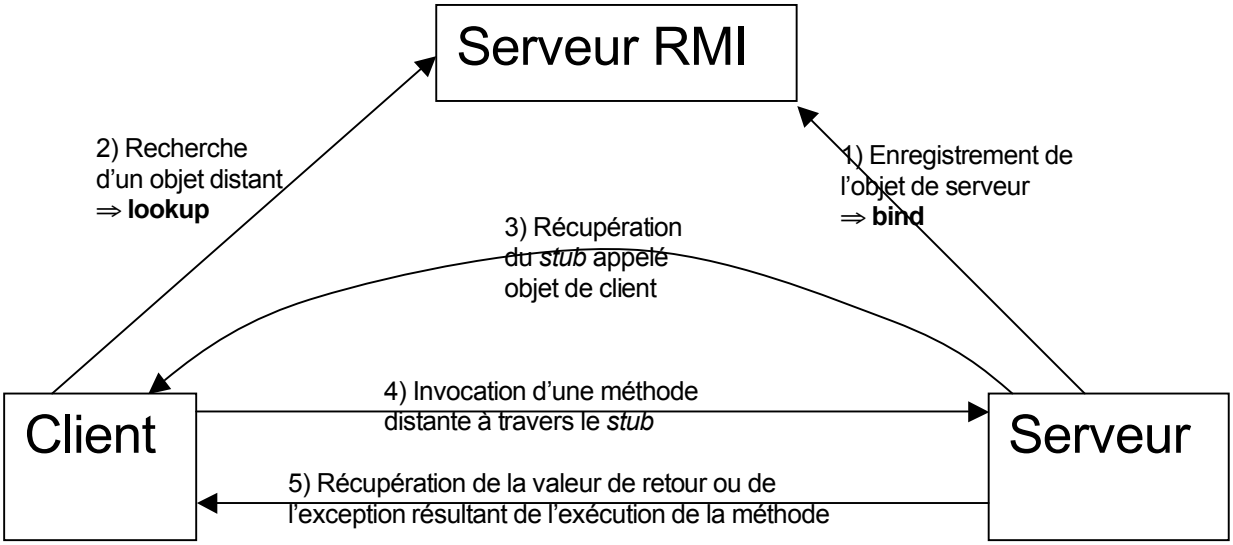
RMI est bâti sur le protocole TCP et son fonctionnement est bien plus compliqué que ce dernier, je décris donc ci-dessous brièvement ses principes.

4.3.1. Invocations de méthodes distantes

RMI est un **Remote Procedure Call** (appel de procédures à distance). Il permet l'invocation de méthodes distantes par le client. Si vous avez accès à un objet d'une machine distante, vous pouvez appeler les méthodes de cet objet distant. Naturellement, les paramètres de ces méthodes doivent être fournis d'une manière ou d'une autre à l'ordinateur distant, l'objet doit être informé de la méthode à exécuter et la valeur de retour doit être renvoyée correctement. RMI gère automatiquement tous ces détails.

Selon la terminologie RMI, l'objet dont la méthode effectue un appel à distance est appelé l'objet de client. L'objet distant est appelé l'objet de serveur.

Le serveur dispose d'un serveur HTTP permettant au client de récupérer l'objet de client.



4.3.2. Stubs et encodage des paramètres

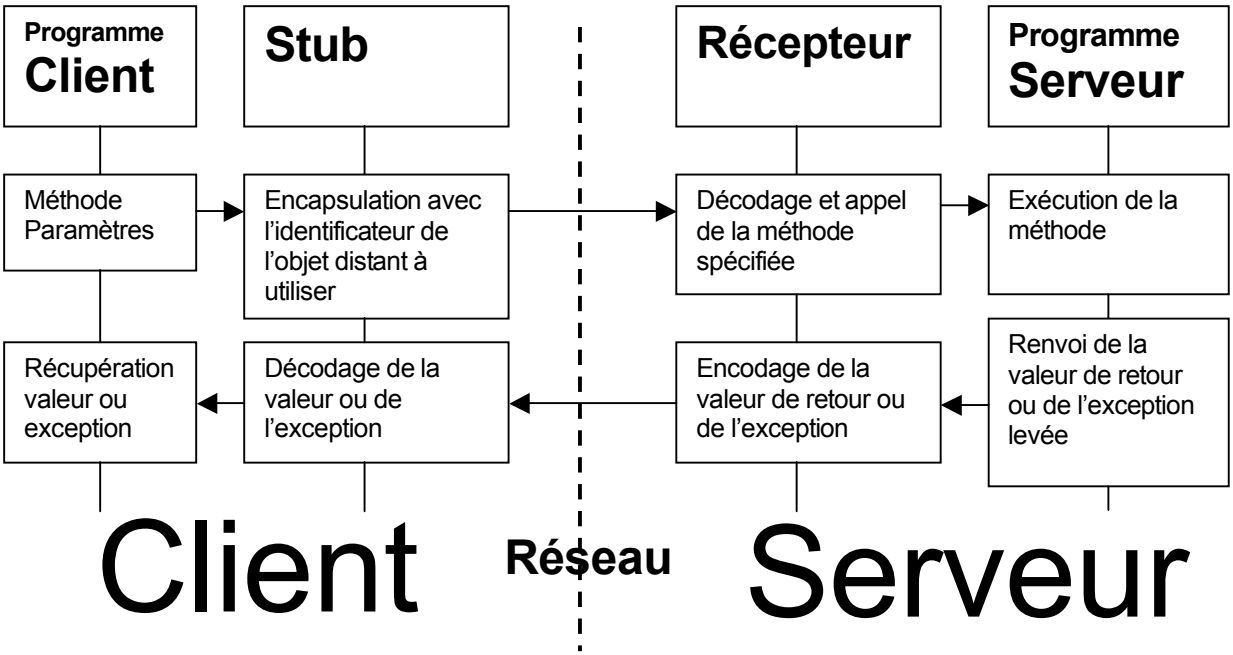
Lorsque le code d'un client veut invoquer une méthode distante sur un objet distant, il commence en fait par appeler une méthode ordinaire du langage de programmation Java, qui est encapsulée dans un objet de substitut appelé un *stub*. Ce *stub* se trouve sur la machine cliente et non sur le serveur. Il rassemble tous les paramètres utilisés par la méthode distante dans un bloc d'octets ce qui lui permet d'obtenir pour chaque paramètre un codage indépendant de la machine.

Pour résumer la méthode du *stub* sur le client construit un bloc de données composé de :

- un identificateur de l'objet distant à utiliser,
- une description de la méthode à appeler,
- les paramètres encodés.

Le *stub* envoie alors ces informations au serveur. Du côté du serveur, un objet de réception effectue les actions suivantes pour chaque appel de méthode distante :

- il décode les paramètres encodés,
- il situe l'objet à appeler,
- il appelle la méthode spécifiée,
- il capture et encode la valeur de retour ou l'exception renvoyée par l'appel,
- il envoie un bloc de données correspondant à la valeur de retour encodée au *stub* du client.



Ce processus est évidemment assez complexe, mais la bonne nouvelle est qu'il est entièrement automatique et totalement transparent pour l'utilisateur. De plus, les concepteurs des objets Java distants ont fait de leur mieux pour que ceux-ci aient le même comportement que les objets locaux. La syntaxe pour un appel à une méthode distante est la même que pour une méthode locale.

4.3.3. Interfaces et implantation

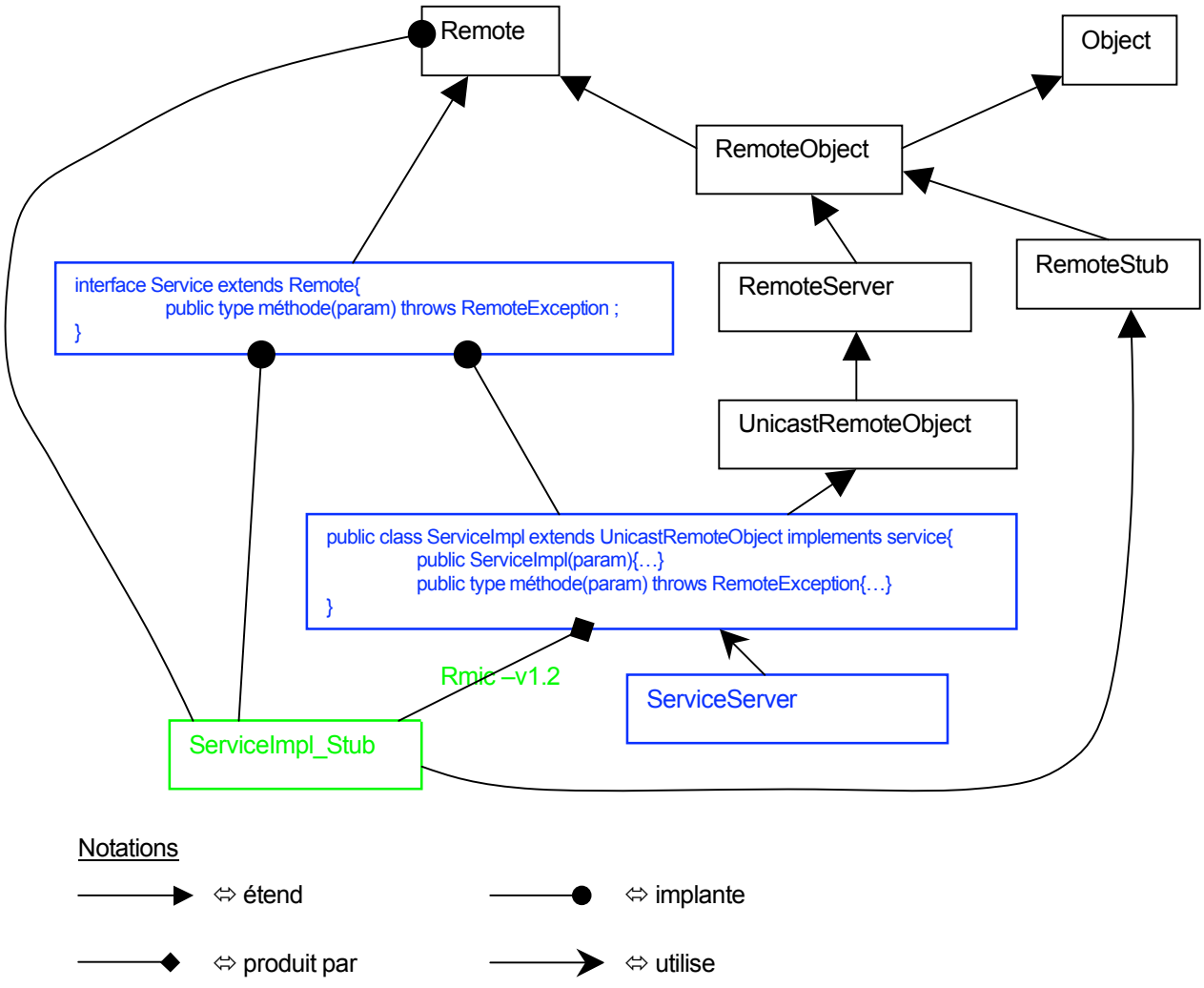
Le code du client se sert toujours de variables d'objet dont le type est une interface dans le but d'accéder à des objets distants. Les interfaces sont des entités abstraites qui se contentent d'indiquer quelles méthodes peuvent être appelées avec les signatures correspondantes. Les variables dont le type est une interface doivent toujours se trouver à l'intérieur d'un objet réel du même type. Lorsque vous appelez des méthodes distantes, la variable d'objet fait référence à un objet de *stub* (le *stub* est généré à partir de l'interface).

Le client doit pouvoir manipuler des objets de serveur, mais il ne dispose pas en réalité de leur copie. Les objets eux-mêmes résident sur le serveur. Le code du client doit en permanence savoir ce qu'il est en mesure de faire avec ces objets. Leurs caractéristiques sont détaillées dans une interface partagée entre le client et le serveur, interface qui se trouve par conséquent sur chacune des deux machines.

Toutes les interfaces des objets distants doivent étendre l'interface *Remote* définie dans *Java.rmi*. Toutes les méthodes de ces interfaces doivent également déclarer qu'elles peuvent déclencher une exception *RemoteException*. Ces déclarations doivent être effectuées parce que les appels de méthodes distantes sont intrinsèquement moins fiables que les appels locaux. Le serveur ou la connexion peuvent être temporairement hors service, ou encore le réseau peut avoir un problème.

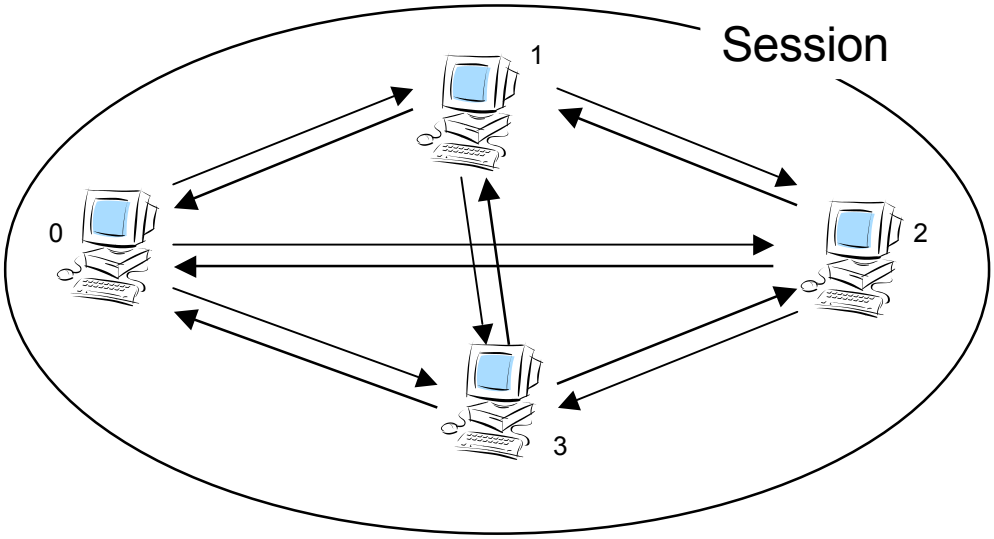
Au niveau du serveur, on implante la classe qui gère réellement les méthodes présentées dans l'interface. Cette classe étend *UnicastRemoteObjet* qui est une classe concrète de la plate-forme Java et qui rend les objets accessibles à distance. Un objet *UnicastRemoteObject* réside sur un serveur. Il doit être actif lorsqu'un service est demandé et il doit être disponible au travers du protocole TCP/IP.

On génère les *stubs* à partir des interfaces en utilisant l'outil *rmic*.



4.3.4. Modélisation d'une session TéléCabriJava avec RMI

4.3.4.1. Première solution : interconnexion de tous les participants



Chaque machine héberge un objet de serveur (implantation de l'interface). Elle récupère aussi un s*tub* par objet distant. Chaque machine fait donc serveur et client. Il faut donc deux connexions TCP pour la communication entre deux machines. Les machines sont numérotées suivant leur ordre d'arrivée dans la session. Les flèches pointent sur les machines serveur.

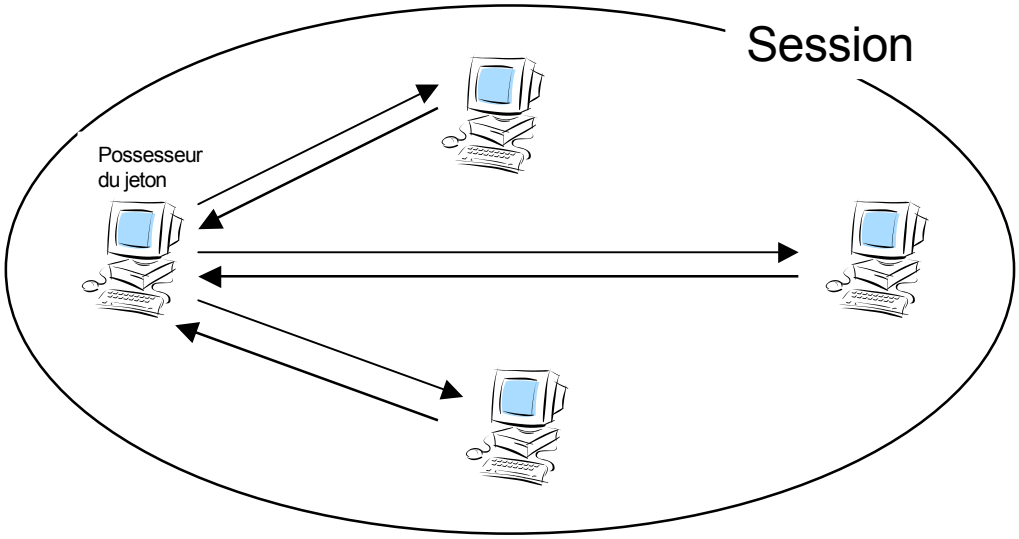
Nombre de sockets nécessaires pour n machines :

2(0+1+2+…+(n-1))
Or, (0+(n-1))+(1+(n-2))+…+((n-1)+0)=n(n-1)
D'où, 2(0+1+2+…+(n-1))=n(n-1)
Soit de l'ordre de n_.

Nombre de méthodes distantes à invoquer par secondes par les participants d'une session:

- par le possesseur du jeton :
  x(n-1) pour informer les participants de ses actions,
  k(n-1) pour les messages,
- par les autres participants :
  k(n-1) pour les messages.

4.3.4.2. Deuxième solution : interconnexion du possesseur du jeton aux autres participants



Seule la machine hébergeant le possesseur du jeton possède des s*tubs* vers chacun des autres participants de la session. Ces derniers ont un seul *stub* qui leur permet uniquement d'appeler des méthodes sur l'objet de serveur du possesseur du jeton. Chacune des machines héberge un objet de serveur. Comme dans la solution précédente, le possesseur du jeton peut informer directement les autres participants d'une modification de la figure.

Toutefois, les inconvénients par rapport à une interconnexion de tous les participants sont :

- les participants ne peuvent plus envoyer directement leurs messages : tout transite par le possesseur du jeton,
- quand le possesseur du jeton change, tous les *stubs* sont à re-générer.

Nombre de sockets nécessaires pour n machines :

2(n-1)

Nombre de méthodes distantes à invoquer par secondes par les participants d'une session:

- par le possesseur du jeton :
  - x(n-1) pour informer les participants de ses actions,
  - k(n-1) pour les messages,
  - k(n-1)(n-2) pour la redirection des messages des participants,
- par les autres participants :
  - k pour les messages.

4.3.5. Avantages et inconvénients de RMI pour TéléCabriJava

4.3.5.1. Avantages

- RMI étant un RPC, il permet d'appeler les méthodes distantes de façon totalement transparente et nous décharge des problèmes liés au transport des paramètres des méthodes (par sérialisation et dé-sérialisation).

4.3.5.2. Inconvénients

- Bâti sur TCP et donc plus lent, dépendance de l'application vis à vis des serveurs RMI qui hébergent les objets de serveurs,
- Les appels aux méthodes distantes sont synchrones et donc bloquants car les méthodes attendent soit un paramètre de retour soit une exception. Un appel d'une méthode distante dure donc le temps d'un RTT (Round Trip Time). Or, dans TéléCabriJava, nous sommes intéressés par l'envoi de message, c'est à dire par l'appel de méthode distante sans retour de paramètre et ne levant pas d'exception. RMI se révèle donc très défavorable puisqu'un RTT avec un modem prend environ 300ms et que l'on désire envoyer plus de 10 messages par seconde. Pour éviter d'être totalement

bloqué par RMI, il faut donc créer un processus léger par appel de méthode distante. Cette stratégie n'est raisonnablement pas envisageable.

- Dans les RPC, les serveurs sont totalement passifs, ils attendent les requêtes des clients. Dès lors, puisque toutes les machines d'une session doivent pouvoir communiquer (envoyer et recevoir), elles doivent être à la fois serveur et client. Or, une machine qui veut faire serveur doit rendre son *stub* disponible à d'éventuels clients et donc disposer d'un serveur HTTP. Rappelons que l'application TéléCabriJava est destinée à des utilisateurs supposés non initiés en Informatique et l'on s'aperçoit que la mise en place d'un serveur HTTP peut être source de problèmes (installation, configuration, firewall, …).

4.4. Réalisation avec Joram

[JORAM] est un MOM (Message Oriented Middleware) qui implante JMS (Java Message Service). Il utilise la technologie basée sur les agents AAA (Agent Anytime Anywhere). Il est actuellement développé par l'équipe SIRAC.

4.4.1. Objectifs de AAA (Agent Anytime Anywhere)

AAA offre un modèle de programmation distribuée par un mécanisme de messagerie asynchrone. C'est à dire qu'il garantit l'arrivée des messages mais pas le moment où ils arriveront. Les caractéristiques principales de AAA sont :

- orienté agents ⇔ un agent est un programme autonome qui a son propre état, qui s'exécute simultanément avec d'autres agents et qui interagit avec eux,
- asynchronisme ⇔ les agents interagissent entre eux par l'intermédiaire de communications asynchrones,
- notifications ⇔ les agents communiquent en envoyant et recevant des notifications. Une notification représente une transition de l'état d'un agent,
- événement, réaction ⇔ les agents réagissent aux notifications qu'ils reçoivent, parfois aussi en envoyant des notifications.

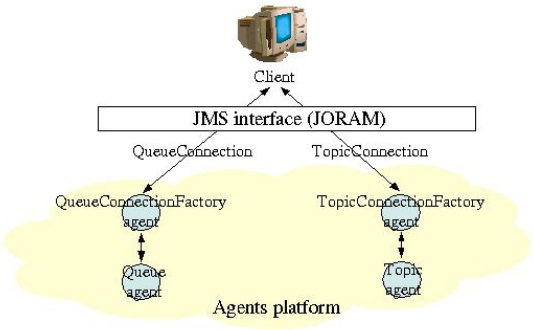
4.4.2. Services fournis par AAA

La technologie AAA fournit à JORAM les services suivants :

- la distribution ⇔ à savoir qu'on peut répartir les agents sur un réseau,
- la persistance ⇔ tous les messages envoyés arriveront d'une manière ou une autre au destinataire, le serveur JORAM stocke de manière fiable les messages reçus de telle sorte qu'il puisse les acheminer après un éventuel plantage,
- l'atomicité ⇔ toutes les actions entreprises par JORAM devront, soit être totalement accomplies, soit annulées.

4.4.3. JMS au sommet des agents

Les spécifications de JMS indiquent que deux types d'objets doivent être crées durant la phase d'administration pour être utilisés ultérieurement comme support de communication par les clients JMS. D'une part, on a les objets de destination, *Queue* et *Topic*, lesquels s'occupent de l'envoi et de la réception des messages. D'autre part, on a les objets *ConnectionFactory* utilisés pour permettre aux clients JMS de se connecter aux objets de destination.

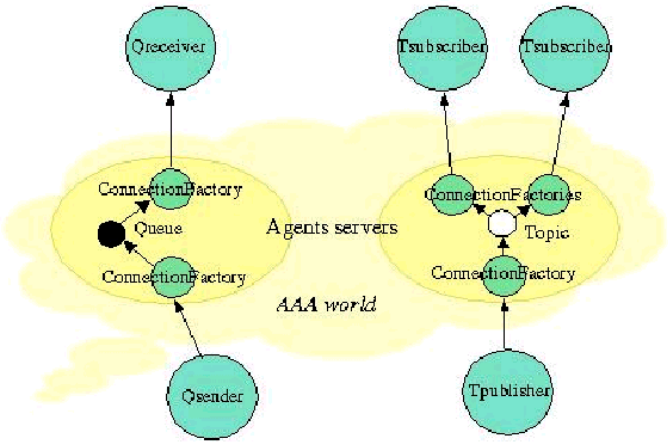


JMS on top of the agents

Les concepts de *Queue* et de *Topic* des MOM sont implantés comme des agents. Après avoir été déployés sur les serveurs à agents, ils sont accessibles et utilisables comme supports de communication par les clients JMS à travers JORAM qui implante l'interface JMS.

4.4.4. Queue et Topic de JORAM

Ainsi, les clients JMS peuvent accéder aux agents *Queue* et *Topic* à travers l'interface JMS et les agents *ConnectionFactory*. Expliquons maintenant ce que sont les *Queue* et *Topic* à l'aide du schéma ci-dessous.



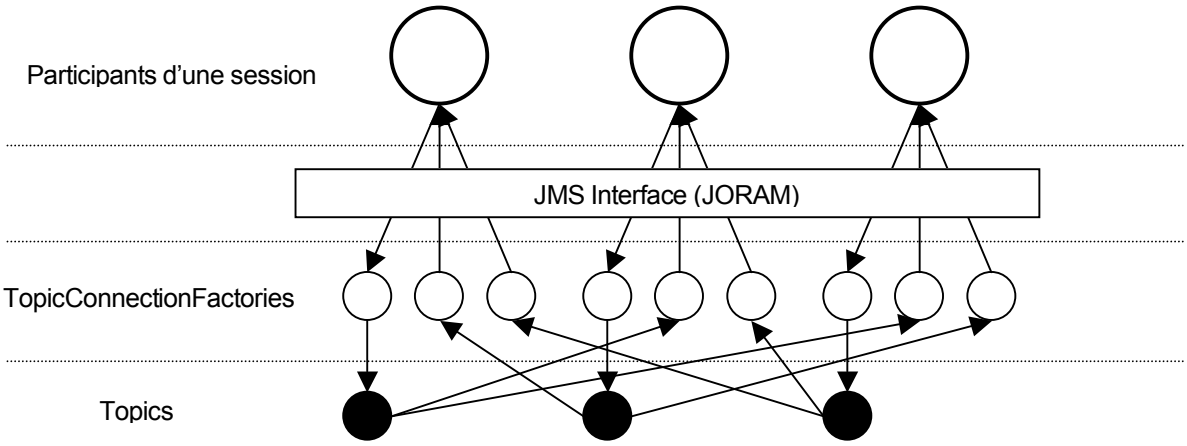
Il y a cinq clients JMS dans ce schéma :

- *Qreceiver* et *Qsender* ; *Qsender* dépose des messages dans une queue et *Qreceiver* vient les chercher quand bon lui semble,
- *Tpublisher* et les deux *Tsubscriber* ; *Tpublisher* publie des messages sur le *Topic* ; les deux *Tsubscriber* sont abonnés au *Topic* et implante la méthode *public void onMessage(javax.jms.Message msg)* de l'interface *MessageListener* qui est appelée à chaque arrivée d'un message sur le *Topic*.

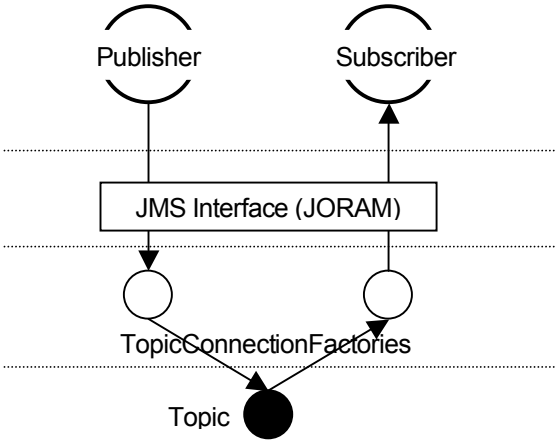
4.4.5. JORAM et TéléCabriJava

4.4.5.1. Queue ou Topic ?

L'anatomie d'une session TéléCabriJava s'accorde parfaitement au concept de *Topic*, en effet, chacun des participants doit envoyer ses messages textuels ou modifications de la figure géométrique (s'il possède le jeton) à chacun des autres participants. Chacun des participants est donc à la fois *publisher* et *subscriber* à chacun des *Topics* des autres participants. Voilà ce que cela donne avec une session de trois participants :



4.4.5.2. Garantie de délivrance des Topics



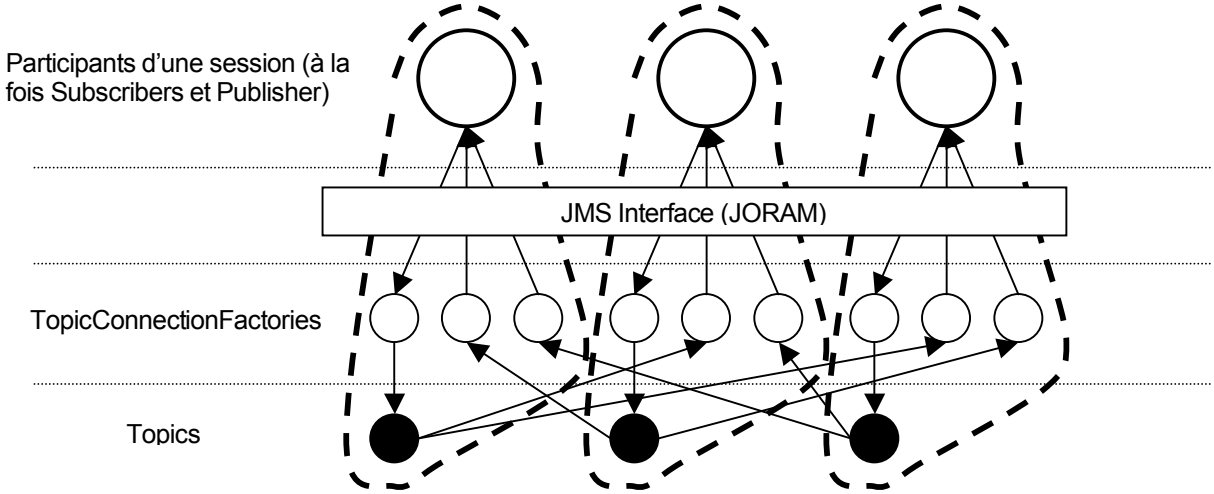
Joram garantit que tout message déposé par le *Publisher* sur le *Topic* sera délivré tôt ou tard au *Subscriber* lorsque le contexte réseau le permet. Les messages sont stockés de manière persistante sur la machine hébergeant le *Topic* (écrits en dur sur disque) de telle sorte que si la machine plante, ils ne soient pas perdus. En outre, grâce à son service d'atomicité (que l'on peut qualifier de transactionnel), Joram peut garantir malgré des problèmes réseaux entre la machine hébergeant le *Topic* et la machine hébergeant le *Subscriber* que le message sera correctement délivré à cette dernière.

4.4.5.3. Répartition des agents dans une session TéléCabriJava

Reprenons le schéma présenté précédemment afin de décider comment répartir sur les machines les différents *agents* ainsi que les *Subscribers* et *Publishers*. Premier point : il est impensable d'allouer des serveurs pour héberger les *Topics* et leurs *ConnectionFactories* et ce pour plusieurs raisons :

- un serveur coûte cher,
- un serveur demande de la maintenance,
- un serveur peut facilement être surchargé et fournir un service insuffisant pour une application exigeante comme notre TéléCabriJava.

La seule solution qui s'offre à nous est alors d'héberger le *Publisher*, son *Topic* et les *ConnectionFactories* sur la même machine. Il en découle la représentation ci-dessous où chacun des trois participants héberge sur sa machine son *Publisher*, le *Topic* associé et ses *ConnectionFactories*.



Cette solution présente de nombreux avantages :

- on se contente des machines des participants,
- l'envoi d'un message du *Publisher* vers le *Topic* est instantané,
- pas de perte de messages entre le Publisher et le Topic puisqu'ils sont sur la même machine et qu'il ne peut donc y avoir de problème réseau (il est à noté que JORAM n'assure pas qu'un message publié par le Publisher arrive bien à son Topic).

4.4.6. Avantages et inconvénients de JORAM pour TéléCabriJava

Comme nous l'avons vu précédemment le concept de *Topic* nous arrangerait bien pour les sessions TéléCabriJava que les participants peuvent joindre ou quitter. En effet, ils n'auraient qu'à souscrire ou se désabonner d'un *Topic*. D'autre part, avec notre configuration dans laquelle les machines des participants hébergeraient *Publisher*, *Topic* et *ConnectionFactories*, nous n'avons pas à nous soucier de stocker dans un tampon les messages puisque le transit entre le *Publisher* et le *Topic* est assuré et que JORAM garantit la délivrance du *Topic* au *Subscriber*.

Toutefois, il y a de sérieux inconvénients à l'utilisation de JORAM. Il s'est avéré que ce dernier génère des délais trop longs, il semble même que le Topic soit incapable de soutenir le débit de messages nécessaire pour un partage raisonnable d'une figure dans TéléCabriJava. Cela provient certainement de la gestion des processus légers du *Topic*. Il y en a deux :

- un pour la réception des messages publiés par le *Publisher,*
- un autre pour l'émission de ces messages vers les *Subscribers* qui ont souscrit au *Topic*.

Comme on peut s'en douter, le processus de réception a la priorité maximale et ce afin d'éviter que le Publisher ait à re-publier des messages qui aurait été perdus pour cause de non-traitement. Le second processus se retrouve donc avec si peu de ressources qu'il n'est pas capable de suivre le débit imposé par TéléCabriJava et les messages non envoyés aux Subscribers s'empilent donc au niveau du Topic qui ne fait quasiment plus que recevoir des messages du Publisher. Il s'ensuit une désynchronisation qui ne cesse de s'allonger au cours du temps entre la figure du possesseur du jeton et les figures des autres participants. Il est à considérer toutefois que cette observation a été faite sur un PC Pentium III 600Mhz qui semble avoir des problèmes de *scheduler* avec son système d'exploitation Windows NT4. Il est aussi possible que la version 2.0 de JORAM sortie début Mai se comporte mieux.

Enfin, chose un peu regrettable, JORAM ne gère pas les problèmes réseaux et se contente de remonter les exceptions de TCP en cas de rupture d'une connexion. Ainsi, JORAM, comme TCP, laisse l'utilisateur se débrouiller en ce qui concerne le bon fonctionnement et les performances du réseau qui sont primordiales pour une bonne synchronisation des figures dans TéléCabriJava.

5. Réalisation avec le protocole TCP

Après une étude et quelques tests des [protocoles] TCP et UDP, des RPC [RMI] et [CORBA] et du MOM [JORAM], il s'est avéré que TCP offrait le meilleur compromis performance, facilité de programmation. Dans la suite du rapport, je commence donc par exposer plus en détail pourquoi notre choix s'est porté sur TCP, puis j'exposerai comment les sessions TéléCabriJava sont gérées et comment le serveur de session sera censé fonctionner. Je terminerai sur la résolution des problèmes réseaux et notamment comment nous avons procédé pour pallier à des performances irrégulières.

5.1. Pourquoi TCP ?

5.1.1. UDP : protocole de trop bas niveau

Comme il a été vu dans la partie précédente, UDP n'est pas orienté connexion, il n'offre donc aucun service et ne garantit même pas l'arrivée des datagrammes émis à bon port. Il aurait probablement été très intéressant d'investir du temps dans la programmation d'une couche au-dessus d'UDP pour garantir l'arrivée et l'ordre d'arrivée des datagrammes émis afin de profiter de la multi-diffusion. Malheureusement, les adresses de groupes ne sont pas allouables dynamiquement et il est très difficile de dire sans multi-diffusion si la programmation d'une telle couche aurait été intéressante d'un point de vue performance par rapport à ce que propose le protocole TCP.

5.1.2. Les RPC : inadaptés pour TéléCabriJava

Au fur et à mesure qu'a progressé mon DEA, je me suis progressivement aperçu que nous aurions besoin d'une communication par messages de type asynchrone pour réaliser TéléCabriJava. Les RPC (Remote Procedure Call) qui ont fait l'objet de mes premières études se sont donc révélés totalement inadaptés.

D'autre part, en ce qui concerne RMI, si l'envie nous était venue d'utiliser ce middleware pour réaliser TéléCabriJava, nous nous serions rapidement aperçus de l'impossibilité de cette entreprise. En effet, comme je l'ai précisé précédemment, les RPC sont par définition synchrones. Un appel de méthode distante est bloquant jusqu'au moment où l'on récupère :

- soit le résultat de l'exécution de la méthode,
- soit l'exception qu'elle a levée.

Sachant qu'un RTT dure environ 300 ms en moyenne avec deux modems 56Kbits, on s'aperçoit que l'on peut au mieux caser 3 appels de procédures distantes par seconde, ce qui est très insuffisant. Sachant qu'au bas mot, le possesseur du jeton aura dans le meilleur des cas à ne transmettre que 15 informations par seconde, on s'aperçoit de l'impossibilité d'une réalisation avec RMI. Il faudrait allouer un processus léger par appel de méthode distante pour éviter d'être bloqué par le synchronisme de RMI, mais cela est tout à fait impensable pour les raisons suivantes :

- on n'a plus de garantie sur l'ordre d'arrivée des appels de méthodes,
- le nombre de processus est ingérable.

Rajoutons à cela que nous ne voulons pas intégrer à TéléCabriJava un serveur HTTP qui est absolument indispensable au bon fonctionnement des RPC et nous pouvons définitivement les rayer de la liste des outils possibles pour sa réalisation.

5.1.3. JORAM : trop lent

JORAM était la solution rêvée pour réaliser TéléCabriJava. Permettant la communication par messages de façon asynchrone et offrant un concept de *Topic* parfaitement adapté à la morphologie de nos sessions, JORAM s'est pourtant révélé extraordinairement lent sur notre machine de test. Nous avons pu observer un débit maximal de 3 messages par secondes quand 20 messages par secondes auraient du passer. Mon DEA touchant à sa fin et ne voulant pas rester sur un échec, j'ai abandonné JORAM aux performances trop décevantes pour la seule solution qui restait : TCP. Néanmoins, je dois préciser que la version 2.0 de JORAM est sortie début Mai 2001 et que ma machine de test avait visiblement des problèmes de Scheduler avec WIN NT4. TCP ne se serait peut-être pas imposé si mon DEA s'était déroulé en 2002.

5.1.4. TCP par forfait

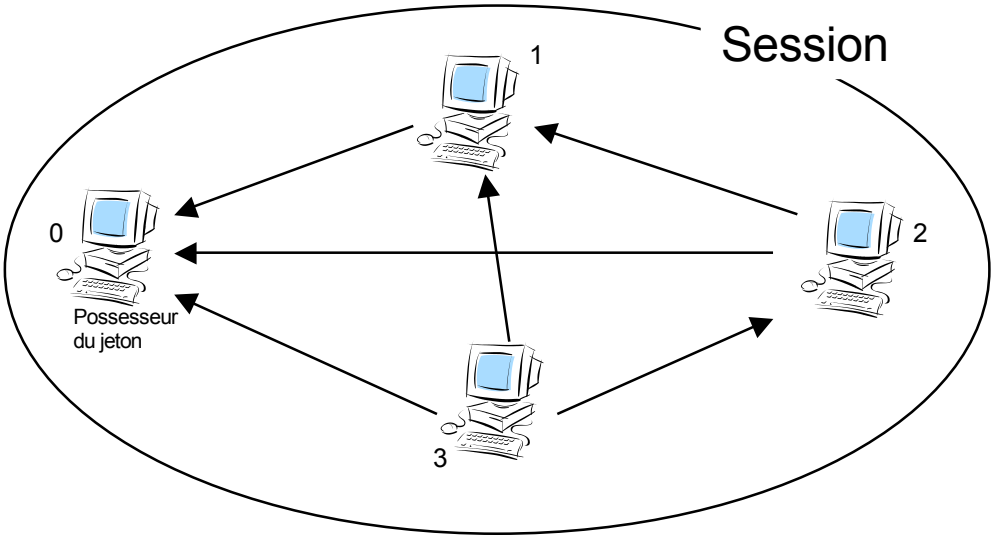
TCP par forfait !!! aucun des middlewares que j'ai étudiés ne répondait correctement à nos besoins pour la réalisation de TéléCabriJava. UDP : trop compliqué à mettre en œuvre. Les RPC : inadaptés, tout particulièrement RMI. JORAM : trop lent pour le moment, mais j'espère que cela ne durera pas pour ce MOM prometteur.

TCP s'est donc imposé par forfait. Néanmoins, TCP possède certaines qualités qu'il convient de remarquer :

- la garantie d'arrivée et de l'ordre d'arrivée des segments,
- une communication par messages et plus particulièrement par *String* ce qui répond parfaitement à nos besoins pour TéléCabriJava,
- des primitives d'utilisation (ouverture d'une connexion, fermeture d'une connexion, envoi d'un segment, réception d'un segment) excessivement simples et intuitives à utiliser,
- la possibilité lors d'une connexion entre deux modems 56Kbits d'envoyer une centaine de messages par secondes avec un RTT (Round Trip Time) moyen de l'ordre de 300ms.

5.2. Modélisation et gestion des sessions TéléCabriJava

La solution retenue pour la modélisation d'une session TéléCabriJava est l'interconnexion de tous les participants.



Les machines sont numérotées suivant leur ordre d'arrivée dans la session. Le sens des flèches indique la machine qui établit la connexion TCP. Les connexions TCP sont full-duplex : elles permettent d'émettre et recevoir simultanément. Pour rappel, tous les participants peuvent envoyer des messages textuels mais seul le possesseur du jeton peut modifier sa figure et donc envoyer des messages informant les autres participants d'une modification de la figure.

Lorsqu'un nouveau participant joint une session, on bloque la session et on lui transfère la figure dans son état actuel dans un souci de synchronisation des figures locales et distantes. De même, lorsque le possesseur du jeton ouvre une figure, il reste bloqué tant qu'il n'a pas fini de la transférer aux autres participants.

5.2.1. Package de gestion d'une session

Je me suis inspiré de JORAM et de son concept de *Topic* pour écrire un package gérant le plus indépendamment possible du reste de l'application :

- l'arrivée et le départ d'un participant,
- l'envoi d'un message à l'ensemble des participants,
- la réception d'un message,
- la gestion des problèmes réseaux majeurs (rupture de la connexion TCP),
- l'uniformisation des performances variables du réseau.

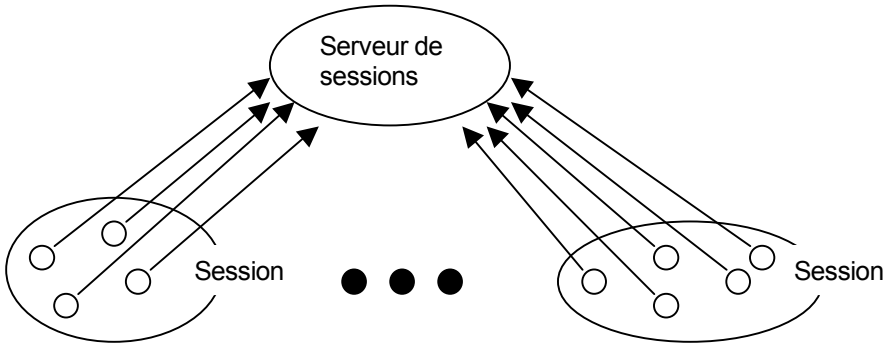
5.2.2. Limiter le nombre de messages à envoyer

Le possesseur du jeton a le droit de modifier la figure géométrique et il peut arriver que CabriJava génère une trentaine de messages par secondes concernant ces modifications. Le possesseur du jeton doit alors les envoyer et ce nombre peut paraître relativement élevé. J'ai donc écrit une classe permettant de le limiter pour les raisons suivantes :

- ne pas surcharger la connexion TCP capable de délivrer une centaine de segments par secondes avec deux modems 56Kbits. En effet, s'il y a cinq participants dans la session, le possesseur du jeton ne pourrait leur faire parvenir les modifications de la figure sans limitation,
- ne pas surcharger le CabriJava distant. On a observé qu'un CabriJava distant auquel on envoyait beaucoup de modifications avait tendance à saccader et à monopoliser toutes les ressources sans en laisser pour le package de gestion des communications réseaux.

5.3. Fonctionnement du serveur de sessions TéléCabriJava

Comme nous l'avons vu en détail dans le cahier des charges, le serveur permet d'administrer les sessions. Les utilisateurs de TéléCabriJava doivent se connecter au serveur par une connexion TCP pour créer ou joindre une session. Ils conservent cette connexion avec le serveur jusqu'au moment où ils quittent l'application, soit de manière conventionnelle, soit sur erreur de l'application ou rupture de la connexion TCP.



Le numéro du port d'une connexion TCP est codé sur 16 bits. Il est donc possible d'avoir 65536 connexions TCP simultanées sur une machine serveur (un peu moins en fait car quelques unes sont réservées par le système), ce qui fait déjà beaucoup de participants à gérer. A terme, il faudra faire un serveur de serveurs pour gérer les serveurs locaux qui seront répartis sur la planète.

La connexion entre un participant et le serveur ne sert qu'à certains moments précis :

- obtenir la liste des sessions,
- joindre une session,
- créer une session,
- quitter une session.

Les participants en quittant une session doivent le signaler au serveur afin que sa liste des sessions et des participants qui s'y trouvent soit maintenue à jour. Si un participant plante et provoque une rupture de sa connexion TCP, le serveur récupère l'exception qui est levée et met à jour ses données en y enlevant le participant. Ce qui est absolument capital, c'est que le serveur soit toujours dans un état cohérent.

5.4. Gestion des problèmes réseaux

5.4.1. Les problèmes réseaux mineurs

Rappelons que TéléCabriJava est architecturé autour du protocole TCP qui est orienté connexion. Ce protocole a été préféré à UDP pour des raisons citées précédemment à savoir la garantie de l'arrivée et de l'ordre d'arrivée des segments.

Les problèmes réseaux mineurs regroupent tous les désagréments qui peuvent être générés par le réseau excepté la rupture de la connexion TCP. On peut citer les fameuses pauses de 30 secondes que l'on rencontre régulièrement dans les jeux sur Internet ainsi que l'arrivée par paquet des segments. Ces désagréments sont occasionnés par les fournisseurs d'accès Internet qui pour la plupart offre un service de qualité vraiment discutable.

Ces problèmes réseaux mineurs n'ont en soit aucune répercussion sur le bon fonctionnement de l'application, c'est au niveau de l'utilisateur que vont se répercuter les mauvaises prestations du fournisseur. Imaginez le déplacement du curseur de la souris distante totalement saccadé voire complètement bloqué pendant plusieurs secondes avant de se déplacer à nouveau à une vitesse hallucinante afin de rattraper le cours des communications réseaux : ça ne serait pas très didactique !

Malheureusement, s'il y a une solution pour compenser l'arrivée par paquet de segments, il n'est pas possible de compenser les pauses occasionnées par les serveurs des fournisseurs d'accès. Je vous expose ci-après la mise à profit du buffer de TCP pour réguler l'arrivée des segments.

5.4.1.1. Mise à profit des buffers de TCP

TCP appartient à la couche « transport » du modèle OSI et dispose d'un buffer pour la réception des segments qui permet à la couche « application » de venir chercher un segment quand nécessaire. Il s'agit pour nous de tirer partie de cette réalité.

On peut réguler dans notre application le nombre de messages par secondes que le TéléCabriJava « maître » va envoyer aux « esclaves ». Dès lors, il est possible d'estimer le nombre de segments que l'on va recevoir sur les TéléCabriJava distants. Si j'utilise le terme « estimer », c'est parce qu'il n'est pas possible de fixer le nombre de messages par secondes émis par un CabriJava « maître » et que l'on ne peut pas quantifier les segments véhiculant des messages liés à la messagerie. Néanmoins, à partir de cette estimation, on va pouvoir déterminer une durée minimale avant d'aller chercher un nouveau segment dans le buffer de réception de TCP. Cette très légère retouche du code de l'application (il suffit d'insérer un « wait(int millisecondes) » avant d'aller lire dans le buffer du socket) permet d'améliorer sensiblement la fluidité de notre application.

5.4.1.2. Intégration d'un estimateur du RTT pour chacun des participants

Comme je l'ai précisé ci-dessus, nous ne pouvons rien contre les pauses réseaux mais il faut tout de même en informer les utilisateurs. De même, les messages textuels des participants doivent être synchronisés par rapport à l'état de la figure et il est impossible de le garantir, de sorte qu'il faudrait les dater et examiner s'ils ne sont pas hors contexte avant de les afficher. Tout ça est bien compliqué et j'ai donc préféré proposer une application vraiment élémentaire aux utilisateurs en comptant sur eux pour la « piloter » au mieux.

Ainsi, l'interface graphique intègre en face de chacun des participants, une estimation du RTT(Round Trip Time) avec ce dernier. Cela permet de savoir s'il y a des participants qui ont des problèmes de synchronisation avec le possesseur du jeton. Je laisse le soin aux participants d'une session de prendre en considération ces indications et de prendre les décisions qui s'imposent : stopper net toute modification de la figure par exemple, ou demander au participant « défectueux » de quitter la session.

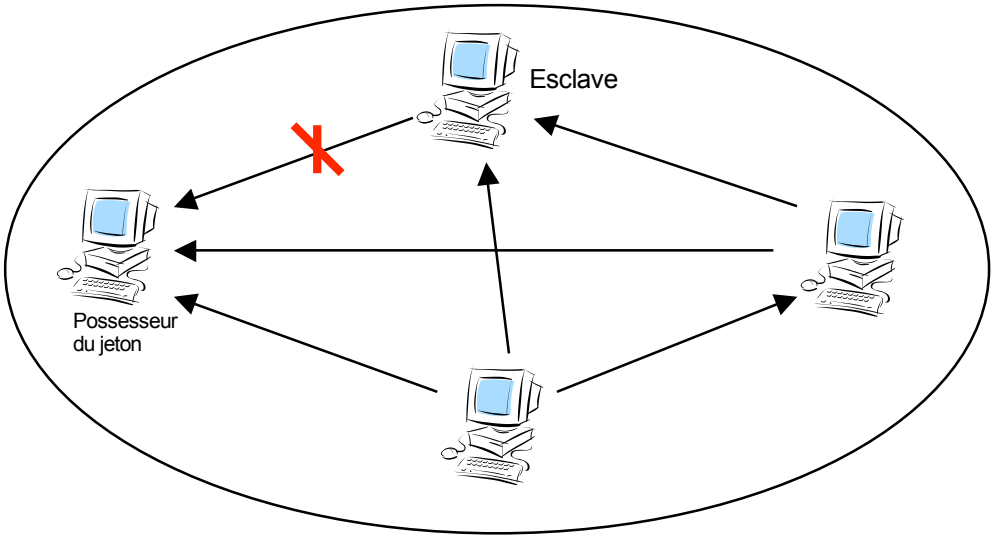
Je suis persuadé qu'il est préférable de laisser l'intelligence humaine résoudre ces problèmes que de tenter de les résoudre dans l'application elle-même. D'une part, l'application serait plus difficile à programmer, plus lourde de fonctionnement et les utilisateurs ne comprendraient peut-être pas ce qui se passe.

5.4.2. Les problèmes réseaux majeurs

A problème réseaux majeurs, j'assimile tous les problèmes qui occasionnent une rupture de la connexion TCP : de l'utilisateur qui arrache le câble réseau, à un plantage de sa machine ou à un arrêt brutal de l'application. Si dans certains cas, l'utilisateur désire réellement mettre fin à une session TéléCabriJava, on ne peut rien faire de plus pour lui, néanmoins, il ne faut pas que la session TéléCabriJava ainsi que le serveur de sessions TéléCabriJava se bloquent. Etudions donc les solutions que nous avons mises en place pour résoudre ces problèmes.

5.4.2.1. Rupture d'une connexion avec le possesseur du jeton

Prenons le cas d'une session TéléCabriJava avec un maître et trois élèves, le maître dispose du jeton pour pouvoir modifier la figure. Le sens des flèches indique lequel des deux participants a établi la connexion.



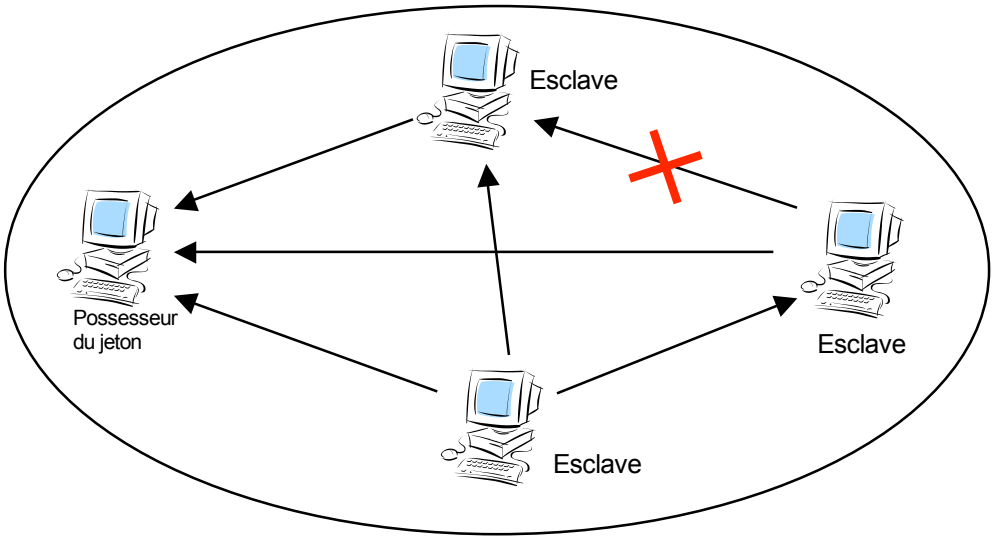
Pour de multiples raisons, si ce cas de configuration se présente, on va totalement abandonner l'esclave de la session. Toutes les connexions TCP avec ce dernier seront fermées et les applications TéléCabriJava distantes informeront les utilisateurs de son exclusion.

Pourquoi une telle mesure ? D'une part, le fait qu'un esclave soit exclu inopinément d'une session n'est pas grave en soit, d'autre part, si l'on tentait la re-connexion du participant à la session, il se poserait les problèmes suivants :

- le possesseur du jeton et tous les autres participants devraient attendre la re-connexion de ce dernier,
- il est possible que ce participant ait des problèmes réseaux permanents, à savoir un câble défectueux, un fournisseur d'accès problématique, … et qu'il soit préférable de laisser les autres participants travailler en paix.

5.4.2.2. Rupture d'une connexion entre deux esclaves

Ce cas de figure n'est pas grave car les connexions inter-esclaves ne servent qu'à la messagerie. On pourra prendre tout notre temps pour rétablir la connexion tout en informant les deux esclaves concernés qu'ils ne peuvent plus émettre et recevoir de messages textuels entre eux.



Toutefois, s'il survient que le possesseur du jeton le donne à l'un des deux esclaves « défectueux » et que la connexion entre eux ne soit toujours pas rétablie, on revient dans le premier cas examiné et l'on éjecte de la session l'esclave qui n'est pas devenu possesseur du jeton.

6. Conclusion

A l'heure ou le télé-enseignement prend de plus en plus d'ampleur, j'ai participé au cours de mon DEA à la réalisation de TéléCabriJava, un outil d'édition coopérative de figures géométriques basé sur l'application Cabri Géomètre II. J'ai étudié les différents outils qui s'offrent à nous pour la réalisation d'une telle application : TCP, UDP, les RPC et les MOM. Après quelques expériences sur ces protocoles et middlewares, TCP s'est imposé pour des raisons de performances par rapport à RMI et JORAM et pour des raisons de simplicité de programmation par rapport à UDP.

Au moment ou j'écris cette conclusion, on peut considérer que tout ce qui concerne une session TéléCabriJava fonctionne correctement et globalement il ne reste plus que le serveur de session à implanter. Les premiers tests de TéléCabriJava ont donné des résultats étonnamment bons sur des accès populaires à Internet (modems analogiques 56Kbits) et cela laisse augurer un TéléCabriJava distribuable d'ici quelques mois. Néanmoins, si l'aspect performance de TéléCabriJava est étonnant, les problèmes réseaux qui peuvent survenir de temps en temps ont peut-être été traités un peu trop radicalement. Seuls des tests plus approfondis avec plusieurs machines connectées à l'Internet avec des modems 56Kbits pourront nous permettre d'améliorer le fonctionnement de TéléCabriJava.

7. Bibliographie

[CabriJava]
- Le projet CabriJava – Gilles Kuntz - http://www-cabri.imag.fr/cabrijava/

[TéléCabriJava]
- Le projet TéléCabriJava – Rapport de première année de Magistère – Adrien Mondot

[swing]
- Au cœur de Java 2 – Volume 1 – Notions fondamentales – Campus Press – Sun Microsystems
- Tutorial Java – http://java.sun.com/docs/books/tutorial/uiswing/index.html

[internationalisation]
- Tutorial Java – http://java.sun.com/docs/books/tutorial/i18n/index.html

[application répartie]
- Construction d'applications réparties – Michel Riveill
- Modes de structuration d'applications réparties – Roland Balter

[protocole] : les protocoles réseaux UDP et TCP
- Tutorial Java – http://java.sun.com/docs/books/tutorial/networking/index.html
- TCP Illustrated – Volume 1 – The protocols – Richard Stevens
- Aspects avancés des réseaux – Andrzej Duda

[CORBA]
- Corba – Des concepts à la pratique – InterEditions
- Systèmes répartis et standards Corba – Virginie Amar – http://cic.cstb.fr/ilc/people/va/corba2/index.html

[RMI]
- Java Remote Method Invocation – Gopalan Suresh Raj – http://www.execpc.com/~gopalan/java/java_rmi.html
- Au cœur de Java 2 – Volume 2 – Fonctions avancées – Campus Press – Sun Microsystems
- Tutorial Java – http://java.sun.com/docs/books/tutorial/rmi/index.html

[Javapod]
- Javapod : une plate-forme à composants adaptable et extensible – Eric Bruneton ; Michel Riveill –http://www.inria.fr/rrrt/rr-3850.html

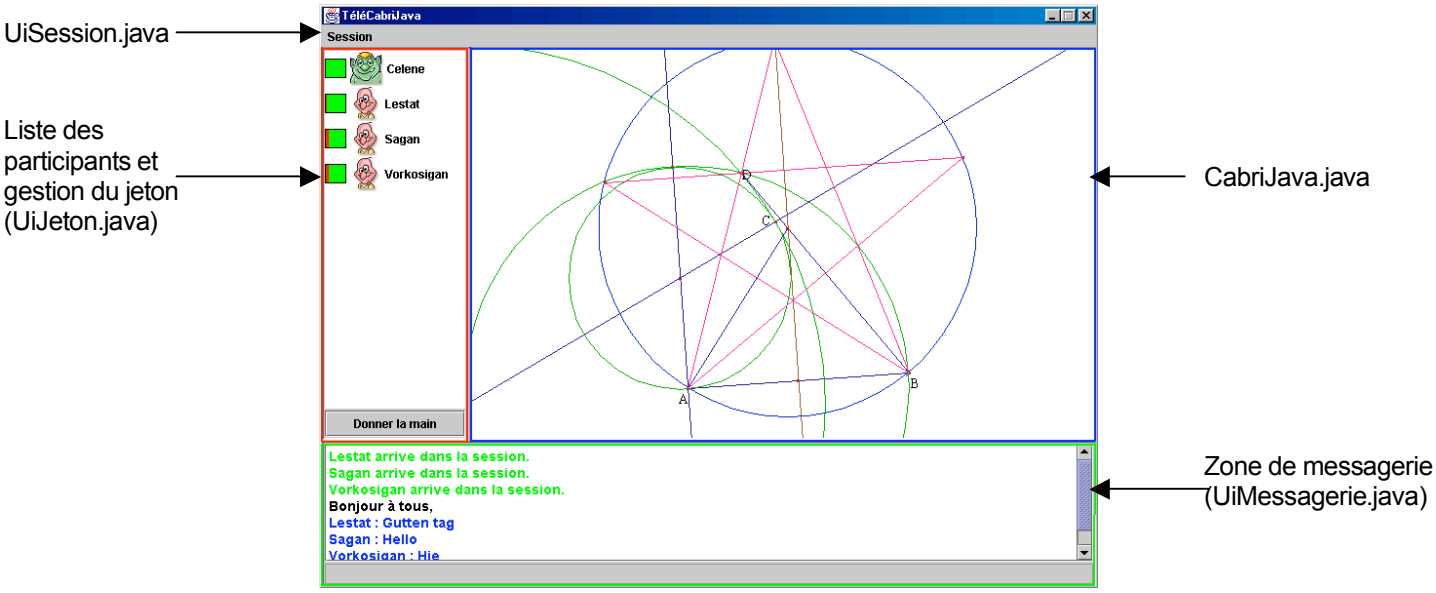
[JORAM]
- Tutorial Joram – http://www.objectweb.org/joram/joramTutorial.html

8. Annexes

Cette partie est relativement riche, j'y présente principalement tous les détails de réalisation de TéléCabriJava que j'ai omis dans les parties précédentes. Je commence par décrire l'interface graphique écrite avec le package [swing] et les classes *LayoutManager* et *ListCellRenderer* que j'ai ré-implantées. Je poursuis sur les fonctionnalités de mon package de gestion de session TéléCabriJava basé sur le protocole TCP. Pour ceux ou celui qui reprendrait mon code afin de l'améliorer et de le compléter, je fournis ensuite la modélisation UML de l'existant de TéléCabriJava. Je donne ensuite la liste de ce que je n'ai pas implanté et je termine sur une exploration en profondeur du protocole TCP que j'ai utilisé pour réaliser TéléCabriJava.

8.1. Détails concernant la réalisation de l'interface graphique

8.1.1. Description de l'interface graphique



La liste des participants répertorie la liste des utilisateurs présents dans la session. Le participant qui possède le jeton est mis en valeur par une icône particulière. L'icône est précédée par une représentation de la durée des ping-pongs avec le participant correspondant.

Dans la fenêtre de messagerie, on peut lire les messages distants, locaux et systèmes. Ils ont chacun une couleur particulière :

- noir ⇔ message tapé par l'utilisateur,
- bleu ⇔ message distant,
- vert ⇔ message système cool,
- rouge ⇔ message système pas cool.

L'utilisateur peut taper les messages au clavier à tout moment. Les caractères saisis sont affichés dans le JTextField non éditable en bas de la fenêtre. L'utilisateur valide un message en appuyant sur la touche ENTREE. Bien sûr, ce procédé interdit l'usage de raccourcis pour les menus (c'est sans importance à mon avis, car une messagerie conviviale est bien plus appréciable que des raccourcis).

Le panneau CabriJava permet quant à lui de modifier la figure si l'on possède le jeton ou de suivre les modifications de ce dernier.

8.1.2. Internationalisation

L'[internationalisation] est un procédé qui permet de concevoir une application qui pourra être disponible dans différentes langues sans avoir à modifier son code.

Java permet l'internationalisation de ses applications et j'en ai profité pour en faire bénéficier TéléCabriJava. Mais comment fonctionne l'internationalisation ?

Il faut tout d'abord créer des fichiers de propriétés. Ces fichiers au format texte ont l'extension ***.properties***. Il en faut un pour la langue par défaut, que l'on nommera MessagesBundle.properties. En voici un extrait tiré de TéléCabriJava :

```
DemanderJeton = Demander la main
iconeMaitre = maitre.gif
```

Maintenant que ces messages sont dans le fichier de propriété par défaut, on peut les traduire dans diverses langues. Chaque langue doit avoir son propre fichier de propriété dont le nom doit respecter la syntaxe suivante : ***MessagesBundle_CodeLangage_CodePays***. Java qui peut récupérer les codes du langage et du pays du système d'exploitation peut ensuite retrouver le fichier de propriétés adéquat à la langue du système. Le fichier de propriétés pour la langue anglaise se nommera ainsi : MessagesBundle_en_US.properties. Voici l'extrait de ce fichier correspondant au fichier de propriétés par défaut vu précédemment :

```
DemanderJeton = Token request
iconeMaitre = maitre.gif
```

Une fois ces fichiers de propriétés écrits, pour les utiliser dans le code Java, il faut commencer par instancier un ***ResourceBundle*** :

```
ResourceBundle messages=ResourceBundle.getBundle("MessagesBundle",Locale.getDefault());
```

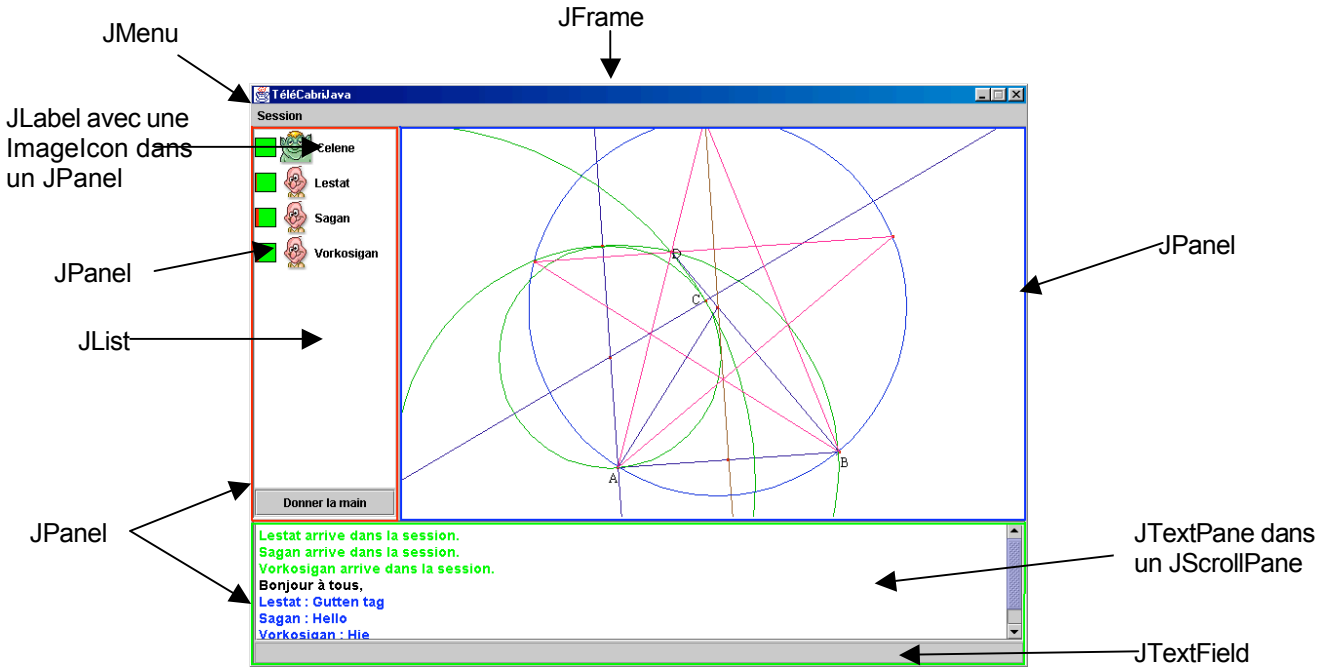
Les deux paramètres de la méthode getBundle permettent d'identifier quel fichier de propriétés doit être utilisé. Une fois qu'on a récupéré le ***ResourceBundle***, on peut récupérer la traduction des messages identifiés par leur clé à l'aide la méthode getString().

```
private String t_DemanderJeton, t_iconeMaitre;
t_DemanderJeton=messages.getString("DemanderJeton");
t_iconeMaitre=messages.getString("iconeMaitre");
```

8.1.3. Utilisation du package swing de Java

8.1.3.1. Description des composants swing de l'interface graphique

Voici l'ensemble des composants [swing] de mon interface graphique. On peut remarquer que les intitulés des composants commencent par J ce qui les différencie de leurs homologues du package AWT.

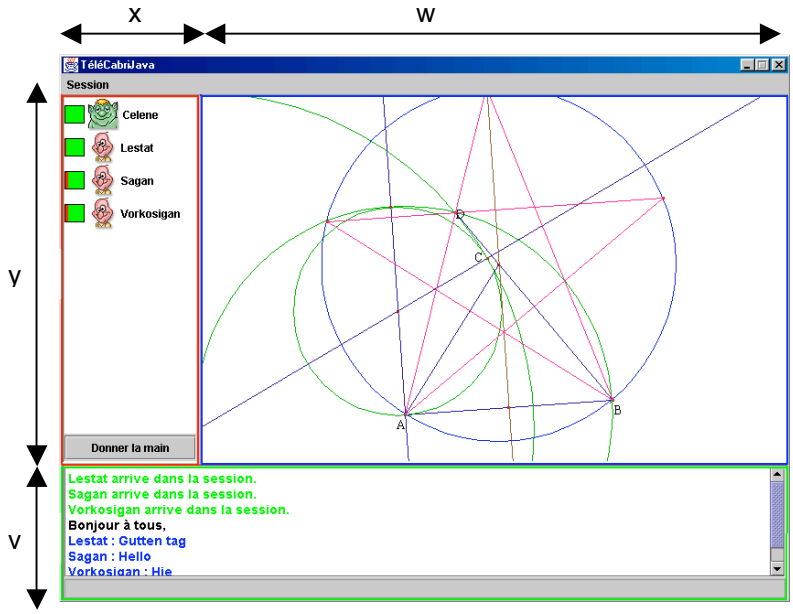


8.1.3.2. Implantation d'un Layout

Les *Layout* fournis par JAVA ne permettaient pas une bonne répartition des *Containers* de l'interface graphique. J'ai donc réalisé mon propre *Layout* en implantant l'interface *LayoutManager*. Voilà comment je fixe les bornes des *Containers* dans la méthode `public void layoutContainer(Container parent)` :

Soit **l** la largeur de la *JFrame* et **h** la hauteur :

- x=minimum(150,l/6)
- y=minimum(150,l/5)
- w=l-x
- v=h-y



Comme on peut le constater, avec mon *Layout*, les *Containers* seront correctement dimensionnés quelle que soit la dimension de la *JFrame*.

Toutefois cette faculté de re-dimensionner la fenêtre ne sera pas offerte à l'utilisateur car on fixe la taille de la JFrame à 800*600 afin que tous les participants aient le même aperçu de la figure. En outre, le 800*600 est une résolution supportée par tous les moniteurs actuels.

8.1.3.3. Implantation d'un ListCellRenderer *pour la liste des participants*

Une fois encore, les classes offertes par JAVA ne convenaient pas pour la réalisation de TéléCabriJava. J'ai été obligé d'implanter l'interface *ListCellRenderer*. Les cellules de la *JList* sont en fait des *JPanels*. Je définis pour chacune d'entres-elles un gestionnaire *BoxLayout* dans lequel j'ajoute un *JPanel* correspondant à la représentation de la durée des ping-pongs puis un *Jlabel* contenant une *ImageIcon*.

8.2. Communication entre CabriJava et TéléCabriJava

Lors de l'instanciation de la classe *CabriJava* (la classe principale du clone de Cabri-Géomètre II en java), on lui passe l'instance de la classe *TeleCabri* avec laquelle elle va communiquer. Cette classe *TeleCabri* permet de faire le lien entre *CabriJava* et l'application TéléCabriJava.

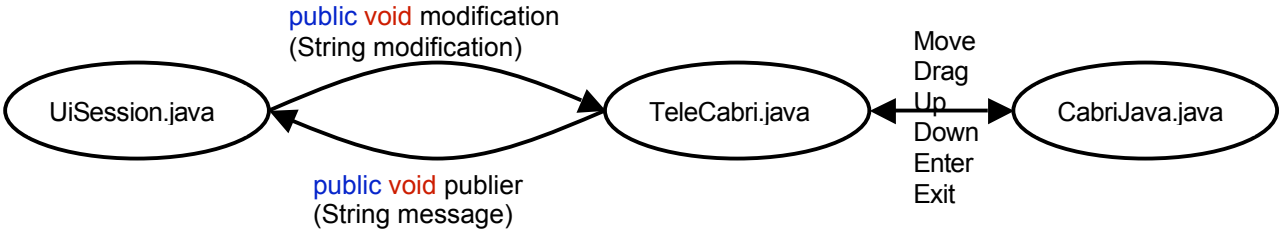
La classe *TeleCabri* peut modifier la figure de *CabriJava* en appelant les méthodes suivantes sur l'objet *CabriJava* :

- public void move(int x, int y, boolean ctrl),
- public void drag(int x, int y, boolean maj),
- public void up(int x, int y, int nbclic),
- public void down(int x, int y, int nbclic, boolean alt, boolean ctrl),
- public void enter(int x, int y),
- public void exit(int x, int y).

Et inversement, *CabriJava* peut informer *TeleCabri* des modifications de la figure locale en appelant les mêmes méthodes sur l'objet *TeleCabri*.

La classe *TeleCabri* propose la méthode « public void modification(String modification) » afin de permettre aux classes de TéléCabriJava de faire parvenir les modifications de la figure distante à l'instance de *CabriJava*.

Enfin, la classe qui instancie la classe *TeleCabri* doit implanter l'interface *Publieur* et passer son instance au constructeur de l'objet *TeleCabri*. Le fait d'implanter cette interface permet à l'objet *TeleCabri* d'appeler la méthode « public void publier(String message)». Cette méthode permet à *TeleCabri* d'informer TéléCabriJava d'une modification de la figure locale.



8.3. Détails sur la communication par messages entre les participants

Les instances de TéléCabriJava d'une session s'échangent des messages correspondant aux différentes actions que peuvent mener les participants. J'ai choisi d'identifier ce à quoi correspond un message par un nombre à deux chiffres qui précède son contenu. Voici la liste de ces identifiants :

- 00 ⇔ un utilisateur a joint la session,
- 01 ⇔ message issu de la messagerie,
- 02 ⇔ don du jeton à un participant,
- 03 ⇔ demande d'un participant du jeton,
- 04 ⇔ un participant quitte la session,
- 05 ⇔ message de CabriJava,
- 06 ⇔ ping,
- 07 ⇔ pong.

Tous les messages que peuvent s'échanger les participants sont destinés à tous les participants de la session excepté le message correspondant à un pong. Détaillons maintenant ces messages :

- « 00login » ⇔ login : le login du participant qui joint la session,
- « 01login » : message ⇔ message : le message issu de la messagerie du participant login,
- « 02login » ⇔ login : le login du participant auquel on donne le jeton,
- « 03login » ⇔ login : le login du demandeur du jeton,
- « 04login » ⇔ login : le login, on quitte la session,
- « 05move int int boolean » ⇔ move du CabriJava du possesseur du jeton,
- « 05drag int int boolean » ⇔ drag du CabriJava du possesseur du jeton,
- « 05up int int int » ⇔ up du CabriJava du possesseur du jeton,
- « 05down int int int boolean boolean » ⇔ down du CabriJava du possesseur du jeton,
- « 05enter int int » ⇔ enter du CabriJava du possesseur du jeton,
- « 05exit int int » ⇔ exit du CabriJava du possesseur du jeton,
- « 06 » ⇔ ping,
- « 07login » ⇔ envoi d'un pong par le participant login.

8.4. Détails concernant le package réseau TCP

8.4.1. Le package GestionCom

J'ai regroupé dans ce package les classes :

- GestionCom.java ⇔ gestion d'une session,
- TheadCom.java ⇔ gestion d'une connexion TCP avec un participant d'une session,

Et les interfaces :

- PbReseauxMajeurs,
- Souscripteur.

Pour obtenir des détails sur le fonctionnement de ce package et de son contenu, le lecteur pourra générer les *javadocs* associés.

8.4.2. Les interfaces du package

Pour permettre une meilleure compréhension de la collaboration entre les classes du package et les classes utilisant ce package, j'ai écrit deux interfaces :

- *PbReseauxMajeurs* ⇔ toute classe qui implante cette interface gère d'une certaine façon les problèmes réseaux et propose aux instances de *ThreadCom* des méthodes à appeler pour les résoudre,
- *Souscripteur* ⇔ toute classe qui implante cette interface peut récupérer les messages reçus par les instances de *ThreadCom* en leur fournissant la méthode « public void recevoir(String message, ThreadCom threadCom) ».

8.4.3. Gestion d'une session

La classe *GestionCom* gère les sessions TéléCabriJava. Elle étend la classe *Thread* et constitue donc un processus léger. Elle se charge :

- d'établir les connexions avec les participants présents dans une session,
- d'attendre les connexions de nouveaux participants arrivant dans la session,
- d'envoyer un message à tous les participants de la session.

La classe *ThreadCom* permet de gérer une connexion avec l'un des participants d'une session. Elle étend aussi la classe *Thread*. Cette classe permet entre autre :

- d'envoyer un message au participant associé à la connexion,
- de recevoir un message de ce même participant.

Il faut noter que j'ai configuré les *sockets* des connexions TCP avec l'option *TcpNoDelay* à *true* afin de désactiver l'algorithme de Nagle et forcer l'envoi immédiat des segments.

La classe *GestionCom* gère ainsi une liste d'instances de *ThreadCom* de manière la plus transparente possible pour les classes utilisant le package.

8.4.4. Limitation du nombre de messages à envoyer

Il a été nécessaire de limiter le nombre de messages que le *CabriJava* du possesseur du jeton envoie par seconde aux autres participants pour deux raisons :

- éviter de surcharger la connexion qui le plus souvent sera de 56Kbits,
- éviter de surcharger le *CabriJava* distant avec plus de modifications qu'il n'est capable d'en afficher.

Voici la liste des messages correspondant à des modifications de la figure que peut envoyer un TéléCabriJava possédant le jeton à un autre :

- move int int boolean,
- drag int int boolean,
- up int int int,
- down int int int boolean boolean,
- enter int int,
- exit int int.

On peut filtrer ces messages au besoin pour réguler le débit, toutefois, il faut respecter la règle suivante : toujours envoyer la dernière action d'une série de la même action. Par exemple, on enverra obligatoirement le dernier *move* d'une série de *moves*.

8.4.5. Résistance aux pannes réseaux

Pour éviter suite à des ruptures de connexions TCP que l'application soit dans un état incohérent, j'ai du modifier l'application initiale à plusieurs niveaux et notamment :

- au moment de joindre une session : tous les participants déjà présents dans une session ne sont pas forcément joignables, si c'est le cas, on ne peut pas joindre la session,
- dans la session : une connexion TCP avec l'un des participants peut-être rompue et suivant les cas, soit le participant est exclu de la session, soit on retente la connexion avec ce dernier (voir la partie « Gestion des problèmes réseaux » du rapport).

8.4.6. Aspects performances

Veuillez consulter la partie $5.4.1.1. intitulée « Mise à profit des buffers de TCP » dans laquelle j'explique comment il a été possible d'améliorer d'une façon notable la fluidité des modifications de la figure.

8.4.7. Test ping-pong

En guise d'introduction, veuillez consulter la partie $5.4.1.2. intitulée « Intégration d'un estimateur du RTT pour chacun des participants ».

On peut trouver l'estimation du RTT (Round Trip Time) pour chacun des participants devant son icône dans la liste des participants. Pour permettre un rafraîchissement permanent de cet indice, j'ai réalisé la classe *BenchPingPong* qui étend la classe *Thread*. Ce processus léger envoie un *ping* et attend la réception du *pong* correspondant avant de représenter la durée du ping-pong. Une fois le *pong* reçu, il récidive en envoyant à nouveau un *ping* et ainsi de suite. Pendant l'attente du *pong*, le processus représente la durée déjà écoulée depuis l'envoi du *ping* à la condition toutefois que cette durée dépasse la durée du dernier ping-pong.

8.4.8. Premiers tests concluants de TéléCabriJava

La version actuelle de TéléCabriJava s'est avérée stable lors des tests. Voici les configurations utilisées pour les tests:

- deux PC sous Windows 98 connectés par un LAN 100Mbits,
- un PC sous Windows NT 4 et un MAC sous Mac OS X connectés par un LAN 100Mbits ou une connexion modem 56Kbits.

Nous avons utilisé le fournisseur d'accès *Free* pour nous connecter à Internet par modem. La qualité d'une telle connexion est plutôt mauvaise avec un ping-pong moyen de 300ms et un débit maximal de 100 segments de petite taille par seconde.

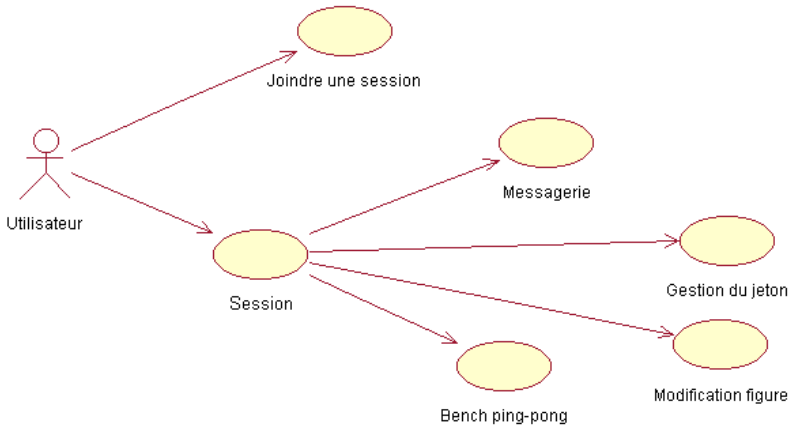
Avec la connexion modem 56Kbits, la fluidité de l'animation d'une figure distante s'est révélé tout à fait concluante. On peut globalement dire qu'il n'y a pas trop de saccades grâce à la régulation opérée à l'aide du buffer de réception des segments de TCP.

Enfin, que ça soit sur LAN ou avec le modem 56Kbits, aucune rupture de connexion TCP n'est survenue pour mettre à l'épreuve la résistance de TéléCabriJava à de tels problèmes réseaux.

8.5. Modélisation UML de l'existant de TéléCabriJava

8.5.1. Cas d'utilisation

J'ai recensé un acteur : l'utilisateur ainsi que six cas d'utilisation de l'application.

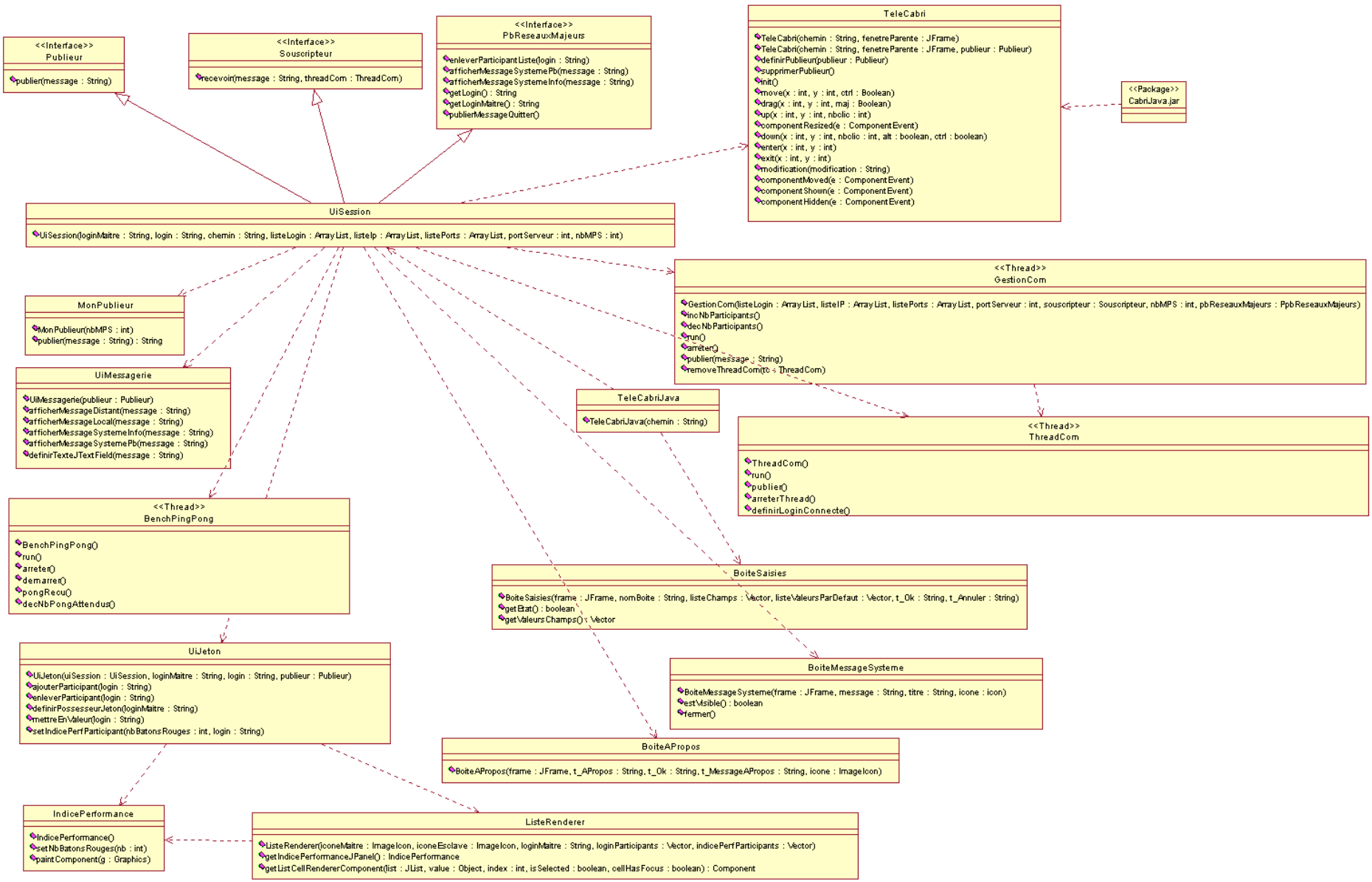


8.5.2. Diagramme de classes

Notations :

⇔ utilise ⇔ implante

La flèche « utilise » indique qu'une classe en instancie une autre. La flèche « implante » indique qu'une classe réalise une interface.

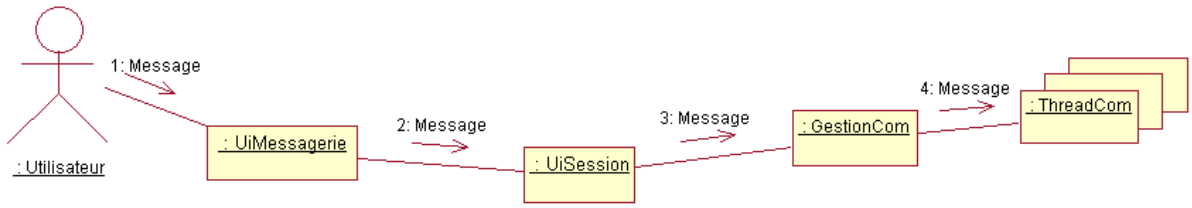

<<Interface>>
Publieur
publier(message : String)
<<Interface>>
Souscripteur
recevoir(message : String, threadCom : ThreadCom)
<<Interface>>
PbReseauxMajeurs
enleverParticipantListe(login : String)
afficherMessageSystemePb(message : String)
afficherMessageSystemeInfo(message : String)
getLogin() : String
getLoginMaitre() : String
publierMessageQuitter()
TeleCabri
TeleCabri(chemin : String, fenetreParente : JFrame)
TeleCabri(chemin : String, fenetreParente : JFrame, publieur : Publieur)
definirPublieur(publieur : Publieur)
supprimerPublieur()
init()
move(x : int, y : int, ctrl : Boolean)
drag(x : int, y : int, maj : Boolean)
up(x : int, y : int, nbclic : int)
componentResized(e : ComponentEvent)
down(x : int, y : int, nbclic : int, alt : boolean, ctrl : boolean)
enter(x : int, y : int)
exit(x : int, y : int)
modification(modification : String)
componentMoved(e : ComponentEvent)
componentShown(e : ComponentEvent)
componentHidden(e : ComponentEvent)
<<Package>>
CabriJava.jar
UiSession
UiSession(loginMaitre : String, login : String, chemin : String, listeLogin : ArrayList, listeIp : ArrayList, listePorts : ArrayList, portServeur : int, nbMPS : int)
MonPublieur
MonPublieur(nbMPS : int)
publier(message : String) : String
<<Thread>>
GestionCom
GestionCom(listeLogin : ArrayList, listeIP : ArrayList, listePorts : ArrayList, portServeur : int, souscripteur : Souscripteur, nbMPS : int, pbReseauxMajeurs : PpbReseauxMajeurs)
incNbParticipants()
decNbParticipants()
run()
arreter()
publier(message : String)
removeThreadCom(tc : ThreadCom)
UiMessagerie
UiMessagerie(publieur : Publieur)
afficherMessageDistant(message : String)
afficherMessageLocal(message : String)
afficherMessageSystemeInfo(message : String)
afficherMessageSystemePb(message : String)
definirTexteJTextField(message : String)
TeleCabriJava
TeleCabriJava(chemin : String)
<<Thread>>
ThreadCom
ThreadCom()
run()
publier()
arreterThread()
definirLoginConnecte()
<<Thread>>
BenchPingPong
BenchPingPong()
run()
arreter()
demarrer()
pongRecu()
decNbPongAttendus()
BoiteSaisies
BoiteSaisies(frame : JFrame, nomBoite : String, listeChamps : Vector, listeValeursParDefaut : Vector, t_Ok : String, t_Annuler : String)
getEtat() : boolean
getValeursChamps() : Vector
UiJeton
UiJeton(uiSession : UiSession, loginMaitre : String, login : String, publieur : Publieur)
ajouterParticipant(login : String)
enleverParticipant(login : String)
definirPossesseurJeton(loginMaitre : String)
mettreEnValeur(login : String)
setIndicePerfParticipant(nbBatonsRouges : int, login : String)
BoiteMessageSysteme
BoiteMessageSysteme(frame : JFrame, message : String, titre : String, icone : icon)
estVisible() : boolean
fermer()
BoiteAPropos
BoiteAPropos(frame : JFrame, t_APropos : String, t_Ok : String, t_MessageAPropos : String, icone : ImageIcon)
IndicePerformance
IndicePerformance()
setNbBatonsRouges(nb : int)
paintComponent(g : Graphics)
ListeRenderer
ListeRenderer(iconeMaitre : ImageIcon, iconeEsclave : ImageIcon, loginMaitre : String, loginParticipants : Vector, indicePerfParticipants : Vector)
getIndicePerformanceJPanel() : IndicePerformance
getListCellRendererComponent(list : JList, value : Object, index : int, isSelected : boolean, cellHasFocus : boolean) : Component

8.5.3. Diagrammes de collaborations

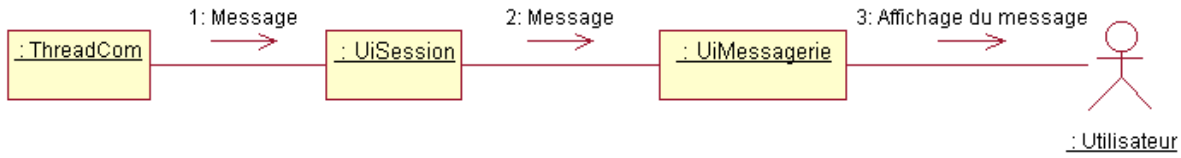
Je donne, ci-dessous, pour chacun des cas d'utilisation de TéléCabriJava, les diagrammes de collaboration entre les classes de l'application.

8.5.3.1. Messagerie

Envoi d'un message

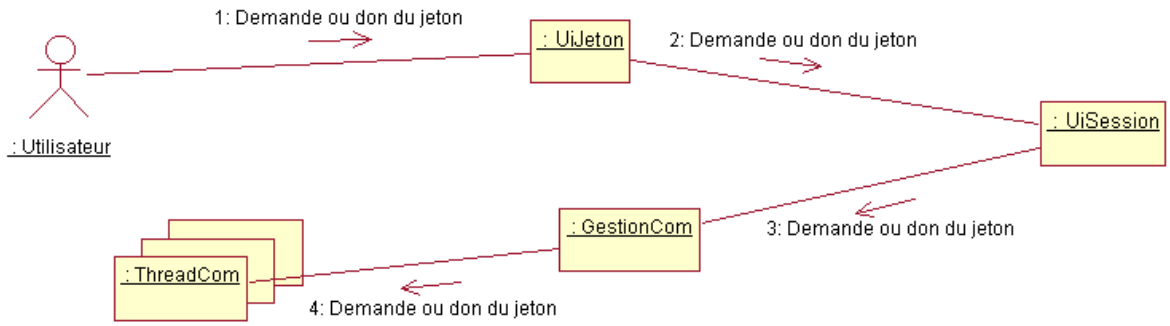


Réception d'un message

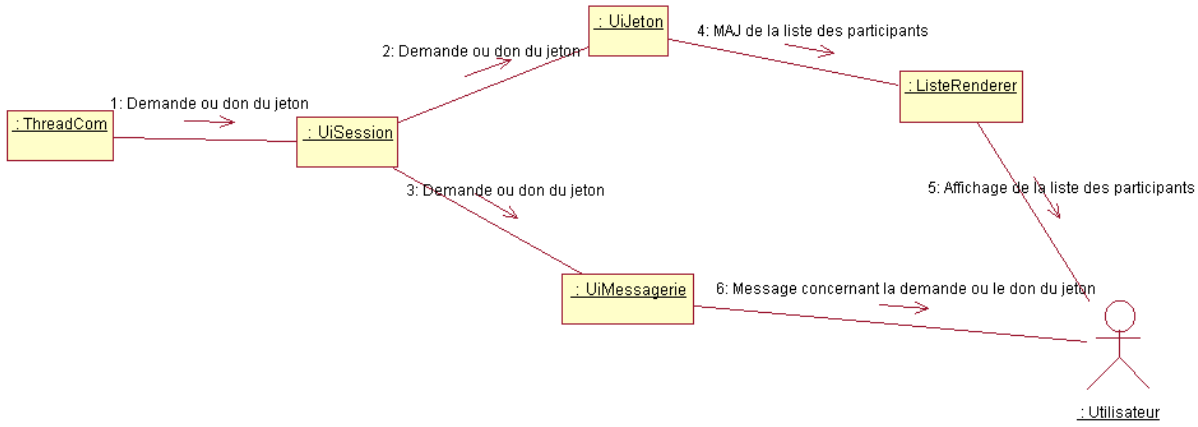


8.5.3.2. Gestion du jeton

Demande ou don du jeton

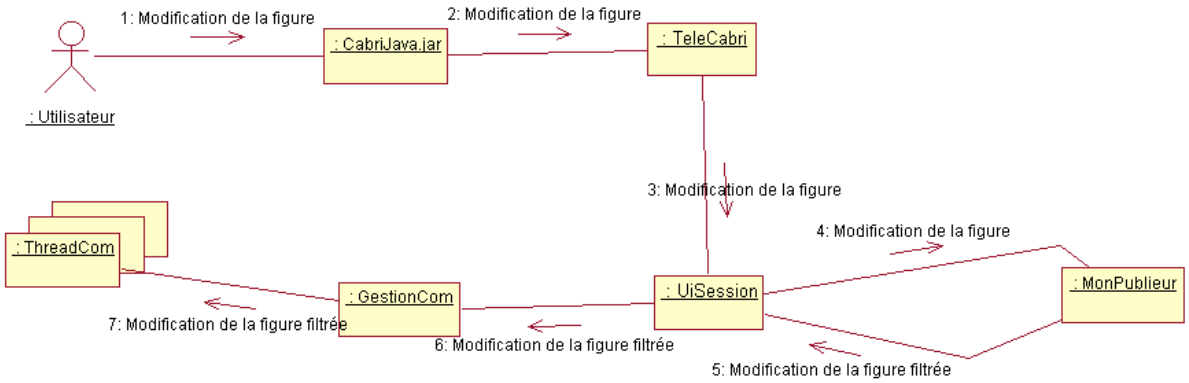


Réception de la demande ou du don du jeton

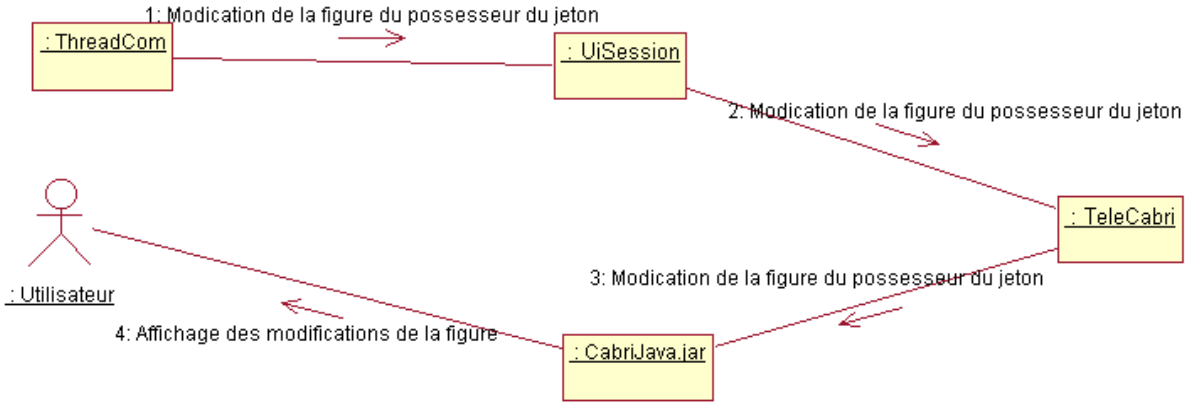


8.5.3.3. Modification figure

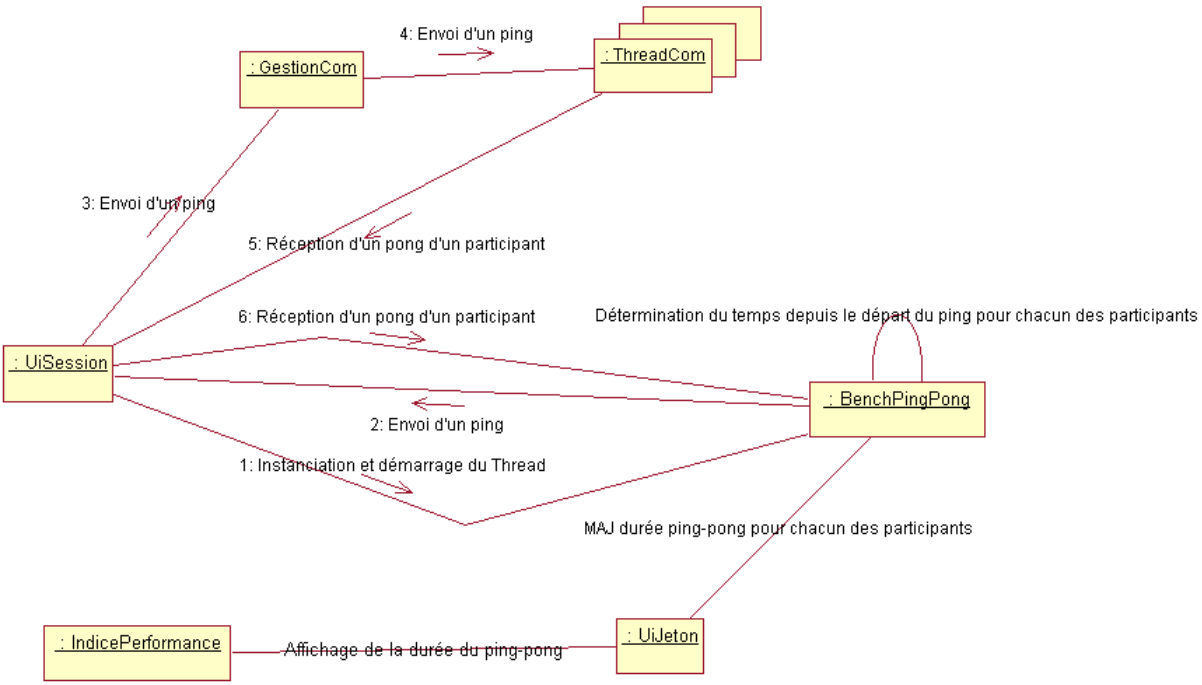
Modification de la figure par le possesseur du jeton



Réception d'une modification de la figure du possesseur du jeton



8.5.3.4. Bench ping-pong



8.6. Ce qui n'est pas encore implanté

8.6.1. Ouvrir une figure

Il est précisé dans le cahier des charges que le possesseur du jeton peut soumettre une figure qu'il a travaillée en mode hors connexion. Pour cela, il devra pouvoir ouvrir une figure et la soumettre aux autres participants.

8.6.2. Transfert d'une figure

Pour l'instant, tel quel, TéléCabriJava démarre avec feuille de géométrie vierge et un nouveau participant ne peut pas récupérer la figure du possesseur du jeton.

8.6.3. Serveur de session et serveur de serveur

Je n'ai pas eu le temps pour réaliser le serveur de session censé administrer les sessions TéléCabriJava. De même pour le serveur de serveur censé administrer les serveurs de sessions répartis sur la Terre.

8.7. Détails approfondis sur le protocole TCP

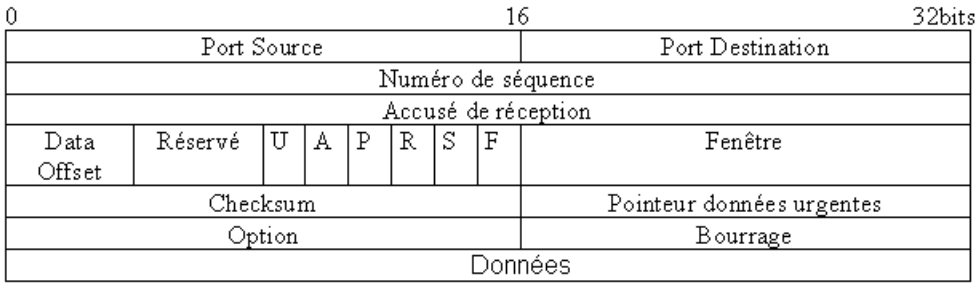
8.7.1. Le protocole TCP : fonctionnement

TCP pour **Transmission Control Protocol** est un service de transport en mode connecté au-dessus d'une couche réseau non fiable en commutation par paquets. Son Implémentation est assez complexe et doit être utilisé dans les cas où il est nécessaire d'avoir une communication sûre entre applications.

8.7.1.1. TCP dans les grandes lignes

- Le récepteur reçoit exactement la séquence d'octets envoyée par le processus source,
- Avant tout envoi de données, il y a un échange de *segments* afin de mettre en place la connexion. De même la déconnexion est négociée,
- Les données envoyées et reçues sont bufferisées afin d'améliorer la communication,
- Connexion Full-Duplex (communication dans les deux sens).

8.7.1.2. Description d'un segment TCP



Voici la description de l'entête d'un segment TCP

Port source: 16 bits - Le numéro de port de la source.
Port Destinataire : 16 bits - Le numéro de port du destinataire.
Numéro de séquence : 32 bits - Le numéro du premier octet de données par rapport au début de la transmission (sauf si SYN est marqué). Si SYN est marqué, le numéro de séquence est le numéro de séquence initial (ISN) et le premier octet à pour numéro ISN+1.
Accusé de réception : 32 bits - Si ACK est marqué ce champ contient le numéro de séquence du prochain octet que le récepteur s'attend à recevoir. Une fois la connexion établie, ce champ est toujours renseigné.
Data Offset : 4 bits - La taille de l'en-tête TCP en nombre de mots de 32 bits. Il indique là où commencent les données. L'en-tête TCP, dans tous les cas à une taille correspondant à un nombre entier de mots de 32 bits.
Réservé : 6 bits - Réservés pour usage futur. Doivent nécessairement être à 0.
Bits de contrôle : 6 bits

- URG: Pointeur de données urgentes significatif,
- ACK: Accusé de réception significatif,
- PSH: Fonction Push,
- RST: Réinitialisation de la connexion,
- SYN: Synchronisation des numéros de séquence,
- FIN: Fin de transmission.

Fenêtre : 16 bits - Le nombre d'octets à partir de la position marquée dans l'accusé de réception que le récepteur est capable de recevoir.
Checksum : 16 bits - Le Checksum est constitué en calculant le complément à 1 sur 16 bits de la somme des compléments à 1 des octets de l'en-tête et des données pris deux par deux (mots de 16 bits).
Pointeur de données urgentes : 16 bits - Communique la position d'une donnée urgente en donnant son décalage par rapport au numéro de séquence. Le pointeur doit pointer sur l'octet suivant la donnée urgente. Ce champ n'est interprété que lorsque URG est marqué.
Options : Les champs d'option peuvent occuper un espace de taille variable à la fin de l'en-tête TCP. Ils formeront toujours un multiple de 8 bits.
Bourrage (padding): Les octets de bourrage terminent l'en-tête TCP: de sorte que le nombre d'octet de celle-ci soit toujours multiple de 4 de sorte que l'offset de données marqué dans l'en-tête corresponde bien au début des données applicatives.

8.7.1.3. Contrôle des erreurs

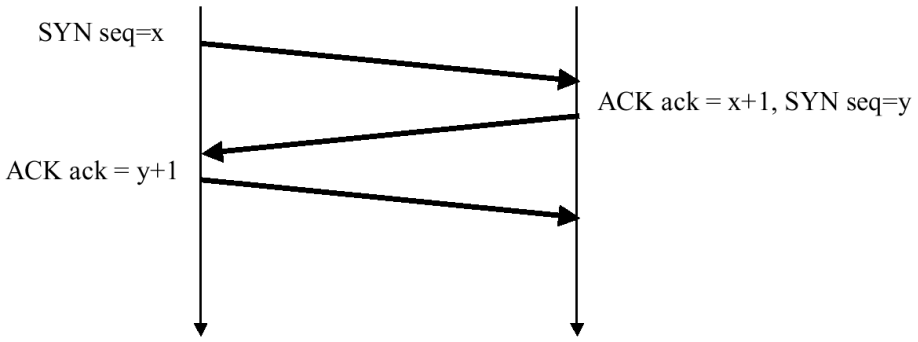
L'émetteur bufferise tous les segments qu'il envoie et attend un acquittement (acknowledgement ACK) avant de les effacer.

- Les segments portent le numéro du premier octet transporté,
- A l'envoi, une temporisation est armée,
- Le récepteur acquitte les segments reçus (le numéro du ACK pourra être égal au numéro du dernier segment reçu incrémenté de 1) :
  - soit immédiatement (segment arrivé en désordre),
  - soit après un retard
    - il a un autre segment à envoyer (piggybacking),
    - reçoit un autre segment (ACK cumulatifs),
    - règle générale : « ACK every other segment »
- Sur épuisement de la temporisation, la retransmission à partir du premier segment non acquitté (dans le cas du TCP Reno : remise à zéro des temporisations des autres segments).

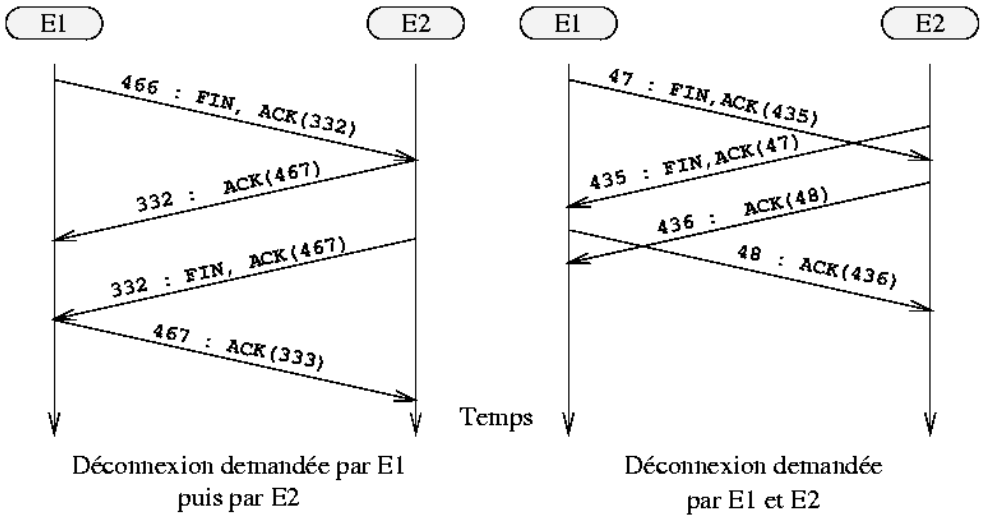
*8.7.1.4. Scénario de connexion (*three-way handshake*)*

Il faut une entente sur les numéros de séquences qui doivent être différents de ceux d'une connexion précédente. Les valeurs x et y sont choisies en fonction de l'horloge.

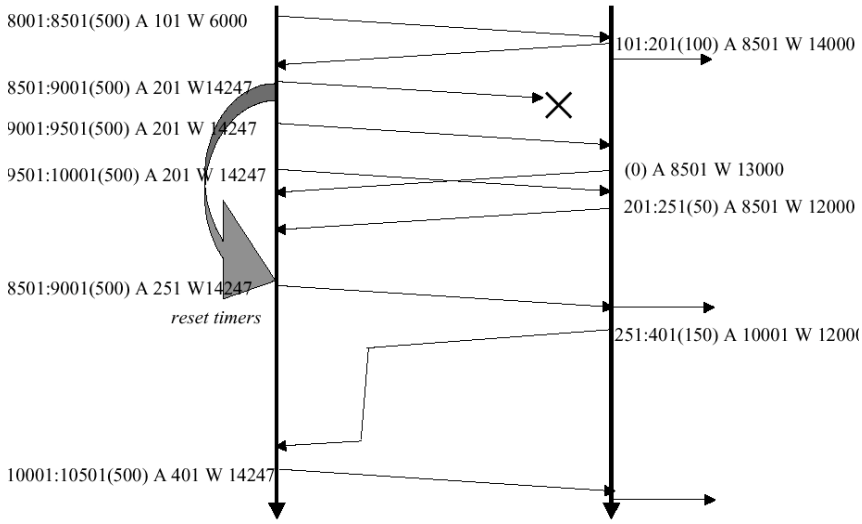
1. Le client envoie un segment de synchronisation (SYN) contenant son numéro de séquence initial.
2. Le serveur reçoit le SYN et renvoie au client un SYN contenant son numéro de séquence initial et un acquittement (ACK).
3. Le client reçoit le SYN/ACK et renvoie un ACK.



*8.7.1.5. Scénario de déconnexion (*two-way handshake*)*



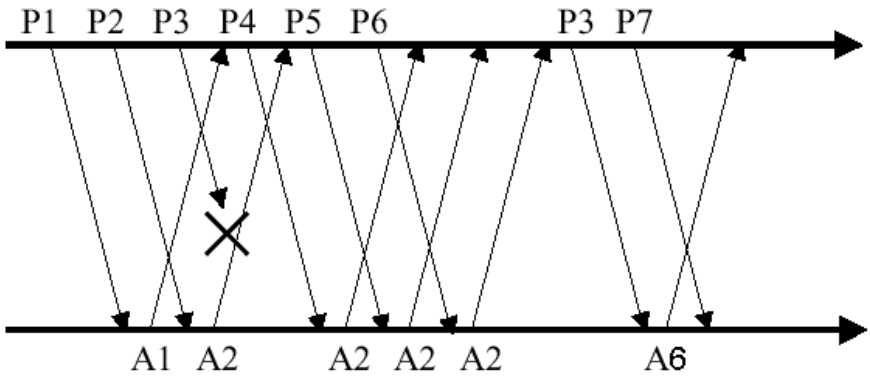
8.7.1.6. Exemple de transfert Reno



Retransmission à partir du segment 8501 :

- retransmission continue (Go-back-N) mais seulement du segment perdu (8501),
- le récepteur stocke les segments reçus en désordre (9001 et 9501),
- dès la réception de 8501, la transmission peut continuer.

8.7.1.7. Exemple de retransmission rapide



Si on reçoit trois ACK dupliqués pour le même segment avant sa temporisation, on retransmet ce segment. L'intervalle de retransmission peut être grand.

8.7.1.8. Mécanismes de TCP

L'algorithme de Nagle

L'envoyeur doit éviter d'envoyer de petits segments. Les octets de données sont accumulés dans un buffer jusqu'au moment où TCP décide de former un segment avec ces données. Le découpage en segment au niveau de TCP est donc indépendant du découpage au niveau de l'application des données à envoyer.

Il est recommandé d'envoyer les données, soit quand la taille maximum du segment est atteinte, soit quand une partie du buffer de réception est pleine. Nagle propose de ne pas envoyer tant que l'on attend un ACK. L'algorithme peut être désactivé par l'application.

Congestion

En cas de congestion (réception d'un message ICMP) on utilise une fenêtre de congestion plus petite que la fenêtre d'émission. Celle-ci augmente au fur et à mesure de la réception des ACK.

Traitement d'erreurs

RTO (Retransmission TimeOut) ⇔ temporisation associée à chaque segment non acquitté.
RTT (Round Trip Time) ⇔ temps d'aller-retour.
R ⇔ estimation lissée du temps aller-retour.

Estimation initiale

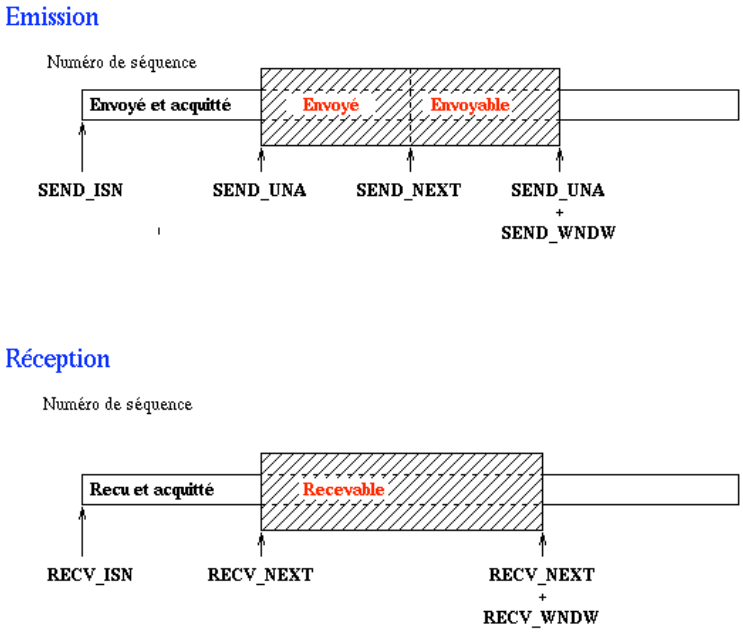
RTO=R*β, avec β=2
R ← qR+(1-q)RTT, avec q=0.9

Règles de Karn et Partridge

- On ne mesure pas de RTT s'il y a des retransmissions
  - on ne sait pas si c'est un segment perdu ou un ACK perdu
- Timer exponential backoff
  - on double la valeur de RTO à chaque retransmission
- Si au début il n'y a pas de mesures
  - RTO = 6s
  - après RTO = 12s et on applique la règle de Karn (2 _12s = 24s)

8.7.1.9. Fenêtre coulissante (sliding window)

Cette fenêtre permet d'améliorer les performances en n'attendant pas d'avoir reçu l'acquittement du segment émis pour en envoyer un nouveau. L'émetteur doit connaître la taille de la fenêtre de réception de son interlocuteur pour ne pas émettre des données pour rien. Il indique la taille de la fenêtre avec un segment TCP (éventuellement *piggybacking*). On déduit la taille de la fenêtre d'émission de celle de la fenêtre de réception et du numéro d'acquittement (prochain octet attendu).

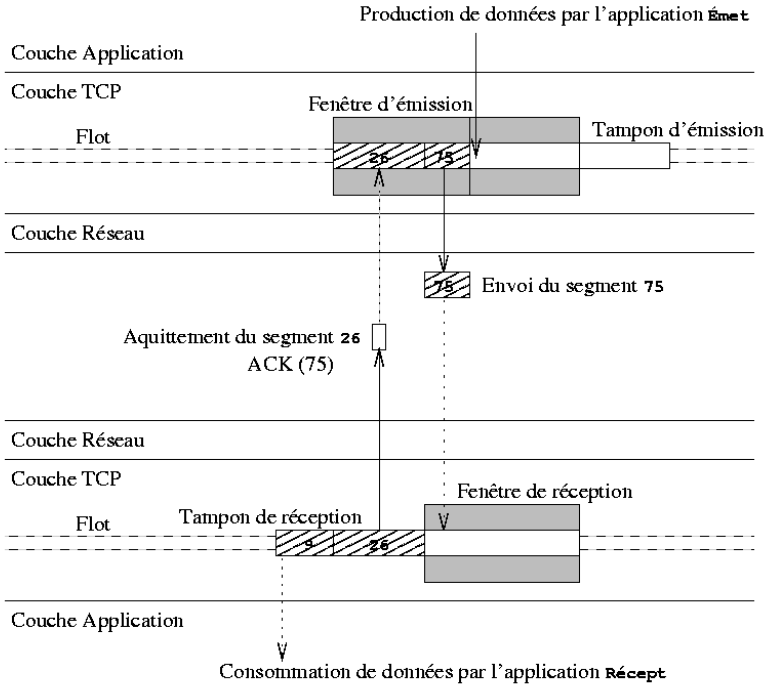


En comportement normal :

- **SEND_UNA** : coulisse en fonction des acquittements qui arrivent et **SEND_WNDW** diminue
- **SEND_NEXT** : coulisse en fonction des segments émis
- **RECV_NEXT** : coulisse en fonction des segments reçus et **RECV_WNDW** diminue
- **RECV_WNDW** : augmente lorsque l'application récupère les données reçues
- **SEND_WNDW** : augmente quand l'émetteur est informé que **RECV_WNDW** a augmenté

Si **SEND_NEXT - SEND_UNA == SEND_WNDW** alors émission stoppée.

La taille du buffer d'émission n'est pas à priori corrélée à la taille de la fenêtre d'émission. Il sert à bufferiser les données qui ne peuvent pas être émises.



8.7.1.10. Taille maximale des segments TCP

Les machines ne sont pas capables de réassembler des datagrammes IP de taille infinie. La taille maximale des datagrammes que peuvent accepter la majorité des machines ne peut excéder 576 octets. Les buffers d'émission/réception doivent pourvoir accueillir des datagrammes de cette taille.

- Les machines ne doivent pas envoyer des datagrammes IP d'une taille supérieure à 576 octets sans être certaines que les machines distantes puissent les accepter.
- La taille maximale d'un segment TCP correspond à la taille maximale d'un datagramme IP diminuée de 40 octets. La taille maximale d'un datagramme IP étant de 576 octets par défaut, la taille maximale d'un segment TCP est de 536 octets par défaut.

8.7.1.11. Contrôle de la congestion dans TCP

Démarrage lent et prévention de la congestion

- twnd ⇔ taille objectif de la fenêtre d'émission
- cwnd ⇔ taille courante de la fenêtre d'émission

Diminution multiplicative

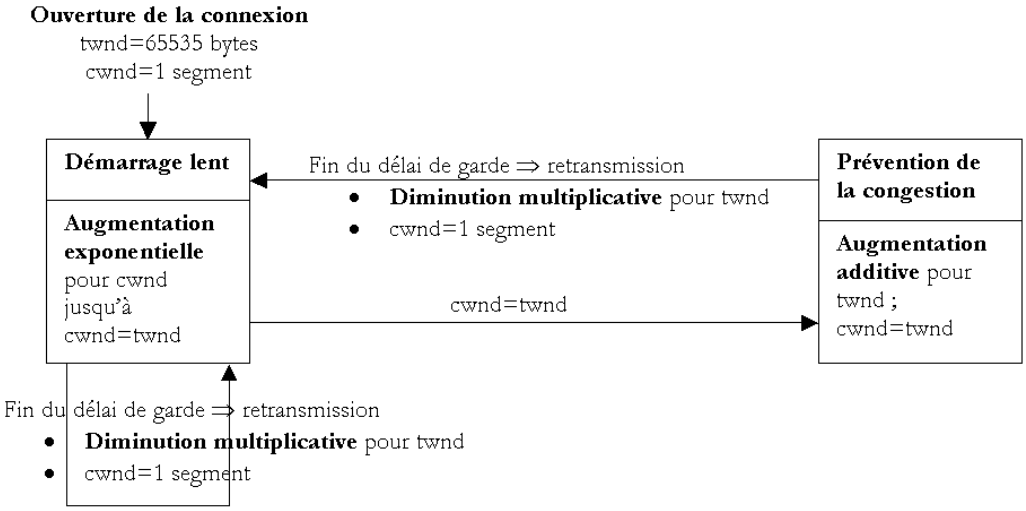
- twnd=twnd/2
- twnd=max(twnd, 2*tailleMaximaleSegment)

Augmentation additive

- A chaque ACK (acquittement de segment),
- twnd=twnd+tailleMaximaleSegment_/twnd
- twnd=min(twnd, tailleMaximale)

Augmentation exponentielle

- A chaque ACK (acquittement de segment),
- cwnd=cwnd+tailleMaximaleSegment



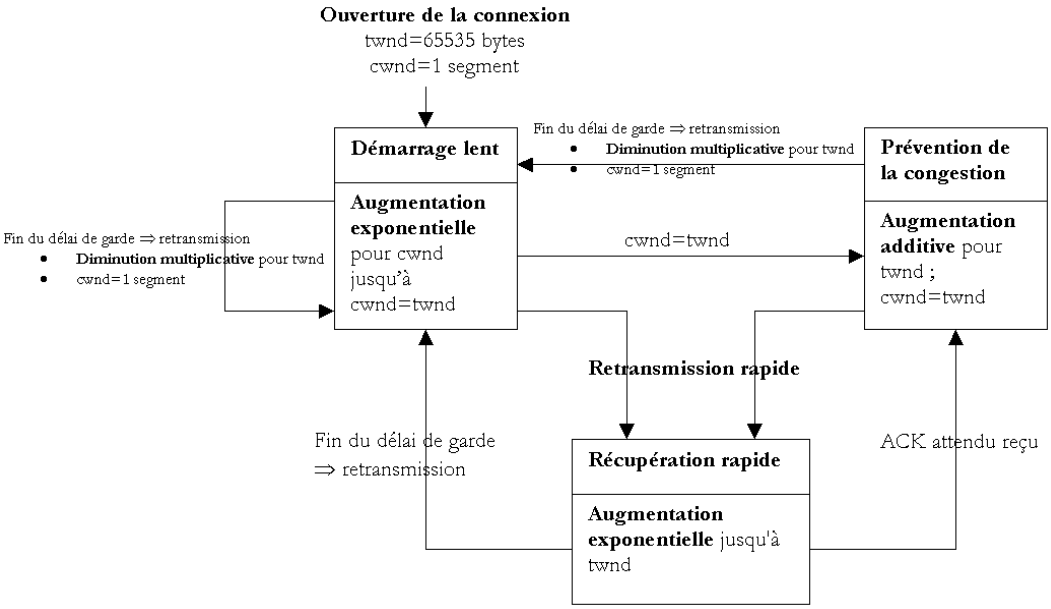
Récupération rapide

Récupération rapide

- cwnd=twnd+3*tailleMaximaleSegment
- cwnd=min(cwnd, 64Kbytes)
- retransmission du segment perdu

Pour chaque ACK dupliqué

- cwnd=cwnd+tailleMaximaleSegment
- cwnd=min(cwnd, 64Kbytes)
- envoyer les segments suivants



8.7.2. Configuration des sockets avec Java

Toutes les méthodes explicitées dans cette partie appartiennent à la classe Socket du paquetage Java.net.

8.7.2.1. ReceiveBufferSize et SendBufferSize

- public void setReceiveBufferSize(int size);
  Permet de stocker les messages reçus.
- public void setSendBufferSize(int size);
  Permet de stocker les messages à envoyer.

8.7.2.2. KeepAlive

- public void setKeepAlive(boolean on);
  Permet d'activer/désactiver l'option SO_KEEPALIVE. Cette option permet d'envoyer un segment à la machine distante si une durée supérieure à deux heures (par défaut) s'est écoulé depuis la dernière fois qu'un segment a été reçu ou envoyé. La machine distante doit répondre par un accusé de réception à ce segment. Cela permet de détecter si la machine distante est devenue inaccessible.

8.7.2.3. SoLinger

- public void setSoLinger(boolean on, int linger);
  Permet d'activer/désactiver l'option SO_LINGER en précisant sa durée en secondes. Sur certains systèmes UNIX, cette option n'a aucun effet. La valeur maximum du timeout est spécifique à chaque plateforme. Cette option ordonne au noyau de suspendre la connexion TCP au lieu de la fermer proprement.

8.7.2.4. SoTimeout

- public void setSoTimeout(int timeout);
  Active/désactive l'option SO_TIMEOUT en précisant sa durée en millisecondes. Avec une durée différente de 0, un appel à la méthode read() sur l'InputStream associé au socket sera bloquant pendant cette durée au maximum. Si le TIMEOUT expire, une interruption java.io.InterruptedIOException est levée. Un timeout fixé à 0 correspond à un timeout infini.

8.7.2.5. TcpNoDelay

- public void setTcpNoDelay(boolean on);
  Permet d'activer/désactiver l'option TCP_NODELAY, c'est à dire l'algorithme de Nagle. Cet algorithme stocke les données à envoyer dans le buffer d'envoi jusqu'à ce qu'il y ait:
  - *une donnée inconnue,*
  - *assez de données dans le buffer pour former un segment complet.*

  Le délai ne peut pas être indéterminé puisque la première condition devient obligatoirement vraie avec le timeout de transmission.
  Cet algorithme est indésirable si l'on veut envoyer de petits messages immédiatement dans une application comme Telnet par exemple. Il faut alors désactiver l'algorithme en positionnant TCP_NODELAY à true.

8.7.3. TCP avec Java

8.7.3.1. Introduction

Un système de protocoles de la dimension de TCP/IP assure les fonctions suivantes :

- fractionnement des messages pour leur transport sur le réseau,
- interface avec la carte réseau,
- adressage de l'émetteur et du récepteur,
- routage des données à travers le réseau qui contient des machines hétérogènes,
- contrôle des erreurs par accusé de réception.

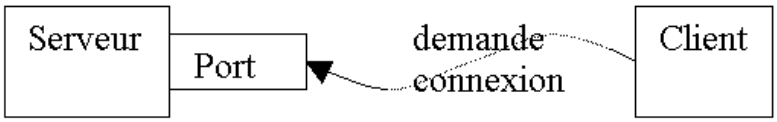
8.7.3.2. Description

- TCP fournit une communication point à point. TCP établit une connexion entre deux machines qui veulent communiquer.
- TCP garantit que les données émises arriveront au destinataire.
- TCP garantit que la réception des données se fait dans le même ordre que leur émission.

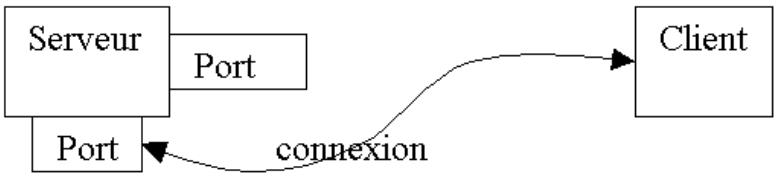
8.7.3.3. Les sockets

Les sockets pour une connexion bi-point TCP

Une application serveur possède un *socket* composé de l'adresse IP de la machine et du numéro de port où on peut la joindre. Une application cliente qui désire s'y connecter doit connaître cette adresse IP et son port.



Le serveur accepte la connexion et fournit un nouveau *socket* avec un nouveau port de communication pour pouvoir continuer à écouter sur le *socket* original.



Du côté du client, si la connexion est acceptée, un *socket* est crée pour permettre la communication avec le serveur.

Les sockets avec Java

Le package java.net fournit une classe `Socket` qui implante une connexion bi-point entre un programme Java et un autre programme du réseau. Ce package inclut aussi une classe `ServerSocket` implantant un *socket* qu'un serveur peut utiliser pour écouter et accepter des clients.

8.7.3.4. Réalisation d'un client-serveur en Java

Réalisation d'un client en Java

On va examiner un exemple de connexion d'un client à un serveur dont voici les étapes :

- ouverture d'un *socket*,
- ouverture d'un flot d'entrée et de sortie sur le *socket*,
- lecture et écriture sur les flots en accord avec les protocoles du serveur (présenté un peu plus loin),
- fermeture des flots,
- fermeture du *socket*.

On commence par créer un *socket* en passant au constructeur le nom de la machine qui héberge le serveur et le port de cette application serveur :

```
echoSocket = new Socket("taranis", 7);
```

Si la machine n'est pas sur le réseau local, il faut fournir son adresse IP à la place de son nom.

On récupère ensuite le flot de sortie du *socket* et on ouvre un `PrintWriter` :

```
out = new PrintWriter(echoSocket.getOutputStream(), true);
```

On récupère aussi le flot d'entrée du *socket* et on ouvre un `BufferedReader` :

```
in = new BufferedReader(new
InputStreamReader(echoSocket.getInputStream()));
```

Puis, on exécute en boucles les actions suivantes :

- on lit une ligne tapée au clavier,
- on l'envoi au serveur,
- on attend une réponse du serveur.

```
String userInput;
while ((userInput = stdIn.readLine()) != null) {
        out.println(userInput);
        System.out.println("echo: " + in.readLine());
}
```

Enfin, une fois les traitements terminés, on ferme les flots de données :

```
out.close();
in.close();
stdIn.close();
```

Et on ferme le *socket* :

```
echoSocket.close();
```

Réalisation d'un serveur en Java

On va examiner un exemple de serveur dont voici les étapes :

- ouverture d'un *socket* serveur,
- attente de connexion et ouverture d'un *socket* client,
- ouverture d'un flot d'entrée et de sortie sur le *socket* client,
- lecture et écriture sur les flots en accord avec les protocoles du client (présenté précédemment),
- fermeture des flots,
- fermeture du *socket* serveur et du *socket* client.

On commence par créer une instance de `ServerSocket` qui va écouter sur un port de la machine :

```
try {
        serverSocket = new ServerSocket(4444);
}
catch (IOException e) {
        System.out.println("Could not listen on port: 4444");
        System.exit(-1);
}
```

La classe `ServerSocket` du package *java.net* fournit une implantation du côté serveur d'une connexion client-serveur.

On va ensuite attendre une demande de connexion d'un client sur l'hôte et le port de l'application serveur :

```
Socket clientSocket = null;
try {
        clientSocket = serverSocket.accept();
}
catch (IOException e) {
        System.out.println("Accept failed: 4444");
        System.exit(-1);
}
```

Quand une demande de connexion est acceptée, la méthode `accept` renvoie une instance de `Socket` avec un port différent du *socket* serveur. Ainsi, le serveur peut communiquer avec ce client et éventuellement continuer à écouter les requêtes de connexion de nouveaux clients.

Remarque : dans le cas où le serveur doit supporter plusieurs clients, il faut créer un `Thread` par client connecté et nous donnons l'algorithme du serveur :

```
TantQue vrai
    accepter une connexion avec un nouveau client
    créer un Thread pour communiquer avec ce client
```

Une fois que le serveur a établi une connexion avec un client, on va récupérer les flux d'entrée et sortie et ouvrir les *readers* et *writers* associés.

```
PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
BufferedReader in = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
```

Puis, on exécute en boucles les actions suivantes :

- on attend un message du client,
- on lit une ligne tapée au clavier,
- on l'envoi au client.

```
String inputLine, outputLine;
while ((inputLine = in.readLine()) != null) {
        System.out.println("echo: " + inputLine);
        outputLine = stdIn.readLine();
        out.println(outputLine);
}
```

Enfin, une fois que le client décide de ne plus rien envoyer et coupe la connexion, on ferme les flots de données :

```
out.close();
in.close();
stdIn.close();
```

Puis, on ferme le *socket* client et le *socket* serveur :

```
clientSocket.close();
serverSocket.close();
```