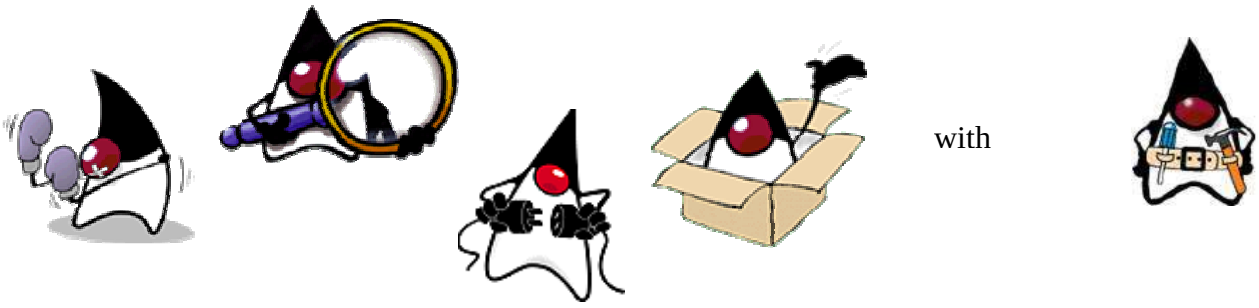
Laboratoire de Réseaux et technologie Internet (TCP-UDP/IP & HTTP en C/C++/Java) : projet "Inpres-Ferries"

3ème Informatique de gestion & 3ème Informatique et systèmes (opt. Informatique industrielle et Réseaux-télécommunications) 2011-2012

Claude Vilvens, Christophe Charlet, Mounawar Madani, Jean-Marc Wagner
(Network programming team)
en collaboration avec
Ludovic Kuty (from Database team)



1. Préambule

Les travaux de programmation réseaux présentés ici constituent la description technique des travaux de laboratoire du cours de "**Réseaux et technologie Internet**". Il s'agit ici de maîtriser les divers paradigmes de programmation réseau TCP/IP et HTTP dans les environnements UNIX et Windows, en utilisant les langages C/C++ et Java. Ces travaux ont été conçus de manière à pouvoir collaborer avec les travaux de laboratoire des cours de "**Compléments programmation réseaux**" (3ème informatique réseaux-télécoms) et "**Technologies du e-commerce**" (3ème informatique de gestion); certains points

correspondants sont donc déjà vaguement évoqués ici (les titres associés portent une astérisque).

Les développements C/C++ UNIX sont sensés se faire avec les outils de développement disponibles sur les machines Sunray [sunray1v440 – 10.59.28.1]. Cependant, Dev-C++ sur les PCs peut vous permettre de développer du code en dehors de ces machines. Les développements Java sont à réaliser avec **NetBeans 6.***. Le serveur Web avec moteur à servlets/JSP utilisé est **Tomcat 6.*** sous Windows (PC) et/ou Unix (machines Indochine, U2 ou Inxs).

Les travaux peuvent être réalisés

- soit par pour **une équipe de deux étudiants** qui devront donc se coordonner intelligemment et se faire confiance, sachant qu'il faut être capable d'expliquer l'ensemble du travail (pas seulement les éléments dont on est l'auteur);
- soit par un **étudiant travaillant seul** (les avantages et inconvénients d'un travail par deux s'échangent : on a moins de problèmes quand il faut s'arranger avec soi-même).

2. Règles d'évaluation du laboratoire

Afin d'éviter tout problème lié à l'évaluation du cours de Réseaux et technologie Internet, il a été établi un document pédagogique définissant les règles de cotation utilisées par les enseignants de l'équipe responsable du cours.

1) L'évaluation établissant la note du cours de "Réseaux et technologie Internet" est réalisée de la manière suivante :

- théorie : un examen écrit en janvier 2011 (sur base d'une liste de questions fournies en novembre et à préparer) et coté sur 20;
- laboratoire : 3 évaluations (aux dates précisées ci-dessous), chacune cotée sur 20; la moyenne de ces 3 cotes fournit une note de laboratoire sur 20;
- note finale : **moyenne pondérée de la note de théorie (poids de 4/10) et de la note de laboratoire (poids de 6/10)**.

Cette procédure est d'application tant en 1ère qu'en 2ème session, ainsi que lors d'une éventuelle prolongation de session.

2) *Dans le cas où les travaux sont présentés par une équipe de deux étudiants*, chacun d'entre eux doit être capable d'expliquer et de justifier l'intégralité du travail (pas seulement les parties du travail sur lesquelles il aurait plus particulièrement travaillé).

3) *Dans tous les cas*, tout étudiant doit être capable d'expliquer de manière générale (donc, sans entrer dans les détails) les notions et concepts théoriques qu'il manipule dans ses travaux (par exemple: socket, signature électronique, certificat, etc).

4) En 2ème session et en prolongation de session, un **report de note** est possible pour chacune des trois notes de laboratoire ainsi que pour la note de théorie **pour des notes supérieures ou égales à 10/20**.
Toutes les évaluations (théorie ou laboratoire) ayant des **notes inférieures à 10/20** sont **à représenter dans leur intégralité**.

La première partie des travaux de programmation réseaux sera **évaluée** par l'un des professeurs du laboratoire **à partir du 24 octobre 2011** (avec rentrée d'un dossier papier limité au code des serveurs et de la librairie - le délai est à *respecter* impérativement).

La deuxième partie de ces travaux sera **évaluée** par l'un des professeurs du laboratoire **à partir du 28 novembre 2011** (avec rentrée d'un dossier papier limité au code des serveurs - le délai est toujours à *respecter* impérativement).

La troisième partie sera évaluée lors de l'examen de laboratoire en janvier-février 2012 (le dossier papier n'est plus nécessaire).

3. Le contexte général du laboratoire

La société "**HAppy Ferry Inpres Company**" (**HAFIC**) existe depuis 1992 et exploite des lignes de ferry entre la France, la Grande-Bretagne et la Scandinavie. Ses concurrents les plus connus sont P&O, SeaFrance, North Sea Ferries, Norfolk Line, …

Outre les navires et leur personnel navigant, la société doit aussi gérer ses terminaux (maritimes, pas informatiques ;-) !) car, à la différence d'autres compagnies, elle est propriétaire de ses installations. C'est à ce niveau que la mise en place d'une structure informatique intervient.

Pour comprendre les besoins de cette structure, considérons le parcours d'un voyageur qui se présente à l'entrée des installations portuaires dans le but d'embarquer sur un ferry HAFIC.

1) La première étape est la frontière et la douane : on quitte un pays pour entrer dans un autre. Il s'agit donc pour le voyageur de produire ses pièces d'identité et celles des autres voyageurs qui sont à bord. Les agents des services des frontières vérifient ces pièces (électroniques ou non) en s'adressant à un serveur **Serveur_Identités** qui fait relais avec les serveurs du Registre national et les serveurs analogues des autres pays. Inutile de dire que toutes les techniques de sécurité logicielle sont ici nécessaires (cryptages et authentification implémentés spécifiquement ou assurés par SSL). En complément, les éventuels contrôles douaniers sont opérés.

2) La deuxième étape est celle de l'enregistrement des voyageurs par la compagnie maritime (le "check-in") : ces voyageurs peuvent déjà disposer d'un titre de traversée (acheté par Internet) mais ils peuvent aussi acquérir ce titre sur place. Le personnel de la compagnie s'adresse pour ces opérations à un **Serveur_Compagnie** chargé de gérer ces informations au moyen d'une base de données BD_HAFIC. Les voyageurs se voient alors confier un ticket d'embarquement et un panonceau à aposer sur le pare-brise avec un numéro de file d'attente où se placer pour accéder au ferry.
A remarquer que les tickets d'embarquement ne sont confiés aux voyageurs que si on a pu vérifier qu'ils ont passé la frontière sans encombres (le serveur des identités doit donc correspondre donc avec le serveur de la compagnie).

3) Les véhicules se retrouvent donc ainsi dans une file. Leurs mouvements vers le ferry sont coordonnés par les agents attachés aux terminaux. Ces agents utilisent des postes clients d'un **Serveur_Terminaux** qui gère tous les mouvements (de véhicules et de navires). Il a connaissance des heures de départ et d'arrivée des ferries parce que Serveur_Compagnie lui envoie ces informations journellement.

4) Avant de partir, et pour patienter, les voyageurs peuvent tuer le temps dans une salle d'attente qui est en fait un petit complexe avec cafétaria-bar, jeux et librairie. Ce complexe

abrite un serveur **Serveur_Information** dont le seul nom explique le rôle : les postes clients fournissent les informations pratiques, les cours des unités monétaires, la liste des articles achetables en free-tax sur le ferry, …

4. Le contexte informatique

Le complexe portuaire doit disposer donc des trois premiers serveurs évoqués ci-dessus (le Serveur_Information interviendra dans d'autres laboratoires). Plus précisément, on aura :

1) un serveur multithread Unix (**Serveur_Terminaux**) qui attend

- sur le port PORT_TERMINAL les requêtes d'une application cliente (**Applic_Terminaux**) de ce serveur qui est utilisée par les agents portuaires; tous deux sont programmés en C/C++;
- sur le PORT_ADMIN les requêtes d'une application Java (**Admin_Terminaux**) pour permettre aux responsables informatiques d'administrer ce serveur;
- sur le PORT_PORT (!) les requêtes d'une application Java dont le seul rôle est de signaler l'arrivé ou le départ d'un ferry dans la zone des terminaux (**Applic_InOut**); physiquement ces postes se trouvent aux extrémités des jetées du port.

2) un serveur multithread Windows (**Serveur_Compagnie**) qui est chargé de satisfaire les requêtes provenant

- de l'application **Applic_Check-In** des agents de la compagnie responsables du check-in des véhicules arrivant de la douane; il utilise pour cela le PORT_VOYAGEURS;
- de l'application **Web Applic_Reservations** qui permet au public d'acheter par Internet des places sur un ferry; il utilise pour cela le PORT_WEB_VOYAGEURS;

Le serveur et toutes ces applications sont programmés en Java. La base de données BD_HAFIC est une base MySQL.

3) un serveur multithread Windows ou Linux (**Serveur_Identités**), en attente sur le port PORT_VERIFY, et son application cliente (**Applic_Identités**) destinée aux agents des frontières chargés d'opérer les vérifications d'identité et les déclarations de douanes; tous sont programmés en Java;

4) un serveur multithread Windows ou Linux (**Serveur_Informations**) et une application cliente (**Applic_Infos**) de ce serveur qui est utilisée par les clients en attente pour obtenir diverses informations; tous deux sont encore programmés en Java.

5. Avertissement

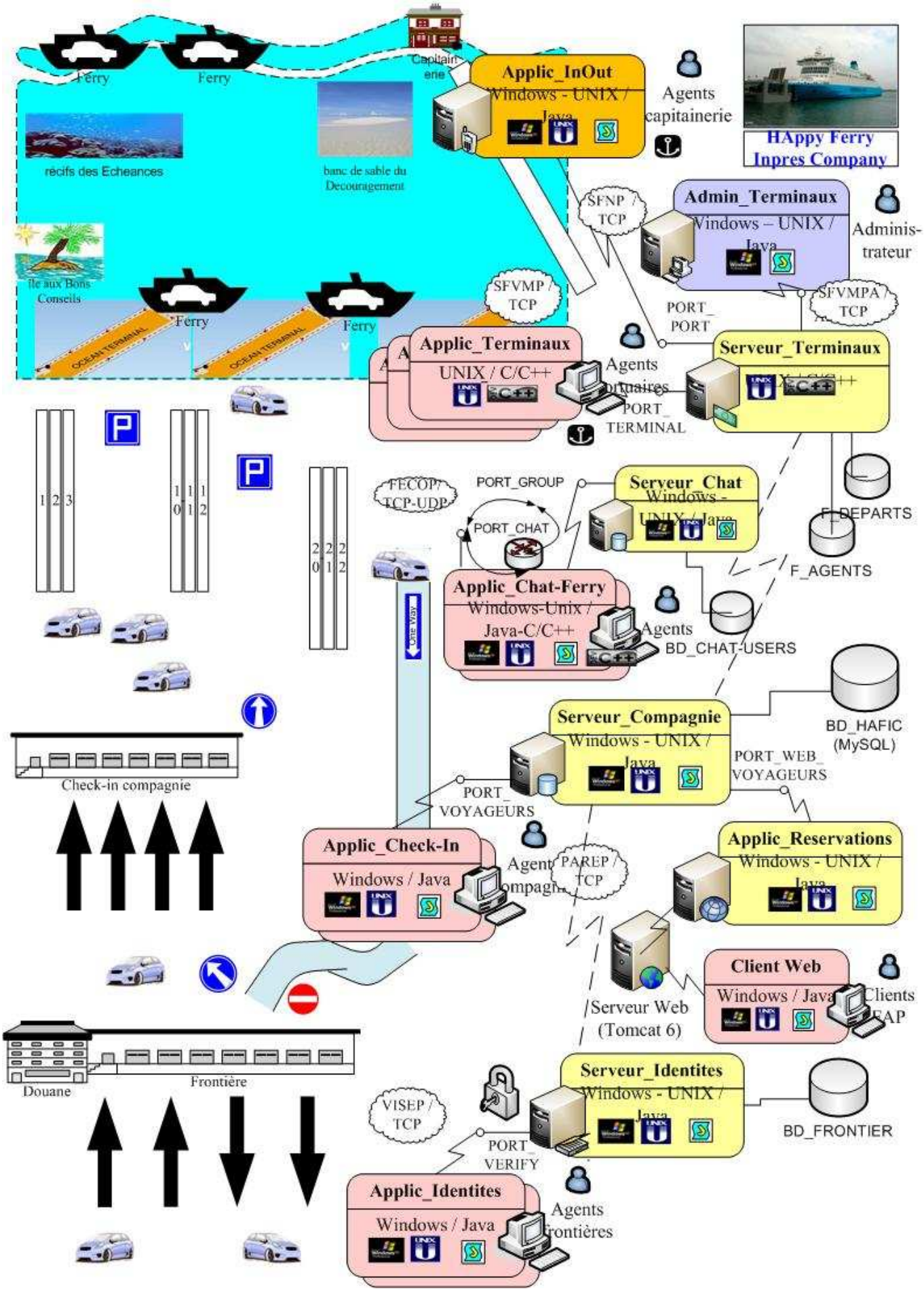
Dans ce qui suivra, un certain nombre de points ont été simplifiés par rapport à la réalité. Certains points ont également été volontairement laissés dans le vague ou l'obscur. Dans tous les cas, l'imagination est la bienvenue pour donner à vos solutions un cachet plus personnel, plus réaliste, plus professionnel. Cependant, pour vous éviter de vous fourvoyer dans une impasse ou perdre votre temps à des options de peu d'intérêt, il vous est vivement recommandé de prendre conseil au près de l'un de vos professeurs concernés par le projet.

Autre chose : une condition sine-qua-non de réussite est le traitement systématique et raisonné des **erreurs** courantes (codes de retour, errno, exceptions).

Une plage de 10 ports TCP-UDP vous est attribuée sur les machines Sun. Il vous est demandé de la respecter pour des raisons évidentes …

6. L'infrastructure informatique générale

La gestion de la plate-forme évoquée ci-dessus nécessite donc la mise en place d'un certain nombre de serveurs, de clients et de bases de données évoqués dans le schéma suivant. Certains éléments relèvent du cours de "Réseaux et technologies Internet" (complètement ou en partie) et seront décrits ici, tandis que d'autres relèvent de "Technologies de l'e-commerce" (informatique de gestion) ou de "Compléments programmation réseaux" (informatique réseaux-télécoms) et seront décrits plus précisément ailleurs. Les descriptions détaillées suivent dans la liste des travaux à effectuer.


Ferry
Ferry
Capitainerie
Applic_InOut
Windows - UNIX / Java
Agents capitainerie
HAppy Ferry
Inpres Company
récifs des Echeances
banc de sable du Découragement
Admin_Terminaux
Windows – UNIX / Java
Adminis-trateur
SFNP / TCP
Ile aux Bons Conseils
SFVMP / TCP
SFVMPA / TCP
PORT_ PORT
Ferry
Ferry
OCEAN TERMINAL
Applic_Terminaux
UNIX / C/C++
Agents portuaires
PORT_ TERMINAL
Serveur_Terminaux
1 2 3
1 1 1 0 1 2
2 2 2 0 1 2
FECOP / TCP-UDP
PORT_GROUP
PORT_CHAT
Serveur_Chat
Windows - UNIX / Java
F_DEPARTS
F_AGENTS
Applic_Chat-Ferry
Windows-Unix / Java-C/C++
Agents
BD_CHAT-USERS
CATEWAY
Serveur_Compagnie
Windows - UNIX / Java
BD_HAFIC (MySQL)
PORT_WEB_ VOYAGEURS
Check-in compagnie
PORT_ VOYAGEURS
Applic_Check-In
Windows / Java
Agent compagn
PAREP / TCP
Applic_Reservations
Windows - UNIX / Java
Client Web
Windows / Java
Clients RAP
Serveur Web (Tomcat 6)
Douane
Frontière
Serveur_Identites
Windows - UNIX / Java
BD_FRONTIER
VISEP / TCP
PORT_ VERIFY
Applic_Identites
Windows / Java
Agents frontières

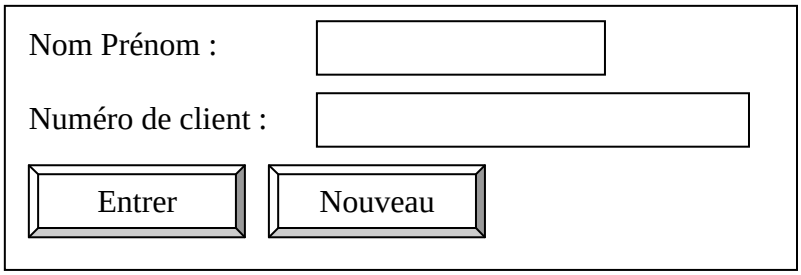
Les travaux de l'évaluation 1

7. Deux applets Java

Il s'agit de confectionner un interface de type GUI qui permettra aux clients utilisant l'application Web **Applic_Reservations** (évoquée au point 13) d'accéder au site Web pour effectuer des réservations de traversées sur un ferry. Bien sûr, à ce stade, il ne saurait y avoir de réaction côté serveur (ce sera le rôle de servlets et de Java Server Pages évoquées plus loin).

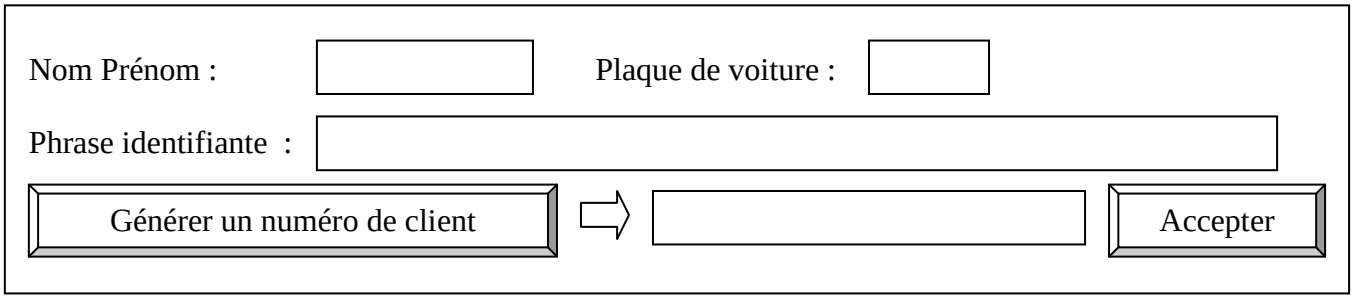
L'interface est constituée d'une page HTML contenant deux applets :

♦ l'applet Login se présente sous la forme d'un GUI demandant l'introduction d'un nom et prénom (séparés par un espace – comme sur l'EV ;-)) et d'un numéro de client, donc selon le canevas suivant :



"Entrer" correspond à un client qui possède déjà un numéro, "Nouveau" à un nouveau client qui n'en possède pas encore (ou à un client qui a oublié son numéro) Dans ce dernier cas, il peut demander un numéro au moyen de la 2ème applet :

♦ l'applet Generator se présente sous la forme du canevas suivant :



Un numéro de client est généré (provisoirement – voir point 13, où l'on verra que l'on s'adresse en fait à une servlet pour obtenir ce numéro) par appui sur le bouton "Générer un numéro de client" au moyen d'un hashage des informations entrées complétées de la date (on utilisera simplement pour cette génération la méthode hashCode() de la classe String). L'appui sur le bouton "Accepter" provoquant la recopie du numéro dans la zone "Numéro de client" de la première applet Login.

L'appui sur le bouton "Entrer" enverra provisoirement sur une page statique de bienvenue. Par après, ce sera une servlet qui réagira (voir encore une fois point 13)

La page HTML et ses deux applets seront déployées sur un serveur Tomcat 6 non local tournant sur un PC.

8. Les accès à la base de données BD_HAFIC

8.1 La base de données

Cette base doit contenir les informations utiles concernant les réservations et les voyageurs. En fait, certains agents zélés de la compagnie avaient déjà confectionné une base de départ sous MySQL (elle est donc fournie) et il est simplement demandé de la compléter si nécessaire. Ses tables, si on admet un certain nombre d'approximations, de simplifications et d'omissions sans importance pour le projet tel que défini ici, seront en première analyse :

♦ **navires** : avec comme champs essentiels un matricule, un nom et une capacité en termes de voitures et de véhicules lourds (cars et camions);

♦ **ports** : nom, pays, nombre de terminaux et nombre de terminaux occupés (information dynamique);

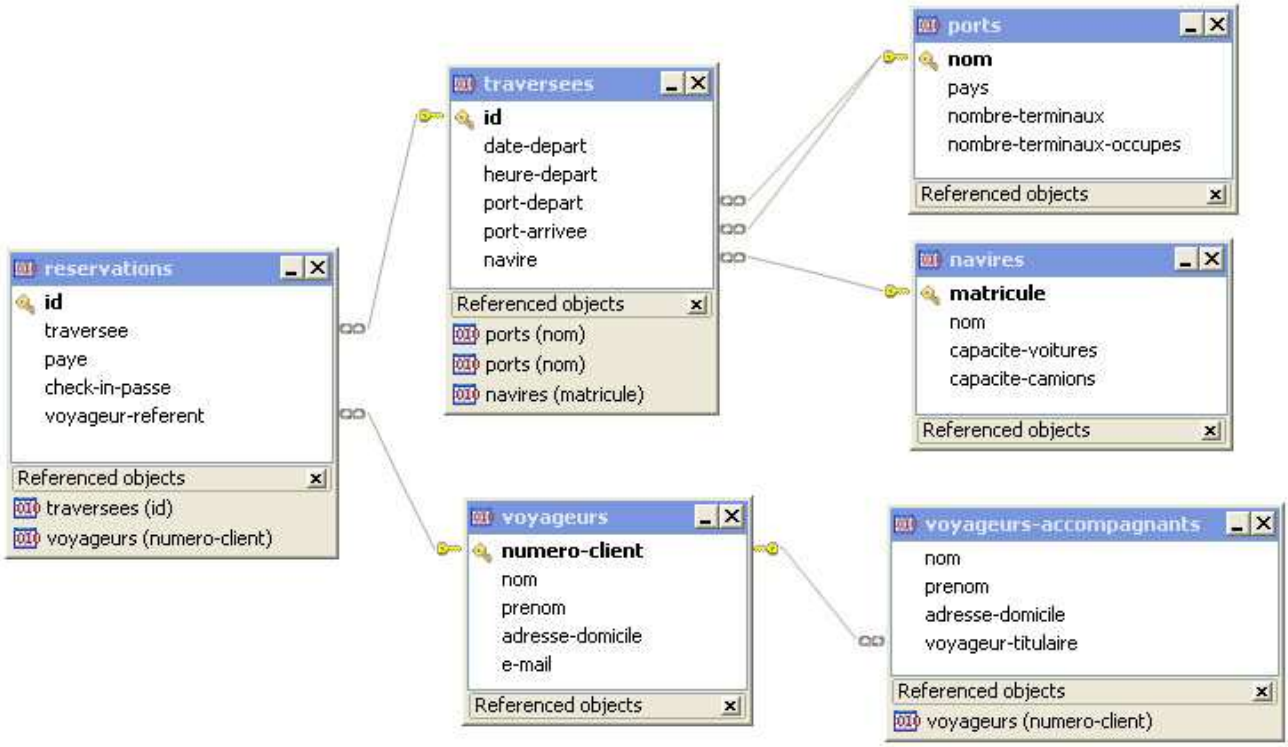
♦ **traversees** : identifiant (de la forme 20110915-TR01), date et heure de départ, port de départ et de destination, navire utilisé;

♦ **voyageurs** (voyageurs au nom de qui la réservation est faite = titulaires) : numéro de client (celui dont il est question dans les applets ci-dessus), nom, prénom, adresse du domicile, adresse e-mail;

♦ **voyageurs-accompagnants** : nom, prénom, adresse du domicile, voyageur titulaire;

♦ **reservations** : identifiant (de la forme 20110915-RES01), traversée, voyageur-titulaire, payé (O/N), passé au check-in (O/N).

On a toute liberté pour ajouter des champs, voire d'autres tables. Le schéma relationnel de base est le suivant :



8.2 Un bean d'accès aux bases de données

L'accès à la base de données ne devrait pas se faire avec les primitives JDBC utilisées telles quelles, mais plutôt au moyen d'objets métiers encapsulant le travail, idéalement des Java Beans. On demande donc de construire un groupe de classes permettant l'accès (c'est-à-dire à tout le moins la connexion et les sélections de base) le plus simple possible.

Outre l'accès aux bases relationnelles de type MySQL ou Oracle, on souhaite aussi pourvoir accéder à des documents CSV, c'est-à-dire des fichiers textes correspondant à une table SQL dont les tuples sont des lignes de texte avec des champs séparés par un séparateur convenu. Un exemple d'un tel fichier est celui qui comporte les champs nom, prénom, niveau, password;

F_AGENTS.csv
nom, prénom, niveau, password Lendormi, Julien, 2, Arglxx007 Lajolie, Julie, 3, UnJourMonPrinceViendra Lenaif, Albert, 5, azerty Sculpturale, Valérie, 5, Johnny69 …

Le programme de test de la petite librairie ainsi construite

- se connectera au choix à une base MySQL ou CSV;
- enverra une requête de type "select count(*) from" et affichera le résultat;
- enverra une requête de type "select * from where" (adaptée à la table visée – pas de tentative de généricité à ce stade) et affichera le résultat.

9. Serveur_Terminaux

9.1 Fonctionnalités demandées pour Applic_Terminaux

Il s'agit d'un serveur multithread UNIX, programmé en C/C++ (à priori sur une machine Sun) selon un modèle "pool de threads".

Son premier rôle est, comme déjà dit, de répondre, sur le port PORT_TERMINAL, aux requêtes des agents portuaires (qui utilisent **Applic_Terminaux**) qui régissent les mouvements des véhicules en attente et des ferries.

L'application cliente est assez simple et utilise le protocole **SFVMP** (**S**imple **F**erries and **V**ehicles **M**anagement **P**rotocol). Ce ***protocole applicatif*** de conversation entre le serveur et ses clients, est basé **TCP**, est **à états** et possède une liste de commandes prédéfinies que chacune des deux parties sait gérer. Ces commandes sont (des tableaux mémoire sont utilisés en certains points au lieu de fichiers, ce qui serait plus réaliste, afin de faciliter le développement) :

protocole **SFVMP**		
commande	**sémantique de la requête**	**réponse éventuelle**
LOGIN	démarrage de l'application : un agent se fait reconnaître *paramètres* : nom, password, numéro de terminal	oui ou non, sur base d'un fichier CSV (F_AGENTS) contenant les nom et prénom des agents avec leur mot de passe – le serveur garde en mémoire l'association agent-terminal

ASK-NEXT-DEPARTURE	demande de l'heure du prochain départ ainsi que du nom du ferry associé *paramètre* : -	- DEPARTURE-KNOWN: si un ferry se trouve au terminal considéré (déterminé au démarrage dans le tableau ArrayRegisteredFerries - voir aussi plus bas), l'heure trouvée pour aujourd'hui pour ce ferry (format 00:00) dans un fichier CSV de départs (F_DEPARTS – structure à imaginer) ➔ point a) – DEPARTURE-UNKNOWN : si pas d'heure trouvée, le ferry est simplement au quai : l'agent reste en attente - NO-FERRY : si pas de ferry au terminal ➔ point b)
a) quand une heure de départ valide a été obtenue, les requêtes suivantes sont utilisables :		
ASK-BEGIN-LOADING	demande d'autorisation du début du chargement du ferry *paramètre* : l'heure (déterminée automatiquement)	ACK ou FAIL applicatif selon que la différence entre l'heure de départ et l'heure envoyée est inférieure à 45 minutes ou pas
NOTIFY-END-LOADING	notifie que le ferry est chargé - demande d'autorisation de départ *paramètres* : l'heure (déterminée automatiquement)	ACK ou FAIL applicatif selon que l'heure envoyée est postérieure d'au moins 15 minutes à l'heure de début de chargement
FERRY-LEAVING	notifie que le ferry quitte le terminal *paramètres* : l'heure (déterminée automatiquement)	le ferry est retiré du tableau ArrayRegisteredFerries et passe dans un tableau ArrayLeavingFerries
b) quand le terminal est libre, les requêtes suivantes sont utilisables :		
ASK-FOR-FERRY	notifie qu'un ferry peut être accepté au terminal	si un ferry est en attente (ces ferries sont référencés dans un tableau ArrayWaitingFerries), le nom du ferry affecté au terminal dans un tableau ArrayRegisteredFerries
FERRY-ARRIVING	notifie que le ferry arrive au terminal *paramètres* : l'heure (déterminée automatiquement) et le nom du ferry	ACK ou FAIL applicatif selon qu'un ferry avait bien été affecté au terminal ou

		pas – si ACK, la commande ASK-NEXT-DEPARTURE peut à présent être envoyée
CLOSE	fermeture de l'application *paramètre* : -	heure de fermeture

On constate donc que le client peut se trouver dans les états "non connecté", "connecté", "départ prévu", terminal libre", …

9.2 Fonctionnalités demandées pour Applic_InOut

Le serveur Serveur_Terminaux attend aussi sur le port PORT_PORT les requêtes de l'application Java **Applic_InOut** dont le seul rôle est de permettre aux agents de la capitainerie de signaler l'arrivé ou le départ d'un ferry dans la zone des terminaux.

L'application cliente est très simple et utilise le protocole **SFNP** (**S**imple **F**erries **N**otification **P**rotocol). Ce ***protocole applicatif*** de conversation entre le serveur et ses clients, est basé **TCP**, est **sans états** et possède deux commandes :

protocole **SFNP**		
commande	**sémantique de la requête**	**réponse éventuelle**
SHIP-IN	notifie l'arrivée d'un ferry dans la rade du port *paramètres* : nom, l'heure (déterminée automatiquement)	le ferry est mis en attente dans le tableau ArrayWaitingFerries – il est sans affectation de terminal
SHIP-OUT	notifie le départ d'un ferry de la rade du port *paramètres* : nom, l'heure (déterminée automatiquement)	le ferry est retiré du tableau ArrayLeavingFerries

9.3 Quelques conseils méthodologiques pour le développement de SFVMP et de SFNP

1) Il faut tout d'abord choisir la manière d'implémenter les requêtes et les réponses des protocoles SFVMP/SFNP et plusieurs possibilités sont envisageables pour écrire les trames;
- uniquement par chaîne de caractères contenant des séparateurs pour isoler les différents paramètres;
- sous forme de structures, avec des champs suffisamment précis pour permettre au serveur d'identifier la requête et de la satisfaire si possible;
- un mélange des deux : une chaîne pour déterminer la requête, une structure pour les données de la requête;
- fragmenter en plusieurs trames chaînées dans le cas d'une liste à transmettre.

On veillera à utiliser des constantes (#DEFINE et fichiers *.h) et pas des nombres ou des caractères explicites.

2) On peut ensuite construire un squelette de serveur multithread et y intégrer les appels de base des primitives des sockets. Mais il faudra très vite (sous peine de réécritures qui feraient perdre du temps) remplacer ces appels par les appels correspondants d'une **bibliothèque** (**librairie**) facilitant la manipulation des instructions réseaux. Selon ses goûts, il s'agira
- soit d'une bibliothèque **C** de fonctions réseaux TCP/IP;

- soit d'une bibliothèque **C++** implémentant une *hiérarchie de classes* **C++** utiles pour la programmation réseau TCP/IP : par exemple, Socket, SocketClient et SocketServeur, éventuellement, si cela est estimé utile, flux réseaux d'entrée et de sortie NetworkStreamBase, ONetworkStream et INetworkStream).

Dans les deux cas, un mécanisme d'erreur robuste sera mis au point.

3) Quelques remarques s'imposent :
3.1) Une remarque pleine de bon sens ;-) : une fonction (ou une méthode) de bibliothèque ne présente d'intérêt que si elle est plus facile à utiliser que la (les) fonction(s) qu'elle remplace …
3.2) Une deuxième remarque pleine de bon sens ;-) : une fonction (ou une méthode) de bibliothèque ne présente d'intérêt que si elle est indépendante du cas particulier du projet "Inpres-Ferries" … Ainsi :

bien ☺	**pas bien ☹**
xxx **send**(void *,int size) /* couche basse : réutilisable dans une autre application */ xxx **AskForFerry**(...,...) /* couche haute : propre à cette application */	xxx send(Product *,int size) // et pas de xxx AskForFerry (...,...) /* une seule couche : la fonction send ne peut être utilisée dans une autre application */

3.3) Une troisième remarque pleine de bon sens ;-) : multiplier les couches exagérément est certainement le meilleur moyen de compliquer le développement et de ralentir l'exécution !

3.4) Enfin, en tenant compte de l'administration du serveur, il serait avisé de faire intervenir dans le code du serveur la notion d'état de celui-ci.

4) Il est impérieux de surveiller les écoutes, les connexions et les communications réseaux
- au moyen des commandes **ping** et surtout **netstat** (savoir expliquer les états TCP);
- en utilisant un **sniffer** comme Wireshark ou autre encore analysant le trafic réseau. Cette pratique sera demandée lors des évaluations.

5) Il serait aussi intéressant de prévoir un fichier de configuration lu par le serveur à son démarrage. A l'image des fichiers properties de Java, il s'agit d'un simple fichier texte dont chaque ligne comporte une propriété du serveur et la valeur de celle-ci :

serveur_terminaux.conf
Port_Terminal=70000 Port_Port=70001 Port_Admin=70009 sep-trame=$ fin-trame=# sep-csv=, nombre-terminaux-max=30 pwd-admin=maitrevcmw …

9.4 L'administration de Serveur_Terminaux

Un type particulier de client est attendu sur le port PORT_ADMIN : ce client peut donc administrer le serveur à distance du point de vue technique (arrêt, reprise, liste des machines du réseau). L'application **Admin_Terminaux** est programmée en Java et fonctionne sur un PC Windows ou sur une machine UNIX (machine Sunray); elle utilise le protocole **SFVMPA** (**SFVMP A**dministration). Ce ***protocole applicatif***, basé TCP, a pour commandes :

protocole **STOPA**		
commande	**sémantique de la requête**	**réponse éventuelle**
LOGINA	un administrateur autorisé se connecte *paramètres* : nom, password	oui ou non
LCLIENTS	List CLIENTS : liste des agents portuaires connectés avec le terminal qu'ils gèrent paramètres : -	pour chaque client : adresse IP, numéro de terminal
PAUSE	PAUSE Server : le serveur est mis en pause temporaire; les clients sont prévenus; les nouveaux clients sont refusés. paramètres : -	oui – serveur suspendu (si une commande est en cours, elle est arrêtée) ou erreur
STOP	STOP Server: on réalise un shutdown du serveur en prévenant les clients de l'imminence de l'arrêt *paramètres* : nombre de secondes avant arrêt	oui ou erreur

9.5 Quelques conseils méthodologiques pour le développement de SFVMPA

Le client administrateur se connecte sur **le port d'administration** PORT_ADMIN et se fait reconnaître comme administrateur en introduisant un mot de passe propre à l'administration. Pour que les clients connectés soient avisés, *de manière asynchrone*, d'un arrêt temporaire ou d'un shutdown du serveur (dans ce dernier cas, afin de se terminer en douceur), il convient de choisir une stratégie :

- ou chaque client normal s'est connecté sur un deuxième port (PORT_URGENCE) et attend des messages urgent par cette voie;
- ou le serveur se connecte sur chaque client en cas de nécessité (le client aura envoyé son port d'écoute lors de sa connexion sur PORT_TERMINAL).

Les travaux de l'évaluation 2

10. Serveur_Compagnie

10.1 Fonctionnalités demandées

Nous en venons à présent au serveur multithread **Serveur_Compagnie**. Nous allons ici implémenter tout d'abord la composante chargée de satisfaire les requêtes provenant de l'application **Applic_Check-In**, utilisée par les agents de la compagnie responsables du check-in des véhicules arrivant de la douane.

Pour rappel, ce serveur est programmé en Java et gère ses données au moyen d'une base de données MySQL BD_HAFIC pour laquelle des beans d'accès ont déjà été développés. Il est architecturé selon le modèle "threads à la demande" et attend, pour cette fonctionnalité, sur le port PORT_VOYAGEURS. Le protocole utilisé par le serveur et l'application cliente **Applic_CheckIn** est **PAREP** (**PA**ssengers **RE**gistering **P**rotocol) : il est basé TCP et comporte les commandes suivantes :

protocole **PAREP**		
commande	**sémantique de la requête**	**réponse éventuelle**
LOGIN	un agent de la compagnie se connecte au serveur *paramètres* : nom, password	oui ou non
VERIF_BOOKING	pour vérifier une réservation *paramètre* : code de la réservation, nombre de passagers (en plus du conducteur)	ACK avec enregistrement du check-in OU FAIL applicatif si réservation non trouvée
BUY_TICKET	pour acheter un ticket d'embarquement pour une traversée *paramètre* : nom conducteur, numéro d'immatriculation de la voiture, nombre de passagers (en plus du conducteur)	ACK avec le nom du ferry et l'heure de départ (c'est celui qui part le plus tôt qui est choisi si il y a encore de la place) ainsi qu'un numéro de client réutilisable dans la suite OU FAIL applicatif si aucune possibilité trouvée ce jour
CLOSE	fermeture de l'application *paramètre* : -	

A remarquer que la confection des horaires de traversées peut se faire manuellement (en ligne de commande, avec Toad for MySQL ou SQL Workbench ou autre) : la fonctionnalité de gestion de ceux-ci n'est pas demandée dans l'application considérée ici.

Une fois le véhicule accepté (soit à cause d'une réservation reconnue par le serveur, soit parce que le ticket vient d'être acheté), l'application cliente choisit un numéro de file d'attente (une file est complètement remplie avant de passer à la suivante) qui est transmis au voyageur. Il peut à présent se diriger vers cette ligne d'attente. Ce type d'information est mémorisé dans une table additionnelle "files-" de la base BD_HAFIC.

11. Serveur_Identités

11.1 Fonctionnalités demandées

Pour rappel, il s'agit donc d'un serveur multithread Windows ou Linux (**Serveur_Identités**), architecturé selon le modèle "pool de threads" et en attente sur le port PORT_VERIFY, et de son application clientes (**Applic_Identités**) destinée aux agents des frontières chargés d'opérer les vérifications d'identité et les déclarations de douanes; tous deux sont programmés en Java. Le serveur dispose d'une base de données BD_FRONTIER (de nature quelconque : Oracle, MySQL, SQl-server, …) contenant d'une part les informations sur les agents (table agents) et d'autre part les dossiers d'identité nationaux (table identites).

Vu le côté confidentiel des données personnelles traitées, les communications réseaux sont ici sécurisées (au moyen d'un système PKI propriétaire) d'une manière d'ailleurs variable en fonction des requêtes. Le protocole applicatif **VISEP** (**V**erifying **I**dentity **SE**cure **P**rotocol) est basé TCP et a pour commandes :

protocole **VISEP**		
commande	**sémantique de la requête**	**réponse éventuelle**
LOGIN	un agent veut entrer en session sur le serveur *procédure*: type HHTP, c'est-à-dire: a) le serveur envoie un nonce n1 b) le client construit un digest avec n1, son mot de passe et un deuxième nonce n2 (qu'il a généré) puis envoie au serveur son nom, n2 et le digest c) le serveur vérifie le digest	oui ou non
KEY_EXCHANGE	l'agent demande une clé de session au serveur *procédure*: type Kerberos simplifié, c'est-à-dire : a) le client s'assure que le serveur possède la clé publique associée à sa clé privée b) le serveur génère une clé de session et l'envoie au client en l'ayant au préalable crypté avec la clé publique de ce client c) le client décrypte avec sa clé privée	ACK si le certificat est valide, FAIL sinon
VERIFY-ID	pour vérifier l'identité d'un voyageur *paramètre* : nom, prénom et numéro de carte d'identité du voyageur, le tout crypté avec la clé de session + vérification de la signature de la réponse	VERIF_SUCCESSFULL ou VERIF_FAILED, réponse signée par le serveur
LOGOUT	fermeture de l'application *paramètre* : -	
TIME_OUT	le serveur envoie cette commande au client après un temps d'inactivité trop	

	long du client : le client est déconnecté	

11.2 Indication méthodologique pour le développement du serveur Serveur_Identités

Les mécanismes de cryptographie asymétrique utilisent des clés publiques et privées. On peut, dans un premier temps, se contenter de clés sérialisées. Mais, dans un deuxième temps, les clés devraient se trouver dans des **keystores**.

Les travaux de l'évaluation 3

12. Ferry-Chat

12.1 Fonctionnalités demandées

Les installations portuaires sont assez étendues et les différents intervenants (compagnie, douane, police maritime, …) ont considéré qu'un chat d'échanges d'informations était un facteur de sécurité supplémentaire : on peut ainsi s'informer et informer tous les autres intervenants, signaler un événement ou un élément suspect, etc. C'est le "Ferry-Chat".

Un agent est admis dans ce chat sur base de ses informations d'identification sur le serveur dont il dépend, donc son nom et mot de passe, complété d'une phrase connue de lui seul. Au moyen d'une application **Applic_Chat-Ferry** écrite en Java, il se joint au groupe en se connectant sur une adresse de classe D précise et un port PORT_CHAT précis qu'il aura obtenu en s'adressant à un serveur **Serveur_Chat** qui vérifie la qualité d'agent (des douanes ou de la compagnie) dans la base donnée BD_CHAT_USERS contenant les informations nécessaires à l'identification.

Les clients utilisent ensuite le protocole **FECOP** (**FE**rries **C**ommunity c**O**nversation **P**rotocol). Basé principalement UDP, ce ***protocole applicatif*** utilise donc au préalable une connexion TCP à **Serveur_Chat** écrit en Java sous Windows; celui-ci n'attend sur le port PORT_GROUP qu'une seule commande permettant de fournir le nom et le mot de passe de l'agent :

protocole **FECOP (partie TCP)**		
commande	**sémantique**	**réponse éventuelle**
LOGIN_GROUP	un agent veut se joindre au groupe *paramètres* : nom, password	oui + envoi de l'adresse et du port PORT_CHAT à utiliser ce jour; ou non

En cas de succès, le client pourra alors réellement participer au chat :

protocole **FECOP (partie UDP)**		
commande	**sémantique**	**réponse éventuelle**
POST_QUESTION	Pose question : un agent pose une question et espère des réponses *paramètres* : la question sous forme de chaîne de caractères (un tag d'identification de question à utiliser dans la réponse est généré)	une réponse à la question précédée du tag
ANSWER_QUESTION	Réponse à une question *paramètres* : le tag d'identification de question et le texte de la réponse	une réponse à la question précédée du tag
POST_EVENT	Un agent signale un fait, donne une information mais n'attend pas de réponse (un deuxième type de tag est cependant généré pour identifier l'événement)	-

On élargira ensuite le chat en ajoutant une application **Applic_Chat-Ferry** écrite en C/C++.

12.2 Indication méthodologique pour le développement de FECOP

Il s'agira évidemment de mettre d'abord en place le Serveur_Chat, qui est un serveur Java qui tourne sur un PC et qui se révèle, du point de vue TCP, des plus simples puisque monothread et mono commande. A noter tout de même qu'il appartient aussi au groupe multicast UDP. Le multicast sera ensuite implémenté, tout d'abord dans le sous-réseau local.

13. Applic_Reservations

13.1 Fonctionnalités demandées

Cette application **Web** permet au public d'acheter par Internet des places sur un ferry; elle est donc cliente du serveur Serveur_Compagnie, lequel utilise pour cela le PORT_WEB_VOYAGEURS.

a) Les clients doivent tout d'abord se connecter au serveur soit en utilisant leur numéro de client obtenu lors d'une autre réservation (ce qui leur donnera droit à une réduction de 3% sur le montant de leurs réservations), soit en en obtenant un. Ceci se passe en utilisant les applets développées au point 7, donc dans un modèle applets-servlets, avec la nuance que le numéro de client est généré côté serveur et non pas côté client. La servlet communique avec Serveur_Compagnie pour obtenir un numéro de client valide (en utilisant le protocole PAREP complété des commandes nécessaires ou un autre protocole à définir).

b) Passée la phase d'identification **absolument indispensable** (rien n'est donc possible tant que le client n'a pas effectué cette démarche), l'utilisation du site suit le canevas suivant avec une page HTML proposant les trois services suivants

1. Liste des promotions last-minute à ne rater sous aucun prétexte
2. Achats selon le principe du caddie virtuel
3. Fin de session avec encodage des données complémentaires du client

Les clients effectuent des réservations selon la technique du **"caddie virtuel"** dans un interface graphique qui illustre très schématiquement les traversées disponibles : différentes destinations et différents horaires selon le jour choisi pour la traversée. La démarche consiste à se promener dans les pages catalogues du site et à choisir au fur et à mesure des éléments. On obtient à la fin le montant total des réservations et on réalise alors les opérations de confirmation avec introduction des informations

- de réservation : nombre de passagers (en plus du conducteur);
- de contact : adresse e-mail, adresse postale, numéro de téléphone;
- de paiement : numéro de carte de crédit, date de validité

Techniquement, ce site est construit

- selon le modèle applet-servlet par tunnel http pour obtenir la liste des promotions;
- selon le modèle MVC pour le caddie virtuel proprement dit, donc avec des java Beans, des Java Server Pages et des servlets; on utilise dans ce contexte un ***protocole applicatif*** basé sur HTTP/TCP et nommé **EASHOP** (**E**Asy **SHO**pping **P**rotocol) – qu'il convient de définir …
- selon le modèle applets-servlets ou le modèle HTML/JavaScript-servlet pour la fin de session.

13.2 Quelques conseils méthodologiques pour le développement de Applic_Reservations

Pour rappel, on veillera impérativement à une "gestion de stocks" cohérente en ce qui concerne les réservations :

- évidemment, le nombre de véhicules sur une traversée n'est pas illimité (pas question de "commander" : les places sont disponibles ou pas);
- si une (ou plusieurs) place(s) est(sont) réservée(s) et mise(s) dans le caddie, elle(s) doit(doivent) rester acquise(s) si le client conclut dans un délai raisonnable (pas question qu'un autre client la(les) prenne pendant la suite de la navigation Web);
- si un client abandonne sa navigation Web, ce qu'il avait réservé doit redevenir disponible dans un délai d'une demi-heure (par exemple).

En pratique, on testera l'application Web sur PC sous Windows avec un Tomcat distant sur PC.

14. Le bonus : la synchronisation des serveurs

En pratique, les différentes parties du projet ne peuvent s'ignorer :

- quand le Serveur_Identites valide les identités, il devrait en aviser le Serveur_Compagnie;
- quand le Serveur_Compagnie a terminé le check-in d'un véhicule, il devrait en aviser Serveur_Terminaux;
- dès qu'un agent des frontières ou de la compagnie se connecte sur le serveur dont il dépend, il est dans la foulée connecté au Serveur_Chat sur base de ses informations d'identification.

En guise de conclusion

Les principaux développements évoqués dans ce dossier de laboratoire ont été définis avec le plus de précision possible. Certains points ont cependant été laissés suffisamment vagues pour vous permettre d'envisager différents scénarios (scénarii ;-) ?) pour satisfaire à la demande. De plus, certaines pistes sont à peine entr'ouvertes - à vous de voir, si vous en avez le temps, ce que vous pouvez ajouter à vos développements pour leur apporter un plus valorisant. Comme toujours, **prudence** : l'avis de l'un des professeurs concernés est sans doute (un peu-beaucoup-très fort) utile ;-).

Soyez créatifs et imaginatifs … mais restez rationnels et raisonnables …

s: CV,CC,MM, JMW

Infos de dernière minute ? Voir l'Ecole virtuelle