Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Institut TIC – Filière Télécommunications

Orientation Réseaux et Sécurité

Projet de semestre de printemps

2015

NetGame on WebSockets

David Rossier

Mandaté par : Rudolf Scheurer

Supervisé par : Rudolf Scheurer

Jean-Frédérique Wagen

Hes·so FRIBOURG FREIBURG
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

Historique du document

Version	Auteur	Description des modifications	Date
0.1	David Rossier	Création du document	20.04.2015
0.2	David Rossier	Ajout des parties existantes	25.04.2015
0.3	David Rossier	Complément d'analyse : NetGame	04.05.2015
0.4	David Rossier	Complément d'analyse : WebSocket	06.05.2015
0.5	David Rossier	Documentation Conception	09.05.2015
0.6	David Rossier	Relecture	10.05.2015
0.7	David Rossier	Spécifications Monitor	11.05.2015
0.8	David Rossier	Tests	11.05.2015
0.9	David Rossier	Implémentation	12.05.2015
1.0	David Rossier	Relecture et finalisation	12.05.2015

Résumé

De nos jours, les technologies du Web sont en constantes évolutions. De nouvelles technologies apparaissent, pour tenter de combler le manque de fonctionnalités des applications Web actuelles. C'est le cas des nouvelles technologies, apportées notamment par la norme HTML5. De plus, une nouvelle forme de communication est apparue récemment : Les WebSockets.

Cette nouvelle technologie permet d'ouvrir une connexion TCP permanente entre le navigateur web et le serveur web. Cela offre un grand avantage au monde de la communication Web. En effet, jusqu'à présent, la communication client – serveur fonctionnait ainsi : Le navigateur envoie une requête vers le serveur, suivie d'une réponse, que ce dernier traitait et affichait à l'écran de l'utilisateur.

Si le serveur souhaitait envoyer régulièrement des mises à jour, c'était donc au navigateur d'effectuer des requêtes régulières pour vérifier si les informations qu'il avait étaient à jour.

Avec la technologie WebSocket, lorsque le serveur web souhaite envoyer une mise à jour des informations, il est capable d'envoyer directement un message au client par l'intermédiaire du WebSocket, sans action préalable du client.

Ce projet a pour but de mettre à jour l'application « NetGame », utilisée lors des cours de Téléinformatique pour apprendre le fonctionnement des couches 1 à 3 du modèle OSI, afin de modifier l'interface de communication client/serveur actuelle, pour y implémenter une interface de communication utilisant la technologie WebSocket. Cela offrira une meilleure réactivité à l'application, et offrira donc aux joueurs une meilleure expérience de jeu.

Durant ce projet, la technologie WebSocket sera étudiée, ainsi que le fonctionnement de l'application NetGame actuelle. Ensuite, les spécifications et la conception d'une application utilisant la technologie WebSocket comme méthode de communication seront établis.

Finalement l'implémentation d'une maquette fonctionnelle sera réalisée et testée afin de vérifier le bon fonctionnement tant de la communication que du jeu NetGame.

Abstract

Nowadays, Web technologies are in constant evolution. New technologies appear, to try to fill functionality gaps of existing Web applications. This is the case of new technologies, especially made by the HTML5 standard. In addition, a new form of communication has emerged recently: The WebSockets.

This new technology opens a permanent TCP connection between the browser and the web server. This offers a great advantage to the world of Web communication. Indeed, until now, communication was limited to a request from the browser to the server, followed by a response. The response was treated by the browser and posted to the user's screen. With WebSocket, when the web server wishes to send an update of the information, it is able to send a message directly to the client via WebSocket, without any client's action.

This project aims to update the « Netgame » application used during Teleinformatique lessons to learn the operation of layers 1 to 3 of the OSI model, in order to change the AJAX client / server communication interface to implement a communication interface using the WebSocket technology. This will make the application more responsive, and therefore offer players a better gaming experience.

During this project, the WebSocket technology and the functionnalities of the current Netgame application will be studied. Then the specifications and design of an application using the WebSocket technology as a communication method will be established.

Finally the implementation of a functional application will be performed and tested to verify proper operation of both the communication and the Netgame.

Table des matières

1. Introduction

NetGame est un jeu interactif utilisé par les étudiants lors des cours de Téléinformatique afin de leur apprendre de manière didactique les couches réseaux 1 à 3 du modèle OSI.

Depuis les débuts du Web, et encore aujourd'hui, l'architecture client/serveur Web a fonctionné selon le schéma de la figure 1, c'est-à-dire que le serveur répondait à une requête d'un client. Pour garder une page avec des informations à jour, le client doit donc envoyer régulièrement des requêtes.

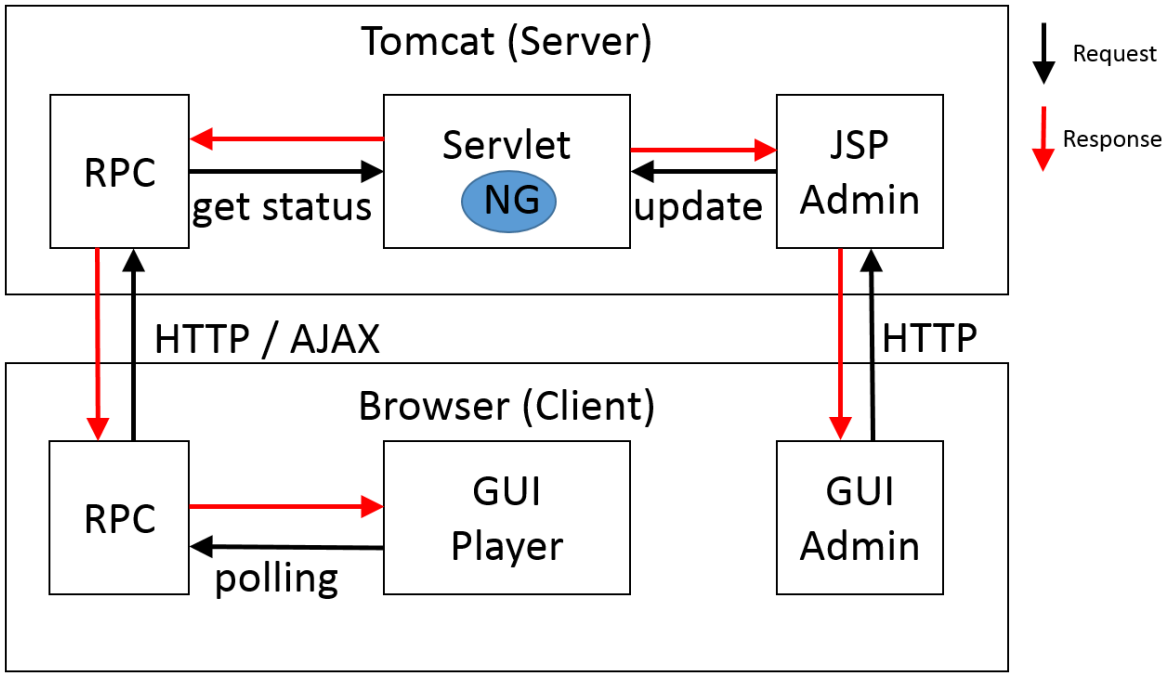


Figure 1 : NetGame : Polling

Dans l'application NetGame (NG) actuelle, un utilisateur a pour mission de remplacer le rôle d'une des couches 1 à 3 du modèle OSI. Pour cela, l'utilisateur doit compléter les entêtes de la couche concernée. Lorsqu'un nouveau paquet doit être traité, l'utilisateur voit le paquet apparaitre sur son écran. La version actuelle fonctionne par polling, c'est-à-dire que le navigateur du client envoie régulièrement des requêtes par une Remote Procedure Call au Servlet. L'utilisateur pourra alors traiter le paquet apparu sur son écran à l'aide d'un formulaire HTML.

L'application actuelle fonctionne en mode client/serveur standard, mais elle utilise des technologies Web anciennes, et certaines des nouvelles technologies apportées par HTML5 offrent des améliorations pour la communication d'un serveur vers un client.

Ce projet a pour but d'améliorer l'application existante afin d'utiliser la technologie WebSocket, qui permet d'ouvrir une connexion TCP permanente entre le client et le serveur, offrant ainsi la possibilité d'envoyer des messages bidirectionnels.

Nous verrons tout d'abord le cahier des charges du projet, puis nous analyserons l'état de l'art, à savoir le fonctionnement actuel de l'application NetGame, ainsi que le fonctionnement de la technologie WebSocket en Java.

Nous verrons ensuite les étapes nécessaires à la conception de cette application, ainsi que certaines particularités de l'implémentation.

Finalement, nous exécuterons un plan de tests, établi précédemment, afin de vérifier le bon fonctionnement de l'application, avant d'établir les conclusions de ce projet.

2. Cahier des charges

Ce chapitre a pour but de fixer le cadre et les objectifs du projet sur la base de la donnée reçue. Il contient les objectifs, le contexte du fonctionnement souhaité, ainsi qu'une planification détaillée du projet.

2.1. Objectifs

Ce projet a pour but dans un premier temps, d'étudier la technologie des Websockets en Java, puis d'adapter le fonctionnement client/serveur de l'application NetGame afin que les messages soient envoyés par le serveur (push), et non récupérés régulièrement par le client (polling). Finalement, une maquette de fonctionnement réelle sera développée, à l'aide de Framework existants tels que JQuery.

Ce projet sera divisé en plusieurs étapes

0. Comprendre et étudier les WebSockets
1. Créer une application simple utilisant les WebSockets
2. Adapter les WebSockets pour fonctionner avec l'interface fournie par le Servlet NetGame
3. Créer une maquette d'utilisation de l'application client avec les WebSockets

2.2. Architecture NetGame on WebSockets

La nouvelle architecture fonctionnera comme dans le schéma de la figure 2. Le joueur connecté au jeu recevra automatiquement les nouveaux messages du serveur par l'intermédiaire des WebSockets. Le joueur pourra alors les traiter à l'aide d'un formulaire HTML. Les numéros illustrent l'objectif lié à chacune des parties de l'application.

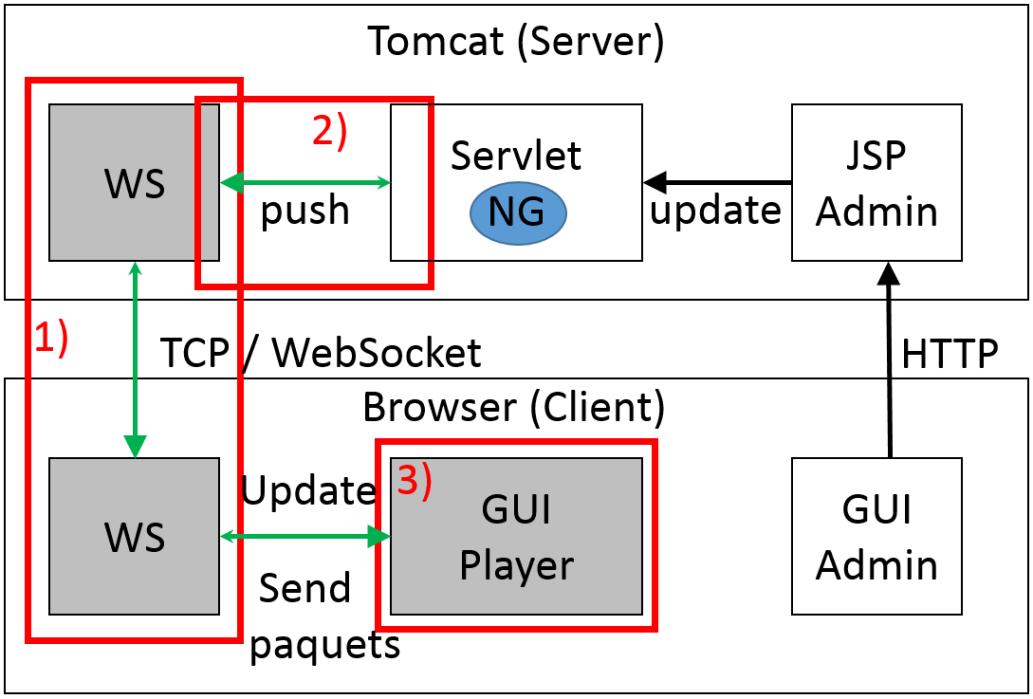


Figure 2 : NetGame : WebSockets

2.3. Planification

Afin de mener à bien le projet et d'avoir une vision globale du temps à disposition, une planification détaillée a été réalisée. Une marge de sécurité a été prévue pour certains jalons.

La planification des tâches à effectuer est disponible dans l'annexe 1.

3. Analyse de l'état de l'art

Ce chapitre a pour but d'analyser tout d'abord l'ancienne application NetGame, et en particulier les échanges entre le client et le serveur. Nous analyserons ensuite le fonctionnement de la technologie WebSocket, tant du côté client (Javascript) que du côté serveur (Java API).

3.1. Analyse de l'application NetGame

Afin de bien comprendre le fonctionnement du jeu NetGame, il est important de rappeler les différentes couches du modèle OSI. Le jeu NetGame met en scène une application Web qui permet de simuler les couches physique (PHL), liaison de données (DLL) et réseau (NL).

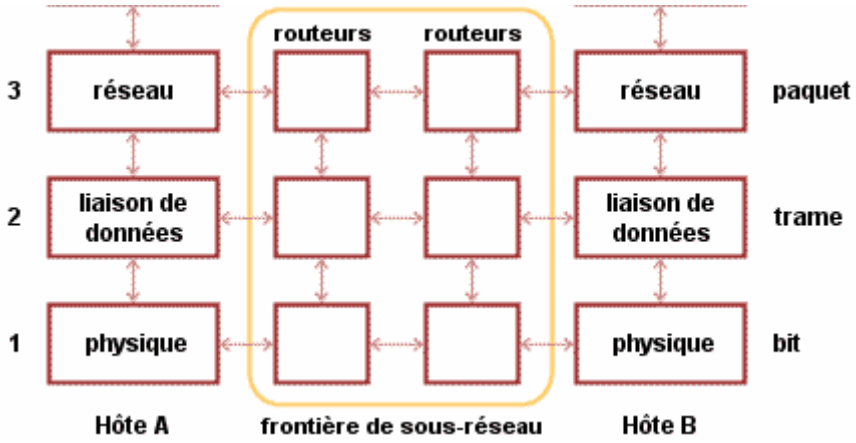


Figure 3 : Le modèle OSI [1]

Le jeu NetGame simule un réseau de neuf nœuds (nodes) du réseau. Un nœud correspond à un hôte ou un routeur du réseau. Chaque noeud implémente les trois couches (layer) inférieures d'une manière légèrement différente des systèmes informatiques traditionnels. Pour démarrer le jeu, un joueur doit choisir un nœud et une couche du réseau, ainsi qu'un pseudo (alias) qui le représentera. Le joueur devra alors traiter les messages reçus par les couches inférieures et supérieures, pour les transmettre à la prochaine couche.

Le jeu NetGame offre plusieurs interfaces Web :

- Jeu : Interface sur laquelle les joueurs se connectent pour jouer.
- Administration : Interface sur laquelle l'administrateur du jeu peut se connecter à l'aide d'un mot de passe pour gérer le jeu.
- Monitor : Interface sur laquelle on peut voir l'état actuel du réseau, la connectivité entre les différents nœuds, et le nombre de message que chacune des couches du réseau doit traiter.

3.1.1. Interface du jeu

Pour chaque couche, il y a des messages arrivant depuis les couches supérieures, et des messages arrivant depuis les couches inférieures. Les points suivants déterminent, pour chaque couche, les actions à effectuer par les joueurs pour valider un message.

3.1.1.1. Physical Layer PHL (1)

La couche physique est la couche la plus basse du modèle OSI. Elle a pour rôle de décoder des bits reçus sur un support physique.

Message montant
Les messages arrivant du bas représentent des messages arrivant sur le lien physique du nœud. Le joueur reçoit le contenu du message en un seul morceau séparé par des « / » pour chacun des champs du message. À l'aide de listes déroulantes, il peut remplir le message avant de l'envoyer à la couche supérieure
Message descendant
Les messages arrivant du haut représentent des messages transmis par la couche DLL. Le joueur reçoit le contenu du message rempli entièrement et doit remplir le champ « PHY ». Le contenu du message est écrit sans espaces dans un champ input, et il doit séparer les différents champs à l'aide de « / » avant de l'envoyer sur le lien physique (vers le bas).

Tableau 1 : Traitement couche Physique

3.1.1.2. Data Link Layer DLL (2)

La couche liaison de données est la seconde couche du modèle OSI. Dans le cas particulier du jeu NetGame, il y a trois types de messages : DAT (Data), ACK (Acknowledge) et NAK (Not Acknowledge). De plus, un numéro de séquence identifie chaque message.

Message montant
Les messages arrivant du bas sont transmis par la couche PHL. S'il s'agit d'un message de type DAT : Le joueur vérifie le champ FCS. • FCS correct : il doit envoyer un message de type ACK, en reprenant le numéro de séquence du message reçu, et envoyer le paquet reçu vers la couche supérieure. • FCS incorrect : il doit envoyer un message de type NAK, en reprenant le numéro de séquence du message reçu, et supprimer le paquet reçu. S'il s'agit d'un message de type ACK : Cela signifie que le dernier message envoyé a bien été reçu. Le nœud peut alors supprimer le message confirmé de son buffer d'envoi. Le joueur doit sélectionner l'option « Supprimer le message confirmé » et supprimer le message ACK. S'il s'agit d'un message de type NAK : Cela signifie que le dernier message envoyé a eu une erreur de transmission. Il doit être retransmis. Le joueur doit donc sélectionner l'option « Retransmission » et supprimer le message NAK.

Message descendant
Les messages arrivant de la couche supérieure sont transmis par la couche NL. Le joueur remplit les champs DLL : - MAC src : Adresse MAC du nœud courant - MAC dst : Adresse MAC du nœud de destination (Next Node) - Type : DAT - Numéro de séquence : Il doit lui-même gérer un historique des numéros de séquence afin qu'il soit supérieur au précédent. - FCS : Le FCS dépend du nombre de caractères contenus dans le champ Message (Application Layer). Il ne prend pas en compte les espaces. Il envoie ensuite le message vers la couche inférieure (PHL) en spécifiant que le nœud doit conserver une copie du message dans son buffer d'envoi.

Tableau 2 : Traitement couche liaison de données

3.1.1.3. Network Layer NL (3)

La couche réseau est la troisième couche du modèle OSI. Cette dernière permet de déterminer le chemin le plus court pour atteindre la destination.

Message montant
Les messages arrivant du bas sont transmis par la couche DLL. Le joueur doit vérifier la valeur du champ « Final Node ». Si ce dernier est le même que l'identifiant de son propre nœud, il doit envoyer le message à la couche supérieure (Application Layer). Si le champ « Final Node » correspond à un autre nœud, il doit effectuer le rôle d'un routeur : Il consulte la page monitor (accessible depuis l'interface Web) pour connaitre l'état du réseau. Il choisit l'identifiant du routeur représentant le prochain nœud à qui envoyer le message et transmet le message vers la couche inférieure (DLL).
Message descendant
Les messages arrivant de la couche supérieure (Application Layer) sont nouvellement créés, soit par l'administrateur du jeu, soit par les joueurs souhaitant envoyer eux-mêmes des nouveaux messages. Le joueur consulte la page monitor (accessible depuis l'interface Web) pour connaitre l'état du réseau. Il choisit l'identifiant du routeur représentant le prochain nœud à qui envoyer le message et transmet le message vers la couche inférieure (DLL).

Tableau 3 : Traitement couche réseau

3.1.2. Interface d'administration

L'interface d'administration actuelle restera inchangée. En effet, l'implémentation des WebSockets dans cette partie n'offre que peu d'avantages, la plupart des fonctions étant des actions client/serveur. Il serait envisageable d'ajouter la mise à jour automatique des messages sur cette interface, mais cela ne fait pas partie du cahier des charges.

Néanmoins, il est important de noter les fonctionnalités qui sont à disposition d'un administrateur. Ces dernières devront être gérées par le client en temps réel.

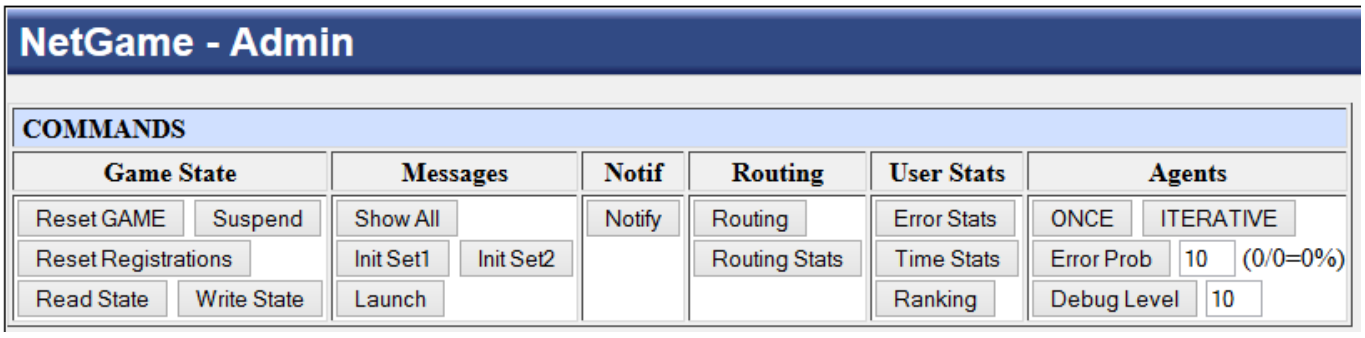


Figure 4 : Interface d'administration

C'est principalement l'administrateur qui a pour rôle de générer les messages à traiter du jeu. Ainsi, les boutons « Init Set » permettent de générer deux messages sur la couche Network Layer de chacun des nœuds du réseau. De plus, la fonction « Launch » permet de générer un message d'un nœud spécifique à un autre. Les nouveaux messages générés devront donc être envoyés aux joueurs immédiatement après leur création.

L'administrateur doit pouvoir envoyer un message de notification. Il peut choisir entre envoyer à tous les joueurs, à un seul nœud, à une seule couche ou à un seul joueur.

L'administrateur peut remettre le jeu à zéro, supprimer tous les enregistrements ou encore suspendre le jeu pour le mettre en pause. Ces trois actions devront notifier les clients.

Afin d'aider les joueurs surchargés, l'administrateur peut également traiter lui-même les messages.

En cas de traitement d'un message par l'administrateur, le joueur concerné devra être informé que le message n'apparait plus dans sa liste des messages à traiter.

3.1.3. Interface monitor

La page monitor fonctionne actuellement de manière autonome, c'est-à-dire qu'elle a directement accès aux informations fournies par la classe NetGame.

Il serait également possible de modifier cette page pour qu'elle se rafraichisse uniquement lorsqu'il y a un changement dans l'état de l'application, et non toutes les 10 secondes à l'aide de la balise HTML `<meta>` comme c'est le cas actuellement.

Afin de réaliser cela, il serait nécessaire de créer un nouvelle classe WebSocket, sur laquelle la page monitor se connecte. Ce dernier fonctionnerait dans tous les cas en broadcast sur tous les clients connectés. En effet, elle devrait réagir de la même manière pour tous les joueurs.

Les informations à envoyer pour garder la page à jour sont les suivantes :

- Nombre de message pour chaque couche (*QueuedMessages* & *BufferedMessages*)
- Noms des joueurs connectés (Pseudos)
- Connectivité complète (entre chacun des nœuds)
- Etat du jeu (*actif/suspendu*)

Ainsi, lorsqu'un événement survient sur l'un des éléments cités ci-dessus, un message devra être transmis à travers le WebSocket *WSMonitor*. Ce dernier enverra les informations dans tous les cas à tous les clients connectés. En effet, la page monitor est la même pour tout le monde et ne nécessite aucune authentification. L'interface monitor actuelle est affichée dans la figure 5.

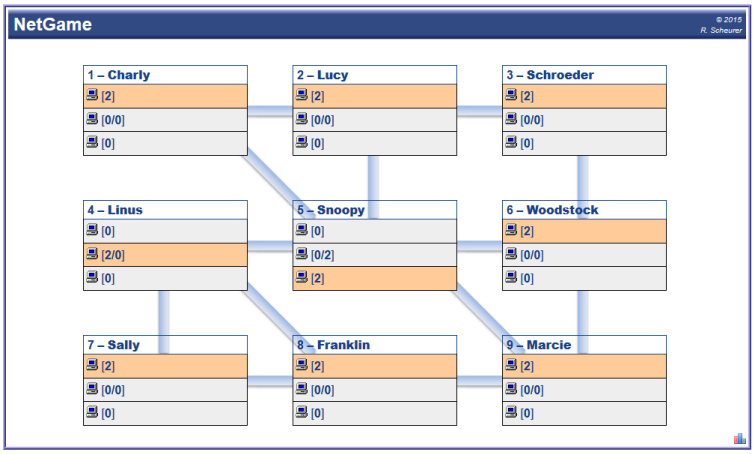


Figure 5 : Interface Monitor

3.2. Interactions client-server

L'application NetGame actuelle a été générée à l'aide du Framework GWT (Google Web Toolkit) qui permet de développer une application cliente uniquement en Java. Lors de la compilation, GWT génère les fichiers (JSP, CSS, Javascript) nécessaires au fonctionnement de l'application.

Elle fonctionne actuellement sous un modèle client-serveur standard avec des requêtes AJAX (Asynchronous Javascript and XML) à l'aide d'une interface RPC.

Comme le décrit la figure 6, AJAX est une technologie Web qui permet, à partir d'une page HTML reçue (flèche verte), d'effectuer une nouvelle requête vers le serveur Web (flèche rouge). Cette dernière peut permettre soit d'effectuer une action sur le serveur, soit de récupérer des informations ou les deux.

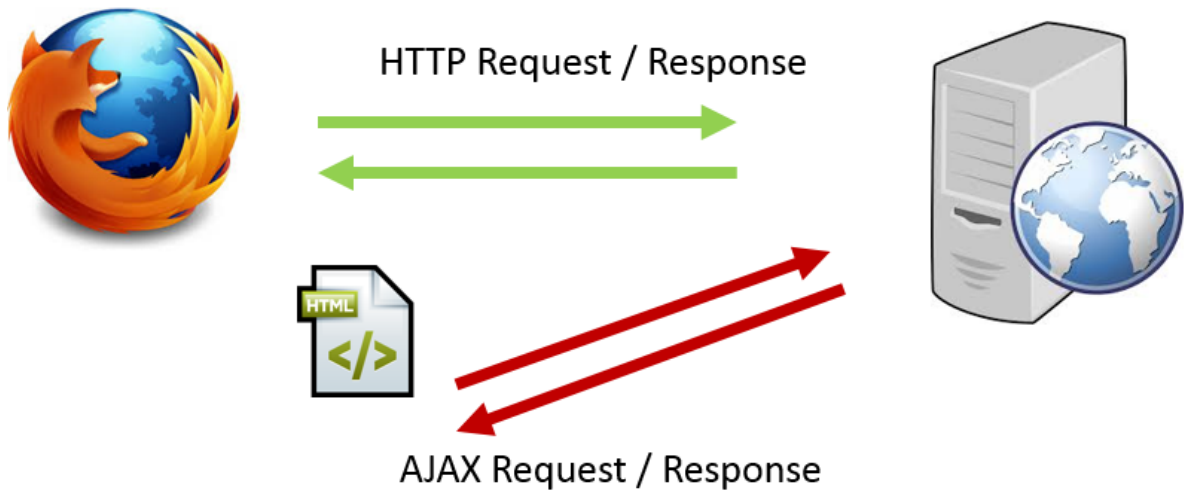


Figure 6 : Fonctionnement AJAX

Le chapitre ci-dessous résume les fonctionalités offertes par la RPC implémentée dans l'ancienne application et leurs utilités.

3.2.1. Classe NetGameServiceImpl.java

La classe NetGameServiceImpl est l'interface RPC sur laquelle les clients se connectent afin d'envoyer des modifications à la classe NetGame.

Fonction	Description
`init()`	Fonction d'initialisation : Vérifie si NetGame existe, si oui, s'y connecte, sinon génère quelques messages de test.
`getNodeAndLayers()`	Récupère les noms des nœuds et couches du jeu en cours sous forme de tableau String[][]
`doRegistration()`	Enregistre un nouvel utilisateur à partir du node ID, layer ID et un alias (pseudo entré par l'utilisateur)
`clearRegistration()`	Supprime l'enregistrement d'un utilisateur
`getNodeState()`	Récupère l'état actuel du jeu pour les utilisateurs des trois couches
`lockMessage()`	Cette fonction est utilisée pour éviter qu'un message en cours de traitement par un utilisateur soit traitée par l'administrateur par le biais de l'interface d'administration
`unlockMessage()`	Débloque tous les messages d'une couche.
`launchMessage()`	Le client envoie un message à une destintation donnée
`processMessage()`	Le client valide un message envoyé. Ce dernier est considéré comme « traité » par le serveur.
`getNetGameMsgWrapper()`	Copie les messages de NetGame vers le NetGameMsgWrapper Voir le chapitre 3.2.1 ci-dessous.

Tableau 4 : Fonctions NetGameServiceImpl

Toutes les fonctions présentes dans ce tableau devront être modifiées pour utiliser WebSocket.

Les points suivants expliquent certaines particularités des fonctions ci-dessus.

3.2.1.1. NodeState

La fonction getNodeState retourne l'état actuel du jeu selon la couche du joueur. En effet, chaque couche reçoit plus ou moins d'informations dont elle a besoin pour déterminer les choix du joueur, et les actions qu'il doit effectuer.

Un NodeState peut contenir quatre types d'informations différentes, selon sa couche. Le tableau ci-dessous détermine quelles informations chaque couche reçoit.

	MessageQueue	BufferQueue	Connectivity	Notification
Physical Layer				
Data Link Layer				
Network Layer				

Tableau 5 : Informations envoyées dans un NodeState

MessageQueue : La liste des messages que l'utilisateur doit traiter.

BufferQueue : La liste des messages que la couche DLL du nœud a enregistré dans son buffer d'envoi, en attente d'un acquittement.

Connectivity : L'état des liens (routing) depuis le nœud courant vers les autres nœuds. Cela représente les voisins connectés par un câble physique.

Notification : Les messages de notifications, envoyés par l'administrateur du jeu.

3.2.1.2. Message NetGame

Un message (NetGameMsg) contient de nombreuses informations sur un message, dont une grande partie ne sont pas utilisées par la partie cliente (navigateur) de l'application NetGame. Le NetGameMsgWrapper est une version réduite du NetGameMsg qui contient les éléments essentiels au fonctionnement de l'application. Le tableau ci-dessous recense les éléments du NetGameMsgWrapper.

Attribut	Description
`ID`	Identifiant du message dans NetGame
`msgFields`	Les champs du message NetGame (voir Tableau 7, ci-dessous)
`ackFields`	Les champs d'un acquittement
`msgString`	Le message String (voir Tableau 7, ci-dessous)
`rawMsg`	Contient la valeur de tous les autres champs séparés par des « / »
`fromLayer`	La couche source du message
`actNode`	Le nœud actuel, utilisé pour déterminer le point de fin du message
`actLayer`	La couche actuelle, utilisée pour déterminer la direction du message
`toNode`	Le nœud de destination, utilisé pour déterminer le point de fin du message
`toLayer`	La couche de destination, utilisée pour déterminer la direction du message.

Tableau 6 : Contenu d'un message NetGameMsgWrapper

3.2.1.3. Champs msgFields

Les champs msgFields est un tableau d'entier. Il contient dans l'ordre les éléments suivants :

`[MAC_SRC, MAC_DST, DATA_TYPE, SEQ_NUM, IP_NEXT_NODE, IP_FINAL_NODE, NODE_SRC, NODE_DST, MESSAGE, FCS]`

Champ	Description
MAC_SRC	Identifiant de l'adresse MAC du nœud source du message (DLL)
MAC_DST	Identifiant de l'adresse MAC du nœud de destination du message (DLL)
DATA_TYPE	Type de donnée transmis (DATA, ACK, NAK)
SEQ_NUM	Numéro de séquence permettant d'identifier le message avec les acquittements
IP_NEXT_NODE	Identifiant du prochain nœud de destination
IP_FINAL_NODE	Identifiant du dernier nœud de destination
NODE_SRC	Identifiant du nœud source du message
NODE_DST	Identifiant du nœud de destination du message
MESSAGE	Le contenu du message est contenu dans l'attribut msgString du NetGameMsgWrapper, car il est de type String, incompatible avec un tableau d'int. La valeur est donc obligatoirement `-1`.
FCS	Nombre de caractères (non-espace) du message msgString transmis.

Tableau 7 : Contenu de l'attribut msgFields

3.3. WebSocket

Un WebSocket est une implémentation d'envoi de messages asynchrone et bidirectionnelle fonctionnant sur une seule connexion TCP. Les WebSockets ne sont pas des connexions http, mais ils utilisent le protocole http pour créer une connexion WebSocket.

Les WebSockets ont initialement été proposés dans la spécification HTML5, qui souhaitait proposer une simplification des communications dans le développement des nouvelles applications Web interactives, mais ils ont été finalement déplacés dans un nouveau document de standardisation. Les WebSockets sont donc décrits par la RFC 6455[1] (communication client/serveur) et par l'API Javascript WebSocket[2] définie par le W3C qui définit les spécifications que les navigateurs Web devraient implémenter.

Ce chapitre présentera le fonctionnement des WebSocket, les intérêts et avantages dans l'application NetGame, ainsi qu'une utilisation basique des éléments fondamentaux nécessaires à l'utilisation d'un WebSocket utilisant le langage Java sur un serveur Tomcat à partir d'un navigateur Web implémentant les WebSockets en Javascript.

[1] RFC 6455 : http://tools.ietf.org/html/rfc6455

[2] API Javascript : http://www.w3.org/TR/websockets/

3.3.1. Fonctionnement des WebSockets (http)

Le protocole WebSocket est une « upgrade » du protocole http. La requête utilisée par le client pour l'ouverture d'un channel WebSocket inclue, en plus des entêtes traditionnels d'une requête HTTP, les entêtes Upgrade, Connection, Sec-WebSocket-Key, Sec-WebSocketProtocol et Sec-WebSocket-Version.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Si le serveur accepte la connexion, il retourne une réponse 101 Switching Protocols indiquant qu'il accepte l'ouverture du channel WebSocket.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

Dès lors, les messages envoyés entre le clients et le serveur au travers du channel WebSocket se font directement au dessus de TCP dans le protocole WebSocket (le port reste le même que pour la requête http)

Le protocole WebSocket contient les entêtes suivants :

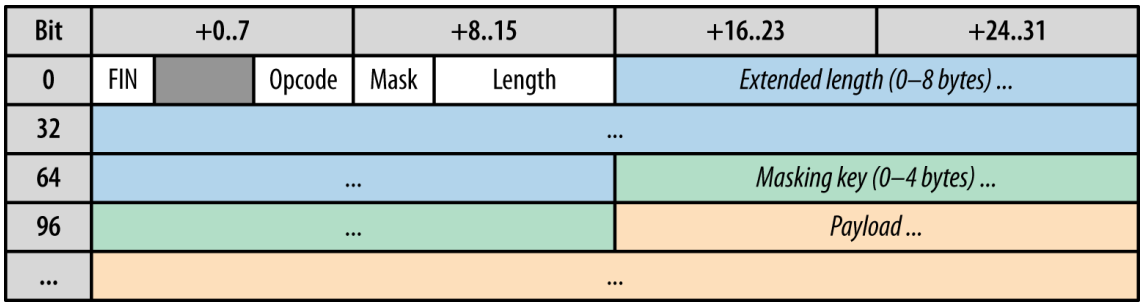


Figure 7 : Entête WebSocket [2]

Le bit FIN indique qu'il s'agit du dernier fragment du message (cela ne signifie pas que la connexion sera fermée)

Les trois bits suivants sont utilisés par certaines extensions. Ils doivent être généralement à zéro.

Les 4 bits Opcode déterminent l'interprétation du payload (suite d'une trame, texte, binaire, fermeture de connexion, ping, pong).

Le bit Mask détermine si le payload est masqué, et si la Masking key doit être présente.

Les 7 bits Length déterminent la longueur du payload. Les bits suivants (bleu) permettent d'étendre la taille jusqu'à 7+64 bytes.

La masking key permet principalement d'empêcher les attaques de type cache poisoning. Ce point est détaillé plus précisément dans la section 5.3 de la RFC 6455[3]. Elle doit être générée aléatoirement avec une grande entropie par le navigateur web.

Le payload détermine les données transmises par le WebSocket. Ce dernier est encodé selon la spécification de la RFC 5234[4].

Tomcat 8 implémente les spécifications de la RFC 6455.

3.3.2. Intérêts dans l'application NetGame

Dans l'application NetGame, l'avantage de la technologie WebSocket est qu'elle permet d'envoyer des messages push aux clients qui se connectent.

La méthode push est nécessaire particulièrement lorsque ce n'est pas le joueur qui génère l'action qui résulte à un résultat. C'est le cas dans l'application NetGame, en particulier pour l'arrivée de nouveaux messages, la source étant soit l'administrateur, soit une action effectuée par un autre joueur.

La méthode push est également utilisée lors d'autres opérations de l'administrateur, notamment l'envoi des notifications, la mise à jour de l'interconnexion des nœuds, la mise en pause du jeu, et la remise à zéro des enregistrements ou du jeu en entier.

Les interfaces du jeu et l'interface monitor devront donc recevoir des messages push contenant les mises à jour sur l'état actuel du jeu. Ces messages seront décrits plus précisément dans les spécifications.

3.3.3. API Javascript

L'API Javascript des WebSockets est implémentée sur la plupart des navigateurs modernes, comme on peut le voir dans le tableau de compatibilité de la figure 8.

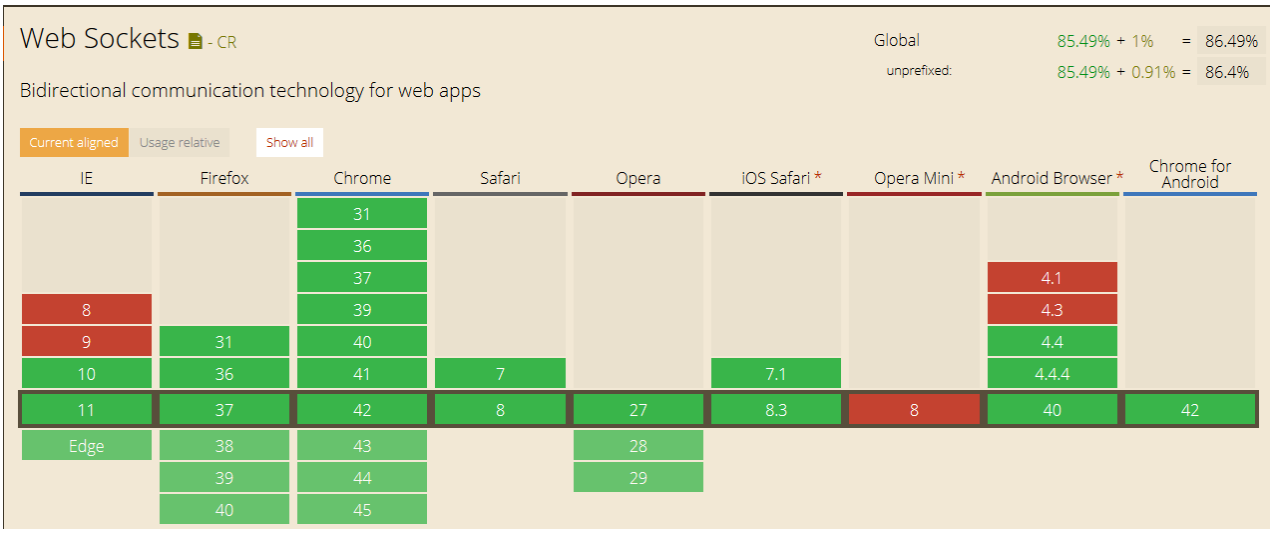


Figure 8 : Compatibilité WebSocket [3]

[3] Masking key : http://tools.ietf.org/html/rfc6455#section-5.3

[4] Encodage Payload : http://tools.ietf.org/html/rfc5234

Le standard WebSocket établit par le W3C définit les méthodes indispensables suivantes :

Méthodes	Description
Constructeur	Pour créer un objet WebSocket, il est nécessaire de lui fournir une URL et éventuellement le protocole associé. L'URL est généralement de type `ws://somesite.com/path/to/app` Ce paramètre correspond à l'URL côté serveur pour communiquer avec le WebSocket. `var ws = new WebSocket(DOMString [url], DOMString protocol ) ;`

`send()`	Cette fonction est utilisée pour envoyer un message au serveur.
	Il peut utiliser les attributs suivants, le plus couramment utilisé étant DOMString (une chaine de caractères unicode)
	`void send(DOMString data);`
	`void send(Blob data);`
	`void send(ArrayBuffer data);`
	`void send(ArrayBufferView data);`

`close()`	Cette fonction permet de fermer le WebSocket. Il est possible d'y ajouter un paramètre « `short code` »et une « `DOMString reason` » afin d'indiquer la raison de la fermeture.

De plus, l'objet WebSocket devrait implémenter les paramètres suivants :

Attributs	Description
`DomString url`	Attribut en lecture seule qui correspond à l'URL créée par le constructeur
`DOMString protocol;`	Attribut en lecture seule qui correspond au protocole passé en paramètre du constructeur.
`Short readyState`	Cet attribut correspond à l'état actuel du WebSocket. Il est automatiquement mis à jour par le navigateur et peut être consulté pour connaitre l'état du WebSocket. Il prend les valeurs suivantes selon son état : `CONNECTING = 0, OPEN = 1, CLOSING = 2, CLOSED = 3`
`Long bufferedAmount`	Cet attribut retourne le nombre d'octets mis en file d'attente par la méthode send(), mais qui n'ont pas encore été envoyés sur le réseau.

Finalement, il est nécessaire de spécifier les méthodes suivantes afin d'effectuer des actions lors d'événements relatifs au WebSocket créé :

Événement	Description
`onopen()`	Méthode invoquée lors de l'ouverture du WebSocket
`onmessage()`	Méthode invoquée lors de la réception d'un message par le serveur
`onerror()`	Méthode invoquée lors d'une erreur
`onclose()`	Méthode invoquée lors de la fermeture du WebSocket

3.3.4. API Java JSR 356

L'API Java JSR est définie par la communauté Java. Une JSR représente une Java Specification Request. Il s'agit d'une demande de modification ou d'ajout de fonctionnalités dans la plateforme Java. Elles doivent être discutées par la Java Community Process avant d'être validées et implémentées dans la plateforme Java.

L'API Java JSR 356[5] est actuellement dans un état stable depuis le 13 août 2014[6].

Les codes sources des exemples Tomcat peuvent être trouvés sur le site du développeur[7]

Il existe deux méthodes pour la création de WebSocket avec l'API JSR 356 :

- Annotation
- Programmatic

De plus, il existe plusieurs utilitaires supplémentaires que nous présenterons dans les chapitres ci-dessous.

3.3.4.1. Annotation Method

Voici un exemple basique d'utilisation côté serveur. Les noms des classes et des fonctions peuvent être changés. Les annotation (@Annotation) servent à indiquer quelle fonction est appelée lorsqu'un événement se déclenche sur le WebSocket.

```
import javax.websocket.* ;

@ServerEndpoint(value = "/path/to/endpoint")

public class WebSocketName{

    @OnOpen

    Public void init(Session session, EndpointConfig conf){

        // Fonction exécutée lors de la connexion d'un client sur le websocket

    }

    @OnClose

    Public void end(Session session, CloseReason reason){

        // Fonction exécutée lors de la déconnexion d'un client

    }

    @OnMessage

    public void incoming(Session session, String message) {

       // Fonction exécutée lors de l'arrivée d'un message sur le websocket

    }

    // Fonction utilisée pour envoyer un message.

    session.getBasicRemote().sendText(msg)

}
```

Lorsqu'un client (navigateur) se connecte sur un WebSocket, une nouvelle instanciation de la classe est créée.

[5] API JSR 356 : http://www.oracle.com/technetwork/articles/java/jsr356-1937161.html

[6] État de la spécification : https://jcp.org/en/jsr/detail?id=356

[7] Apache Tomcat source code http://svn.apache.org/viewvc/tomcat/trunk/webapps/examples/

Il est important de noter que la librairie annotation-api.jar est nécessaire pour la compilation d'un annotation EndPoint.

3.3.4.2. Programatic Method

```
public class MyEndpoint extends Endpoint {
   @Override
   public void onOpen(final Session session, EndpointConfig config) {
      session.addMessageHandler(new MessageHandler.Whole<String>() {
         @Override
         public void onMessage(String msg) {
            try {
               session.getBasicRemote().sendText(msg);
            } catch (IOException e) { ... }
         }
      });
   }
   @Override
   public void onClose(final Session session, EndpointConfig config) {
      System.out.println("The EndPoint has been closed") ;
   }
}
```

La méthode addMessageHandler permet d'ajouter le déclenchement d'un événement lorsqu'un nouveau message arrive.

Pour déployer le EndPoint dans notre application, il suffit d'utiliser la commande suivante :

```
ServerEndpointConfig.Builder.create(MyEndpoint.class,
"/path/to/endpoint").build();
```

3.3.4.3. Paramètres dans le chemin du EndPoint

Il est possible de créer un EndPoint paramétrisable avec des accolades. Le paramètre définit peut être accédé dans la méthode `OnOpen` à l'aide de l'annotation `@PathParam("param-name") String param`.

```
@ServerEndpoint("/example/{param-name}")
public class MyEndpoint {
   @OnOpen
   public void open(Session session, EndpointConfig c,
                    @PathParam("param-name") String paramName) {
      // Use the parameter
   }
}
```

Ce cas n'est pas vraiment applicable pour l'application NetGame on WebSocket, car nous devons récupérer dynamiquement les noms des nœuds et couches lors de la connexion au jeu NetGame.

3.3.4.4. Gestion des erreurs

En cas d'erreur lors de la connexion, dans la conversion lors du décodage, il est possible de traiter l'erreur à l'aide de l'annotation `OnError` :

```
@OnError
   public void error(Session session, Throwable t) {
      t.printStackTrace();
   }
```

3.3.4.5. Définir une configuration de EndPoint

Un `EndPointConfig` permet de sauvegarder des paramètres envoyés lors de la requête initiale du WebSocket (HandShakeRequest).

```
public class GetHttpSessionConfigurator extends
ServerEndpointConfig.Configurator
{
  @Override
    public void modifyHandshake(ServerEndpointConfig conf,
                                HandshakeRequest req,
                                HandshakeResponse resp) {
     // Saves the HandshakeRequest for use in the WebSocket EndPoint.
         conf.getUserProperties().put("handshakereq", req);
    }
}
```

Il est ensuite possible de récupérer le paramètre stocké dans `userProperties` à l'aide de la méthode `get("param")` pour l'utiliser ensuite dans la méthode `OnOpen`. L'exemple ci-dessous montre comment récupérer l'identifiant de la session Java.

```
@OnOpen
public void open(Session session, EndpointConfig config) {
  HandshakeRequest req = (HandshakeRequest) config.getUserProperties()
       .get("handshakereq");
  HttpSession httpsession = (HttpSession) req.getHttpSession();
  this.cookie = httpsession.getId();
}
```

On peut noter que l'implémentation JSR 356 de WebSocket ne permet pas de récupérer l'adresse IP du client avec cette méthode.

3.3.4.6. Encodeurs / Decodeurs

L'API JSR 356 offre la possibilité de créer des encoders et décoders pour les messages échangés entre le navigateur Web et le ServerEndpoint WebSocket. Cela permet de standardiser les messages échangés. Pour configurer les encodeurs et décodeurs, il suffit de modifier la ligne de création du EndPoint pour y ajouter les paramètres suivants :

Configuration EndPoint

```
@ServerEndpoint(
   value = "/myendpoint",
   encoders = { MessageATextEncoder.class, MessageBTextEncoder.class },
   decoders = { MessageTextDecoder.class }
)
```

Décodeur

La classe décodeur nécessite les fonctions `destroy()`, `decode(String s)` qui permettra de décoder le message reçu et `willDecode(String s)` qui permettra de déterminer si le décodeur est capable de décoder le message reçu.

```
public class MessageTextDecoder implements Decoder.Text<Message> {
   @Override
   public void init(EndpointConfig ec) { }
   @Override
   public void destroy() { }
   @Override
   public Message decode(String string) throws DecodeException {
      // Read message...
      if ( /* message is an A message */ )
         return new MessageA(...);
      else if ( /* message is a B message */ )
         return new MessageB(...);
   }
   @Override
   public boolean willDecode(String string) {
      // Determine if the message can be converted into either a
      // MessageA object or a MessageB object...
      return canDecode;
   }
}
```

Encodeur

La classe encodeur nécessite la fonction encode(MessageA msgA) qui permettra d'encoder le message A en format JSONString pour le transmettre au client.

```
public class MessageATextEncoder implements Encoder.Text<MessageA> {
   @Override
   public void init(EndpointConfig ec) { }
   @Override
   public void destroy() { }
   @Override
   public String encode(MessageA msgA) throws EncodeException {
      // Access msgA's properties and convert to JSON text...
      return msgAJsonString;
   }
}
```

Envoi de messages

Pour envoyer un message d'un type spécifique, il suffit d'utiliser les méthodes suivantes :

```
MessageA msgA = new MessageA(...);
MessageB msgB = new MessageB(...);
session.getBasicRemote.sendObject(msgA);
session.getBasicRemote.sendObject(msgB);
```

Conclusion

Les Encoders/Decoders n'ont pas été implémentés dans l'application NetGame on WebSocket. En effet, il aurait fallu une classe par type de message (voir Chapitre 4.3 « Spécifications ») et cette méthode n'offre pas d'avantages.

3.4. Format de message JSON

Nous utiliserons le format JSON pour transférer des messages à travers le WebSocket.

Le JSON a été choisi car il s'agit d'un format de message simple et intuitif. En effet, il est pris en charge par la plupart des langages de programmation et il s'agit du format objet par défaut de Javascript, ce qui simplifie son utilisation dans le cadre de ce projet.

Les prochains chapitres décriront l'utilisation du JSON en Java et en Javascript.

3.4.1. Java (Serveur)

Plusieurs librairies Java existent pour utiliser JSON dans le langage Java. C'est le cas des librairies JSON-Simple, mais également de la librairie développée par Douglas Crockford. C'est cette dernière qui a été choisie pour l'implémentation de ce projet. En effet, elle offre une utilisation simplifiée des méthodes pour manipuler les objets.

Du côté WebSocket, il faudra traiter le message JSON reçu par l'application client. C'est-à-dire vérifier le type de message (s'il existe) et effectuer l'action contenue dans le message.

Soit msg le String JSON reçu depuis le client (exemple avec la récupération des paramètres pour le node et layer du register), on peut récupérer les informations d'un objet JSON avec la méthode `get` et un casting du type, ou directement à l'aide de la méthode `getType` où `type` est le type de l'objet en sortie.

```
JSONObject js = new JSONObject(msg);
String command = (String) js.get("type");
JSONObject param = (JSONObject) js.get("param");
if(command.equals("Register")){
   this.nodeId = param.getInt("node");
   this.layerId = param.getInt("layer");
}
```

Pour l'envoi de message, il est possible d'utiliser la méthode `put` d'un objet JSON (exemple avec l'unicast d'un message)

```
JSONObject fullmsg = new JSONObject();
fullmsg.put("type", "Notification");
JSONObject param = new JSONObject();
param.put("Message", "Alerte : Le jeu est mis en pause");
fullmsg.put("error", "0");

unicast(nodeId, layerId, fullmsg.toString());
```

Il y a deux comportements différents avec la méthode `put()` de la classe JSONObject :

- Un objet simple
- Un tableau d'objet

Pour illustrer le fonctionnement de chacun des scénarios, nous allons utiliser l'objet Example suivant :

```
public class Example implements Serializable{
   public int id;
   public int getID() {
      return id;
   }
   public int getNextID() {
      return id + 5;
   }
   public String getOtherInfos(){
      return "This id is : "+id;
   }
   public String toString(){
      return "myObjectToString()";
   }

}
```

Objet simple :

Lorsqu'on ajoute un objet simple à un objet JSON, il invoque sa méthode `toString()`

```
public class ExampleTest {

   public static void main(String[] args) {
      Example example = new Example();
      example.id = 2;

      JSONObject examplejson = new JSONObject();
      examplejson.put("array", example);
      System.out.println(examplejson.toString());

   }
}
```

Résultat :

```
{"array":"myObjectToString()"}
```

Tableau d'objet :

Dans le cas d'un tableau d'objet, ce n'est plus la fonction toString de l'objet qui est appelée, mais tous les getters de l'objet Java. Ainsi, l'exemple suivant, avec des getters non liés à des attributs générera un tout autre résultat qu'avec un objet simple.

```
public class ExampleList {
   public Example[] array = null;
}
public class ExampleTest {
   public static void main(String[] args) {
      Example example = new Example();
      example.id = 2;
      ExampleList exampleList = new ExampleList();
      exampleList.array = new Example[1];
      exampleList.array[0] = example;
      JSONObject examplejson = new JSONObject();
      examplejson.put("array", exampleList.array);
      System.out.println(examplejson.toString());
   }
}
```

Résultat :

```
{"array":[{"otherInfos":"This id is : 2","ID":2,"nextID":7}]}
```

L'objet généré correspond aux getters Java de l'objet. Il est important de noter que ces getters doivent être en CamelCase et non nuls pour que le JSON les ajoute à l'objet généré.

3.4.2. Javascript (Client)

Du côté client, chaque nouveau message sera traité de la même manière, et mettra à jour l'interface graphique du joueur.

Le message reçu sera parsé à l'aide de la méthode Javascript « `JSON.parse(JSONString)` » et la méthode `JSON.stringify(JSONObject)` pourra être utilisée pour envoyer les données au serveur.

Il est possible d'accéder à n'importe quel paramètre de la même manière en Javascript.

```
<script>
// Un exemple de message reçu depuis le WebSocket
var msg = JSON.parse("{ \"type\":\"Notification\", \"param\":{\"Message\":\"Hey
you got an alert!\"}, \"error\":\"0\" }");

alert(msg.type); // Affiche le type de message : Notification
alert(msg.param.Message); // Affiche le contenu du message : Hey you got an
alert!
alert(msg.error); // Affiche l'erreur : 0

// Un exemple d'objet Javascript à envoyer
var message = {
    type:"Register",
    param:{
        NodeId:1,
        LayerId:2,
        Alias:"MyAlias",
        IP:"127.0.0.1"
    },
    error:0 };
alert(JSON.stringify(message));
// Affiche l'objet transformé en format texte :
{"type":"Register","param":{"NodeId":1,"LayerId":2,"Alias":"MyAlias","IP":"127.
0.0.1"},"error":0}
</script>
```

Il est possible de vérifier si un paramètre existe ainsi :

```
If(msg.param.theParam){
    // Actions à effectuer s'il existe
}
```

En effet, s'il n'existe pas, la valeur est undefined, et elle est évaluée à false dans une condition.

3.5. Conclusion

L'analyse de l'état de l'art des différents éléments qui seront utilisés lors de ce projet étant terminée, nous pouvons désormais passer à la conception d'une application basée sur la technologie WebSocket.

4. Conception

Ce chapitre de conception nous permet de décrire le développement de notre application en mettant en avant les démarches et idées utilisées. Cela permettra à une personne externe de reprendre le développement en vue d'y ajouter des fonctionnalités.

4.1. Organisation des fichiers

L'application NetGame on WebSocket sera développée en Java à l'aide de fichiers JSP et de classes Java.

Le tableau 8 nous permet de mieux nous représenter l'organisation des fichiers utiles à l'application :

Chemin	Description
`/NetGame/`	Contient tous les fichiers de l'application
`/NetGame/index.html`	Point d'arrivée des utilisateurs, avec les liens vers les autres fichiers
`/NetGame/dumps/`	Contient les sauvegardes existantes de l'état du jeu.
`/NetGame/images/`	Contient toutes les images utilisées dans l'application
`/NetGame/js/`	Contient les fichiers Javascript utilisés par l'application
`/NetGame/incl/`	Contient les fichiers inclus par les fichiers JSP
`/NetGame/jsp/`	Contient les fichiers JSP admin, monitor et NetGame (jeu)
`/NetGame/css/`	Contient les fichiers de styles utilisés pour la mise en page des fichiers HTML et JSP
`/NetGame/WEB-INF/`	Contient les classes Java et les librairies associées

Tableau 8 : Organisation des fichiers

4.1.1. Organisation des classes Java

Les classes Java du dossier `/NetGame/WEB-INF/classes` utilisés dans l'application seront divisés en plusieurs packages.

org.json

Ce package contient les classes Java de la librairie de Douglas Crockford permettant d'utiliser les objets JSON en Java.

ch.eif.netgame2

Ce package contient les classes de l'application. Elles sont encore divisées en trois parties :

- server : contient tous les fichiers liés à la classe principale NetGame et à son fonctionnement sur le serveur
- websocket : contient tous les fichiers utilisés pour faire fonctionner les WebSockets
- shared : contient les classes utilitaires utilisées pour le transfert entre la classe NetGameWS et le client du navigateur Web.

4.2. Communication Client-Server

Nous allons créer une classe NetGameWS ayant une interaction avec les clients par l'intermédiaire des WebSockets, et une classe NGWSBridge faisant l'intermédiaire entre le jeu NetGame et la classe NetGameWS. La partie d'administration (admin) reste inchangée. La classe NGWSBridge a été imposée par M. Scheurer afin d'effectuer le moins possible de changements dans l'application NetGame, et de le limiter à un simple appel de fonction. Cela permet de séparer au mieux les différents éléments de l'application.

Le schéma de la figure 9 résume l'interconnexion entre les différents éléments :

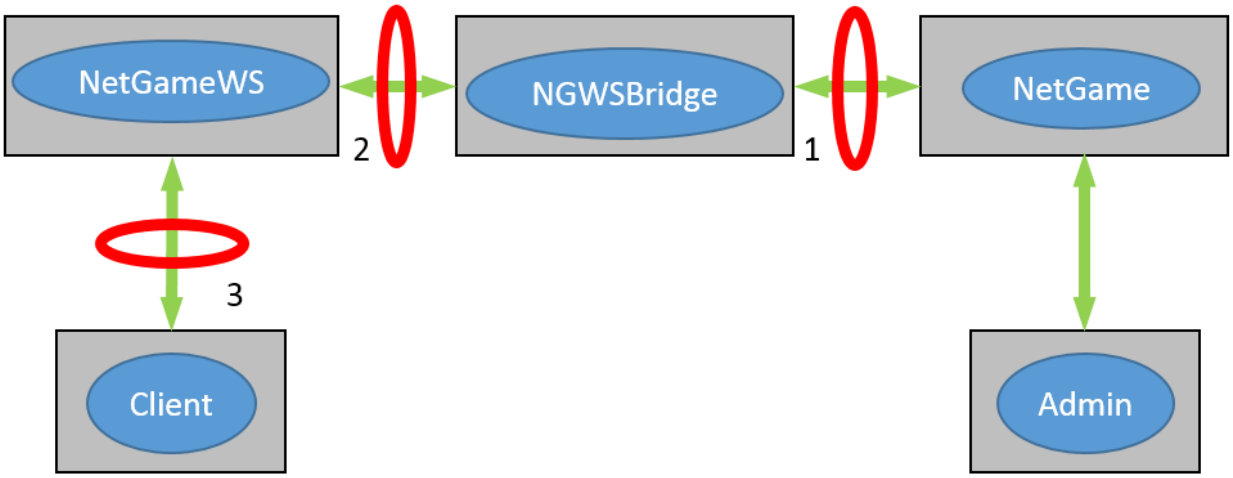


Figure 9 : Interfaces de l'application

Les cercles rouges représentent les interfaces définies par les spécifications.

Tout au long de ce chapitre, nous utiliserons les mots « message » pour désigner un paquet du jeu NetGame, et « notification » pour désigner un message de notification envoyé par le jeu.

En effet, l'application NetGame a été créée avec le terme `Message` pour les messages du jeu et nous avons conservé cette nomenclature pour éviter les confusions.

4.3. Spécifications des interfaces du jeu

Nous allons définir les interfaces des classes NGWSBridge et NetGameWS.

4.3.1. Classe NGWSBridge

La classe NGWSBridge a pour but de faire l'intermédiaire entre la classe NetGameWS (WebSockets côté serveur) et le jeu NetGame.

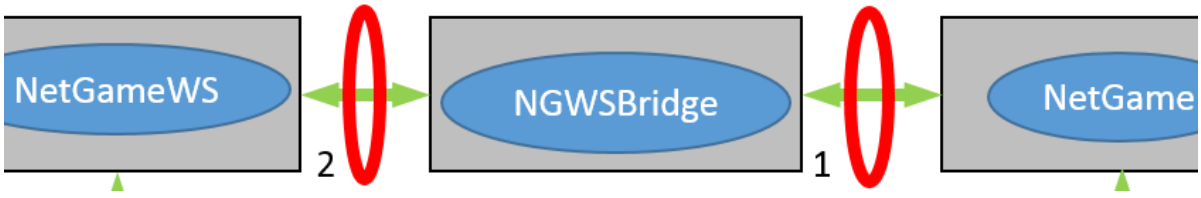


Figure 10 : Interface NGWSBridge – NetGame

Interface NetGameWS → NGWSBridge (2)

La liste des fonctions qui seront appelées par la classe NetGameWS pour accéder au NGWSBridge sont définies dans le tableau 9.

Fonction	Description
`String[][] wsngGetNodeAndLayers`	Récupère la liste des noms de chaque couche et de chaque nœud chacun dans un tableau.
`String wsngDoRegistration()`	Enregistre un joueur sur le jeu
`int nodeID`	Identifiant du nœud
`int layerID`	Identifiant de la couche
`String alias`	Pseudo du joueur
`String ipAddress`	Adresse IP du joueur
`String cookie`	Cookie identifiant unique du joueur
`return String status`	Résultat de l'enregistrement
`void wsngClearRegistration()`	Supprime l'enregistrement d'un joueur
`int nodeID`	Identifiant du nœud
`int layerID`	Identifiant de la couche
`public boolean wsngLockMessage()`	Bloque un message en cours de traitement.
`int nodeID`	Identifiant du nœud
`int layerID`	Identifiant de la couche
`int msgID`	Identifiant du message
`return boolean`	Résultat de l'opération
`public void wsngRemoveMessageLocks()`	Permet de supprimer le blocage du message
`int nodeID`	Identifiant du nœud
`int layerID`	Identifiant de la couche
`public String wsngProcessMessage()`	Permet de valider le traitement d'un message
`int nodeID`	Identifiant du nœud
`int layerID`	Identifiant de la couche
`int msgID`	Identifiant du message traité
`int changedValue`	FCS du message
`return String status`	null ou un message d'erreur
`public void wsngLaunchMessage()`	Permet de créer un nouveau message
`int nodeID`	Identifiant du nœud
`int layerID`	Identifiant de la couche
`int toNodeID`	Identifiant du nœud de destination
`String msgText`	Contenu textuel du message

Tableau 9 : Interface NetGameWS - NGWSBridge

Interface NetGame → NGWSBridge (1)

La liste des fonctions qui seront appelées par le jeu NetGame pour accéder aux clients, par l'intermédiaire du pont NGWSBridge sont définies dans le tableau 10.

Fonction	Description
`static void ngwsGameReset()`	Remise à zéro du jeu
`static void ngwsGameActivityUpdate()` `boolean active`	Change l'état du jeu (actif / inactif)
`static void ngwsGameResumeFromFile()` `String msg`	Reprend la partie à partir d'un fichier de configuration
`static void ngwsGameRegistrationReset()`	Supprime les enregistrements de tous les joueurs

`static void ngwsPlayerRegistrationReset()`
`int nodeID`
`int layerID`
	Supprime l'enregistrement d'un joueur
	Identifiant du nœud
	Identifiant de la couche

`static void ngwsGameRoutingUpdate()` `String debugMsg`	Modifie les informations de routage pour tous les joueurs
`static void ngwsGameStateUpdate()`	Mise à jour de l'état du jeu pour tous les joueurs

`static void ngwsPlayerStateUpdate()`
`int nodeID`
`int layerID`
	Mise à jour de l'état du jeu pour un seul joueur
	Identifiant du nœud
	Identifiant de la couche

`static void ngwsPlayerNotification()`
`int nodeID`
`int layerID`
`String notificationMsg`
	Envoi d'une notification pour un seul joueur
	Identifiant du nœud
	Identifiant de la couche
	Notification

`static void ngwsNodeNotification()`
`int nodeID`
`String notificationMsg`
	Envoi d'une notification pour tous les joueurs d'un nœud
	Identifiant du nœud
	Notification

`static void ngwsLayerNotification()`
`int layerID`
`String notificationMsg`
	Envoi d'une notification pour tous les joueurs d'une couche
	Identifiant de la couche
	Notification

`static void ngwsGameNotification()` `String notificationMsg`	Envoi d'une notification pour tous les joueurs Notification
`static void setNetGame()` `NetGame netgame`	Etablissement du lien entre NetGame et le NGWSBridge. Objet NetGame

Tableau 10 : Interface NetGame - NGWSBridge

4.3.2. Classe NetGameWS

La classe NetGameWS fait le lien entre les clients (à l'aide des WebSockets) et les fonctions du serveur. Il s'agit d'un EndPoint compatible avec les WebSocket.

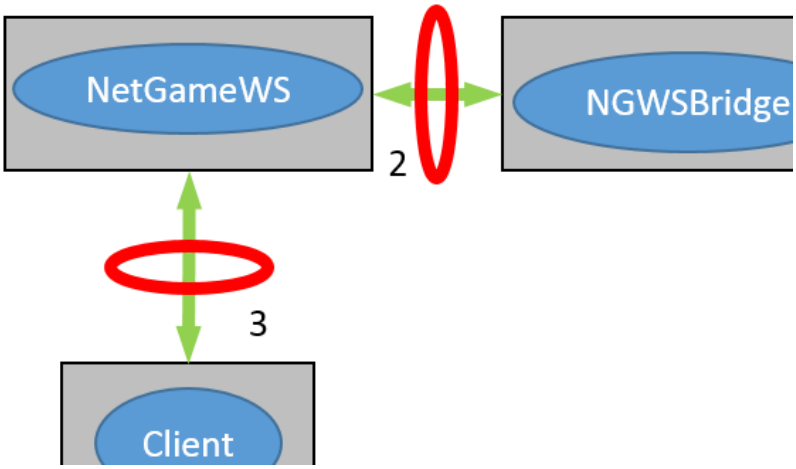


Figure 11 : Interface NetGameWS

Interface NetGameWS – NGWSBridge (2)

La classe NetGameWS utilise les WebSockets et a pour rôle de faire le lien entre le NGWSBridge et les clients. Les fonctions définies dans le tableau 11 seront appelées par la classe NGWSBridge pour envoyer des mises à jour aux joueurs.

Fonction	Description

`static boolean unicast()`
`int nodeId`
`int layerId`
`String message`
	Envoi d'un message WebSocket à un seul client.
	Identifiant du noeur
	Identifiant de la couche
	Message JSON

`static boolean nodeMulticast()`
`int nodeId`
`String message`
	Envoi d'un message WebSocket à tous les clients d'un nœud
	Identifiant du nœud
	Message JSON

`static boolean layerMulticast()`
`int layerId`
`String message`
	Envoi d'un message WebSocket à tous les clients d'une couche
	Identifiant de la couche
	Message JSON

`static boolean broadcast()` `String message`	Envoi d'un message WebSocket à tous les clients connectés Message JSON

Tableau 11 : Interface NetGameWS - NGWSBridge

Interface Navigateur – NetGameWS (3)

L'interface entre la classe NetGameWS et le navigateur est géré en Javascript du côté navigateur et en Java du côté serveur, par l'intermédiaire d'un ServerEndPoint WebSocket.

Le format des messages transmis sera JSON. Ils sont facilement transformables à l'aide de l'implémentation de Dooglas Crockford[8] (org.json.JSONObject). Cette dernière fournit plusieurs fonctions d'encodage et de décodage. Elle permet de transformer des objets JSON en objets Java et inversement.

Le format des messages qui seront envoyés dans l'interface Navigateur-NetGameWS sera le suivant :

{« type » : »*theType* », »param » :{« p1 » : »v1 », »p2 » : »v2 »...} , « error » : »ErrorText »}

Définition des types de messages entre le navigateur et la classe NetGameWS

Le « *type* » du message est en format `String`. Le tableau 12 définit l'utilité de chacun des types de messages

Types	Description	Source
`Notification`	Envoi d'une notification	NetGameWS
`NodeStatus`	Mise à jour des messages à traiter pour chacun des joueurs	NetGameWS
`RegisterResult`	Cookie d'enregistrement, identifiant unique du joueur.	NetGameWS
`Connectivity`	Envoi des modifications des connexions entre les nœuds	NetGameWS
`GameReset`	Remise à zéro du jeu	NetGameWS
`GameChangeState`	Changement d'état du jeu (actif / inactif)	NetGameWS
`RegistrationReset`	Remise à zéro d'un enregistrement	NetGameWS
`ProcessResult`	Résultat d'un message traité	NetGameWS
`NodesLayers`	Envoi les noms des nœuds et des couches	NetGameWS
`Register`	Enregistrement d'un joueur	Client
`ClearRegister`	Désenregistrement d'un joueur	Client
`Lock`	Bloque un message à traité	Client
`Unlock`	Débloque un message précédemment bloqué	Client
`ProcessMsg`	Traitement d'un message	Client
`LaunchMsg`	Lance un message	Client

Tableau 12 : Types de messages JSON

[8] https://github.com/douglascrockford/JSON-java

Définition des paramètres

Le tableau 13 définit pour chaque type de message WebSocket les paramètres envoyés, ainsi que leurs types.

Type de message	Paramètres
`Notification`	`(String) Message` : notification
`NodeStatus`	`(Object NetGameNodeLayerState) Status` : état du jeu pour le nœud et la couche spécifiée L'objet `NodeLayerState` contient la liste des messages (InputQueue) à traiter pour tous les nœud. La bufferedQueue est également ajoutée pour les joueurs de couche 2.
`RegisterResult`	`(String) Message` : Message d'enregistrement `(Boolean) Status` : Status (true = réussi)
`Connectivity`	`(boolean[]) Routing` : connectivité entre les différents nœuds.
`GameReset`	Aucun paramètre
`GameChangeState`	`(boolean Status` : actif / inactif
`RegistrationReset`	Aucun paramètre
`ProcessResult`	`(String) Message` : résultat du message traité
`Register`	`(int) NodeId` : le nœud du joueur `(int) LayerId` : la couche du joueur `(String) Alias` : le pseudo du joueur `(String) IP` : L'adresse IP du joueur
`ClearRegister`	Aucun paramètre.
`Lock`	`(int) MsgId` : Id du message en cours de traitement
`Unlock`	Aucun paramètre
`ProcessMsg`	`(int) MsgId :` l'identifiant du message à traiter `(int) changedValue :` La valeur changée dans le paquet
`LaunchMsg`	`(int) ToId` : identifiant du nœud de destination `(String) MsgTxt` : Contenu textuel du message

Tableau 13 : Paramètres des messages JSON

ProcessMsg : Valeur changée

La valeur changée prend plusieurs valeurs, selon la couche du joueur. Elle permet de modifier le contenu du message envoyé.

Dans le cas où le joueur est enregistré en tant que couche Network Layer, la valeur changée envoyée permet de spécifier l'identifiant du prochain nœud (routing).

Dans le cas où le joueur est enregistré en tant que couche Data Link Layer, la valeur changée envoyée permet de spécifier le numéro de séquence du message. Cela permet d'identifier le message en couche DLL pour les acquittements.

Dans le cas où le joueur est enregistré en tant que couche Physical Layer, la valeur changée représente le FCS envoyé sur la couche physique « réseau », c'est-à-dire, dans les câbles. Si le FCS est incorrect, cela signifie qu'il y a eu une erreur dans l'envoi des données et que le message doit être renvoyé.

4.4. Authentification

Chaque utilisateur sera automatiquement connecté au WebSocket de la classe NetGameWS à son arrivée sur la page NetGame.jsp. Lorsque l'utilisateur choisit un nœud et une couche et s'enregistre à l'aide d'un message Register, le WebSocket envoie une requête d'enregistrement à l'application NetGame. Si l'enregistrement est validé par la classe NetGame, alors la classe NetGameWS modifiera ses attributs `nodeId` et `layerId`.

Ces attributs seront utilisés par la classe NetGameWS pour identifier le joueur vers la classe NetGame, et également pour déterminer la destination des messages (*unicast*, *multicast*, *broadcast*) à travers le WebSocket.

4.5. Traitement d'un message

Dans le cas du traitement d'un message standard, l'application suivra le fonctionnement suivant :

Un utilisateur choisit un nœud et une couche, ainsi qu'un pseudo, puis clique sur le bouton « S'enregistrer ». L'interface Web envoie un message Register au serveur, avec les informations de connexion. Le serveur enregistre l'utilisateur, et renvoie un message RegisterResult avec le Status de l'enregistrement. Si celui-ci n'est pas réussi, un message d'erreur s'affiche à l'écran et le joueur peut choisir une autre combinaison nœud/couche. Si l'enregistrement a réussi, la liste des messages du joueur s'affiche.

Le joueur sélectionne alors un message à traiter dans la liste à l'aide de la flèche à côté du message. Une nouvelle fenêtre s'ouvre, et le joueur peut alors traiter le message, ou annuler l'opération pour revenir à la liste des messages. L'interface client vérifie à l'aide du Javascript si les informations entrées sont correctes. Si ce n'est pas le cas, elle affiche un message d'erreur à l'écran, et le joueur doit corriger les informations du message. Si les informations entrées par l'utilisateur sont correctes, alors le message est envoyé au serveur, signalant que le message a bien été traité par l'utilisateur.

La valeur changée a été décrite plus précisément dans l'analyse de l'application.

Le schéma de la figure 12 résume les différentes actions effectuées par chacun des acteurs.

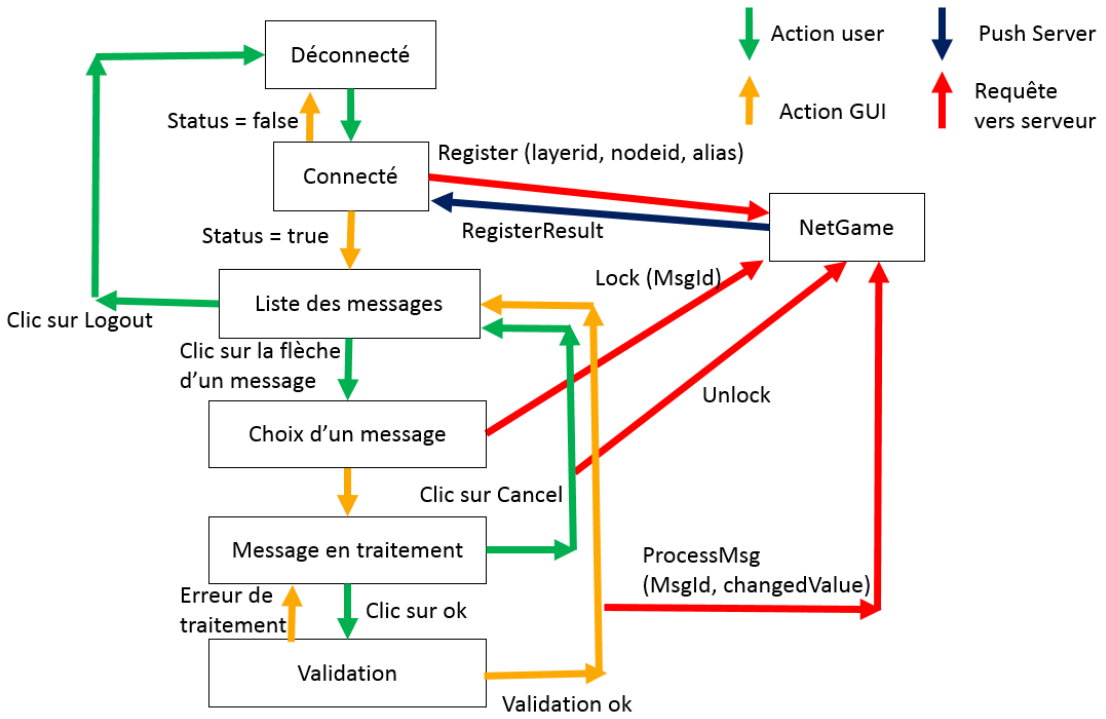


Figure 12 : Interactions GUI

4.6. Mise à jour des liste de messages

La liste des messages des joueurs sera mise à jour automatiquement par l'interface graphique. Lorsqu'un nouveau message est créé ou lorsqu'un message est traité, l'application NetGame enverra un message NodeStatus contenant la nouvelle liste des messages aux utilisateurs concernés. L'interface graphique du joueur se mettra alors immédiatement à jour pour afficher les nouveaux messages ou supprimer les messages traités.

4.7. Gestion du langage

Dans la maquette, une gestion dynamique du langage est prévue à partir d'un simple fichier de configuration JSON. Les fichiers contenus dans `NetGame/js/languages/*.js` contiennent un objet « `obj_lang` » de type JSON. Ce dernier contient toutes les associations « identifiant :value » séparés par une virgule, qui viendront modifier le fichier HTML pour adapter la langue.

Pour ajouter un nouvel élément géré par ce fichier de configuration, il suffit d'ajouter un élément HTML (de préférence inline de type `<span></span>`) et de lui assigner un id de format « `text-*` », puis de modifier les fichier contenus dans `js/languages/`. Le Javascript modifiera automatiquement le HTML affiché aux joueurs.

4.7.1. Définition des langues

La langue est passée en paramètre `lang` au fichier `NetGame.jsp`. Ce dernier incluera le fichier de langue. La langue par défaut est l'anglais.

4.7.2. Ajout d'une langue

Pour ajouter une nouvelle langue, il faut ajouter un fichier de configuration dans `/NetGame/js/languages/` et modifier le fichier `/NetGame/jsp/NetGame.jsp` pour y ajouter la langue.

Le fichier de langue est inclus directement à l'aide de la méthode suivante :

```
<script src="../js/languages/<%
if(request.getParameter("lang") == null){
    out.print("en");
} else {
    String lang = request.getParameter("lang");
    if(lang.equals("fr") || lang.equals("de")){
        out.print( lang );
    } else {
        out.print("en");
    }
} %>.js"></script>
```

Il faut donc ajouter une condition pour confirmer que le nouveau fichier de langue est valide. Le contrôle effectué dans le code ci-dessus permet de protéger le paramètre `lang` contre une attaque de type XSS.

4.7.3. Ajout d'un nouveau mot

Le code Javascript suivant est utilisé pour modifier tous les éléments HTML avec un id **`text-*`** correspondant dans l'objet `obj_lang`.

```
// Dynamically fill all fields with the good language (see js/languages/fr.js)
$(function(){
    for (var i in obj_lang){
        $("#text-"+i).html(obj_lang[i]);
    }
});
```

4.7.4. Exemple

HTML

```
<span id="text-example"></span>
```

Javascript

```
var obj_lang = {
    autres_valeurs:"valeur",
    example:"Nouveau libellé pour example"
}
```

4.8. Spécifications de l'interface Monitor

L'interface Monitor est utilisée dans le cadre de l'application NetGame on WebSockets pour afficher l'état de chacun des nœuds, à savoir la liste des pseudos, le nombre de messages à traiter par couche, et savoir quels nœuds ne sont pas occupés par un joueur.

Pour ce faire, la classe NGWSBridge enverra ses messages au travers de la classe WebSocket WSMonitor. À la différence du WebSocket NetGameWS, celui-ci n'enverra aucun message vers NetGame.

Le tableau 14 définit les messages envoyés par NGWSBridge au travers du WSMonitor.

Types	Description	Source
`GameStatus`	Changement d'état du jeu (actif / inactif)	NGWSBridge
`Connectivity`	Envoi des modifications des connexions entre les nœuds	NGWSBridge
`MessagesCount`	Envoi des modifications du nombre de messages pour chaque couche / nœud.	NGWSBridge
`UsersList`	Envoi des modifications des joueurs connectés	NGWSBridge

Tableau 14 : Spécification des messages NetGame – WSMonitor

Le tableau 15 définit les paramètres des messages envoyés.

Type de message	Paramètres
`GameStatus`	`(Boolean) Status` : Status (actif / inactif)
`Connectivity`	`(Boolean[][]) Connectivity` : Matrice de connexion entre chacun des nœuds
`MessagesCount`	`(int[][][]) MessagesCount` : Matrice du nombre de messages pour chaque couche / noeud. La troisième dimension du tableau permet de transférer également pour la couche DLL la taille du buffer d'envoi.
`UsersList`	`(String[][]) UsersList` : Matrice des pseudos pour chaque couche / noeud

Tableau 15 : Paramètres des messages NGWSBridge – WSMonitor

4.9. Conclusion

Le chapitre de conception nous aura permis de définir les interfaces entre chacun des éléments dde notre application. Nous pouvons désormais nous concentrer sur l'implémentation de l'application NetGame on WebSocket.

5. Implémentation

5.1. Introduction

Ce chapitre a pour but de décrire les points importants de la configuration de l'environnement, et certains aspects spécifique de Tomcat. Nous verrons ensuite les différentes étapes de l'implémentation lors de ce projet.

5.2. Configuration Tomcat & Eclipse

Dans le cadre de ce projet, nous avons choisi de développer sur linux avec le programme eclipse.

5.2.1. Installation Tomcat & NetGame

Pour installer Tomcat 8, il faut suivre la marche à suivre suivante :

- Télécharger la dernière version de Tomcat 8 sur le site Apache[9]
- Décompresser le fichier « `tar xzf apache-tomcat-8.0.20.tar.gz` »
- Déplacer le dossier décompressé dans le dossier de votre choix « `sudo mv apache-tomcat-8.0.20 /opt/tomcat8` »
- Ajouter les variables d'environnement :
  - Ouvrir le fichier .bashrc
  - Ajouter:
    - `export JAVA_HOME=/usr/lib/jvm/java-7-openjdk-amd64`
    - `export CATALINA_HOME=/opt/tomcat`
- Démarrer le serveur Tomcat « `$CATALINA_HOME/bin/catalina.sh run` » ou pour démarrer en mode daemon : « `$CATALINA_HOME/bin/start.sh`»
- Déplacer le dossier NetGame dans les applications: « `mv NetGame /opt/tomcat8/webapps/` »
- L'application est désormais accessible sur : http://localhost:8080/NetGame

5.2.2. Configuration Eclipse

À présent que Tomcat est correctement configuré et installé, il faut suivre la marche à suivre suivante afin de pouvoir modifier le code de l'application.

- Ajouter le projet dans eclipse :
  - New => Java Project
  - Sélectionner "Use project folder as root for sources and class files
  - Choisir Location : `/opt/tomcat/webapps/NetGame/classes`
  - Cliquer « Next » puis « Finish »
- Il faut encore modifier le build Path du projet pour y ajouter certaines librairies essentielles :
  - Clic-droit sur le projet → Properties → Java Build Path
  - Ajouter les libraries suivantes se trouvant dans `/opt/tomcat/lib`:
    - **annotation-api.jar**
    - websocket-api.jar
    - servlet-api.jar
  - Ajouter les librairies se trouvant dans `Netgame/WEB-INF/lib`
    - xstream-1.3.1.jar
    - xpp3_min-1.1.4.jar
- Votre Eclipse est désormais prêt à être utilisé et vous devriez pouvoir modifier les classes.
- Note : Il est nécessaire de redémarrer Tomcat lors d'une modification du code Java.

[9] Site Internet d'Apache : https://tomcat.apache.org/download-80.cgi

5.2.3. Encoding

Par défaut, l'encodage envoyé par le serveur Tomcat pour les pages JSP est iso-8859-1. Si l'on souhaite envoyer des pages au format UTF-8, il est nécessaire d'ajouter la ligne suivante au début du fichier :

```
<%@page contentType="text/html; charset=UTF-8" session="true" %>
```

Note : Cette ligne ajoute également une session.

Pour plus d'informations concernant l'encodage avec Tomcat, consulter la documentation officielle[10]

5.3. Implémentation

L'implémentation de cette application s'est faite en trois parties. Tout d'abord une application simple utilisant un WebSocket a été créée, puis nous allons lier cette application à NetGame. Finalement nous allons créer une interface GUI complète permettant d'avoir une maquette complète du jeu.

5.3.1. Application simple WebSocket

Cette première partie consistait à prendre en main le fonctionnement des WebSockets, tant du côté serveur que client. Elle a été confrontée à un premier problème. En effet, la librairie annotation-api.jar utilisée pour la génération de WebSocket annotés, n'était pas incluse dans le build path. Ce manque de générant pas d'erreur, il a été difficile de déterminer la source du problème.

Dès lors que ce problème a été découvert et corrigé, les échanges de messages ont pu être effectués sans problèmes.

5.3.2. Interaction WebSocket – NetGame

Cette seconde partie consistait à lier le WebSocket créé avec l'application NetGame. M.Scheurer s'est occupé d'effectuer les modifications dans le code de la classe NetGame pour effectuer les appels des méthodes push vers la classe NetGameWS. Dans un premier temps, ces modifications n'étant pas effectuées, le fonctionnement a été simulé à l'aide de la classe NetGameTest, afin de valider le fonctionnement des envois de messages depuis une classe externe à la classe WebSocket.

Lorsque les modifications ont été effectuées, M.Scheurer a transmis la classe NetGame modifiée et le squelette de la classe NGWSBridge. Dès lors, nous avons pu valider le fonctionnement de l'envoi des nouveaux messages lors d'actions sur l'interface d'administration.

5.3.3. Implémentation du client GUI

La dernière partie consistait à implémenter une interface graphique pour les joueurs. Cette dernière devait permettre de mettre en place une maquette du fonctionnement complet de l'application. C'est-à-dire que chaque couche devait avoir une interface graphique permettant de traiter en particulier ses messages. Cette implémentation a été effectuée en s'inspirant du design de l'ancienne application.

La librairie JQuery a été utilisée pour simplifier les interactions avec le client. Cette dernière offre une utilisation simplifiée du Javascript. De plus la librairie JQuery UI, utilisant comme base la librairie JQuery, a également été utilisée. En effet, elle offre une panoplie de fonction utilitaires pour les interactions avec les utilisateurs.

[10] Apache tomcat character encoding : http://wiki.apache.org/tomcat/FAQ/CharacterEncoding#Q1

5.4. Conclusion

L'implémentation des différentes classes et de l'interface GUI étant terminés, il s'agit désormais de valider leur fonctionnement.

6. Tests et validation

6.1. Introduction

Ce document de tests est divisé en plusieurs parties selon l'ordre d'implémentation, avec tout d'abord des tests du fonctionnement de l'application simple WebSocket, puis de l'interaction entre WebSocket et une classe Java, et finalement le fonctionnement complet de l'application.

Note : Dans ce chapitre le terme client est utilisé pour définir un client (navigateur web) connecté à un WebSocket. Le terme joueur sera utilisé pour définir un client connecté au jeu NetGame par l'intermédiaire d'un WebSocket.

6.2. Application WebSocket simple

Dans un premier temps, une application simple a été créée pour étudier le fonctionnement des WebSockets. Le tableau 16 résume les tests effectués :

Test	Description	Validation
Message Echo	Le message revient en retour	**OK**
Message Broadcast	Plusieurs clients connectés, un client envoie un message, celui-ci est envoyé à tous les clients connectés (simulation de Chat)	**OK**

Tableau 16 : Tests WebSocket

La page HTML utilisée côté client pour effectuer les tests ne comporte qu'un champ permettant d'envoyer un message ainsi qu'une console affichant le contenu de tous les messages reçus.

6.3. Interaction WebSocket – NetGame

Dans un premier temps, l'application NetGame n'étant pas fonctionnelle, une classe Servlet NetGameTest a été développée. Son rôle était de générer des messages push sur une action externe à un utilisateur afin de simuler le fonctionnement final avec la classe NetGame.

Test	Description	Validation
Enregistrement	Message d'enregistrement JSON définissant le nœud et la couche pour valider le fonctionnement. Le retour confirme l'enregistrement en affichant le nœud et la couche séléctionnés.	**OK**
Message vers un seul client	Origine : NetGameTest : Le message envoyé est envoyé à un seul client (avec plusieurs navigateurs connectés).	**OK**
Message vers tous les clients	Origine : NetGameTest : Le message envoyé est envoyé à tous les clients connectés (même principe que Broadcast, sauf qu'il est appelé depuis une fonction externe)	**OK**
Message multicast (layer)	Origine : NetGameTest : Le message envoyé est envoyé à tous les clients enregistrés sur la couche spécifiée.	**OK**
Message multicast (nœud)	Origine : NetGameTest : Le message envoyé est envoyé à tous les clients enregistrés sur le nœud spécifié.	**OK**

Tableau 17 : Tests interaction WebSocket - NetGame

Lorsque l'application NetGame a été mise à jour pour intégrer le fonctionnement du WebSocket, nous avons validé que les messages définis dans les spécifications étaient bien envoyés dans le bon format.

Test	Description	Validation
Notification	L'administrateur envoie un message de notification. Les clients concernés reçoivent le message dans le bon format	**OK**
NodeStatus	Un nouveau message est créé. Le client correspondant à la couche reçoit le message dans le bon format.	**OK**
RegisterResult	Après un message Register du client, celui-ci reçoit un RegisterResult contenant le message dans le bon format	**OK**
Connectivity	L'administrateur change la configuration du réseau. Un message est envoyé à tous les clients de couche 3.	**OK**
GameReset	L'administrateur remet le jeu à zéro. Tous les clients reçoivent le message GameReset.	**OK**
GameChangeState	L'administrateur met le jeu en pause. Tous les clients reçoivent l'état du jeu.	**OK**
RegistrationReset	L'administrateur remet à zéro les enregistrements. Les clients concernés reçoivent le message RegistrationReset	**OK**
ProcessResult	Après un message ProcessMsg du client, le message ProcessResult est retourné contenant le résultat de l'opération	**OK**
Register	Le client envoie un message Register. Le WebSocket le traite et lui retourne un RegisterResult. Le pseudo du joueur apparait désormais dans l'interface d'administration	**OK**
ClearRegister	Le client envoie un message ClearRegister. Le WebSocket le traite et le jeu NetGame supprime l'enregistrement. Un message de notification lui est envoyé lui indiquant le résultat de l'opération.	**OK**
Lock	Le client envoie un message Lock. Le WebSocket le traite le message. Le message concerné apparait bloqué dans l'interface d'administration	**OK**
Unlock	Le client envoie un message Unlock. Le WebSocket le traite le message. Le message concerné apparait débloqué dans l'interface d'administration	**OK**
ProcessMsg	Le client envoie un message ProcessMsg. Le message est traité par le WebSocket et apparait comme tel dans l'interface d'administration.	**OK**
LaunchMsg	Le client envoie un message LaunchMsg. Le message est créé et apparait dans l'interface d'administration.	**OK**

Tableau 18 : Tests des messages NetGame → WebSocket

6.4. Application NetGame on WebSockets

Les tests précédents ont permis de valider le fonctionnement de la communication entre le jeu NetGame, les WebSockets, et une page HTML simple affichant le contenu des messages reçus.

L'application NetGame on WebSocket est l'application complète avec une interface graphique représentant une maquette du fonctionnement du jeu.

L'annexe 2 contient le plan des tests qui ont été réalisés sur l'application NetGame on WebSocket.

7. Conclusion

7.1. Conclusion du projet

L'application NetGame on WebSockets fournit désormais une maquette fonctionnelle du jeu NetGame utilisant en tant que méthode de communication la technologie WebSocket. Elle offre en outre une gestion du langage adaptable et dynamique.

Il a été convenu avec M.Scheurer que la validation côté client serait écartée, et pourrait faire l'objet d'un nouveau travail de semestre.

Le plan de test a permis de valider le fonctionnement de chacune des fonctionnalités implémentées.

Le développement des fonctionnalités étapes par étapes a permis une meilleure compréhension du fonctionnement des WebSockets, et de NetGame. Au final, les quatre objectifs ont été réalisés selon le cahier des charges.

Néanmoins, certaines fonctionnalités sont encore nécessaires pour la mise en production de l'application. Elles seront décrites plus précisément dans le chapitre 7.3 « Opportunités et Améliorations ».

7.2. Conclusion personnelle

Le projet NetGame on WebSocket m'aura permis d'approfondir mes connaissances, tant dans les langages Web clients tels que HTML, CSS et Javascript que dans le fonctionnement du Java en tant que langage Web Serveur. Il m'aura permis en particulier de découvrir et de mieux comprendre la technologique WebSocket, et les particularités du langage Java dans ce domaine.

Il m'a également permis de bien me rendre compte de la difficulté de reprendre un projet existant, et l'importance de la collaboration avec les anciens développeurs. Il m'a également permis de réaliser l'importance de la documentation précise des fonctionnalités d'une application.

En effet, certains aspects de l'application n'étaient pas toujours instinctifs et il a été nécessaire de discuter avec M.Scheurer afin de connaitre l'utilité de certaines classes ou paramètres.

7.3. Opportunités & Améliorations

L'application NetGame on WebSocket développée lors de ce projet reste une maquette simple du fonctionnement du jeu. Les points suivants permettent de définir les améliorations qui devraient y être apportées avant de pouvoir l'utiliser en production pour les cours de Téléinformatique.

7.3.1. Validation des données

L'application NetGame on WebSocket fournit uniquement un modèle de toutes les fonctionnalités de l'ancienne application NetGame. En effet, aucune validation des informations entrées dans les champs des formulaires par les joueurs n'est faite.

Dès lors, deux solutions existent pour valider si les joueurs ont entré des informations correctes dans les champs, à savoir la validation côté client (navigateur) et la validation côté serveur (dans la classe NetGame). L'ancienne version de l'application fonctionnait avec une validation côté client, et c'est cette optique qui a été conservée dans le développement de l'application NetGame on WebSockets. En effet, les joueurs sont des étudiants, et n'ont pas d'intérêts à passer au travers des validations, car le but de l'exercice est l'apprentissage.

Le chapitre 3.1 « Analyse de l'application NetGame » définit les points à valider.

Les points suivants permettent de donner les lignes directrices de chacune des validations et les modifications à apporter à l'applicatoin.

Validation cliente :

Pour réaliser la validation du côté client, il suffit de modifier la fonction `processMsg()` du fichier Javascript se trouvant dans `/NetGame/js/netgamews.js`.

Pour récupérer la valeur d'un champ, il suffit de regarder l'identifiant le représentant et utiliser la fonction Javascript suivante :

```
var value = $("#id").val();
```

L'identifiant peut être trouvé dans le fichier JSP `/NetGame/jsp/NetGame.jsp` et est de la forme « `layerfield` ». Exemple : « `dllfcs` » pour le champ FCS.

Pour connaitre les informations du message initial, nous pouvons utiliser la fonction msgQueue[msgId]. L'objet contient les informations comme la direction et la valeur des champs (msgFields), qui permettront de savoir comment le message doit être traité. De plus, les valeurs des variables myLayer et myNode seront utilisées pour connaitre l'identifiant et la couche actuelle.

Note : Voir le chapitre 3.2.1.3 « Message NetGame » pour connaitre les détails du contenu d'un message et le chapitre 3.1 « Analyse de l'application NetGame » pour savoir comment traiter les messages.

Validation serveur :

Pour réaliser la validation du côté serveur, il sera nécessaire d'envoyer d'avantages d'informations avec le message WebSocket de type ProcessMsg. En effet, dans la version actuelle, il n'envoie que la valeur changée. Cette dernière n'est utile que pour des cas précis où l'utilisateur doit entrer une valeur qui sera utilisée par l'application NetGame pour déterminer le chemin à prendre pour le message, son numéro de séquence, ou encore son FCS.

Il faudrait donc envoyer au travers du WebSocket toutes les informations des champs remplis par l'utilisateur, et les valider de la même manière que pour la validation cliente.

7.3.2. Style & mise en page

La maquette présentée dans ce projet est très simpliste et a pour but principal de montrer les fonctionnalités du jeu. Il n'y a eu que très peu d'ajout de mise en forme de la page.

Afin d'offrir une meilleure interface aux joueurs, il serait appréciable d'effectuer quelques modifications pour la rendre plus « user-friendly ».

7.4. Remerciements

Je tiens à remercier toutes les personnes qui m'auront assisté, de près ou de loin, pour ce projet de semestre 6. Je tiens à remercier particulièrement Monsieur Scheurer pour son encadrement tout au long du projet et pour sa collaboration dans le développement de l'application NetGame on WebSockets, ainsi que M.Wagen pour ses remarques et suggestions au fil du projet. Leur aide m'aura permis d'améliorer certains aspects de l'application afin de rendre un résultat plus complet.

Merci également à M. Sébastien Rossier pour la relecture du présent document.

7.5. Déclaration sur l'honneur

Je, soussigné, David Rossier, déclare sur l'honneur que le travail rendu est le fruit d'un travail personnel. Je certifie ne pas avoir eu recours au plagiat ou à toutes autres formes de fraudes. Toutes les sources d'information utilisées et les citations d'auteur ont été clairement mentionnées.

Fribourg, le 12.05.2015
David Rossier

8. Contenu du CD

Chemin	Contenu
/README.pdf	Liste le contenu du CD
/Administration/	Donnée du projet
/Application/	Fichiers du jeu
/Documentation/	Documentation du projet
/Documentation/Final.pdf	Rapport final du projet (PDF)
/Documentation/Final.docx	Rapport final du projet (DOCX)
/Documentation/GUI.pdf	Interfaces GUI
/Documentation/Tests.pdf	Tests effectués sur l'application finale
/Documentation/Minutes/	PV et invitations des séances
/Documentation/Planification.pdf	Planification du projet
/Documentation/JournalBord.pdf	Journal de bord du projet

9. Sources

9.1. Analyse

[AN01] Documentation Oracle, "Tutoriel WebSocket",
http://docs.oracle.com/javaee/7/tutorial/websocket.htm [mars 2015]

[AN02] Rudolf Scheurer, "Application NetGame" : Documentation des fonctions [mars 2015]

[AN03] Douglas Crockford, "JSON",
https://github.com/douglascrockford/JSON-java [mars 2015]

[AN04] Yidong Fang, "JSON Simple",
https://code.google.com/p/json-simple/ [mars 2015]

[AN05] Wayne Ye, "L'essentiel des Web-Socket",
http://www.codeproject.com/Articles/209041/HTML-Web-Socket-in-Essence [mai 2015]

[AN06] W3C Specification, "WebSocket API Javascript",
http://www.w3.org/TR/websockets/ [mai 2015]

[AN07] Spécification WebSocket : RFC 6455
http://tools.ietf.org/html/rfc6455 [mai 2015]

[AN08] Spécification ABNF (Encodage Payload WebSocket)
http://tools.ietf.org/html/rfc5234 [mai 2015]

9.2. Implémentation

[IM01] Documentation Apache "Tomcat 8",
https://tomcat.apache.org/tomcat-8.0-doc/index.html [mars 2015]

[IM02] Documentation de la librairie JQuery
https://api.jquery.com/ [avril 2015]

[IM03] Documentation de la librairie UI de JQuery
https://jqueryui.com/ [avril 2015]

10. Sources des figures

[1] Le modèle OSI :

http://www.frameip.com/osi/#2.1_-_Les_7_couches

[2] Protocole WebSocket :

http://orm-chimera-prod.s3.amazonaws.com/1230000000545/images/hpbn_1701.png

[3] Support WebSocket :

http://caniuse.com/#feat=websockets

Les autres figures ont été réalisées par M.Rossier.

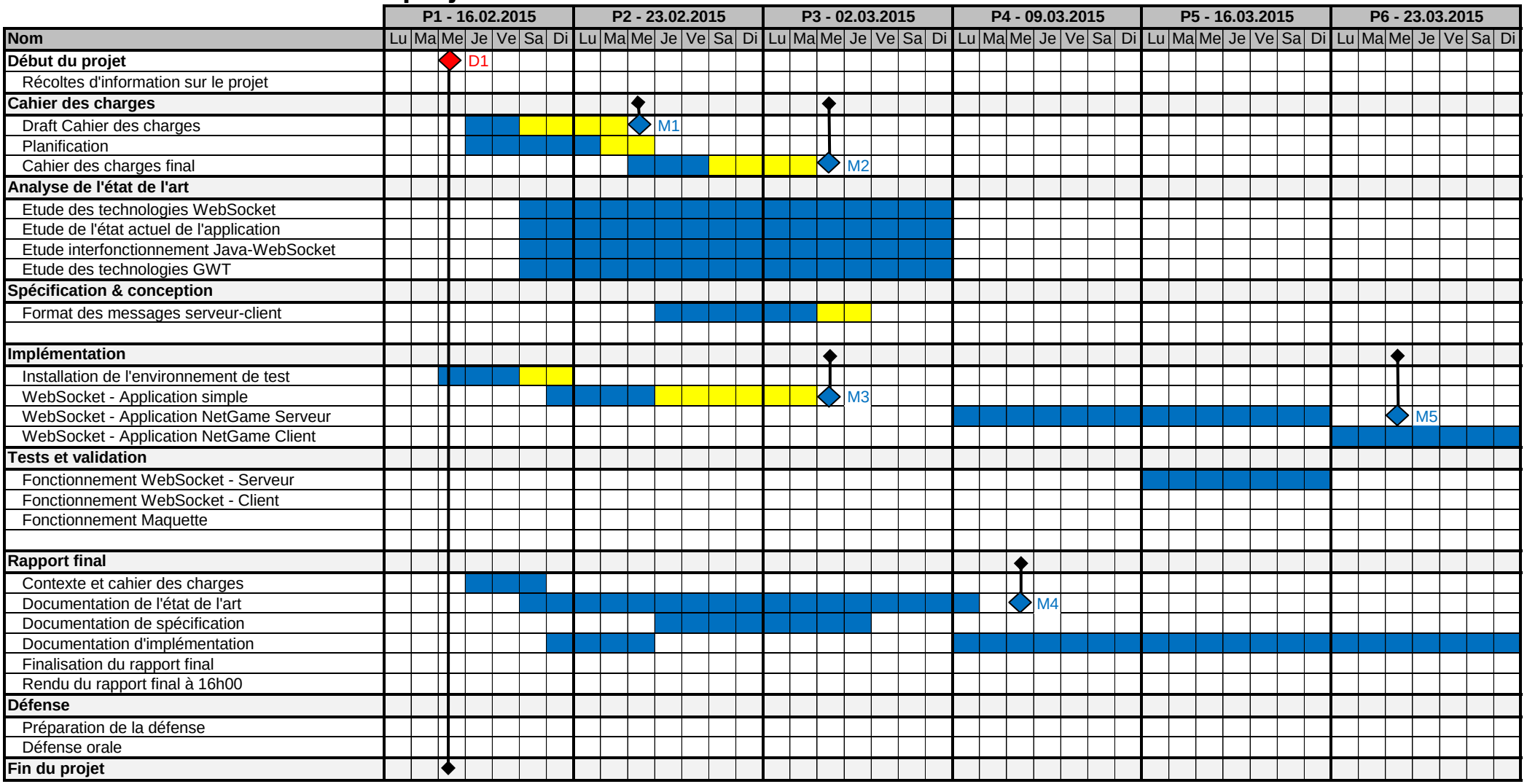
11. Annexes

1) Planification
2) Tests effectués
3) Journal de bord
4) Interfaces GUI

12. Glossaire

Abbréviation	Définition
HTML	HyperText Markup Language est le format de données conçu pour réaliser les pages Web
JS	JavaScript est un langage de programmation principalement utilisé côté client (navigateur Web)
CSS	Cascading Style Sheets est un lanage de programmation définissant la présentation des documents HTML
Java	Java est un langage de programmation haut niveau
WebSocket	WebSocket est une technologie amenée par le Web 2.0 permettant la communication bidirectionnelle entre un client (navigateur) et un serveur Web
PHL	Physical Layer est la couche physique (1) du modèle OSI
DLL	Data Link Layer est la couche liaison de donnée (2) du modèle OSI
NL	Network Layer est la couche réseau (3) du modèle OSI
IP	Internet Protocol
API	Application Programming Interface
JSR	Java Serving Request
OSI	Open System Interconnection
JSP	Java Server Page

Annexe 1. Planification du projet



Version 1,0
Date : 25.02.2015

Légendes:

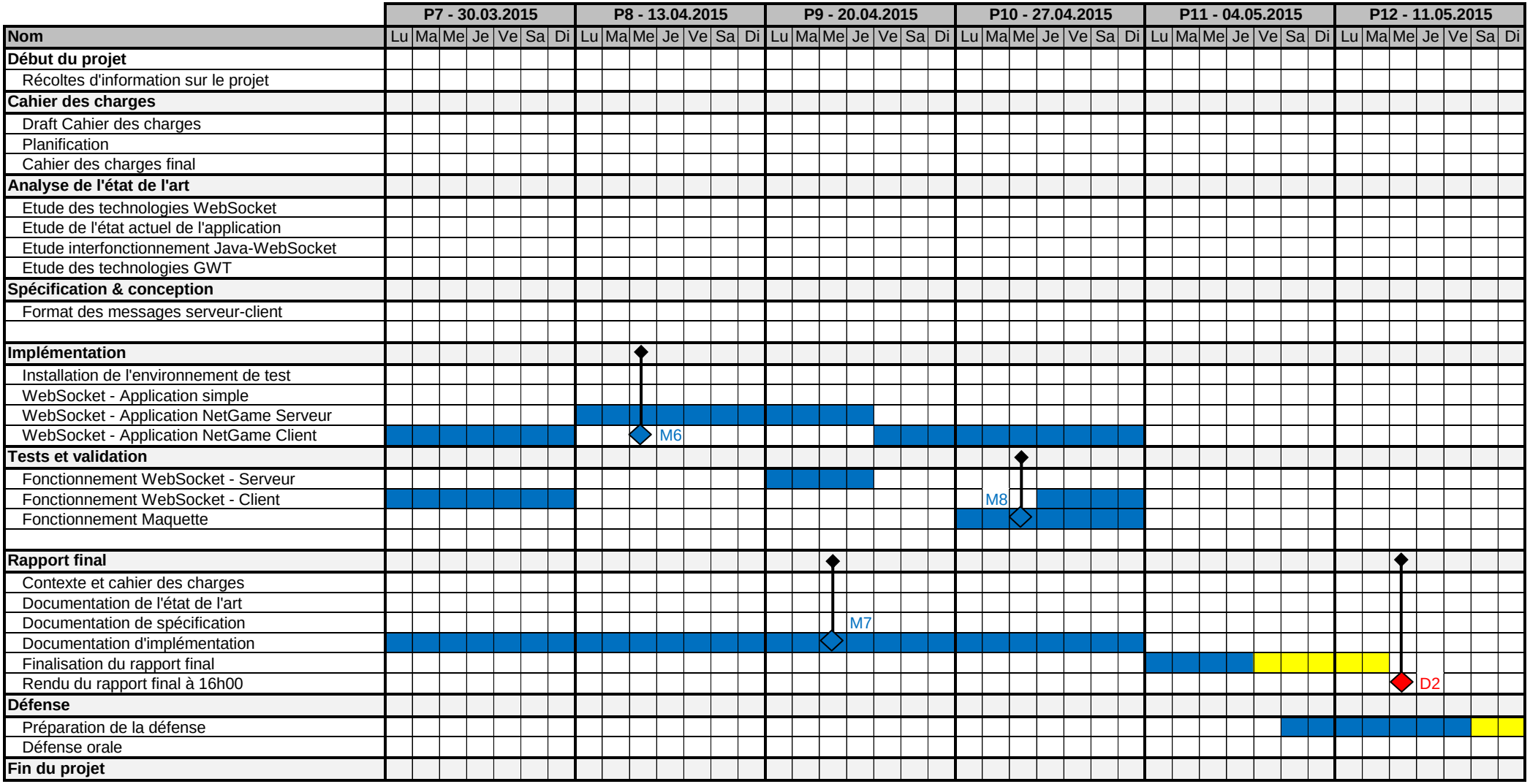
Temps planifié
Planifié
Marge
DeadLine
MileStone
Réel

D1 Début du projet
D2 Rendu du rapport final
D3 Défense orale
D4 Fin du projet

M1 Draft cahier des charges
M2 Cahier des charges
M3 Application Websocket
M4 Première version analyse
M5 WebSocket Server-interface Netgame
M6 WebSocket Client-Maquette
M7 Première version implémentation
M8 Démonstration maquette

David Rossier

Annexe 1. Planification du



Version 1,0
Date : 25.02.2015

Annexe 1. Planification du

Nom	P13 - 18.05.2015 Lu	Ma	Me	Je	Ve	Sa	Di
Début du projet							
Récoltes d'information sur le projet							
Cahier des charges							
Draft Cahier des charges							
Planification							
Cahier des charges final							
Analyse de l'état de l'art							
Etude des technologies WebSocket							
Etude de l'état actuel de l'application							
Etude interfonctionnement Java-WebSocket							
Etude des technologies GWT							
Spécification & conception							
Format des messages serveur-client							
Implémentation							
Installation de l'environnement de test							
WebSocket - Application simple							
WebSocket - Application NetGame Serveur							
WebSocket - Application NetGame Client							
Tests et validation							
Fonctionnement WebSocket - Serveur							
Fonctionnement WebSocket - Client							
Fonctionnement Maquette							
Rapport final							
Contexte et cahier des charges							
Documentation de l'état de l'art							
Documentation de spécification							
Documentation d'implémentation							
Finalisation du rapport final							
Rendu du rapport final à 16h00							
Défense							
Préparation de la défense							
Défense orale				D3			
Fin du projet				D4			

Version 1,0
Date : 25.02.2015

Annexe 2 : Tests et validation

N°	Description	Attendu	Test
100	**Fonctions d'enregistrement**		
101	Enregistrement accepté	Message de notification : Enregistré en tant que Nœud x, Layer y; GUI Logged	**OK**
102	Enregistrement refusé par le serveur	Message de notification : Message d'erreur retourné par le serveur; GUI Logout	**OK**
103	Désenregistrement d'un utilisateur non enregistré	Message de notification : Message d'erreur retourné par le serveur; GUI Logout	**OK**
104	Désenregistrement d'un utilisateur enregistré	Message de notification : "Registration for this node/layer cleared!"; GUI Logout	**OK**
105	Enregistrement alors que le jeu NetGame n'est pas initialisé	Un message d'erreur apparait.	**OK**
200	**Liste des messages**		
201	Nouveau message à traiter venant d'une couche supérieure	Le message apparait dans sa liste de messages à traiter; GUI ListeUp	**OK**
202	Nouveau message à traiter venant d'une couche inférieure	Le message apparait dans sa liste de messages à traiter; GUI ListeDown	**OK**
203	Nouveau message à traiter venant de la même couche (L1 only)	Le message apparait dans sa liste de messages à traiter; GUI ListeSameLayer	**OK**
204	Nouveau message dans la bufferQueue (L2 only)	Le message apparait dans sa liste de messages en mémoire; GUI BufferQueue	**OK**
205	Aucun message	Pas de messages affichés	**OK**
300	**Physical Layer**		
301	Paquet à destination du nœud (direction up)	Le message apparait dans GUI TraitementL1Up, et peut être traité par l'utilisateur	**OK**
302	Paquet partant du nœud (direction down)	Le message apparait dans GUI TraitementL1Down, et peut être traité par l'utilisateur	**OK**
303	Erreur dans le traitement (down only)	Un message d'erreur apparait.	**OK**
304	Paquet sortant : Numéro FCS donné faux	Le numéro FCS est enregistré dans la trame (Génère un NAK dans L2)	**OK**
305	Paquet sortant : Numéro FCS donné juste	Le numéro FCS est enregistré dans la trame (Génère un ACK dans L2)	**OK**
400	**Data Link Layer**		
401	Paquet à destination du nœud (direction up, DAT)	Le message apparait dans GUI TraitementL2 et peut être traité par l'utilisateur	**OK**
402	Paquet partant du nœud (direction down, DAT)	Le message apparait dans GUI TraitementL2 et peut être traité par l'utilisateur	**OK**
403	Acquittement à destination du nœud (direction up, ACK)	Le message apparait dans GUI TraitementL2 et peut être traité par l'utilisateur	**OK**
404	Acquittement partant du nœud (direction down, ACK)	Le message apparait dans GUI TraitementL2 et peut être traité par l'utilisateur	**OK**
405	Erreur dans le traitement (down)	Un message d'erreur apparait.	**OK**
406	Paquet sortant : Numéro de séquence spécifié par l'utilisateur	Le numéro de séquence est enregistré dans les couches inférieures	**OK**
500	**Network Layer**		
501	Paquet à destination du nœud (direction up)	Le message apparait dans GUI TraitementL3 et peut être traité par l'utilisateur	**OK**
502	Paquet partant du nœud (direction down)	Le message apparait dans GUI TraitementL3 et peut être traité par l'utilisateur	**OK**
503	Paquet entrant, à destination d'un autre nœud (direction down)	Le message apparait dans GUI TraitementL3 et peut être traité par l'utilisateur	**OK**
504	Paquet sortant : Destination spécifiée dans NextNode	L'information NextNode est conservée dans les couches inférieures.	**OK**
600	**Fonctions d'administration**		
601	L'administrateur suspend le jeu	Message de notification : "Game has been stopped by admin. Wait for instructions."; GUI Wait	**OK**
602	L'administrateur reset le jeu	Message de notification : "Reset by admin"; GUI Logout	**OK**
603	Suppression d'un enregistrement par l'admin	Message de notification : "L'administrateur a supprimé votre enregistrement."; GUI Logout	**OK**
604	Suppression de tous les enregistrements par l'admin	Message de notification : "L'administrateur a supprimé votre enregistrement."; GUI Logout	**OK**
605	L'administrateur envoie une notification	Message de notification : "{Contenu de la notification}";GUI Notification	**OK**
606	L'administrateur process un message	Le message disparait de la liste du joueur	**OK**
607	L'administrateur change le Routing	Message	**OK**
800	**Interface Monitor**		
801	Affiche la connectivité en temps réel	Les interconnexions entre les nœuds se mettent à jour lors d'une modification de la connectivité	**OK**

802	Affiche les pseudos des joueurs	Les pseudos se mettent à jour lors d'un enregistrement / désenregistrement	**OK**
803	Affiche le nombre de messages à traiter par couche	Le nombre de message dans les files buffer/messageQueue se mettent à jour à chaque nouveau paquet	**OK**

Annexe 3 : Journal de bord

Date	Activité	Durée
18.02.2015	Séance de Kickoff, administration	1,0
	Recherche Bootstrap & Websockets	2,0
	Installation d'une VM pour développement et tests	3,0
19.02.2015	Cahier des charges	3,0
	Invitation séance 2	0,5
	Planification	1,0
25.02.2015	Séance 2	0,5
	Administration	2,0
	Problème examples tomcat not working	2,0
	Cahier des charges	2,0
	Planification	0,5
	Recherche différence tomcat http://tomcat.apache.org/tomcat-8.0-doc/changelog.html	0,5
26.02.2015	Finalisation cahier des charges	1,0
	Schémas de fonctionnement	1,0
	Application Java WebSocket simple	0,0
04.03.2015	Réinstallation de l'environnement : Bug Ubuntu passé sur Debian	1,0
	Séance 3, administration	1,0
06.03.2015	Etude du code de l'ancienne application	3,0
11.03.2015	Séance 4, administration	2,0
	Installation et configuration eclipse	2,0
12.03.2015	Tests de configuration des examples	2,0
	Bug : Les examples ne fonctionnent pas lorsqu'ils sont compilés par eclipse	2,0
14.03.2015	Recherche sur le problème, tests de différentes solutions proposées	4,0
	Solution : Manque la librairie "annotation-api.jar" de tomcat.	1,0
16.03.2015	Problème : Les classes de NetGame ne sont pas reconnues par l'application	1,0
	Solution : Inclure les class dans le buildpath "add classes"	1,0
	Version n'existe pas dans les fichiers serveurs; utilisation de GetTrafficData	1,0
18.03.2015	Administration	1,0
	Séance 5, PV, Invitation	1,0
	Finalisation de l'interface client	1,0
19.03.2015	Spécifications & discussion spécification	2,0
	Nouvelle architecture	1,0
25.03.2015	Administration, séance 6, PV, Invitation	2,0
	Spécification, recherche de solutions Context	1,0
26.03.2015	Fin Spécification (étude du format des messages des nœuds)	2,0
	Implémentation NetGameWS (unicast, multicast, broadcast des notifications)	2,0
	Implémentation de NetGameTest (http Servlet remplaçant NetGame le temps des tests)	1,0
	Choix des formats de messages et tests avec JSON (non fonctionnels)	1,0
27.03.2015	Fonctions d'enregistrement avec une base JSON (Crockford)	2,0
	Application de tests NGTest fonctionnant avec JSON (unicast, multicast, broadcast)	2,0
29.03.2015	Implémentation et tests de l'application	2,0
	Modification des spécifications pour correspondre	
30.03.2015	Implémentation et tests de l'application	2,0
01.04.2015	Finalisation de l'application	1,0
	Séance 7, administration, PV, Invitation	1,5
	Spécifications : Interface NGWS => NG	1,0
02.04.2015	Spécifications : Interface client <=> NGWS	2,0
10.04.2015	Spécifications : Mise à jour différentes interfaces	1,0
	Début d'implémentation GUI	2,0
11.04.2015	Spécification : Mise à jour selon le NGWSBridge	3,0
	Implémentation des fonctions du serveur	1,0
13.04.2015	Implémentation d'un GUI simple HTML / JS	2,0
14.04.2015	Spécification : Mise à jour après discussion avec M.Scheurer	2,0
	Implémentation des fonctions simples du serveur (notifications)	1,0
15.04.2015	Administration / Séance 8, PV, Invitation	2,0
	Spécification : Dernières modifications	1,0

16.04.2015	Développement GUI & SRV	4,0
17.04.2015	Développement GUI & SRV	2,0
18.04.2015	Développement GUI & SRV	2,0
19.04.2015	Développement GUI & SRV	2,0
21.04.2015	Développement GUI & SRV	5,0
22.04.2015	Développement GUI & SRV	2,0
	Séance 9, administration, PV, invitation	1,0
	Documentation de tous les cas du NetGame client actuel	1,0
23.04.2015	Développement GUI & SRV	7,0
24.04.2015	Développement GUI & SRV	2,0
25.04.2015	Développement GUI & SRV	2,0
27.04.2015	Développement GUI & SRV	5,0
28.04.2015	Développement GUI & SRV	4,0
29.04.2015	Finalisation GUI & SRV	5,0
	Séance 10	1,0
30.04.2015	Correction Encoding & doc	1,0
	Administration	1,0
	Gestion de la langue & Finalisation interface GUI	2,0
	Discussion sur le code et problème d'encoding	1,0
01.05.2015	Documentation : Implémentation ; Langues	2,0
03.05.2015	Documentation : Correction SPEC	1,0
	Documentation : Implémentation ; Encoding & changedValue	1,0
04.05.2015	Documentation : Analyse : WS	1,0
05.05.2015	Documentation finale : Structure	1,0
06.05.2015	Documentation : Analyse : Liste des fonctions à intégrer au WebSocket & NetGameServiceImpl	1,0
	Complément au document d'analyse	2,0
09.05.2015	Complément d'analyse : WebSockets	2,0
10.05.2015	Documentation : Conception	2,0
10.05.2015	Implémentation : Interface Monitor	4,0
12.05.2015	Documentation : Interface Monitor & Implémentation	2,0
	Relecture et finalisation	1,0
12.05.2015	Annexes : Mise en forme	0,5
	Documentation WebSocket	2,0
	Documentation Implémentation	1,0
	Total	158,0

Annexe 4. Interfaces GUI

Cette annexe contient toutes les interfaces graphiques référencées dans le rapport final.

4.1. GUI Logout

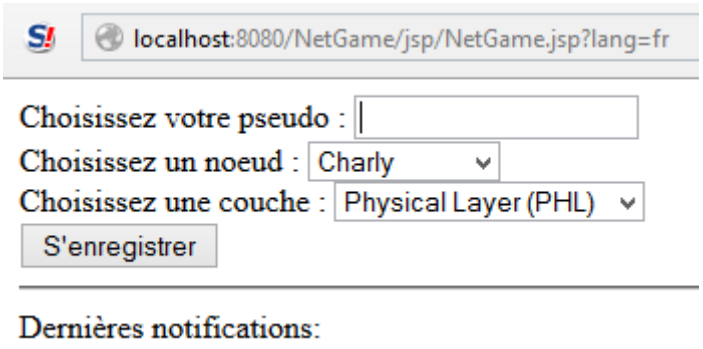


Figure 1 : GUI Logout

4.2. GUI Logged L1, L3

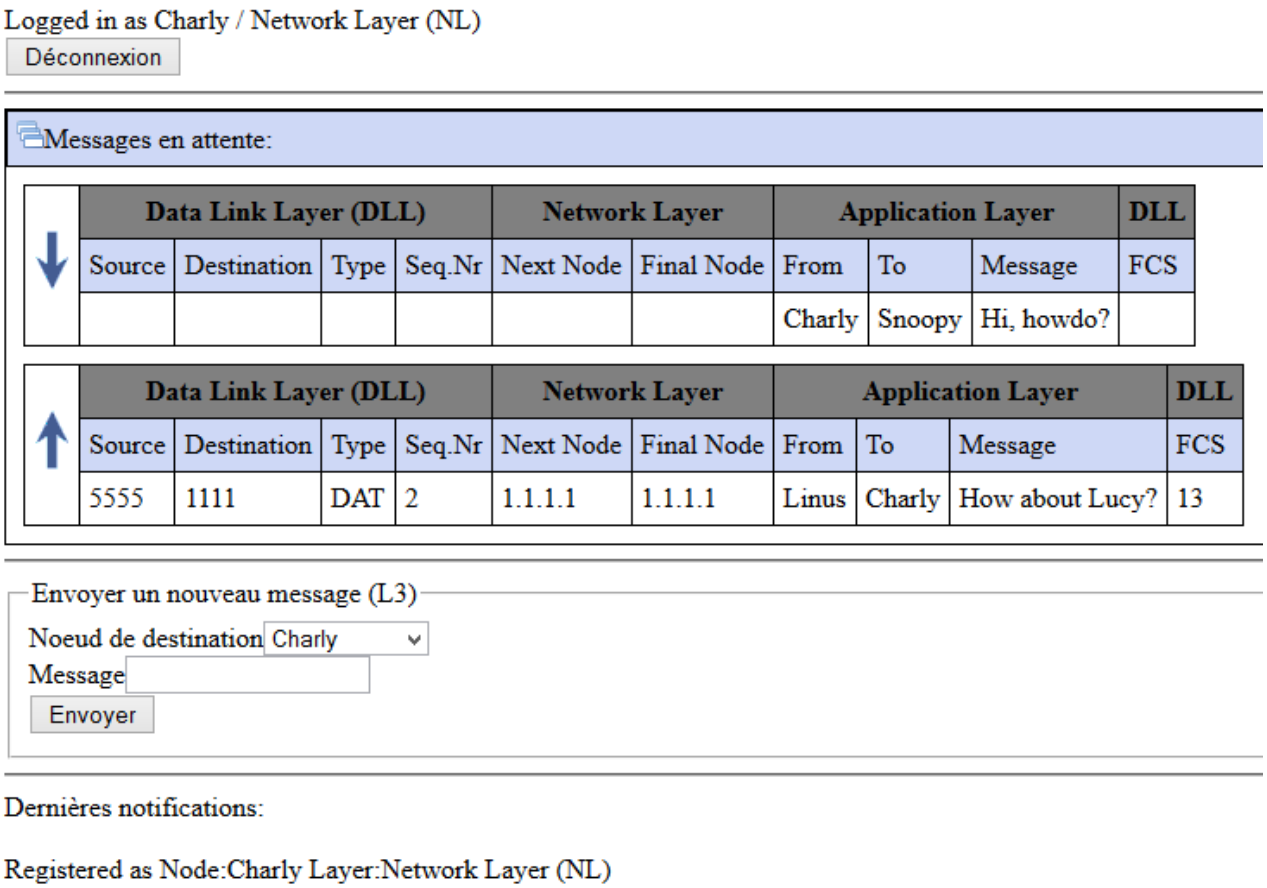


Figure 2 : GUI Logged L1 L3

4.3. GUI Logged L2

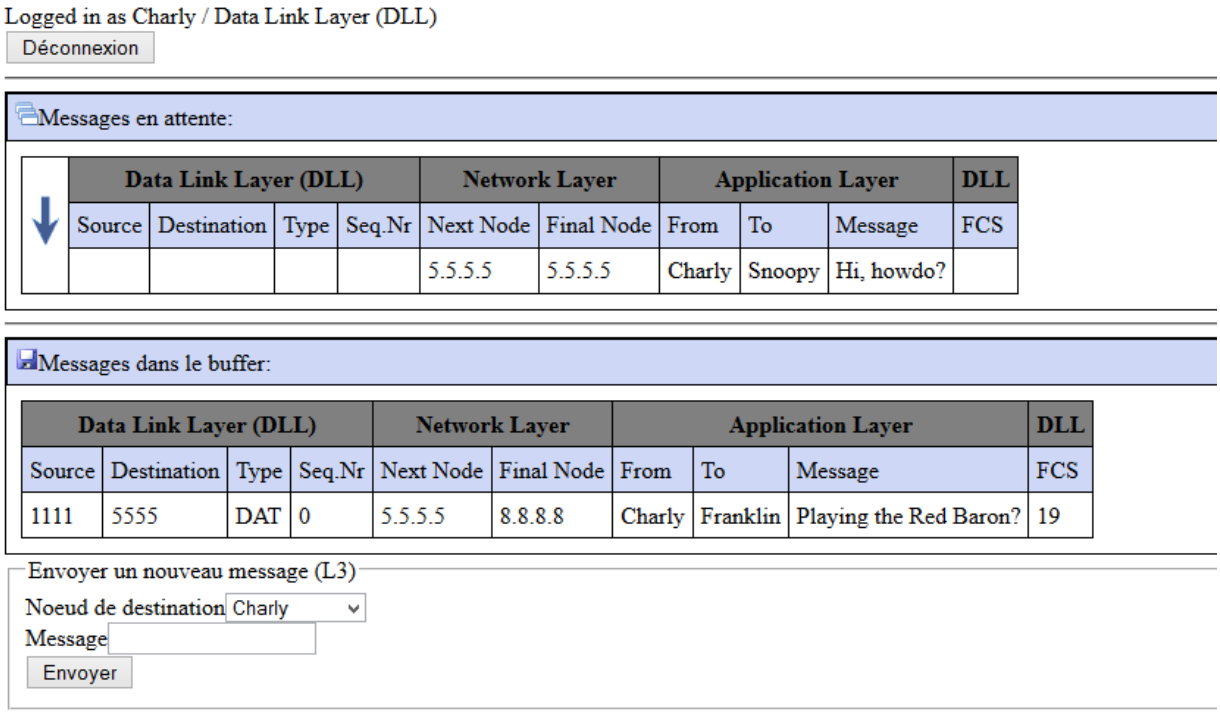


Figure 3 : GUI Logged

4.4. GUI Liste

	Data Link Layer (DLL)				Network Layer		Application Layer			DLL
→	Source	Destination	Type	Seq.Nr	Next Node	Final Node	From	To	Message	FCS
	5555	1111	ACK	1						

Figure 4 : Direction : Same Layer

	Data Link Layer (DLL)				Network Layer		Application Layer			DLL
↓	Source	Destination	Type	Seq.Nr	Next Node	Final Node	From	To	Message	FCS
	1111	5555	DAT	2	5.5.5.5	5.5.5.5	Charly	Snoopy	Hi, howdo?	9

Figure 5 : Direction : bottom

	Data Link Layer (DLL)				Network Layer		Application Layer			DLL
↑	Source	Destination	Type	Seq.Nr	Next Node	Final Node	From	To	Message	FCS
	1111	5555	DAT	1	5.5.5.5	8.8.8.8	Charly	Franklin	Playing the Red Baron?	19

Figure 6 : Direction : Top

4.5. GUI Notification

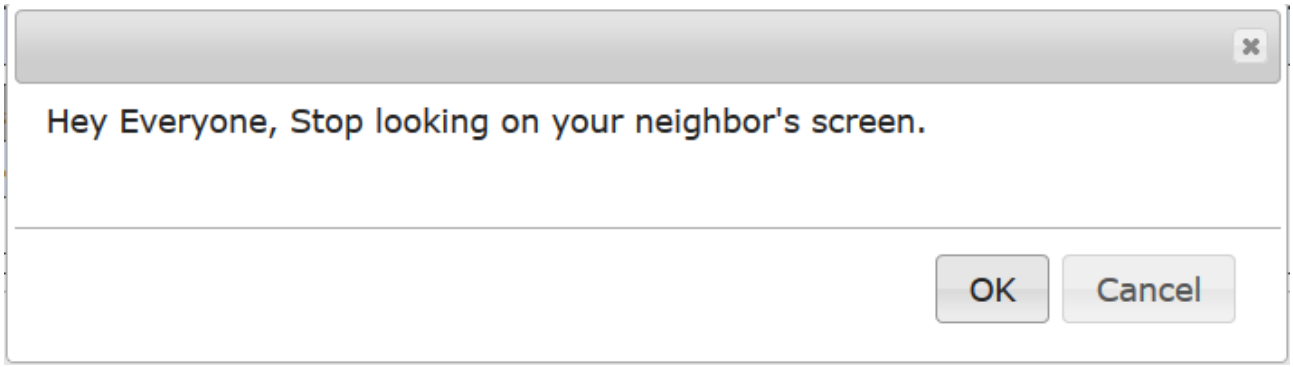


Figure 7 : GUI Notification

4.6. GUI Traitement

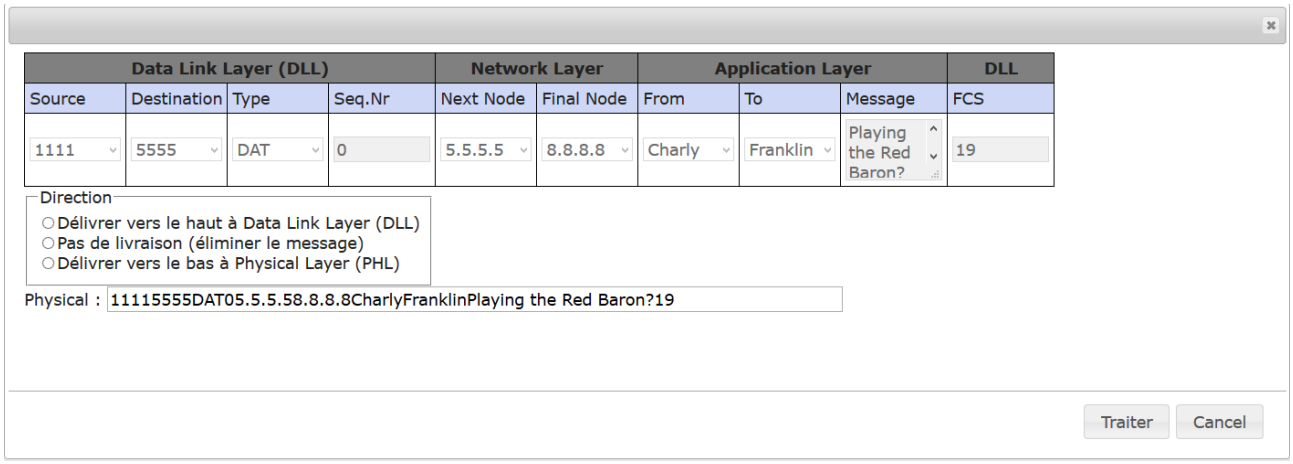


Figure 8 : GUI Traitement, Layer 1, Direction : Down

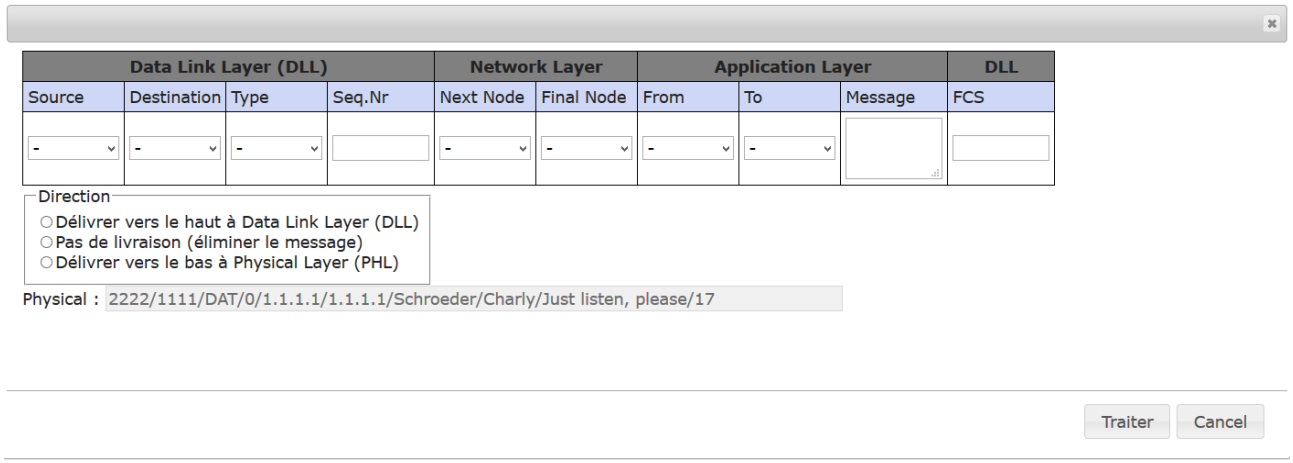


Figure 9 : GUI Traitement, Layer 1, Direction : Up

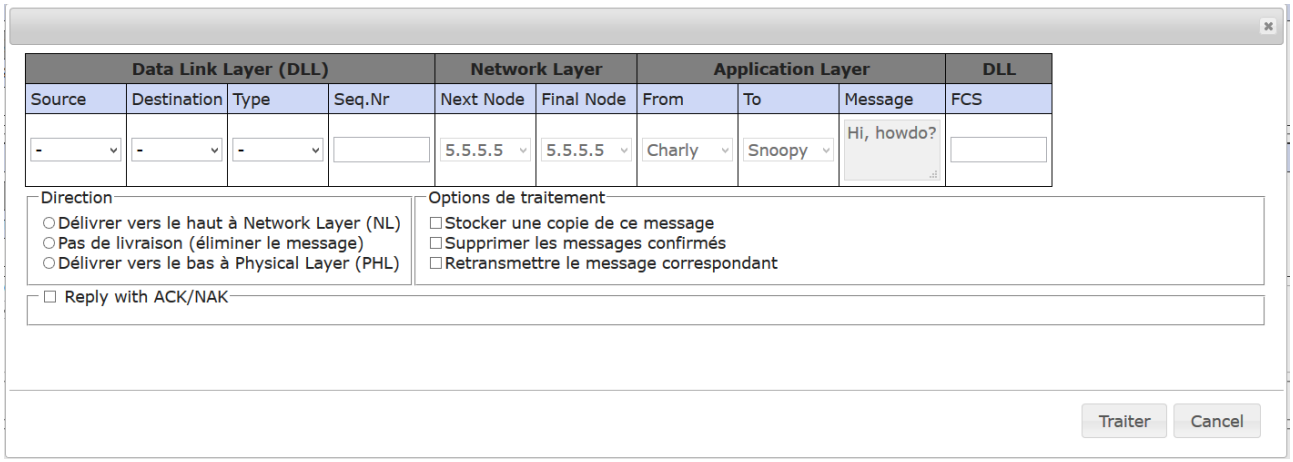


Figure 10 : GUI Traitement Layer 2

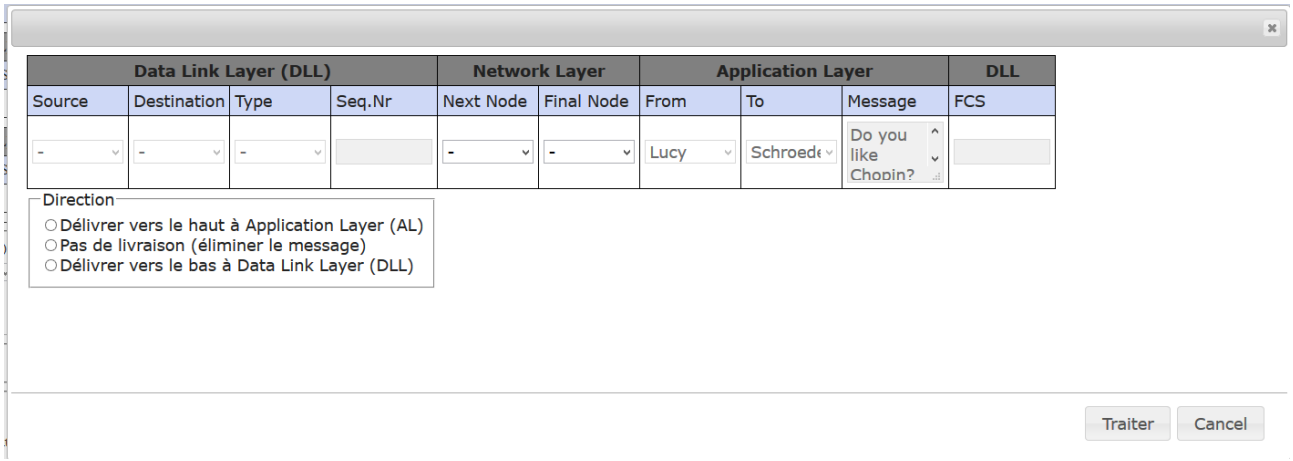


Figure 11 : GUI Traitement Layer 3

4.7. GUI Jeu suspendu

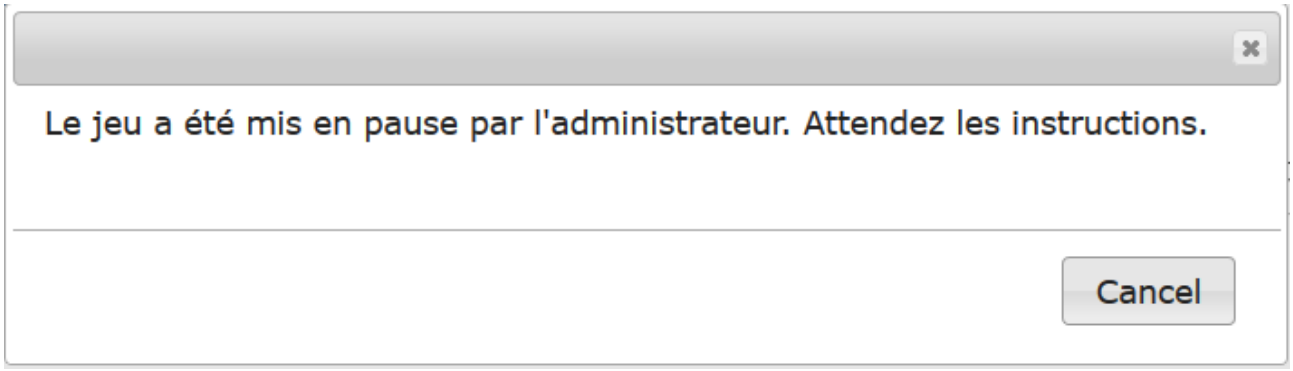


Figure 12 : GUI Jeu suspendu