
# TCP avec UPPAAL

Encadrants : Marie Duflot-Kremer, Stephan Merz
Préparé par : Nicolas Blin, Driss Amrani
2 février 2018

# Sommaire de Projet

## Objectif

Nous nous sommes fixé comme objectif de réaliser ce projet, d'une complexité non-négligeable, en procédant par itération de version. Ainsi nous allons présenter dans ce rapport les différentes étapes successives majeurs qui nous amèneront possiblement à une version aboutie et réaliste.
Ce protocole étant encore une fois complexe, nous décidons de nous consacrer plus sérieusement que sur un seul aspect que nous détaillerons par la suite.

## Contexte du projet

Nous nous sommes intéressé à TCP de part le fait qu'il s'agit d'un protocole qui est réellement utilisé et qui demeure très difficile à optimiser aujourd'hui. Ce nombreux algorithmes existent tels que : Reno, new Reno, Vegas, etc., qui essayent dans certains contextes particuliers d'être dans cette optimalité.
Nous avons donc décidé de voir jusqu'où et dans quelles mesures il pouvait être utilisé/vérifié avec UPPAAL.

## Termes et définitions

TCP désigne simplement le nom d'un protocole permettant un échange fiable entre deux entités sur un réseau ou canal de communication. Il est décomposable en trois parties distinctes :

- Phase d'établissement de la connexion (Figure 1)
- Phase d'échange de messages (Figure 2)
- Phase de fin de la connexion (Figure 3)

C'est principalement sur la phase d'échange de messages/paquets que nous travaillerons. Les deux autres, bien que très intéressantes également, ne sont pas au coeur de la problématique d'optimisation des implémentations du protocole.

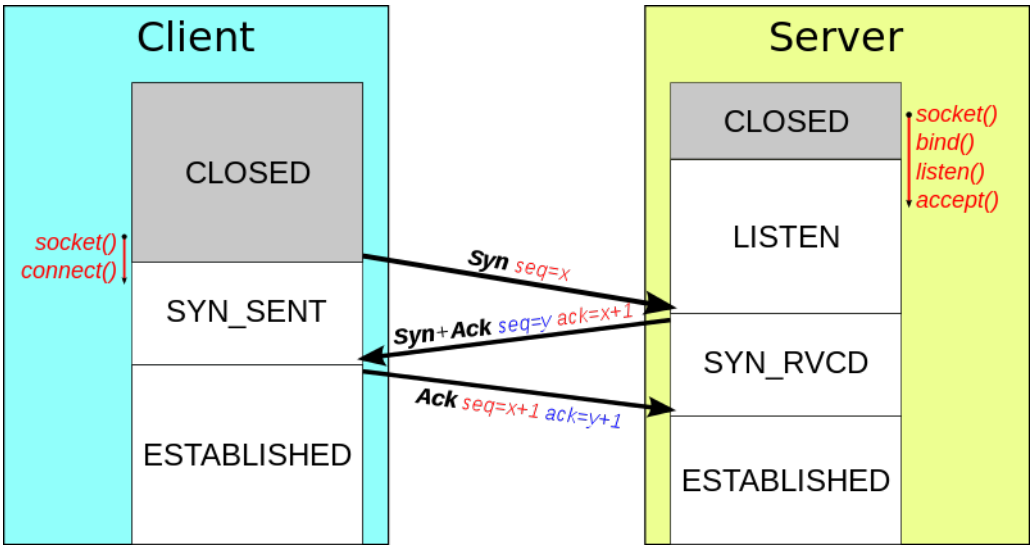


Figure 1 : Ouverture de la connexion

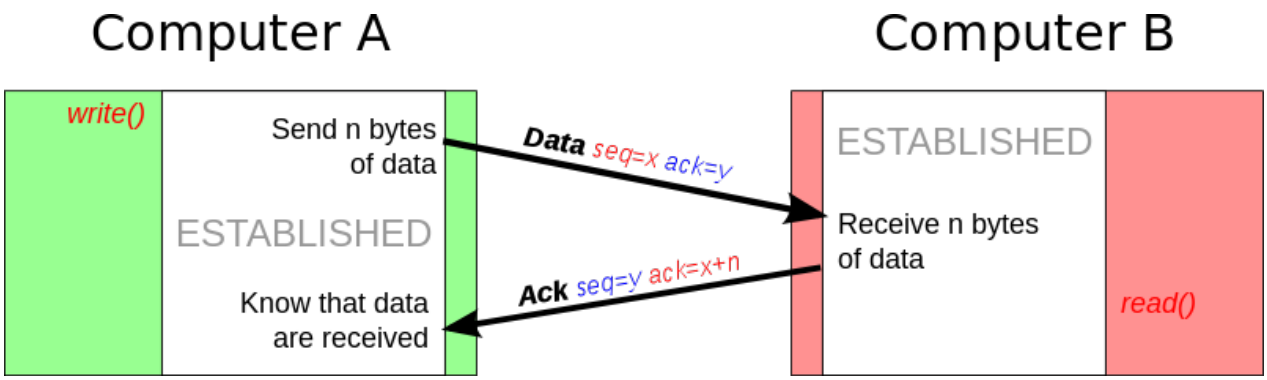


Figure 2 : Les échanges de messages

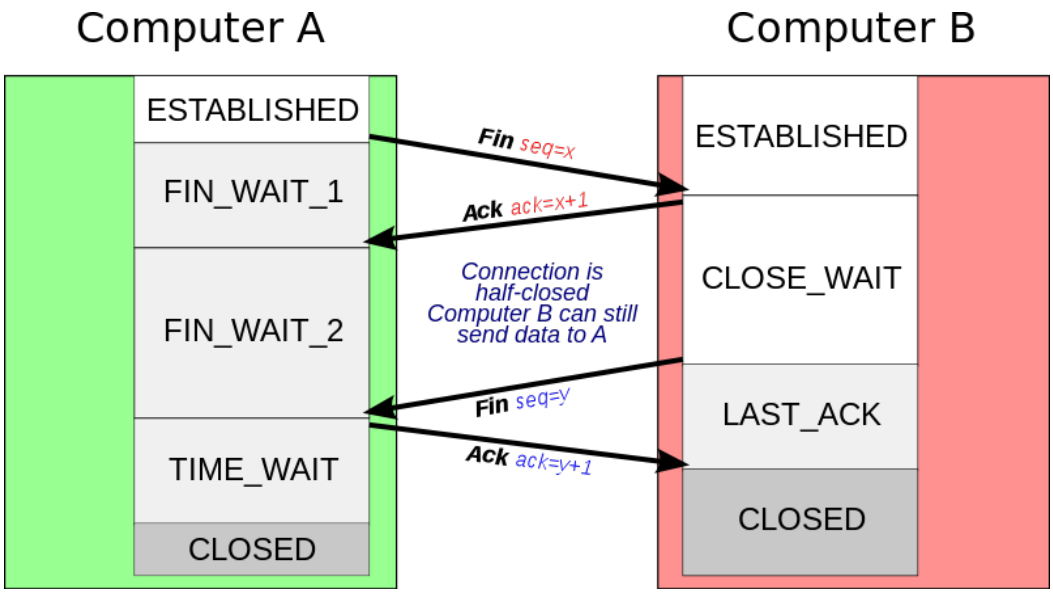


Figure 3 : Fin de la communication

# CE QU'IL FAUT SAVOIR

Le protocole TCP n'étant pas forcément évident à comprendre au premier coup d'oeuil, nous allons donner ici l'algorithme général, ainsi que des notions nécessaires sur la phase d'échange de paquets.

**Algorithme :**

1. Handshake en 3 temps
2. LOOP sur le nombre de messages à envoyer (un par un)
   - Client ⇨ Serveur :

     Flag DATA = true;
     Flag ACK = false;
     N° ack = 1;
     N° seq = 0;
     envoie(message);
     Si tout se passe bien Alors
     - Serveur ⇨ Client :

       Flag DATA = false;
       Flag ACK = true;
       int tmpSwap = N° ack;
       N° ack = N° seq + n;
       N° seq = tmpSwap;
       envoie(acquittement);
       Si tout se passe bien Alors
       - C'est fini;

       Sinon
       - TTL++;

     Sinon
     - TTL++;

     Fin Si
     Si TTL trop grand Alors
     - recommencer(Client ⇨ Serveur);

     Fin Si

   Fin LOOP
3. Fin de connexion

TCP est composé d'une multitude de Flags nécessaires sa la fiabilité. Ces Flags permettent aux deux parties de s'échanger leurs états respectifs.
Le TTL (time to live) représente la durée de vie d'un message et par conséquent le temps qu'attendra le Client avant de renvoyé un message identique. Dans notre cas, nous fixerons le TTL à une valeur arbitraire que l'horloge de l'automate aura comme plafond à la durée de renvoie des messages.
On peut voir aussi des numéros de séquence ainsi que des numéros d'acquittement. Ils vont permettre dans une version plus avancée à déterminer un numéro d'envoie de message et un numéro d'acceptation de message. Si plusieurs messages sont envoyés en même temps sur le réseau, ces deux nombres vont permettre d'identifier quel message est reçu et ainsi permettre de savoir l'ordre dans lequel ces messages ont été envoyés. N'oublions pas que le protocole TCP est un protocole fiable et que par conséquent l'ordre des messages est important.

# VERSION 1

## Objectif

Lors de cette première version, nous avons décidé de ne pas aller trop loin dans le protocole en le simplifiant au maximum. Cette simplification avait pour but de simplement vérifier que l'envoi d'un seul message était fiable. Mais qu'est-ce que la fiabilité ?
Pour ce premier lancé, nous avons défini la fiabilité comme étant la certitude que l'envoi d'un message serait bien acquitté malgré les perturbations que nous allions injecter.

## Mise en place

Nous allons tout d'abord commencer à mettre en place l'ouverture de connexion entre les deux entités que nous appellerons Client et Serveur, ou respectivement Machine A et Machine B.
Puis nous verrons comment effectuer la terminaison d'une connexion.
Finalement nous verrons la partie qui nous intéresse principalement; l'échange de données.
Une réflexion que nous avons eu était de savoir comment créer la perturbation dont nous avons parlé un peu plus tôt. Le choix s'est finalement porté sur ces deux principaux éléments :

- Le réseau par lequel transit les messages est lent. Ainsi un Client pourrait avoir à attendre un acquittement pendant des périodes définies plus ou moins longues. De même le Serveur pourrait attendre un message du réseau.
- Des messages vont disparaître ou bien se détruire lors des échanges. C'est ainsi que nous avons introduis un Voleur dans notre protocole. Son rôle sera d'altérer ou de détruire certains messages/paquets.

Ne venons nous pas ainsi de créer une nouvelle entité à implémenter ?
C'est donc sur la base de 3 automates (Voleur, Réseau, Machine), que nous sommes arrivés à compléter la fameuse V1 que nous attendions.

# HANDSHAKE EN 3 TEMPS

## Définition

Le handshake en 3 temps (Figure 1) est la première phase qui permet d'établir le canal de communication entre le Client et le Serveur.
La première difficulté rencontrée vient du fait qu'une communication dans ce genre de protocole peut être bidirectionnelle. Autrement dit, il n'y a pas de différence entre le Client/Machine A et le Serveur/Machine B.

## Observation et réalisation

Si on regarde de plus prêt la Figure 1, nous pouvons remarquer que le Serveur est dans l'état Listen avant même que le Client lui ai envoyé un message. C'est parce que nous avons choisi de représenter un serveur en mode passif, qui se contentera d'accepter les communications du premier à le demander.
Ce choix nous a simplifié la tâche dans le sens où la différence entre le Client et le Serveur se ferra par cette dissociation d'état.

La Figure 4 représente l'implémentation (basique) que nous avons faite de l'automate représentant les deux machines. On pourrait analyser le patron comme si la partie du haut représentait le Client et la partie du bas le Serveur. Ils partagent en commun l'état Established qui permet la connexion entre les deux machines et ainsi le début des échanges de paquets "intéressants".

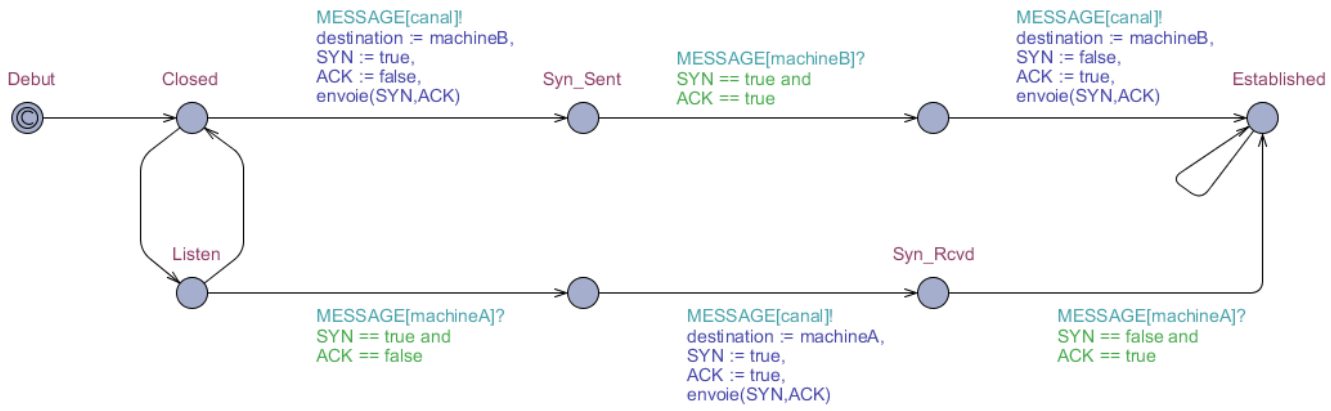


Figure 4 : Implémentation Handshake

Sur la Figure 5 nous pouvons voir le réseau tel qu'il a été représenté.

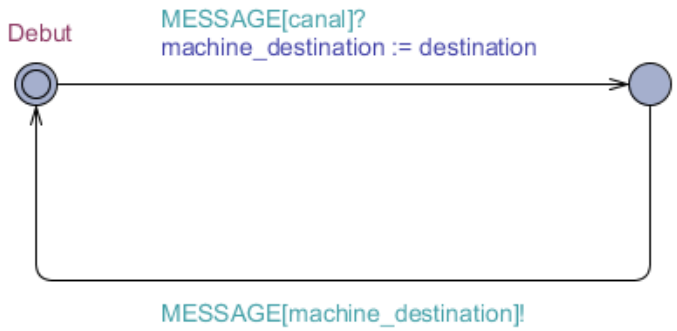


Figure 5 : Implémentation handshakes côté Reseaux

Nous avons fait trois vérifications pour cette partie du protocole, elle permettent de vérifier 3 choses :

- Il n'y a aucun deadlock possible
- Vérifie qu'un lien peut être établi entre le Client et le Serveur (à un moment donné ils sont tous les deux dans l'état Established)
- Une machine ne dois pas être Established alors qu'une autre est dans les états Closed ou Listen ou Syn_Sent. Si cette condition n'est pas remplie, la phase d'établissement de connexion n'est pas certaine d'ouvrir un canal de communication entre la Machine A et la Machine B.

Ces requêtes sont visibles sur la Figure 6.

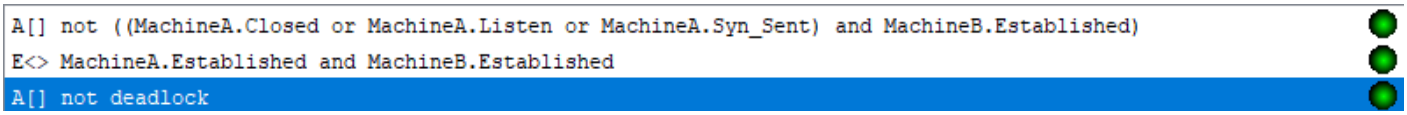


Figure 6 : Requêtes du vérifieur

# TERMINAISON D'UNE CONNEXION

## Définition

La phase de terminaison d'une connexion utilise un handshaking en quatre temps, La Machine A et la Machine B peuvent tous les deux décider de la fin de la connexion. Elle utilise donc le même principe que pour l'ouverture. Les paquets échangés lors de cette phase seront composés des deux booléens FIN et ACK; ainsi que des deux entiers seq et ack (Figure 3).

## Observation et réalisation

La Machine A et la Machine B s'occupent toutes les deux de la fermeture de la connexion. Dans un premier temps, ces dernières se retrouvent dans l'état "Established". Ils commencent à envoyer des messages entre eux pour s'informer de l'état de la connexion sans prendre  en compte les messages perdus. L'architecture de l'implémentation a été basée sur l'implémentation de l'ouverture de connexion. La figure ci-dessous représente la fermeture de connexion entre les deux machines.

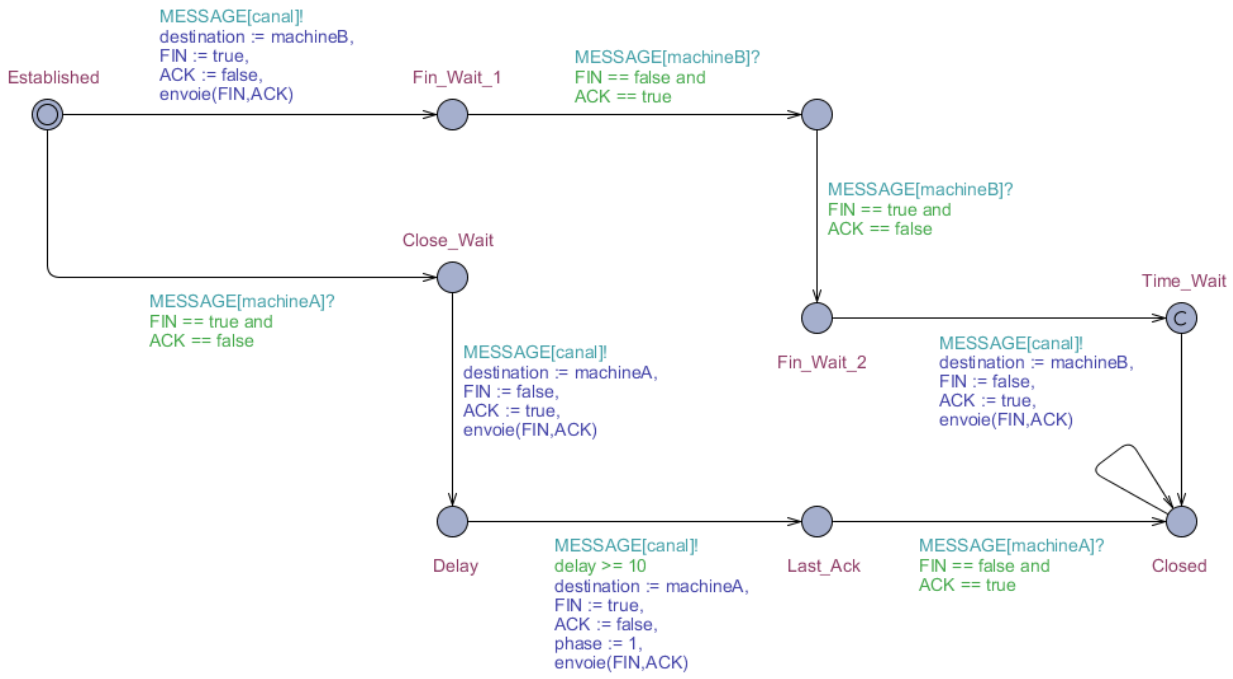


Figure 7 : Implémentation de la terminaison d'une connexion

Le réseau sera le même pour la fermeture de la connexion.

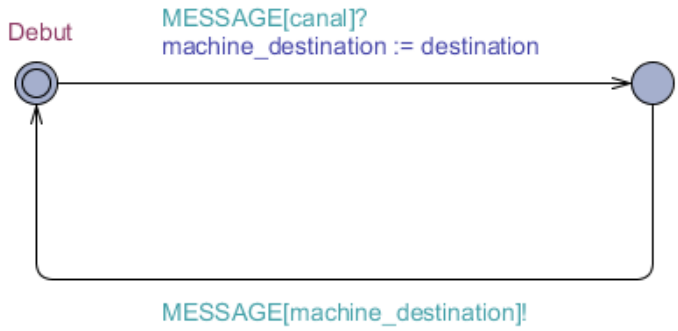


Figure 8: Implémentation de la fin de connexion côté Réseau

Nous avons fait trois vérifications pour cette partie du protocole, elle permettent de vérifier 3 choses :

- Il n'y a aucun deadlock possible
- Vérifie que la fermeture peut bien s'effectuer entre le Client et le Serveur (à un moment donné ils sont tous les deux dans l'état Closed)
- Une machine ne dois pas être Closed alors qu'une autre est dans les états Establised ou Fin_Wait_1 ou Fin_Wait_2 ou Tim_Wait. Si cette condition n'est pas remplie, la phase d'établissement de connexion n'est pas certaine d'ouvrir un canal de communication entre la Machine A et la Machine B.

Ces requêtes sont visibles sur la Figure 9.

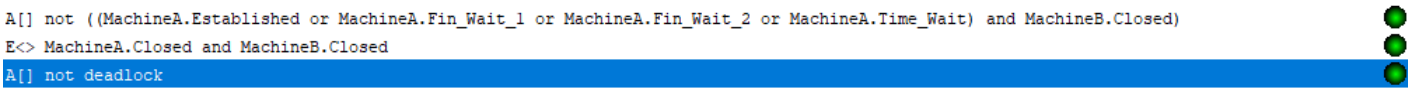


Figure 9 : Requêtes du vérifieur

# Échange des données:

## Définition

Tout échange d'information repose nécessairement sur un ensemble de conventions partagées entre l'émetteur et le destinataire d'un message : il faut que l'un et l'autre sachent notamment à quel moment commence la communication, selon quelles procédures elle s'effectue, et à quel moment elle se termine. Pour cela, on a défini deux machines A et B qui permettent la simulation de ce protocole avec un Voleur A qui permet de perturber les échanges de données.

## Observation et réalisation

Comme le montre la figure ci-dessous, l'échange de données se base sur un mode de communication client-serveur. Dans cette version, nous nous sommes basé sur l'envoie et la réception d'un seul message. Donc on se retrouve devant deux possibilités :

- La première possibilité : Envoie du message de la MachineA qui sera dans ce cas le Client vers la machine B (le Serveur) sans intervention de la part du voleur.

- La deuxième possibilité : Injection du Voleur dans le protocole de communication de données, ce qui nécessite une procédure de renvoi du message (voir la figure ci-dessous l'état "Data_Send2").


Figure 10 : Phase d'échange de données

Un réseau est indispensable pour envoyer un message d'une machine à une autre. Pour cela  nous avons utilisé un automate Réseau, comme montre la figure ci-dessous, avec une horloge (clock) qui permet d'augmenter le temps d'échanges de données entre les 2 machines.

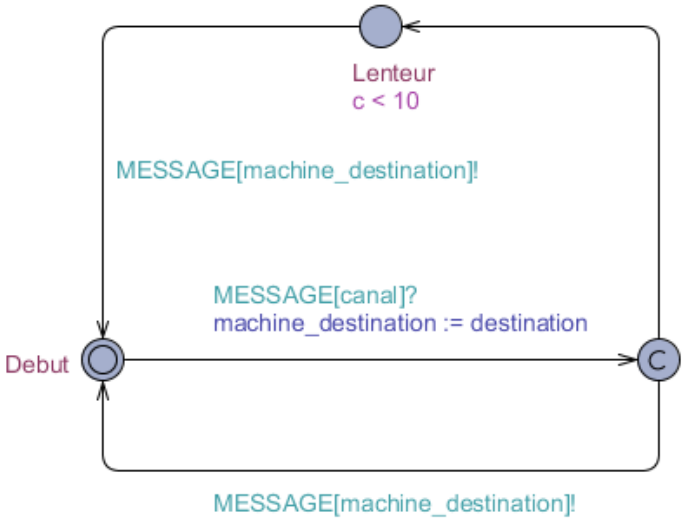


Figure 11 : Implémentation du réseau pour l'échange des données

Nous avons fait deux vérifications pour cette partie du protocole, elle permettent de vérifier 2 choses :

- Il n'y a aucun deadlock possible
- Il existe un jour un message envoyé depuis une machineA qui sera reçu surement par la machine B.

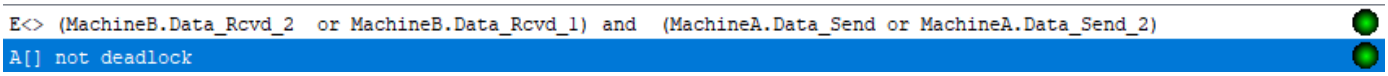


Figure 12 : Requêtes du vérifieur

# VERSION 2

## Objectif

Cette fois-ci nous tenterons de mettre en place un système de communication entre une Machine A et une Machine B, qui s'enverraient de 1 à N messages simultanément. Il faut donc prévoir qu'une perturbation peut s'effectuer à n'importe quel moment. Ainsi le Voleur pourra décider de s'emparer du message $i \in [1..N]$. De même l'horloge pourra atteindre le TTL pour n'importe quel message *i*. Une fois que ces vérifications seront en place, il faudra vérifier l'ordre dans lequel ces messages seront retrouvés par la Machine A. En effet si le Client venait à recevoir le message 2 avant le 1, il faudrait qu'il recommence l'envoie des N messages.

## Implémentation

La complexité de tout ces automates étant trop longue à implémenter pour le temps qui nous était imparti, nous n'avons pas pu aboutir à une version finale. Malgré le fait qu'il s'agisse réellement de la partie que nous voulions à tout prix voulu finir, nous n'avons pas réussi à dépasser la complexité de l'automate.
Ainsi nous parlerons de ce que nous avons réussi à implémenter et les problèmes qu'il nous reste à corriger.

L'envoie de plusieurs messages sur le réseau fonctionne, ces envois en parallèle sont possible grâce à la démultiplication du nombre d'automate "Reseaux". Nous avons dans notre version instancié 3 de ces automates.

```
// déclaration des réseaux, ce qui revient à déclarer le nombre de messages en parallèle
Reseau1 = Reseaux(0);
Reseau2 = Reseaux(0);
Reseau3 = Reseaux(0);
// déclaration des machines communicantes
MachineA = Machines(1,2,0);
MachineB = Machines(2,1,0);
// déclaration du voleur
Voleur1 = Voleurs(0);

// définition du système d'automates
system Reseau1, Reseau2, Reseau3, MachineA, MachineB, Voleur1;
```

Figure 13 : Instanciations

Ces automates ne sont pas différents des précédents :

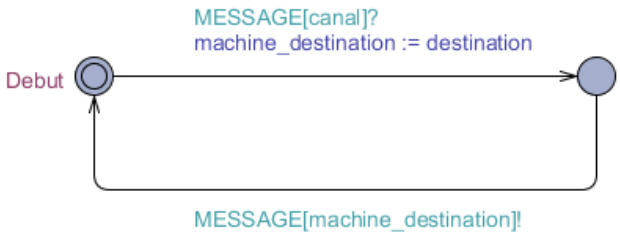


Figure 14 : Automate Reseaux

## Problèmes

Le principal problème que nous avons rencontré est, comme nous l'avons dit, la complexité de l'automate "Machines". Tel que présenté dans la Figure 13, il ne peut qu'effectuer l'envoi de N messages de façon désordonnée (ce qui n'est pas un problème s'il s'agit de l'envoie du Client vers le Serveur) et résister au vol de message qu'une seule fois.

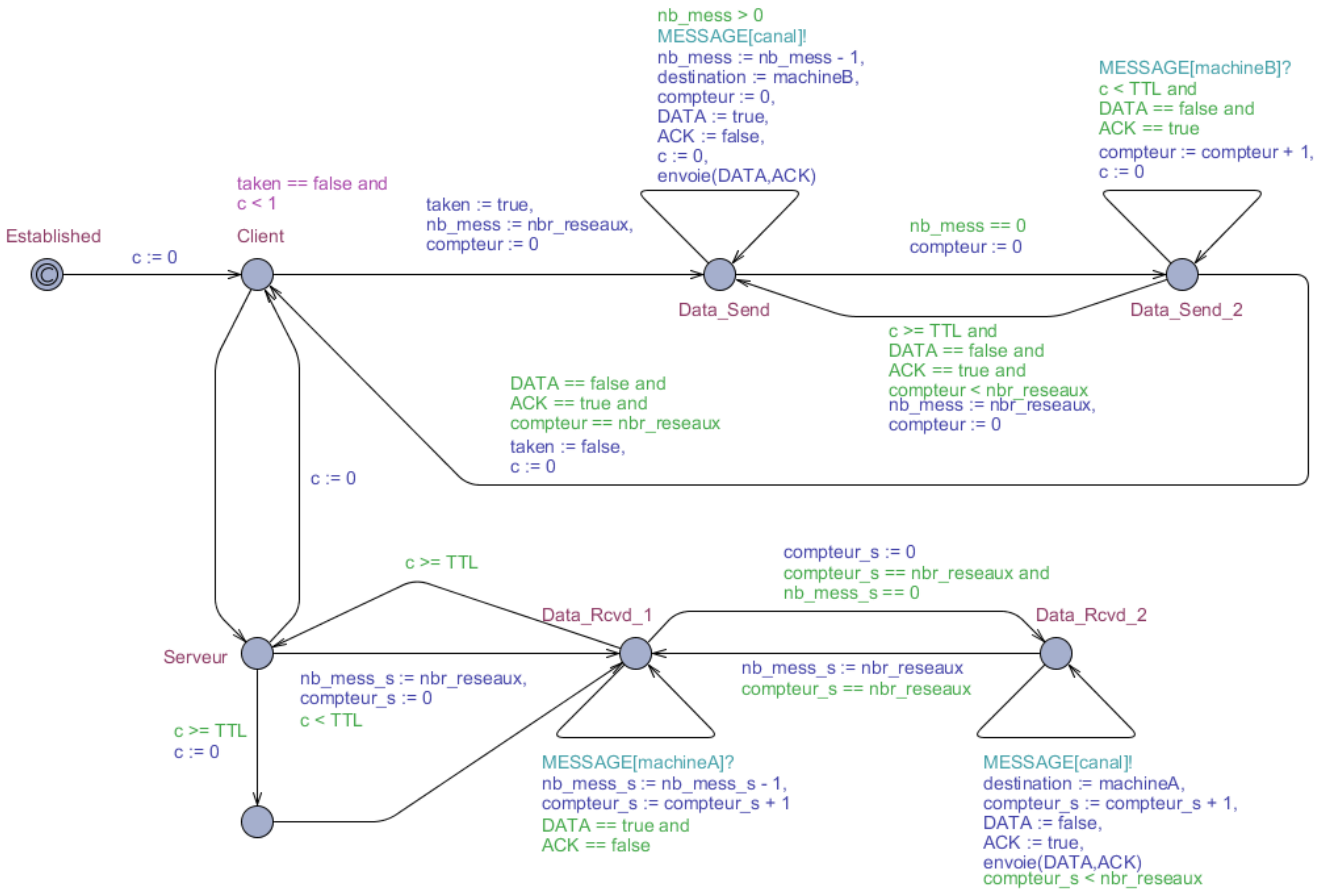


Figure 15 : Automate Machines

Les autres problème sont les suivants :

- Deadlock trouvé, mais non résolu.
- Pas de vérifications possibles sur cet automate non fini
- Nous aurions voulu disposer d'une méthode qui aurait vérifié que les messages soient bien reçu dans le même ordre que l'envoi

## Résumé

Nous sommes déçu de ne pas avoir réussi à implémenter cette V2.
C'est en effet cette partie qui aurait permis  de réellement tester les notions d'optimalité dont nous parlions au début de ce rapport.

Nous sommes cependant très satisfait dans la Version 1 du Handshake en 3 temps, l'échange de données et la fin de connexion, qui apportent tout de même une certitude qu'une connexion peut être établie entre un Client et un Serveur, et que les messages qui y transitent soient certains d'être acquittés.

# Bibliographie

Pour ce projet nous avons fait appel à ces ressources en plus de notre cerveau :

http://sebsauvage.net/comprendre/tcpip/

https://fr.wikipedia.org/wiki/Transmission_Control_Protocol

https://www.it.uu.se/research/group/darts/uppaal/small_tutorial.pdf

http://people.cs.aau.dk/~adavid/publications/21-tutorial.pdf

http://www.seas.upenn.edu/~lee/09cis480/lec-part-4-uppaal-input.pdf

https://tel.archives-ouvertes.fr/tel-00132057/document

# Annexe

Nous disposons d'un Git sur lequel nous avons déposé notre code et les vérifications qui ont été faites lors de ce projet :

https://github.com/MagicKitty/uppaal-lmfi