Rapport d'Alternance
Master Informatique : Parcours Réseaux

Implémentation du protocole Tendermint dans un simulateur basé agent

Encadrant : **Önder Gürcan**

Etudiant : **Mohamed Aimen Djari**

2018/2019

Remerciements

Je tiens à remercier toutes les personnes qui ont contribué au succès de mon alternance et qui m'ont aidé lors de la rédaction de ce rapport.

Je voudrais dans un premier temps remercier, mon maitre d'apprentissage M.GÜRCAN, ingénieur de recherche au CEA-LIST, pour sa patience, sa disponibilité et surtout ses judicieux conseils, qui ont contribué à alimenter ma réflexion.

Je remercie également toute l'équipe du LICIA avec qui j'ai travaillé tout au long de l'année pour leur professionnalisme, leur efficacité et leur bonne humeur au quotidien.

Je tiens à témoigner toute ma reconnaissance aux enseignants du Master informatique : Parcours Réseaux, qui m'ont permi de venir compléter ma formation d'ingénieur avec leurs cours. Ces connaissances complémentaires m'ont permi d'être encore plus performant lors de mon apprentissage en entreprise et de trouver des solutions auxquelles je n'aurai peut être pas pensé auparavant.

Parmi mes professeurs du Master, je tiens à remercier tout particulièrement M.BAEY, co-responsable du master qui a su me guider avec ses précieux conseils, ainsi que Mme.POTOP-BUTUCARU qui m'a offert ma première expérience dans les *blockchains*, et qui m'a recommandé pour ce poste en Alternance.

Mes plus profonds remerciements vont à mes parents. Tout au long de mon cursus, ils m'ont toujours soutenu, encouragé et aidé. Ils ont su me donner toutes les chances pour réussir. Qu'ils trouvent, dans la réalisation de ce travail, l'aboutissement de leurs efforts ainsi que l'expression de ma plus affectueuse gratitude.

Je remercie enfin toutes les personnes intéressées par mon travail, en espérant qu'elles puissent trouver dans mon rapport des explications utiles pour leurs propres travaux.

Résumé

Dans ce rapport, l'implémentation d'un protocole *blockchain* dans un simulateur dédié est présentée. Ce simulateur permet l'étude de systèmes distribués comme les *blockchains*, afin d'en découvrir les limites et proposer des solutions.

La nécessité d'étudier les protocoles *blockchain*, que ce soit avec des approches analytiques ou expérimentales, vient du fait qu'il s'agit d'un domaine relativement jeune. Étant donné qu'il s'agit avant tout de systèmes ouverts, ils sont plus difficiles à cerner mathématiquement de par la complexité de leurs algorithmes. Voilà pourquoi on opte pour des solutions plus expérimentales tel que la simulation. Le but étant de définir les limites du système modélisé pour pouvoir l'améliorer.

Il existe aujourd'hui un grand nombre de protocoles blockchain, dont les plus connus, Bitcoin [1], Ethereum [2], ainsi que Tendermint [3]. Au cours de cet apprentissage, j'implémente Tendermint pour permettre la simulation de différents scénarios impliquants ce protocole, afin de permettre et de faciliter la proposition d'améliorations en fonction de différents aspects tel que la tolérance aux pannes, l'équité, etc. Pour ce faire, on utilise un simulateur basé agents qui permet d'implémenter le fonctionnement d'un agent en particulier et de le dédoubler pour en dégager une fonctionnalité globale propre au système distribué. Cette implémentation repose sur la modélisation de l'algorithme étudié en prenant en compte les aspects qui semblent les plus pertinents et en négligeant d'autres qui n'ont pas forcément d'impact direct sur la fonctionnalité globale du système.

Abstract

In this report, the implementation of a blockchain protocol in a dedicated simulator is presented. This simulator allows the study of distributed systems like blockchains, in order to discover the limits and propose solutions.

The need to examine blockchain protocols, whether with analytical or experimental approaches, comes from the fact that it is a relatively young field. Since they are primarily open systems, they are more difficult to understand mathematically because of the complexity of their algorithms. This is why we opt for more experimental solutions such as simulation. The objective is to define the limits of the modeled system to be able to improve it.

Today, there are a large number of blockchain protocols, including the best known, Bitcoin [1], Ethereum [2], and Tendermint [3]. As part of this apprenticeship, I implement Tendermint to allow the simulation of different scenarios involving this protocol, to enable and facilitate the proposal of improvements based on different aspects such as fault tolerance, fairness, etc. To do this, we use an agent-based simulator which makes it possible to implement the behavior of a particular agent, then duplicate it in order to derive a global functionality specific to the distributed system. This implementation is based on the modeling of the studied algorithm taking into account the aspects that seem most relevant and neglecting others that do not necessarily have a direct impact on the overall functionality of the system.

Table des matières

Table des figures

Table des abréviations

- **ABCI** : **A**pplication **B**lock**C**hain **I**nterface
- **ABM-R** : **A**gent **B**ased **M**odelling in **R**
- **AGR** : **A**gent-**G**roup-**R**ole
- **API** : **A**pplication **P**rogramming **I**nterface
- **BFT** : **B**yzantine **F**ault **T**olerance
- **CD** : **C**ontinuous **D**eployment
- **CEA** : **C**ommissariat à l'**E**nergie **A**tomique
- **CGR** : **C**ommunity-**G**roup-**R**ole
- **CI** : **C**ontinuous **I**ntegration
- **CTReg** : **C**EA **T**ech **Reg**ions
- **DACLE** : **D**épartement **A**rchitecture **C**onception et **L**ogiciels **E**mbarqués
- **DAM** : **D**irection des **A**pplication **M**ilitaires
- **DDoS** : **D**istributed **D**enial **o**f **S**ervice
- **DEN** : **D**irection de l'**E**nergie **N**ucléaire
- **DIASI** : **D**épartement **I**ntelligence **A**mbiante et **S**ystèmes **I**nteractifs
- **DILS** : **D**épartement **I**ngénierie **L**ogiciels & **S**ystèmes
- **DISC** : **D**épartement **I**magerie & **S**imulation pour le **C**ontrôle
- **DM2I** : **D**épartement **M**étrologie **I**nstrumentation & **I**nformation
- **DRF** : **D**irection de la **R**echerche **F**ondamentale
- **DRT** : **D**irection de la **R**echerche **T**echnologique
- **GNU** : **G**NU is **N**ot **U**nix
- **GPS** : **G**lobal **P**ositioning **S**ystem
- **HTLC** : **H**ash **T**ime **L**ocked **C**ontract
- **IDE** : **I**ntegrated **D**evelopment **E**nvironment
- **IP** : **I**nternet **P**rotocol
- **IS-IS** : **I**ntermediate **S**ystem to **I**ntermediate **S**ystem
- **JaCaMo** : **J**ason-**C**ArtAgO-**M**oise
- **LECS** : **L**aboratoire **E**xigences & **C**onformité des **S**ystèmes
- **LETI** : **L**aboratoire d'**E**lectronique et de **T**echnologies de l'**I**nformation
- **LICIA** : **L**aboratoire systèmes d'**I**nformation de **C**onfiance, **I**ntelligents et **A**uto-organisants
- **LIDEO** : **L**aboratoire **I**ngénierie des **L**angages **E**xécutables & **O**ptimisation
- **LIST** : **L**aboratoire d'**I**ntégration de **S**ystèmes et des **T**echnologies
- **LITEN** : **L**aboratoire d'**I**nnovation pour les **T**echnologies des **E**nergies nouvelles et les **N**anomatériaux
- **LSEA** : **L**aboratoire conception des **S**ystèmes **E**mbarqués et **A**utonomes
- **LSL** : **L**aboratoire **S**ûreté & sécurité des **L**ogiciels
- **MaDKit** : **M**ulti-**a**gent **D**evelopment **Kit**
- **MAS** : **M**ulti-**A**gent **S**ystem
- **MAX** : **M**ulti-**A**gent e**X**perimenter
- **NS-3** : **N**etwork **S**imulator 3

— **ODAC** : **O**rganisme **D**ivers d'**A**dministration **C**entrale
— **OMNET** : **O**bjective **M**odular **N**etwork **T**estbed
— **OS** : **O**perating **S**ystem
— **OSPF** : **O**pen **S**hortest **P**ath **F**irst
— **P2P** : **P**eer to **P**eer
— **POM** : **P**roject **O**bject **M**odel
— **PoS** : **P**roof-**o**f-**S**take
— **PoW** : **P**roof-**o**f-**W**ork
— **PRTT** : **P**lateformes **R**égionales de **T**ransfert **T**echnologique
— **RAM** : **R**andom **A**ccess **M**emory
— **REST** : **RE**presentational **S**tate **T**ransfer
— **STP** : **S**panning **T**ree **P**rotocol
— **TCP** : **T**ransmission **C**ontrol **P**rotocol
— **URL** : **U**niform **R**esource **L**ocator
— **UUID** : **U**niversally **U**nique **ID**entifier

1 Sujet d'alternance

La *blockchain* est une technologie d'enregistrement et de transmission d'informations, transparente, sécurisée, et fonctionnant sans organe central de contrôle.

Une *blockchain* constitue un historique qui contient tous les échanges effectués entre ses utilisateurs depuis sa création. Cet historique est sécurisé et distribué : il est partagé par ses différents utilisateurs, sans intermédiaire, ce qui permet à chacun de vérifier la validité de la chaîne.

Le caractère décentralisé et ouvert de la *blockchain*, couplé avec sa sécurité et sa transparence, promet des applications bien plus larges que le domaine monétaire, afin d'en évaluer les capacités mais également les limites, il est important d'en simuler le fonctionnement.

Pour cela, un simulateur de *blockchains* a été développé au sein du Laboratoire de Systèmes d'Information de Confiance, Intelligents et Auto-Organisants (LICIA) au CEA-LIST.

L'objectif de cet apprentissage est de développer en langage Java une extension au simulateur de *blockchains* pour le protocole Tendermint et de développer les différents cas d'études en utilisant cette extension afin d'étudier les implications du protocole.

Cette mission se fera sur une durée de 11 mois au cours de laquelle, il s'agira d'apprendre les bases des *blockchains* et du travail de développement au sein d'un laboratoire de recherche.

2 Environnement de travail

2.1 Introduction

Dans le cadre de ma seconde et dernière année de Master Informatique, Parcours Réseaux au sein de Sorbonne Université, j'ai choisi le parcours en Alternance pour clôturer mon cursus tout en passant plus de temps en entreprise qu'avec un stage classique. Ceci m'a permis de passer une année au sein d'une entreprise dans le but d'approfondir mes connaissances et d'appliquer celles déjà acquises au cours de mon cursus.

Dans ce rapport, je présente l'entreprise qui m'a accueilli, ainsi que l'équipe avec laquelle je travaille pour mener à bien le projet qui m'a été confié, qui consiste à développer un simulateur de *blockchains*, et plus précisèment le protocole Tendermint [3].

2.2 Présentation de l'entreprise

Figure 1 – Logo du CEA

Le **Commissariat à l'énergie atomique et aux énergies alternatives (CEA)** est un Organisme Divers d'Administration Centrale (ODAC) contrôlé et financé majoritairement par l'État. Le CEA est spécialisé dans la recherche scientifique, et plus particulièrement dans les domaines de l'énergie, de la défense, des technologies de l'information et de la communication, ainsi que des sciences de la matière. Il est implanté sur dix sites en France.

Historiquement dénommé **Commissariat à l'Energie Atomique** (**CEA**), il a changé de nom en 2010 en élargissant son champ aux énergies alternatives.
Les principaux centres de recherche sont implantés à Saclay, à Fontenay-aux-Roses (Île-de-France), à Marcoule, à Cadarache (Provence) et à Grenoble (Isère). Le centre CEA de Saclay se trouve au cœur de la grappe industrielle technologique Paris-Saclay, tandis que le centre CEA de Grenoble se trouve au cœur du Polygone scientifique.

Créé il y a plus de 70 ans afin de poursuivre « les recherches scientifiques et techniques en vue de l'utilisation de l'énergie atomique dans divers domaines de la science, de l'industrie et de la défense nationale », le CEA reste fidèle à sa vocation d'origine et contribue à la réindustrialisation de la France.

Fin 2017, il emploie 16 000 salariés, pour un budget annuel de 4,5 milliards d'euros.

La structure organisationnelle du CEA étant complexe, je vais présenter l'organigramme du CEA jusqu'à arriver à l'équipe avec laquelle je travaille, la figure 2 ci-après renseigne plus de détails concernant l'entreprise.

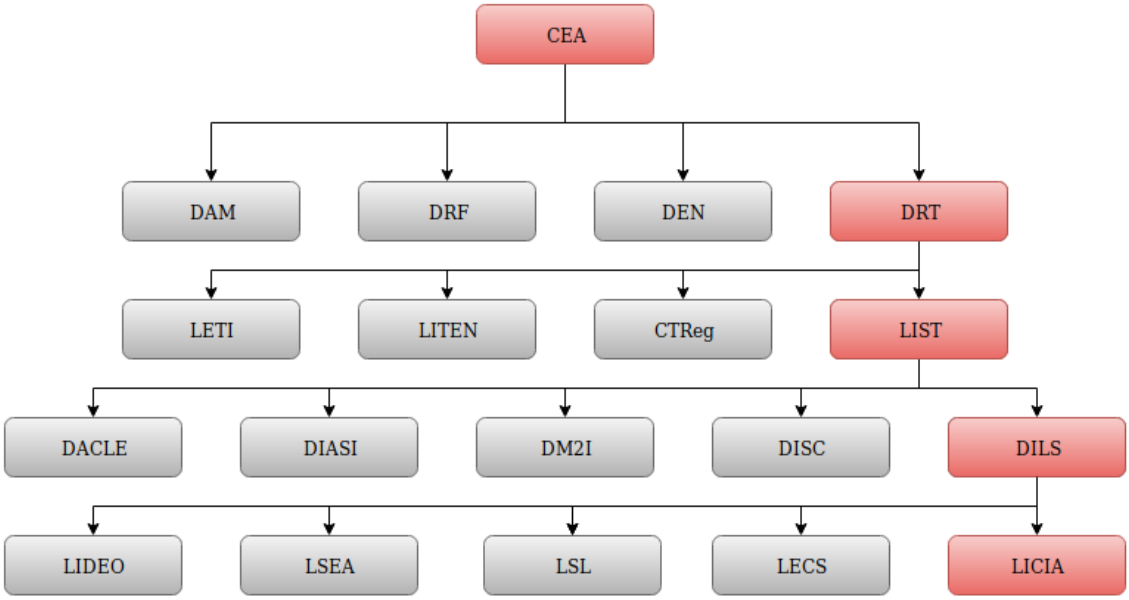

Figure 2 – Organisation du CEA

Le CEA est structuré autour de 4 grandes missions :

— L'indépendance stratégique de la France via la **DAM** (Direction des Application Militaires)
— La production d'une recherche fondamentale d'excellence via la **DRF** (Direction de la recherche fondamentale)
— L'indépendance énergétique de la France via la **DEN** (Direction de l'énergie nucléaire)
— La réindustrialisation de la France via la **DRT** (Direction de la Recherche Technologique) à laquelle je suis rattaché.

La **Direction de la Recherche Technologique** (**DRT**) compte 4 000 personnes basées à Grenoble, Saclay et au sein des Plateformes Régionales de Transfert Technologique (**PRTT**). Elle a pour mission de contribuer à la compétitivité des entreprises françaises via le développement technologique et le transfert de connaissances, de compétences et de technologies.

La **DRT** est elle même divisée en 4 instituts :

— Le Laboratoire d'Électronique et de Technologies de l'Information (**LETI**), qui concentre son activité sur les micros et nanotechnologies et leurs applications allant du spatial aux objets communicants.
— Le Laboratoire d'Innovation pour les Technologies des Énergies nouvelles et les Nanomatériaux (**LITEN**), qui joue un rôle décisif dans le développement de technologies d'avenir au service de la transition énergétique.
— CEA Tech Régions (**CTReg**), qui englobe les **PRTT**, situées à ce jour à Lille, Metz, Cadarache et Gardanne, Toulouse, Bordeaux et Nantes.
— Le Laboratoire d'intégration de systèmes et des technologies (**LIST**) qui est labellisé « Institut Carnot Technologies numériques », ce qui implique que c'est une structure de recherche publique labélisée par le Ministère de l'Enseignement supérieur, de la Recherche et de l'Innovation pour son engagement à mener et développer une activité R&D pour l'innovation des entreprises.

Basé à Saclay (région parisienne, France), le LIST est l'un des instituts de recherche technologique du CEA constituant la division technologique du CEA. Dédié aux systèmes numériques intelligents, notre mission est de réaliser un développement technologique d'excellence pour nos partenaires industriels.

À son tour, le **LIST** est constitué de cinq départements :

— Département Architecture Conception et Logiciels Embarqués (**DACLE**)
— Département Intelligence Ambiante et Systèmes Interactifs (**DIASI**)
— Département Ingénierie Logiciels & Systèmes (**DILS**)
— Département Imagerie & Simulation pour le Contrôle (**DISC**)
— Département Métrologie Instrumentation & Information (**DM2I**)

Le département où je travaille est le DILS, et il est lui-même organisé en cinq laboratoires de recherche :

— Laboratoire Ingénierie des Langages Exécutables & Optimisation (**LIDEO**)
— Laboratoire Conception des Systèmes Embarqués et Autonomes (**LSEA**)
— Laboratoire Systèmes d'Information de Confiance, Intelligents et Auto-organisants (**LICIA**)
— Laboratoire Exigences & Conformité des Systèmes (**LECS**)
— Laboratoire Sûreté & Sécurité des Logiciels (**LSL**)

Au début de mon contrat, le laboratoire LICIA, l'équipe avec laquelle je travaille quotidiennement était constituée de huit personnes :

- Sara TUCCI-PIERGIOVANNI, Chef du laboratoire.
- Agnès LANUSSE.
- Zaynah DARGAYE.
- Önder GÜRCAN, mon maitre d'apprentissage.
- Antonella DEL POZZO.
- François LE FEVRE.
- Yackolley AMOUSSOU-GUENOU, en deuxième année de doctorat.
- Moi-même, Mohamed Aimen DJARI, apprenti ingénieur de recherche.

Au cours de l'année, d'autres personnes ont rejoint l'équipe, celles-ci sont par ordre d'arrivée :

- Pablo TORIBIO, stagiaire.
- Thibault RIEUTORD, post-doctorant.
- Nicolas LAGAILLARDIE, stagiaire.
- Ludovic DESMEUZES, stagiaire.

2.3 Ressources fournies

Dès mon arrivée, il m'a été fourni un ordinateur DELL tournant sous l'OS Ubuntu avec un processeur i7 et 8GB de RAM. Connecté au réseau du CEA, j'ai eu accès à l'intranet, ainsi qu'au simulateur que j'allais développer et étoffer.

Après quelques mois, j'ai dû demander le remplacement de mon ordinateur par un nouveau plus puissant à cause des exigences en termes de ressources matérielles du projet.

3 Analyse théorique

3.1 Blockchain

La *blockchain*, aussi appelée « Chaine de blocs » en français, est un registre partagé par tous ses utilisateurs dans lequel on retrouve tout l'historique de transactions effectuées depuis le jour de sa création, c'est-à-dire depuis la création du *block* 0, aussi appelé le jour de la génèse. Mis à jour en permanence et distribué, les transactions et leur validation sont basées sur un système cryptographique, ce qui rend toute modification quasiment impossible, ce qui accroit le caractère sécurisé de la *blockchain*.

La *blockchain* étant contrôlée et gérée par ses utilisateurs sans aucune institution faisant office de tiers de confiance, elle présente à ce jour une alternative aux monnaies classiques grâce aux cryptomonnaies, mais elle offre également de nouvelles perspectives dans bien d'autres domaines que la finance.

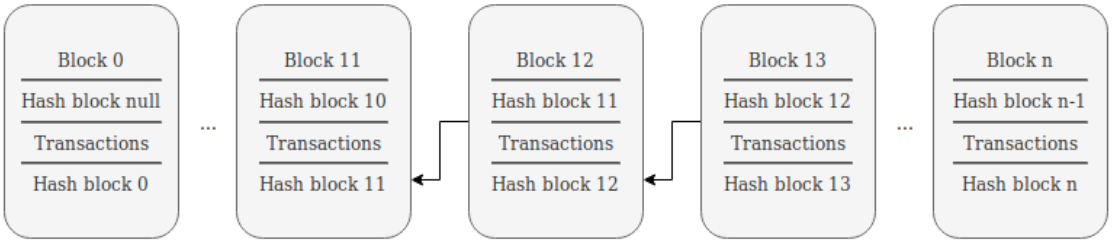

Figure 3 – Structure d'une *blockchain*

Qu'elles soient publiques ou privées, les *blockchains* sont toutes basées sur des réseaux pair-à-pair pour permettre la distribution de la *blockchain* ; de ce fait, chaque utilisateur, qui est également un nœud du réseau, possède un "portefeuille numérique" (*wallet*) qui contient la clé privée associée au compte, ainsi que l'historique des transactions faites sur la *blockchain* dont il garde une copie. Il existe néanmoins deux types d'utilisateurs, ceux qui vont simplement échanger des transactions et garder une copie de la chaine, ou encore ceux qui contribuent plus activement à sa gestion en créant des *blocks*, ces derniers sont appelés des créateurs de *block* (*block creators*).

Lorsqu'un utilisateur envoie une transaction, celle-ci est reçue par les autres utilisateurs du réseau et stockée dans leurs *mempools*, un espace mémoire où sont stockées les transactions en attente de validation. Ayant la faculté de créer des *blocks*, les *block creators* vont regrouper ces transactions en un *block* grâce à des techniques cryptographiques, ce qui aura pour effet de valider les transactions contenues dans ce *block*, qui pourront donc être retirées du mempool.

Une fois le *block* créé, il est diffusé au reste du réseau et attaché au reste de la chaine pour former une structure comme illustrée sur la figure 3, un mécanisme qui rend chaque transaction contenue dans la *blockchain* impossible à modifier ou à effacer.

Mécanismes de création de *blocks* :

C'est au travers d'un mécanisme que s'effectue la création des *blocks* [4], et donc la mise à jour de la *blockchain*. Ce mécanisme assure un ordonnancement clair et sans ambiguïté des transactions et des *block*. Il garantit également l'intégrité et la consistance du contenu de la *blockchain* entre les différents utilisateurs. Il en existe par ailleurs différents types, parmi ceux-ci :

— **Proof-of-Work (PoW)**, ou preuve de travail, est un protocole ayant pour objectif principal de dissuader les cyberattaques, telle qu'une attaque par déni de service distribuée (DDoS), qui a pour objectif d'épuiser les ressources d'un système informatique en envoyant plusieurs fausses demandes.
Appliqué aux cryptomonnaies depuis l'apparition du Bitcoin, Le concept de PoW est nécessaire pour définir un calcul informatique coûteux, également appelé "mining", qui doit être effectué par les "mineurs" afin de créer un nouveau *block*. [5]

— **Proof-of-Stake (PoS)**, ou preuve d'enjeu, est une manière différente de valider les transactions et d'atteindre le consensus distribué.
Contrairement au PoW, où le mineur est le premier qui a résolu le problème cryptographique, le PoS permet de choisir le créateur d'un nouveau *block* de manière déterministe, en fonction de sa la richesse, également définie comme le "stake".
Dans ce système, il n'y a pas de récompense, donc les mineurs prennent les frais des transactions. [5]

— **Modèles basés sur un consensus :** Dans ces modèles, les *blocks* sont créés et diffusés par un "comité", contrairement aux autres modèles où un seul utilisateur est responsable de la création d'un *block*.
On considère qu'un algorithme implémente un modèle de consensus s'il satisfait les propriétés suivantes :
1- Résiliation : Chaque nœud doit avoir décidé une valeur à la fin de l'algorithme.
2- Intégrité : Aucun nœud ne peut voter deux fois.
3- Accord : Si à la fin de l'algorithme, un nœud décide une valeur, tous les nœuds doivent avoir décidé la même valeur.
4- Validité : Si un nœud décide une valeur, elle doit être valide.

Exemple : Byzantine Fault Tolerance (BFT) [6], ou la tolérance face aux pannes byzantines, est la caractéristique qui définit un système qui tolère le type de pannes appartenant au problème des généraux byzantins.
Ce problème est l'un des plus difficiles car il n'implique aucune restriction et ne fait aucune hypothèse sur le type de comportement qu'un nœud peut avoir, il peut par exemple transmettre et poster de fausses transactions tout en se faisant passer pour un utilisateur honnête, ce qui remet en cause la fiabilité de la *blockchain*.

3.2 Tendermint

La capacité à garder le bon fonctionnement d'une application malgré certaines machines défaillantes, malveillantes, ou qui ne respectent tout simplement pas le protocole, est appelée tolérance face aux pannes byzantines (BFT).

La théorie du BFT remonte à plusieurs décennies, mais récemment, à cause du succès de la *blockchain*, les implémentations logicielles du BFT sont devenues populaires.

Tendermint [3] est un exemple d'implémentation logicielle du BFT permettant de répliquer de manière sécurisée et cohérente une application sur de nombreuses machines.

Cette réplication est dite sécurisée, car Tendermint, étant un protocole BFT, arrive à fonctionner correctement même si jusqu'à 1/3 des machines sont des byzantins, c'est à dire qu'ils fonctionnent de manière arbitraire et ne suivent pas le protocole.

La cohérence de l'application répliquée réside dans le fait que tous les utilisateurs non byzantins ont le même état, et la même copie de la *blockchain*.

Dans les systèmes distribués, la sécurité et la cohérence sont fondamentales ; Elles jouent un rôle essentiel dans la tolérance aux pannes d'un large éventail d'applications.

3.2.1 Fonctionnement de Tendermint

Le fonctionnement de Tendermint [7] repose sur deux composants principaux : un moteur de consensus *blockchain* et une application *blockchain* :

— Le moteur de consensus, appelé Tendermint Core, garantit que les transactions sont enregistrées dans la *blockchain*, qui elle-même est répliquée sur toutes les machines du réseau appartenant à l'application.
— L'application est associée au Tendermint Core pour mettre en relation les utilisateurs, leur permettre d'échanger des transactions, mais également pouvoir les valider afin de les introduire dans la *blockchain*.

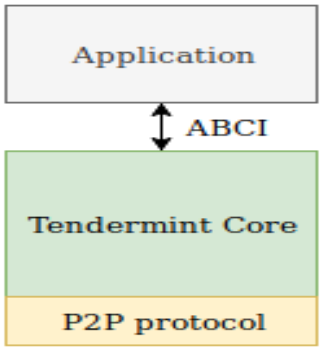

Figure 4 – Architecture de Tendermint

Comme illustré par la figure 4, l'application *blockchain* et le Tendermint Core communiquent à travers l'interface ABCI (Application BlockChain Interface) [7], qui permet de traiter les transactions dans n'importe quel langage de programmation.

Contrairement aux autres solutions *blockchain* et consensus, qui obligent l'utilisateur à se conformer à un tout nouveau framework, les développeurs peuvent utiliser Tendermint pour la réplication d'applications écrites dans le langage qu'ils veulent.

À ce jour, toutes les *blockchains* ont eu un design monolithique. Autrement dit, chaque application *blockchain* est un programme unique qui traite toutes les préoccupations, ce qui inclut la connectivité P2P, la diffusion "mempool" des transactions, le consensus, les soldes des comptes, les autorisations au niveau utilisateur, etc.

À l'inverse de ces applications, Tendermint est conçu de manière modulaire, ce qui en fait une solution performante et utile pour une grande variété d'applications distribuées.

Le premier module, le Tendermint Core, est un moteur de consensus, ce qui fait qu'il traite ce qui est lié à la connectivité P2P, la diffusion "mempool" des transactions et le consensus. Tandis que le second module, l'application, prend en charge les soldes des comptes, les autorisations au niveau utilisateur, etc.

La communication entre ces deux couches se fait via l'ABCI en échangeant trois types de messages :

- **DeliverTx**, est un message utilisé pour remonter les transactions validées par la *blockchain* à l'application, ce qui implique des changements d'état.
- **CheckTx**, est un message utilisé pour vérifier les transactions contenues dans le *mempool* avant de passer à la création de *block*.
- **Commit**, est un message utilisé pour la validation d'un *block*, ce qui bloque momentanément le *mempool*.

3.2.2 Création de *blocks* dans Tendermint

Dans Tendermint [7], avant d'être regroupées dans des *blocks*, les transactions sont stockées dans le *mempool* de chaque nœud. Ce *mempool* est un cache mémoire local où sont stockées les transactions après avoir été validées grâce au message CheckTx. Elles sont ensuite diffusées à d'autres pairs sous la forme d'une liste ordonnée de laquelle on tire des transactions pour créer des *blocks*.

Ces *blocks* sont ensuite soumis au vote des autres membres du comité. Ce mécanisme est appelé consensus, il se déroule comme illustré par la figure 5 [7] et se fait en rounds.

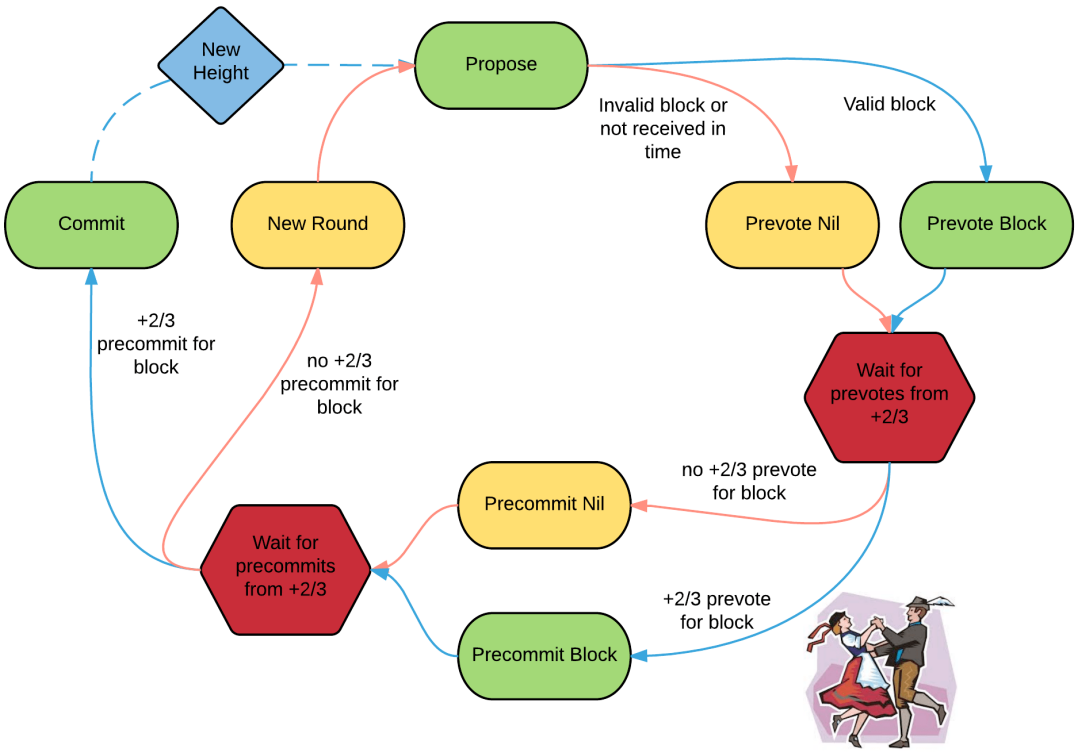

Figure 5 – Moteur de consensus Tendermint Core

Chaque round est constitué de trois phases, (i) une phase d'envoi où l'on va créer le comité, choisir un nœud parmi d'autres pour être le proposant et lui demander de proposer un *block*. (ii) Une phase intermédiaire où chaque nœud va faire état de tout ce qu'il a reçu jusqu'à l'instant t. Puis (iii), une dernière phase de calcul où chaque machine va prendre en compte toutes les données reçues pour définir son nouvel état.

(i) Phase d'envoi

— **Création du comité :**
À chaque hauteur de la *blockchain*, des validateurs sont choisis en fonction de leur "stake" pour faire partie du comité, qui sera une liste de validateurs triés de façon déterministe. Chaque membre de ce comité va exécuter un protocole de consensus pour déterminer le *block* suivant. Ce protocole se déroule en rounds, et chaque round est composé de trois étapes (pre-vote, pre-commit et commit).

— **Désignation du proposant :**
Un des validateurs du comité est désigné grâce à un algorithme Round Robin pour être le proposant. De cette façon, un seul proposant est valide pour un round donné, et il change à chaque changement de round pour une hauteur donnée.

— **Proposition de *block* :**
Le validateur désigné crée un *block* avec les transactions qu'il a dans son mempool et le diffuse aux autres membres du comité. L'algorithme du consensus est ensuite exécuté, on passe à la phase (ii).

(ii) Phase intermédiaire

— **Étape 1 : Pre-vote**
Le *block* étant proposé, vient l'étape de pre-vote où chaque nœud diffuse un message "pre-vote", qui signale s'il a vu une proposition à temps et s'il écoute les pre-votes des autres nœuds.
Si le validateur reçoit une proposition correcte, ce dernier signe un pre-vote pour cette proposition et la diffuse sur le réseau. En revanche, s'il ne reçoit pas de proposition correcte avant que le timeout n'expire, il envoie un pré-vote *nil* à la place.

— **Étape 2 : Pre-commit**
Après la phase de pre-vote, si un nœud a reçu 2/3 des messages qui concernent un seul *block*, il diffuse un pre-commit pour signaler que le réseau est prêt à valider le *block*. Sinon, il attend l'expiration d'un timeout, considère le *block* invalide, et envoie *nil*.
Un pre-commit est donc un vote pour réellement valider un *block*, par contre un pre-commit *nil* est un vote pour passer au prochain tour. Avec assez de votes pour valider le *block*, c'est à dire, au moins deux tiers, le *block* est validé et diffusé.

— **Étape 3 : Commit**
Si un validateur a reçu plus de deux tiers de pre-commit positifs pour un seul *block*, il le valide, puis passe à la hauteur suivante (prochain *block*). Sinon, il passe au round suivant pour la même hauteur.
Une fois l'étape du commit finie, le *block* est validé.

(iii) Phase de calcul

Une fois qu'un *block* proposé est validé, toutes les transactions incluses dans le *block* sont retirées du mempool. Les transactions restantes sont revalidées par l'application car leur validité peut être modifiée à cause d'autres transactions en cours d'exécution.

3.3 Généralités sur la simulation

En tant que scientifiques, on se pose souvent la question : quel est le résultat que j'obtiens si j'exerce telle action sur tel élément ?

Le moyen le plus simple serait de tenter l'expérience pour pouvoir observer le résultat. Dans de nombreux cas l'expérience est soit irréalisable, soit trop chère. On a alors recours à la simulation [12], un outil informatique servant à étudier les résultats d'une action sur un élément sans avoir à réaliser l'expérience sur l'élément réel.

Dans le cadre de l'étude de la dynamique des systèmes, la simulation est l'une des approches possibles pour comprendre le comportement d'un système et en évaluer les performances. Elle consiste à modéliser et étudier un système réel par des modèles informatiques implémentés sous la forme d'un programme informatique appelé simulateur.

La simulation à événements discrets est une technique utilisée dans ce cadre, il s'agit d'une suite d'événements discrets qui sont exécutés à un instant donné, ce qui modifie l'état du système. Cet état ne peut donc changer que lors d'instants temporels distincts où il y a un événement, qui peut-être défini comme étant une circonstance qui permet au système de changer d'état, par exemple la réception d'un message...

De nos jours, cette technique est couramment utilisée tant par les industries et les entreprises de services afin de concevoir, optimiser et valider leurs organisations que par les centres de recherche dans l'optique d'étudier les systèmes complexes non-linéaires.

Le fonctionnement d'un simulateur est basé sur l'exécution de certains évènements selon un ordre chronologique, ce qui est géré par un mécanisme appelé l'horloge, servant à indiquer le temps courant de la simulation, dont l'unité est le tick.

La simulation suit la liste d'événements, exécute les événements les uns après les autres et met à jour l'horloge de la simulation suivant les paramètres de l'événement exécuté, on distingue deux types d'exécution selon la figure 6 ci-dessous.

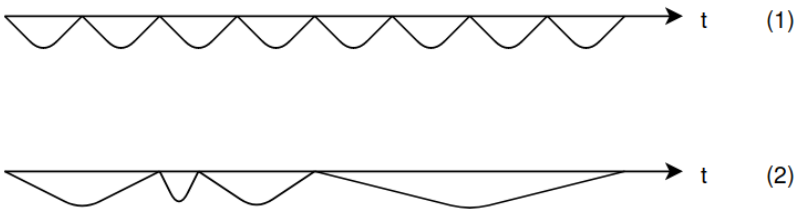

Figure 6 – Types de simulateurs

— **Simulation avec avancement par valeur fixe (1) :** La simulation avance en incrémentant l'horloge d'une certaine valeur et exécute tous les événements (s'il y en a) jusqu'à ce qu'elle atteigne une certaine limite (par exemple, le temps de fin de la simulation).

— **Simulation avec avancement par événement (2) :** Une simulation avec avancement par événement avance d'un événement à un autre de manière chronologique en les exécutant jusqu'à ce que la simulation se termine.

Dans une simulation à événements discrets, tous les événements ne sont pas forcément créés lors du démarrage de la simulation. Quand la simulation avance, un événement peut introduire un ou plusieurs nouveaux événements, qui sont alors insérés dans une liste triées selon le moment d'exécution de chaque évènement. Le processus continue jusqu'à ce que tous les événements soient exécutés ou que la simulation se termine.

Cette liste qui contient tous les évènements à exécuter est appelée écheancier, et le mécanisme de gestion qui y est associé est un ordonnanceur. L'ordonnanceur gère l'identification et l'exécution du prochain événement dans l'echéancier, l'insertion d'un événement futur dans la liste ou encore le retrait d'un événement après son exécution.

Sur la figure 7 ci-dessous, un exemple d'echéancier où on aura d'un côté des actions à exécuter et de l'autre, le compte à rebours qui y est associé. Á chaque intervalle, les nombres de cette colonne sont decrémentés, et les actions à 0 ticks sont exécutées puis retirées. Pour ce qui est de l'ajout d'évènements, si on devait ajouter une action qui s'exécuterait dans 17 ticks, alors elle viendrait se loger entre l'action 4 et l'action 5.

Evénements	Temps restant
Action 1	5
Action 2	7
Action 3	10
Action 4	15
Action 5	23
Action 6	29

Figure 7 – Exemple d'echéancier

Il existe aujourd'hui deux principales techniques de simulation :

— **Test-bed** : Travailler sur un prototype ou un réseau expérimental, une technique coûteuse mais dont les résultats sont réalistes. Toutefois, il est quasiment impossible de reproduire les mêmes résultats à cause de variables sur lesquelles nous n'avons aucun contrôle.

— **Modélisation** : Utiliser les modèles mathématiques (approche analytique) ou informatiques (approche simulation) pour construire une abstraction du système réel en négligeant certains détails jugés peu importants. Cette technique est moins coûteuse que le Test-bed et permet d'avoir des résultats qui sont, certes, moins réalistes à cause de la simplification apportée au système mais reproductibles.

3.4 Simulation basée agents

Dans le but de simuler un système de nœuds distribués, une des approches serait d'utiliser des objets auto-organisants et autonomes, c'est la simulation basée agents [13]. Elle consiste à décomposer la fonctionnalité globale du système en sous-fonctionnalités, et ainsi modéliser le comportement de chaque sous-système (nœud).

Les objets auto-organisant qui composent ce système sont appelés "agents". Un agent pourrait être défini comme étant une entité capable d'agir ou de réagir selon ses compétences et ce qu'elle perçoit de l'environnement dans lequel elle évolue.

Un des agents qui compose ce système a un rôle assez spécial puisqu'il va servir d'intermédiaire entre les autres agents. Il s'agit de "l'environnement", qui est l'entité la plus importante de ce système car elle permet la mise en relation des autres composants.

Utiliser la simulation basée agents permet de créer des systèmes composés de ces entités autonomes, qui vont interagir entre elles dans un certain environnement, ce qui va conduire à l'émergence d'un schéma global décrivant la fonctionnalité du système que l'on veut simuler.

Ainsi, un système n'est autre que l'ensemble des actions des agents qui le constituent. De cette façon, les agents d'un même système peuvent avoir des comportements différents du moment que la fonctionnalité globale est fournie.

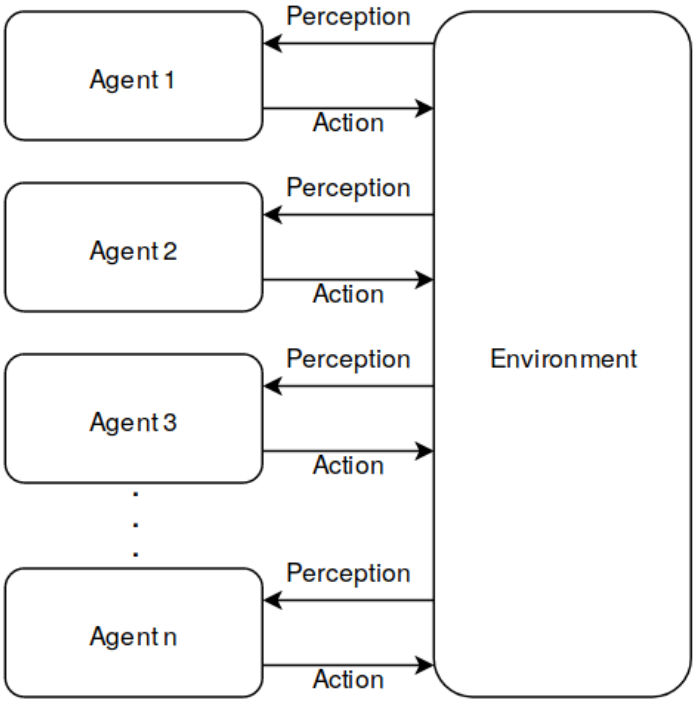

Figure 8 – Simulation basée agents

3.5 Simulation basée agents dans les *blockchains*

Par définition, la *blockchain* est une réplication d'application de façon distribuée, on parle donc ni plus ni moins d'un réseau de nœuds distribués. L'approche présentée en 4.2 convient donc tout à fait dans le sens où elle permet de modéliser des systèmes ouverts, dynamiques, distribués et intelligents, quatre des caractéristiques d'un réseau *blockchain*.

En somme, une application *blockchain* est indépendamment exécutée sur plusieurs nœuds distants qui vont interagir pour maintenir ou faire évoluer l'état de ce système auto-organisant. Un système de ce genre est défini comme étant capable de modifier le comportement de ses composants, sans aucune implication qui lui soit extérieure, afin de maintenir la fonctionnalité attendue.

La simulation basée agents nous permet de réduire la complexité liée à la création de ce système en se concentrant sur ses composants. Dans notre cas, elle nous permet de simuler un réseau de nœuds distribués (voir figure 9 [14]) en modélisant simplement les nœuds par des agents qui suivront un certain protocole (comportement) pour agir et/ou réagir dans l'environnement où ils évoluent. Cet environnement est également un des acteurs principaux du bon fonctionnement de la *blockchain*, car il incarne des paramètres tel que le délai de transfert ou encore la fiabilité de transmission qui seront des paramètres influants sur les propriétés de la *blockchain*.

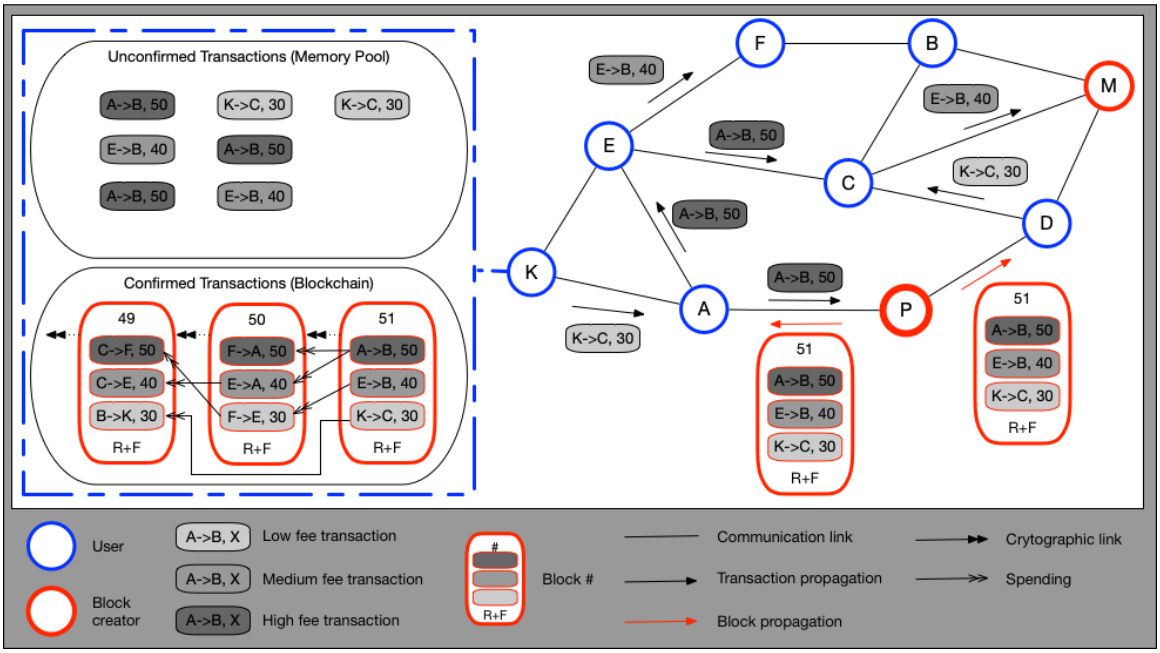

Figure 9 – Réseau *blockchain*

4 Outils de travail

Pour mon travail d'implémentation, certains outils de développement sont mis à disposition tel que l'IDE, outil indispensable du développeur. Ainsi que d'autres, qui sont obligatoires car ils permettent le travail en équipe tel que la forge Gitlab et les outils d'intégration continue qui y sont liés.

4.1 IDE

Le logiciel de dévloppement qui m'a été conseillé est Eclipse [8] (voir figure 10), un environnement de développement integré utilisé en programmation informatique. Il est également l'un des IDE les plus utilisés en développement Java.

Figure 10 – Logo d'Eclipse

Étant écrit en Java, le but premier d'eclipse est le développement dans ce langage précis, mais il peut très bien être utilisé pour d'autres tel que C, C++, Javascript, etc. Il existe évidemment des versions différentes qui vont embarquer différentes fonctionnalités selon le langage utilisé.

L'un des atouts principaux d'eclipse est le système de plug-in qui permet d'ajouter des fonctionnalités au besoin, mais également le fait qu'il soit gratuit et open-source. Il peut également être utilisé sous OS GNU, et ce même s'il est sous licence propriétaire de la fondation Eclipse.

C'est pour toutes ces raisons que mon choix s'est porté sur Eclipse, un outil que j'allais utiliser tous les jours pour mener à bien le projet qui m'a été confié.

4.2 Gitlab

Eclipse étant un IDE utilisé pour le développement local, et le projet étant à faire en équipe, nous avons utilisé Gitlab [9] (voir figure 11), un outil DevOps utilisé sous forme de plateforme web qui fournit un gestionnaire de projets Git, de suivi de problèmes, mais également un service d'intégration continue.

Figure 11 – Logo de Gitlab

Á l'origine écrit en Ruby, Gitlab a été initialement créé comme une solution de gestion de code dans le but d'améliorer et de faciliter la collaboration entre membres d'une même équipe en charge d'un projet. Ce n'est qu'après que la plateforme a été complétée par d'autres fonctionnalité tel que l'intégration continue pour former une véritable solution DevOps utilisée par bon nombre de grandes entreprises technologiques.

Dans Gitlab, les projets sont organisés en branches dont l'une, qui représente la version la plus stable du programme en cours de développement, est toujours présente et obligatoire, la branche *master*.

L'utilisation de cette plateforme est soumise à des conventions. Par exemple, lors de l'ajout d'une nouvelle fonctionnalité, il est nécessaire de d'abord créer un sujet (*issue*) ouvert au débat, ce qui permet aux autres membres de poser leurs questions et de proposer des alternatives. Une nouvelle branche est ensuite créée par le développeur en charge de cette fonctionnalité, il pourra donc faire ses changements localement puis pousser sa branche sur gitlab pour qu'un autre membre puisse vérifier son travail et l'ajouter à la branche principale.

4.3 Testing

Le test logiciel (*software testing*) est un outil utilisé pour fournir des informations sur la qualité du logiciel et renseigner le développeur sur les différents problèmes à corriger.

Les différents tests existants impliquent l'exécution du logiciel ou de la fonction créée dans le but de s'assurer que certaines propriétés indispensables au bon fonctionnement de l'application sont vérifiées. En général, ces propriétés indiquent dans quelle mesure le composant ou le système testé :

— répond aux exigences qui ont été définies.
— répond correctement à toutes sortes d'entrées.
— remplit ses fonctions.

Pour le projet qui m'a été confié, j'ai utilisé JUnit 5 [10], la dernière version d'un framework de test unitaire créé exclusivement pour le langage Java. L'utilisation de ce dernier a été facilitée par Maven, un gestionnaire de dépendances que je vais présenter par la suite.

4.4 Maven

Le projet sur lequel je travaille est divisé en plusieurs sous-projets Git dépendants entre eux, cette dépendance est gérée grâce à Maven [11].

Maven est un logiciel de gestion de projets et de dépendances très utilisé dans les projets java. Il est basé sur le concept "*Project Object Model*" et utilise un fichier *.xml* (*pom.xml*) pour décrire comment le projet doit être compilé, mais également les dépendances de ce projet ainsi que d'autres modules qui permettent par exemple de préciser l'adresse du dépôt d'artefact où trouver les dépendances.

La figure 12 illustre un exemple de pom.xml qui contient des informations sur le projet Maven qui utilise ce fichier, tel que son *groupID*, *artifactID*, ou encore le numéro de version. On peut également spécifier dans ce fichier des *plugins* supplémentaires selon le besoin. Pour finir, ce fichier décrit les dépendances dont le projet a besoin pour être compilé, ainsi que les adresses de *repositories* supplémentaires où l'on peut trouver certaines dépendances (en général privées).

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>groupExample</groupId>
  <artifactId>artifactExample</artifactId>
  <version>1.0</version>
    <build>
        <plugins>
            <plugin>
                <groupId>pluginGroupExample</groupId>
                <artifactId>pluginArtifactExample</artifactId>
                <version>2.3.2</version>
            </plugin>
        </plugins>
    </build>

    <repositories>
    <repository>
      <url>l'URL du dépôt d'artefacts où trouver les dépendances</url>
    </repository>
  </repositories>


  <dependencies>
    <dependency>
      <groupId>dependencyGroupID</groupId>
      <artifactId>dependencyArtifactID</artifactId>
      <version>3.8.1</version>
    </dependency>
  </dependencies>

</project>
```

Figure 12 – Exemple de pom.xml

4.5 CI/CD

CI/CD est une technique utilisée dans gitlab pour une meilleure gestion des apports de chaque membre d'un projet. Comme son nom l'indique, cette technique est un mélange de fonctions d'intégration continue (**CI**) et de déploiement continu (**CD**).

En développement logiciel, l'intégration continue (**CI**) consiste à faire travailler ensemble les membres d'un projet en les faisant fusionner leur travail régulièrement sur un environnement partagé, la branche *master*.

Le déploiement continu (**CD**) est une autre approche d'ingéniere logicielle dans laquelle les logiciels sont déployés fréquemment sur, par exemple, des dépôts d'artefacts comme un serveur **Nexus**.

Le déploiement continu n'est possible que grâce à des *gitlab-runners* et des pipelines de déploiement. Les *gitlab-runners* sont des instances utilisées par Gitlab pour exécuter différentes tâches liées au déploiement des projets en cours de développement. Les pipelines, quant à eux, ont pour but de vérifier le bon fonctionnement du système après les modifications qu'il a subi.

Cette vérification se fait en trois phases liées entre elles, c'est à dire, qu'on ne passe à la phase suivante que si le résultat de l'actuelle est concluant. Ces phases consistent en une première étape de compilation, une seconde où des *gitlab-runners* vont exécuter les tests liés à ce projet, et enfin, une dernière phase qui représente le déploiement sur un serveur tel que **Nexus** pour permettre l'utilisation de la nouvelle version du système.

4.6 Multi-Agent eXperimenter (MAX)

Multi-Agent eXperimenter (**MAX**) est un framework permettant à n'importe quel développeur *blockchain* de créer un modèle dans son propre environnement de développement, en s'approchant au maximum de l'algorithme réel à modéliser, en vue de le tester et l'améliorer.

En simulant la fonctionnalité d'un système au niveau agents, l'impact du comportement de chaque algorithme sera testé et analysé en détails selon différents scénarios d'exécution. Dans ce but, le plus important est de se concentrer sur les caractéristiques fondamentales qui font un tel système distribué, à savoir, l'organisation des élements et les systèmes économiques utilisés.

4.6.1 Architecture de MAX

MAX est un framework complexe, créé de façon à ce qu'il soit générique, simple d'utilisation et ait donc une architecture modulaire. Comme illustré par la figure 13, cette dernière peut être décomposée en trois sous-composants :

— **max.core :** qui représente le moteur du simulateur, va contenir les implémentations des abstractions basiques présentées précedemment, les rôles les plus généraux, ainsi que des structures de communication comme les messages, les connexions pair-à-pair, les listes de connexion...

— **max.datatype.ledger :** regroupe les structures de données propres à la *blockchain* tel que la *blocktree*, les *blocks*, les transactions...

— **max.model :** qui comprendra les différents modèles créés, et donc les protocoles implémentés, les messages qui leur sont propres, ou encore les rôles qui leur sont spécifiques.

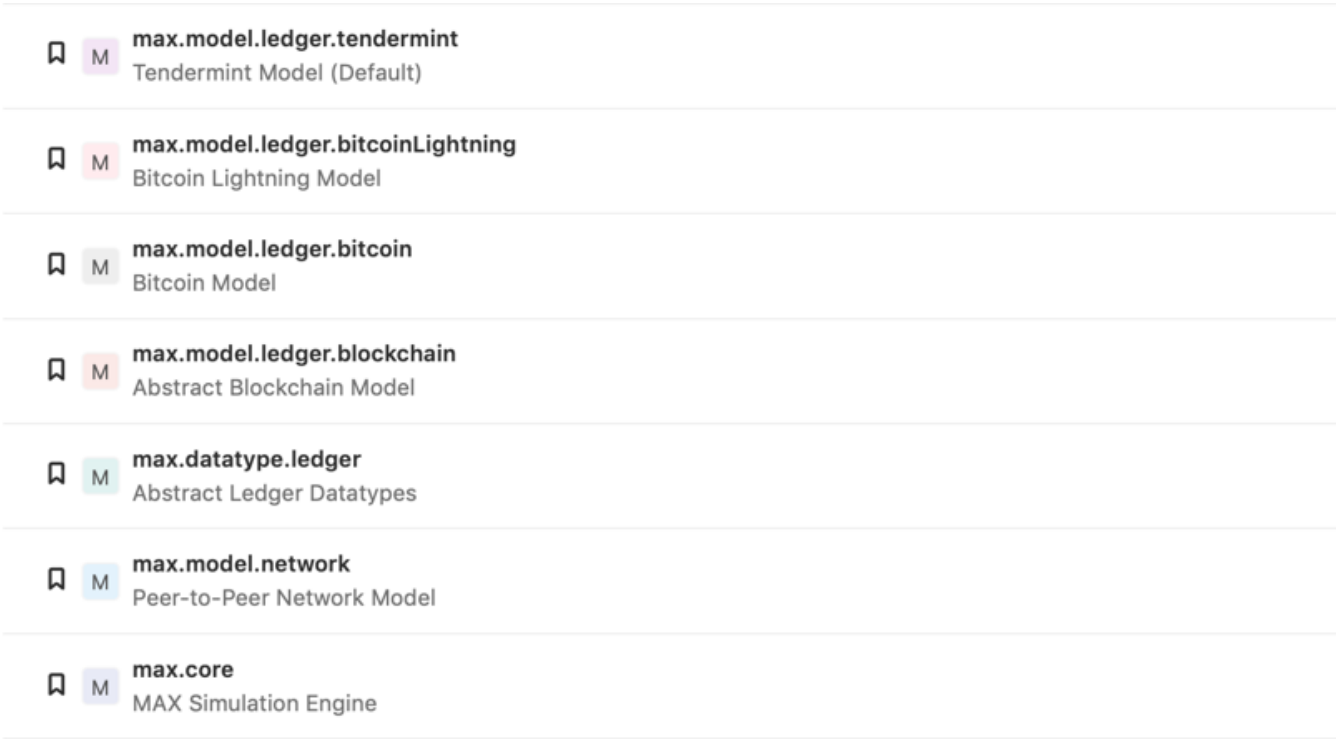

Figure 13 – Architecture de MAX

La création d'un modèle dans MAX repose sur des abstractions dont certaines sont indispensables :

— **Agent**
Un agent est une entité active qui peut jouer un ou plusieurs rôles dans un ou plusieurs environnements, il peut planifier des actions selon ce que les rôles qu'il joue lui permettent.
Quand un agent prend un rôle, il gagne le droit de planifier des actions qui sont propres à ce rôle, s'il délaisse ce rôle pour quelque raison que ce soit, les actions à venir que seul ce rôle lui permet de réaliser seront annulées par MAX.

— **Scheduler**
Le *scheduler* (ordonnanceur) est responsable de la gestion de l'horloge globale, de la planification et de l'exécution des actions des différents agents.

— **Environment**
Dans le monde des réseaux et de la qualité de services précisément, le réseau est l'un des élements les plus importants lorsqu'il s'agit de communication distante, ce qui peut donc s'appliquer aux réseaux distribués. La qualité du réseau influe directement sur le comportement des utilisateurs (retransmission, baisse de débit...).

Voilà pourquoi dans MAX, l'environnement est l'une des pièces maitresses quand on modélise des systèmes multi-agents.
Dans MAX, l'environnement est abstrait, il peut être physique (internet) ou logique (réseau social), et est modélisé comme étant un agent qui planifie des événements en fonction des requêtes des autres agents actifs. Il joue donc le rôle de médiateur entre les agents et permet leur interaction. En d'autres termes, les agents ne peuvent pas interagir directement entre eux, ils passent obligatoirement par un environnement. Les environnements sont modélisés comme des agents appartenant à un groupe et jouant le rôle d'environnement dans ce groupe, les agents qui feront également partie de ce groupe pourront prendre les rôles de ce groupe et intéragir via l'unique agent qui a le rôle environnement dans ce grouoe.

— **Experimenter**
Il s'agit de l'agent responsable de la création et de la mise en place du scénario à simuler. Étant donné que c'est avant tout un agent, il peut observer mais également intervenir sur le déroulement de la simulation. Ce qui permet la dynamicité d'un scénario où des agents peuvent être ajoutés et retirés selon certaines conditions précisées dans le scénario initial. Par exemple, il est possible d'ajouter un agent à un instant t1 et de le retirer à l'instant t2 afin d'étudier la réaction du système face à un environnement dynamique.

La création d'un scénario tel que celui-ci se fait en étapes :
— Initialisation du scheduler
— Initialisation de l'environnement
— Initialisation des agents

— **Community-Group-Role**
MAX permet une organisation hiérarchique des agents grâce au concept CGR (*Community Group Role*). De ce fait, deux agents peuvent faire partie de groupes différents mais appartenir à la même communauté. Étant donné qu'il s'agit d'une abstraction, une communauté est simplement le conteneur d'un ensemble d'agents divisé en sous ensembles appelés groupes dans lequel les individus jouent des rôles.

— **Context**
Le contexte permet à un agent de différencier les informations qu'il a perçu dans plusieurs environnements afin de garder une séparation entre les groupes, et rendre possible l'appartenance d'un agent à plusieurs environnements sans altérer la fonctionnalité du système.

4.6.2 MAX : Core

Le moteur de MAX est responsable de la gestion des fonctionnalités basiques du simulateur, à savoir l'ordonnancement grâce au scheduler, l'organisation hiérarchique grâce au groupe, au rôle ou encore à l'environnement, ainsi que la scénarisation grâce à l'experimenter. *max.core* est basé sur un gestionnaire d'évènements discrets du nom de **MaDKit**, qui permet l'organisation hiérarchique des agents, leur adressage, ainsi que la différenciation entre les rôles.

Multi-agent Development Kit, ou **MaDKit** [15] est un outil de développement de plateformes multi-agents, il repose sur un modèle organisationnel basé sur des groupes contenant des agents, et des rôles qui y sont associés.

MaDKit fonctionne sur le modèle AGR (*Agent-Group-Role*). Ce modèle servira à organiser les agents de façon hiérarchique, et leur donner des capacités en fonction de leur position dans la hiérarchie. Cette fonctionnalité permet donc la modélisation d'un système hétérogène tel qu'un réseau distribué.

Un agent est défini comme une entité réactive qui joue un rôle dans un ou plusieurs groupes. Un rôle est représentation abstraite d'une fonction attribuée à un agent. Pour avoir un rôle, un agent doit en faire la requête.

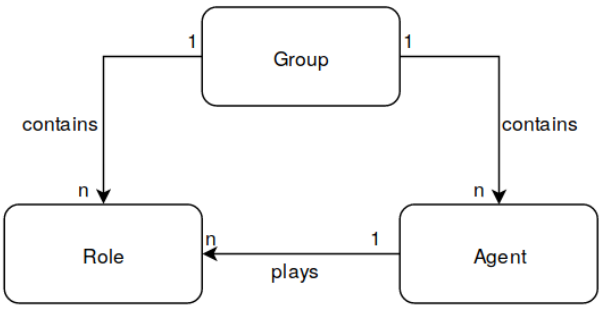

Figure 14 – Organisation AGR

Dans **MaDKit**, certains élements sont indispensables au bon fonctionnement de la simulation :

- **Abstract Agent :** l'agent actif qui peut interagir avec les autres agents appartenant à son groupe en adoptant un comportement défini par le.s rôle.s qu'il a dans ce groupe. La perception de l'*Abstract Agent* se limite à ce que les autres agents veulent bien lui montrer en lui envoyant un message.
- **Watcher :** Un autre type d'agent qui a un rôle de surveillant dans un groupe, et qui a accès à toutes les informations intra et inter agents. Il sert à la scénarisation ou pour veiller au bon fonctionnement de la simulation. Pour ceci, il utilise l'objet *probe* qui lui permet d'accéder aux informations des agents que le *probe* va "surveiller".

— **Activator :** Pour lancer un agent, il faut lui attribuer un rôle initial qui doit être valide. C'est là que l'activateur entre en jeu, il permettra d'activer les rôles que l'on veut pour la simulation.

Par définition, la *blockchain* est une application distribuée, donc basée sur un réseau de nœuds qui communiquent entre eux en suivant un certain protocole. Cet aspect de communication est nécessaire au bon fonctionnement de la simulation de n'importe quel type de modèle, voilà pourquoi un autre module a été ajouté au core, qui regroupe les structures de données nécessaires à la communication entre les nœuds. Ces structures de données sont :

— **Address :**
Une adresse est un identifiant qui a une représentation unique et qui est attribué à chaque agent dans chaque groupe auquel il appartient. Dans MAX, l'adresse est de type **UUID** (*Universally Unique IDentifier*), et c'est une structure de 128 *bits* utilisée pour identifier un agent parmi d'autres.

— **Message :**
Les messages sont utilisés pour les interactions entre les agents dans un environnement, ils constituent une structure de données qui prend comme paramètre l'adresse de l'agent emetteur, l'adresse de l'agent destinataire et l'objet du message.

— **Contact :**
Une structure de donnée qui représente l'agent, son nom, ainsi que ses adresses dans les différents groupes.

— **ContactList :**
À l'image du répertoire d'un télephone, il s'agit d'une liste que chaque agent possède et qui renferme les connaissances de l'agent, ses contacts, leurs noms, leurs adresses, etc.

4.6.3 MAX : Datatype

MAX étant un simulateur dédié à la *blockchain*, l'un des premiers modules que nous avons implémenté regroupe les structures de données propres à la *blockchain*, celles-ci sont :

— **Transactions :**
Une transaction est un transfert de biens entre deux nœuds. Après sa création, elle est diffusée au réseau pour qu'elle soit mise dans des *blocks*. Elle est modélisée comme un message où le *payload* représente la somme transférée.

— **MerkleTree [16] :**
Comme le montre la figure 15 [16], le *merkleTree* est un arbre créé à partir de transactions, celles-ci sont hashées deux à deux jusqu'à ce qu'il n'y ait plus qu'un seul *hash*, la racine de l'arbre.

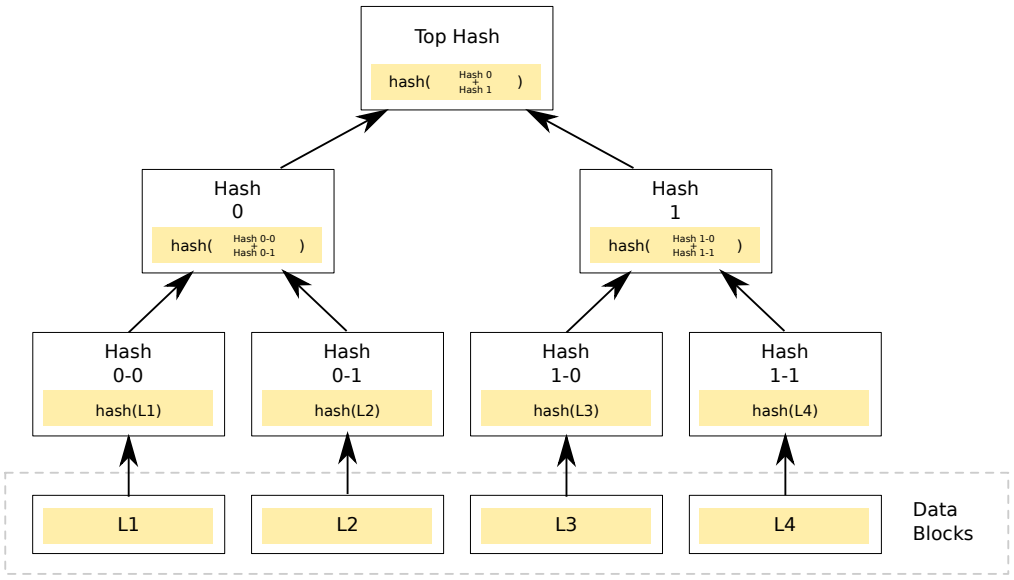

Figure 15 – Arbre de Merkle

— ***Block* :**
Un *block* est une structure de données contenant un en-tête et des données qui ne sont autres que des transactions organisées grâce à un MerkleTree. L'ensemble des transactions est choisi par le mineur dans son mempool, il y rajoute ensuite la *coinbase transaction* qui représente sa récompense. L'en-tête du *block* quant à elle contient des informations tel que le numéro de version du protocole utilisé, le *hash* du *block* précedent, un *timestamp* et le *nonce* qui a permi au mineur de résoudre le challenge cryptographique.

— ***Blockchain* :**
Notre modélisation de la *blockchain* est une liste dynamique de *blocks* configurée en "ajout seul" (*append-only*) où chaque *block* contient une référence cryptographique au *block* précedent. Le tout premier *block* est appelé *genesis-block* et est créé à la création de la *blockchain*.

— ***Blocktree* :**
Un arbre contenant tous les *blocks* créés et donc toutes les chaines de *blocks* qui ont pu être créées à la suite de *forks*.

— ***Wallet*** **:**
Chaque utilisateur d'une *blockchain* a un *wallet* qui contient des informations importantes tel que son adresse, sa paire de clés cryptographiques, son historique de transactions...

— ***Multi-sig Wallet*** **:**
Un *multi-sig wallet* est un *wallet* partagé entre plusieurs utilisateurs, il a donc plusieurs propriétaires et les transactions dont il est l'emetteur doivent être signées par tous les propriétaires. Ce type de *wallet* est essentiellement utilisé pour réaliser des transactions *off-the-chain*.

4.6.4 MAX : Model

MAX est un framework de modélisation essentiellement utilisé dans les *blockchains*. On l'utilise notamment pour implémenter des protocoles de *blockchains* afin de les utiliser, ces implémentations sont des modèles car elles se rapprochent du fonctionnement du protocole avec quelques suppositions ou omissions. MAX est créé de façon à être modulaire, c'est à dire que les modèles créés ne seront que des modules supplémentaires qui viendront s'ajouter à la librairie de MAX.

1- Création d'un modèle :

MAX repose sur une organisation hiérarchique des agents en *Community-Group*, des classes définissant ces conteneurs sont donc nécessaires. Il est également indispensable de spécifier les rôles à prendre dans un groupe donné. Dans les modèles de MAX, les agents communiquent à travers un environnement, ce qui le rend indispensable à toute modélisation. Il n'est cependant pas nécessaire de créer un nouvel environnement à chaque modèle créé. De par leurs rares différences à ce niveau, les modèles *blockchains* peuvent tout à fait utiliser l'environnement d'un module qu'ils étendent.

La dernière étape pour la création d'un modèle est évidemment la création d'un agent avec ses caractéristiques, ses propriétés, ainsi que ses différents comportements en fonction de son rôle.

Le modèle à présent créé, il est mis en pratique grâce à des scénarios, qu'ils soient exécutés comme des tests automatisés ou comme des expériences.

2- Étude d'un modèle :

Après l'étape de création du modèle, on passe à l'étude du protocole modélisé. Pour faciliter la tâche à toute personne désirant étudier l'exécution d'un certain protocole, j'ai été en charge de la création de graphes censés aider à avoir une meilleure compréhension d'un système donné.

— **Affichage de l'environnement avec ses propriétés :** aide à avoir une bonne vision de la topologie, des paramètres de l'environnement tel que le délai de transmission, ainsi que de l'évolution du nombre d'agents connectés au cours du temps.

— **Affichage de la *blocktree* :** un affichage complet de la *blocktree* est proposé pour pouvoir voir son évolution, le nom du mineur qui l'a créée, les forks s'il y en a également.

— **Diagrammes relatifs aux *blocks* :** observer le nombre de mineurs, leur capacité à créer des *blocks*, ainsi que leur activité.

— **Diagrammes relatifs à l'environnement :** afficher la connectivité de chaque agent (connexion / déconnexion et à quel moment) ou encore la synchronisation de l'application (avoir le même état chez tous les agents).

3- Utilisation d'un modèle :

Comme expliqué en section 4.6.1, le scénario nécessite l'initialisation de certains éléments comme le scheduler et l'environnnement pour son bon déroulement. Après cette phase d'initialisation, l'utilisateur de MAX peut également choisir d'afficher certains graphes qu'il jugera utiles et pertinents pour une meilleure compréhension du déroulement de la simulation.

Tous les modèles présentés ont été implémentés en utilisant MAX, un environnment de simulation basée agents, écrit en Java au sein de mon équipe, le LICIA.

5 Implémentation de Tendermint dans MAX

Tendermint [3] est un protocole très étudié au sein de mon équipe. Il est très intéressant du fait que son algorithme est complexe et donne des résultats satisfaisant qu'aucun autre protocole ne propose. Il résout le problème du *fork*, du gaspillage d'énergie, mais permet également de supporter un certain nombre d'utilisateurs mal intentionnés. Jusqu'ici, les méthodes utilisées pour étudier ce protocole reposaient sur des approches mathématiques longues et difficiles à mettre en place. Cette nouvelle approche expérimentale permet de mieux cerner l'algorithme, et de pouvoir le modifier pour proposer des améliorations.

Pour ce faire, nous avons commencé par implémenter des modèles abstraits qui vont servir de bases aux différents protocoles *blockchains* qu'on pourrait implémenter, par exemple l'échange de messages ou encore la gestion des structures de données *blockchains* sont deux aspects communs à la majorité des protocoles existants.

5.1 Network Model

Le système des *blockchains* est un réseau distribué, donc des nœuds connectés les uns aux autres sans aucun serveur. La capacité de communiquer de ces nœuds permet le bon fonctionnement de la *blockchain*, c'est donc par là que nous avons commencé.

Avant de commencer à modéliser un protocole *blockchain*, et toujous dans le but d'avoir un simulateur modulaire, la première étape fut de créer un module de communication "Network" (voir diagramme UML en annexe B.1).

Nous avons donc commencé la création de max.model.network.p2p (voir annexe A.1)qui est la modélisation d'un réseau pair-à-pair non structuré classique, pour ensuite passer à un modèle plus complexe et plus complet, "*broadcast*", dont la fonctionnalité principale est le routage et le transfert de messages entre les nœuds.

Broadcast Model

L'inconvénient du protocole *p2p* mis en place est qu'il est pratiquement impossible de faire un réel *broadcast* lorsque le réseau devient trop grand. De plus, certains protocoles *blockchain* utilisent le *gossiping*, qui se définit par le *forwarding* d'un message à ses propres voisins à sa réception. Mon équipe, qui est en charge du projet, a donc voulu implémenter cette fonctionnalité, et j'ai eu l'idée d'utiliser l'algorithme de création d'arbre du plus court chemin le plus connu et certainement le plus complet, Dijkstra.

1- Algorithme de Dijkstra :

L'algorithme de Dijkstra [17] est un algorithme permettant de trouver le plus court chemin entre deux nœuds d'un graphe. Essentiellement utilisé dans les réseaux informatiques, cet algorithme est à la base de beaucoup de protocoles de routage tel que IS-IS(*Intermediate System to Intermediate System*), ou encore OSPF(*Open Shortest Path First*).

Au fil du temps, il y a eu beaucoup de versions du protocole Dijkstra qui ont été créées. Aujourd'hui, la version la plus connue et la plus utilisée est celle qui permet de créer un arbre du plus court chemin d'un nœud vers tous les autres. En fixant le nœud source dans un graphe, Dijkstra permet de trouver le plus court chemin entre cette racine et tous les autres nœuds du graphe. Ce qui est semblable au protocole STP (*Spanning Tree Protocol*) qui crée un arbre dont la racine est l'un des commutateurs STP.

Cet algorithme est utilisé dans de nombreux domaines, pas seulement les réseaux. Il peut par exemple être utilisé comme GPS où les villes seraient représentées par des nœuds dans un graphe, et dont les liens sont les routes qui joignent ces villes. Dijkstra permettrait donc de trouver le plus court chemin entre deux villes, donc entre deux positions, et ainsi avoir une fonction de GPS.

2- Modélisation de Dijkstra dans MAX :

Dans notre modélisation de Dijkstra, nous avons voulu avoir un protocole simple qui puisse converger rapidement, et que les messages envoyés empruntent les chemins les plus courts, ce qui est en réalité impossible avec Dijkstra étant donné que cet algorithme crée un seul arbre du plus court chemin avec une seule source.

L'idée était donc de créer autant de graphes que de nœuds, avec chaque graphe ayant une source différente des autres. Tout d'abord, pour la création d'un arbre, nous avons choisi GraphStream [18], une librairie Java utilisée pour créer des graphes de nœuds. Cette librairie permet de créer des arbres Dijkstra à partir d'un graphe. Cette librairie permet également de mieux gérer l'arbre créé grâce à une classe "Dijkstra" qui a été définie, et qui permet, par exemple, de trouver le chemin entre deux nœuds en faisant appel à une méthode créée à cet effet.

Pour faire en sorte que les messages empruntent toujours les chemins les plus courts, l'environnement calcule un graphe pour chaque agent, c'est à dire qu'à chaque agent, on attribue un graphe dont il est la source. L'environnement s'occupe au fur et à mesure de la mise à jour des graphes en fonction de ceux des voisins sans avoir besoin de les diffuser.

L'environnement s'occupe également du changement d'adresses dans les messages lors du relai. Lorsqu'un agent doit relayer un message, il ne fait que renvoyer ce message à l'environnement qui trouve le bon destinataire, modifie les champs appropriés avec les bonnes valeurs et délivre le message.

Cette modélisation a necessité la création d'une nouvelle structure de données :

— **RelayableMessage :**
Á l'image du modèle TCP/IP, nous avons créé un nouveau type de messages qui étend le type "Message" du module de communication présenté en section 4.6.2. En effet, ce message a non plus une, mais deux couches d'adressage, une première qui contient l'emetteur et le récepteur final (niveau L3), et une seconde qui contient les nœuds relais emetteur et recepteur (niveau L2) à chaque saut.

5.2 Blockchain Model

Le simulateur MAX étant dédié à la *blockchain*, j'ai été en charge de créer un modèle abstrait représentant la *blockchain* et ses agents de la façon la plus générique possible. Cette façon de faire nous a ensuite permis de créer des protocoles qui suivent ce modèle abstrait et dont la création était plus simple, étant donné qu'il n'y avait plus qu'à implémenter le protocole sans se soucier des fonctions basiques de la *blockchain*.

Modélisation de la *blockchain* dans MAX :

Après avoir implémenté les modules servant à la communication des agents, nous avons créé un autre modèle mettant en relation cet aspect de communication avec les structures de données *blockchain* décrites en section 4.6.3. Dans notre représentation, l'agent Blockchain ne serait donc qu'un agent Network ayant accès aux structures de données *blockchain* (voir diagramme UML en annexe B.2).

Oracle :
Parmi les propriétés des protocoles *blockchain*, la création de *block* n'est pas l'aspect sur lequel mon équipe voulait se concentrer le plus. Pour modéliser cet aspect, on a donc opté pour l'utilisation de l'abstraction Oracle.
L'oracle est un agent "*watcher*" qui peut accèder aux propriétés de tous les agents. Grâce à ces informations, il peut choisir quel agent doit créer un *block*. Dans MAX, la création d'un *block* est une méthode de l'oracle qui prend en compte la puissance de calcul (*computational power*) ou alors les richesse de l'agent selon le modèle de création. L'oracle prend comme paramètre le mode de création de *blocks* selon le protocole simulé.

Rôles :
Le modèle Blockchain étend le modèle network, de ce fait, il possède également les rôles de ce dernier. Les comportements des agents ayant ces rôles vont quant à eux différer. Ajouté à cela, le rôle "Oracle" qui permet à l'Oracle de mener à bien sa mission, ainsi que le rôle "Viewer" qui va permettre à certains agents de type *watcher* (voir section 4.6.2) de surveiller l'environnement, ainsi que les agents qui s'y trouvent.

Hypothèses :
Pour les besoins de la simulation, nous avons eu recours à certaines hypothèses qui nous ont permis de simplifier le protocole, celles-ci sont :

- **Délai :** Le même délai de transmission est appliqué entre tous les agents et il n'est pas variable.
- **Création de *blocks* :** La création de *blocks* se fait régulièrement en fonction de la propriété *block creation rate* propre à chaque protocole.
- **Pas de signature :** Nous avons omis l'utilisation de signatures cryptographiques car on suppose qu'aucun agent ne peut usurper l'identité d'un autre.

— **Validité des *blocks*/transactions :** Aucun mécanisme de validation n'a été implémenté, on suppose toutes les transactions et *blocks* créés valides.
— **Synchronisation de l'horloge :** Les agents sont synchronisés, il n'y a donc pas de retard causé par une désynchronisation.

5.3 Tendermint Model

Après avoir créé les modèles abstraits nécessaires à l'implémentation de protocoles. On est passé à la création de modèles plus concrets qui pourront être exécutés et nous permettront donc de simuler les protocoles et ainsi, les étudier. Le but de mon alternance étant d'implémenter Tendermint [3] dans MAX, nous avons décidé de commencer par d'autres protocoles plus simples, que sont Bitcoin [1] et BitcoinLightning [19] pour me familiariser avec le framework, mais également pour en apprendre plus sur la blockchain en général avant de passer à un protocole complexe tel que Tendermint (voir annexe A).

5.3.1 Étude de l'algorithme de Tendermint

Après avoir présenté le protocole Tendermint de façon générale en 3.2, nous allons passer à une étude plus approfondie du protocole pour pouvoir ensuite l'implémenter et l'utiliser. Le déroulement du protocole Tendermint [7] se fait essentiellement en trois étapes, propose, prevote et precommit :

Propose Step (height :H,round :R)
Au cours de cette première étape, le validateur choisi par l'algorithme de sélection pour être proposant, propose un *block* de hauteur H.

L'étape est finie :
- Quand le timeout de la phase Propose expire, on passe à l'étape Prevote.
- S'il y a eu réception de plus de 2/3 de precommits pour un *block* de même hauteur H, on passe à l'étape Commit(H).
- S'il y a eu réception de plus de 2/3 de prevotes pour un *block* de même hauteur H, mais pas dans le même round ($R' \neq R$), on passe à l'étape prevote(H,R').
- S'il y a eu réception de plus de 2/3 de precommits pour un *block* de même hauteur H, mais pas dans le même round ($R' \neq R$), on passe à l'étape precommit(H, R').

Prevote Step (height :H,round :R)
Au cours de cette seconde étape, chaque validateur devra diffuser son prevote, cette valeur dépendra de certaines conditions :
— Si le validateur a verrouillé un *block* mais qu'il peut à présenter verrouiller une autre valeur à un round supérieur, il verrouille cette nouvelle valeur.
— Si le validateur a une valeur vérrouillée, il la prevote.
— Sinon, il prevote la valeur qu'il a reçu dans propose si elle est valide, si elle ne l'est pas, il prevote *nil*.

L'étape est finie :
- Quand le timeout de la phase prevote expire, on passe à l'étape Precommit.
- À la réception de plus de 2/3 de prevotes pour une même valeur ou pour *nil*, alors on passe au Precommit.
- S'il y a eu réception de plus de 2/3 de precommits pour un *block* de la même hauteur H, on passe à l'étape Commit(H).
- S'il y a eu réception de plus de 2/3 de prevotes pour un *block* de la même hauteur H, mais pas dans le même round ($R' \neq R$), on passe à l'étape prevote(H,R').
- S'il y a eu réception de plus de 2/3 de precommits pour un *block* de même hauteur H, mais pas dans le même round ($R' \neq R$), on passe à l'étape precommit(H, R').

Precommit Step (height :H,round :R)

Au cours de cette dernière étape de l'algorithme, chaque validateur diffuse son precommit. Cette valeur dépendra de certaines conditions :
- Si le validateur a choisi une valeur (*block* ou *nil*) qu'il peut verrouiller, il la verrouille en enregistrant le round, il precommit ensuite cette valeur.
- Sinon, il ne déverrouille pas et precommit *nil*.

Un precommit *nil* signifie qu'il n'a trouvé aucune valeur qu'il peut verrouiller mais qu'il a quand même attendu de recevoir assez de messages ($> 2/3$).

L'étape est finie :
- Quand le timeout de la phase precommit expire, on passe à l'étape propose du prochain round.
- S'il y a eu réception de $> 2/3$ de precommit *nil*, on va directement à l'étape propose du prochain round.
- S'il y a eu réception de $> 2/3$ precommit pour un *block* de la même hauteur H, on commit le *block* (on l'enregistre dans sa *blockchain*) et on passe à la hauteur suivante.
- S'il y a eu réception de $> 2/3$ prevote pour un *block* de la même hauteur H, mais pas dans le même round ($R' \neq R$), on passe à l'étape prevote(H,R').
- S'il y a eu réception de $> 2/3$ precommit pour un *block* de la même hauteur H, mais pas dans le même round ($R' \neq R$), on passe à l'étape precommit(H, R').

5.3.2 Modélisation de Tendermint dans MAX

Dans notre implémentation, le modèle Tendermint étend le modèle Blockchain qui à son tour étend le modèle p2p, de la même façon que la *blockchain* est une application exécutée sur un réseau pair-à-pair (voir diagramme UML en annexe B.3).

Nous avons essayé de représenter Tendermint de la manière la plus réaliste possible avec les hypothèses formulées précedemment en 5.2, mais comme avec toute modélisation, il est indispensable d'omettre certains points qu'on juge peu importants ou sur lesquels on ne voudrait pas se focaliser dans l'immédiat.

Oracle :

La création de *block* dans Tendermint utilise l'oracle créé dans le modèle Blockchain avec comme mode de création, du BFT exclusivement comme le stipule le protocole Tendermint.

Cette méthode est basée sur le consensus, c'est à dire qu'un *block* n'est pas créé par un seul utilisateur comme dans les protocoles précedemment cités, mais par un groupe, un comité qui échange des messages pour décider d'un *block* à diffuser.

Ce comité est sélectionné via un modèle PoS où on choisit un ensemble de nœuds selon leurs richesses, on donne ensuite à ce comité la responsabilité de créer un *block* et de le diffuser à l'ensemble du réseau pour ensuite être récompensé par le comité suivant.

Messages :

Comme avec *network.p2p*, nous avons mis en place certains types de messages nécessaires au fonctionnement du protocole, et correspondants aux étapes de l'algorithme présenté en 5.3.1.

Au sein de mon équipe, nous avons jugé préférable de modifier les noms des étapes dans Tendermint ; De cette façon, propose devient prepropose, prevote devient propose, et precommit devient vote.

Ces messages que nous avons implémenté ont une structure nouvelle, ils ont des champs correspondants à l'emetteur du message, à la hauteur, au numéro de round et bien sûr à la valeur envoyée. Les messages utilisés sont donc :

- **PREPROPOSE :** correspondant au message envoyé à la fin de l'étape prevote.
- **PROPOSE :** correspondant au message envoyé à la fin de l'étape propose.
- **VOTE :** correspondant au message envoyé à la fin de l'étape vote.

Hypothèses :

Pour les besoins de la simulation, nous avons eu recours à des hypothèses supplémentaires qui nous ont permis de simplifier le protocole, celles-ci sont :

- **Byzantins :** Tendermint est un protocole qui est tolérant aux failles byzantines, c'est là son intérêt. Dans cette première implémentation, nous n'avons ajouté aucun exemple de comportement byzantin, c'est d'ailleurs un sujet de travail futur.
- ***Stakes* :** En réalité, les validateurs sont triés selon leurs "*stakes*", une forme de richesse qui correspond à une certaine somme gelée pour pouvoir participer au consensus. Dans notre implémentation, on assigne directement une valeur qu'on appelle *stake* sans avoir à passer par cette première phase d'ajout de *stakes* via transaction.
- **Pénalités :** En cas de mauvais comportement, des pénalités sont définies selon le degré de l'infraction. Ces pénalités n'ont pas été implémentées, car d'une part, il n'y a aucun exemple d'agent byzantin et d'autre part, les pénalités impliquent en général la perte de stakes, et l'ajout de stakes via transactions n'a également pas été implémenté. Cet aspect de Tendermint est vaste et soumis à l'étude.

5.3.3 Utilisation du modèle Tendermint dans MAX

Comme expliqué au début de la section 4.6.1, l'utilisation de MAX est basée sur des scénarios mettant l'accent sur certains aspects du protocole, voici quelques exemples de scénarios que j'ai créé :

— **Tendermint Network :**
Ce scénario a été créé afin de montrer l'utilisation de Tendermint, son bon fonctionnement, ainsi que l'évolution des états des agents pendant l'exécution. Le but est de montrer l'environnement Tendermint et l'activité de création de *blocks* grâce à ce nouveau modèle de création, le BFT.

— **Tendermint Network with Delay :**
Avec les mêmes propriétés que le scénario précedent excepté le délai de transmission entre les agents qui sera modifié. Ce scénario a pour but d'étudier l'impact de la modification du délai sur Tendermint, un algorithme de création de *blocks* basé sur le consensus, et donc sur la communication.

— **Tendermint Network with Reliability :**
Dans ce dernier scénario, nous mettons l'accent sur la fiabilité du réseau. En effet, Tendermint est basé sur l'échange de messages, leur bonne réception et leur traitement sont donc des étapes cruciales pour le bon foncitonnement de l'algorithme. Ce scénario aura donc pour but d'étudier cet aspect.

Voici un exemple de l'exécution d'un scénario "Tendermint Network" (voir la création du scénario en annexe C) :

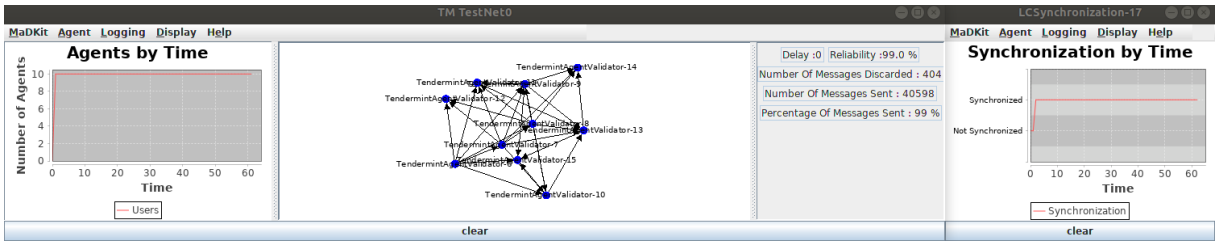

Figure 16 – Environnement Tendermint

Comme le montre la figure 16, ce scénario crée un environnement contenant 10 agents qui utilisent le protocole Tendermint.

Les propriétés de l'environnement également apparentes sur la figure montrent les conditions optimales pour l'exécution d'un tel protocole. à ce moment de l'exécution, les agents sont parfaitement synchronisés, ce qui est une conséquence au fait que le délai de transmission est nul.

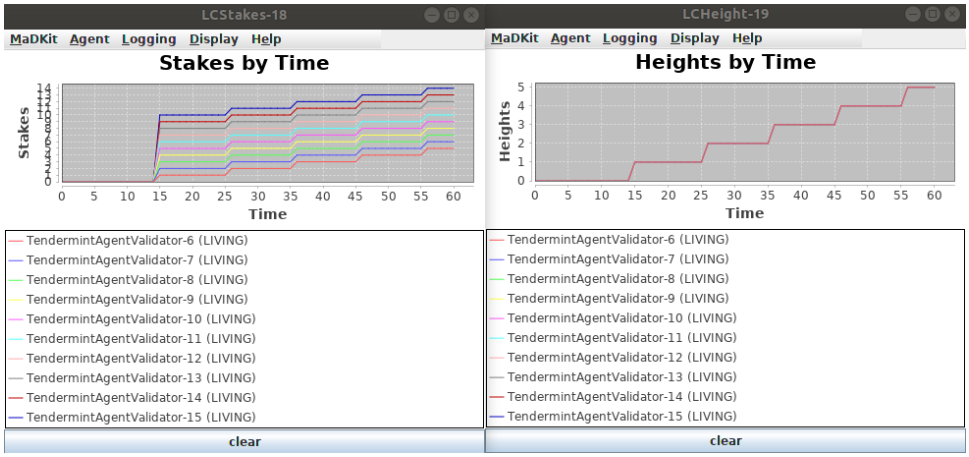

Figure 17 – Résultats Tendermint

La figure 17 en revanche, met l'accent sur l'évolution des états des agents au cours de la simulation. En effet, au fur et à mesure que les *blocks* sont créés, les agents sont récompensés et ont donc plus de stakes qu'ils pourront convertir pour les utiliser.

Cet exemple de scénario montre l'utilité du protocole Tendermint en termes d'efficacité et de stabilité de la chaine car il est à noter que grâce à ce modèle de création à plusieurs, il n'y a pas de forks possibles. Ce modèle utilise également beaucoup moins d'energie que le PoW, mais demande en contrepartie l'échange d'un plus grand nombre de messages.

6 Travaux Connexes

Dans la littérature, afin de tester efficacement un système *blockchain*, il existe deux approches principales, les test-nets et la simulation. Les deux approches présentent des avantages et des inconvénients. Dans la simulation, il y a des simulateurs dédiés à la *blockchain* comme MAX (*Multi-Agent eXperimenter*), ou encore des simulateurs plus généraux qui peuvent être utilisés pour de la *blockchain*, c'est ce qui sera présenté dans la suite :

6.1 Test-Nets

Les test-nets servent à faciliter le développement d'un système en proposant une solution d'implémentation réaliste du système. Lorsqu'il s'agit de cryptomonnaies, les test-nets permettent aux utilisateurs d'échanger des faux jetons sans réelle valeur pour tester le système.

De cette façon, un utilisateur peut se connecter à un test-net et l'essayer sans avoir peur de perdre de l'argent.

Les compagnies commercialisant des services *blockchain* mettent donc en place des test-nets pour, à la fois tester leur produit, et permettre à de potentiels utilisateurs de faire un essai gratuit. Il existe par exemple, pour Bitcoin [1], Bitcoin Faucet Testnet, pour Ethereum [2], on a Ropsten, RinkeBy, Kovan, ainsi que Görli, et pour Tezos [20], le réseau Alphanet est mis à disposition des utilisateurs.

Il est également possible de déployer un réseau privé dans un environnement local où on connecterait des nœuds utilisant un certain protocole.

Dans ces réseaux privés, le seul paramètre sur lequel on a accès, est le nombre de nœuds puisque pour configurer ce réseau, il faut avoir les adresses des nœuds, ainsi que les numéros de port si on en a plusieurs sur une même machine. Dans Tezos, on peut lancer l'application en mode "sandboxed" et spécifier les adresses des pairs du réseau [21]. Dans Tendermint, on a juste à modifier le fichier de configuration en y ajoutant les adresses des pairs.

6.2 Simulateurs dédiés à la *blockchain*

— **SimBlock [22]**
SimBlock est un simulateur avec avancement par évènement où chaque nœud génère des évenements de message et de création de *blocks*. Utilisé en fonction de paramètres relatifs aux *blocks* (taille, intervalle de création), aux nœuds(nombre, connections maximales, position...), mais également au réseau (délai de transmission).
Chaque nœud a une capacité de création de *block* définie, étant donné que dans ce simulateur, les *blocks* sont créés en PoW, cela signifie que créer un *block* a une certaine difficulté (paramètre également défini). Même si les nœuds et la topologie du réseau sont dynamiques, SimBlock ne précise pas si le système est dynamique, c'est-à-dire si les nœuds peuvent entrer et sortir du réseau.

Dans SimBlock, il n'existe qu'un seul environnement sur lequel les agents peuvent interagir. De plus, il n'y a aucun mécanisme permettant l'orchestration d'un scénario, ni même la verification et la validation des modèles.

— **Le travail de Kaligotla et al. [23]**
Le travail de Kaligotla et al. propose un framework basé agents servant à la modélisation de systèmes *blockchains*, en se concentrant sur les élements essentiels de la *blockchain* et en implémentant des mécanismes de mesure de l'efficacité du système du point de vue écologique (économie d'énergie dans le PoW).
Ce simulateur modélise un réseau *blockchain* avec ses participants et des transactions. Les élements les plus importants d'un tel système comme le comportement des agents, et leurs prises de décisions sont détaillés dans leur modèle. Les agents sont divisés en deux catégories, les utilisateurs normaux qui créent des transactions, et les mineurs qui créent des *blocks* en utilisant le PoW. Il n'est cependant pas précisé si leur environnement est paramètrable, c'est à dire, si on peut modifier des paramètres tel que le délai ou encore la fiabilité de transmission.
Dans ce simulateur également, comme dans SimBlock, les agents n'appartiennent qu'à un seul environnement sur lequel ils interagissent. De plus, aucune fonction de planification de scénario ou même de test n'est mise à disposition.

— **Le travail de Memon et al. [24]**
Memon et al. proposent une modélisation d'un système *blockchain* en se basant sur une simulation de file d'attente. Ils assimilent le processus de création de *block* en PoW, le mining à une file d'attente dans un routeur. Ils ont utilisé les paramètres de Bitcoin et l'ont simulé en utilisant un système de file d'attente M/M/n/L avec JSIMgraph. Dans cette simulation, il n'y qu'une seule file d'attente et un certain nombre de mineurs. Les transactions sont créées de façon aléatoire. Ils ne modélisent pas tout à fait le système distribué qu'est la *blockchain*, ni aucun protocole de communication d'aucune sorte. Comme pour les autres simulateurs que nous avons étudié, celui-ci ne propose pas non plus de mécanismes de vérification ou d'orchestration, et ne permet pas non plus l'appartenance d'un agent à plusieurs environnements.

— **Le travail de Piriou et al. [25]**
Piriou et al. proposent une modélisation stochastique de la *blockchain*. Un modèle stochastique est un outil permettant d'estimer la distribution probabiliste liée à la distribution des biens pour une cryptomonnaie. Ceci étant réalisé grâce à la variation aléatoire de certains paramètres au fil de l'exécution.
Comme les autres simulateurs étudiés, aucune fonction de vérification ou d'orchestration et les nœuds ne peuvent appartenir qu'à un environnement. Néanmoins, Cette solution étudie le PoW, ainsi que le PoS, contrairement aux simulateurs susmentionnés.

6.3 Simulateurs généraux

Pour de la simulation de *blockchains*, il n'est pas nécessaire d'utiliser des simulateurs dédiés, étant donné qu'il s'agit avant tout d'un réseau distribué, il est également possible d'utiliser des simulateurs plus généraux.

1) Simulateurs de systèmes distribués :

Dans la simulation de systèmes distribués, les deux simulateurs les plus utilisés sont OMNeT++ et NS-3.

— **OMNeT++ [26] :** est un framework et une librairie de simulation à évènements discrets, utilisé essentiellement dans les réseaux. Ce simulateur est dédié à la modélisation réseau, il se concentre donc sur les aspects réseaux, tel que la connection de pairs ou encore le transfert à travers des sockets. Dans OMNeT++, toutes les actions sont basées sur des messages.
Le simulateur propose une fonction de planification d'envoi de messages via la fonction scheduleAt(). Si un nœud veut créer planifier une action, la seule façon de faire est de s'envoyer un message avec scheduleAt(), puis d'exécuter la fonction à la réception du message.
Dans OMNeT++, les topologies sont définies par le développeur qui doit créer les nœuds et les connecter entre eux manuellement. Il est possible d'ajouter des nœuds à un moment donné, mais impossible pour un nœud d'en créer d'autres, et donc d'avoir un nœud gestionnaire de simulation. Grâce à cette fonctionnalité, ce simulateur rend possible la création de réseaux dynamiques avec l'ajout ou la suppression de nœuds en cours de simulation. Les nœuds n'ont accès qu'à un seul environnement, mais il leur est possible d'avoir plusieurs rôles (en exécutant différents protocoles par exemple).
Pour ce qui est des tests, opp_test est un outil mis à disposition pour des tests unitaires ou encore des tests de régression.

— **NS-3 [27] :** est un simulateur à évenements discrets de réseaux. Essentiellement utilisé à des fins de recherche ou d'éducation, il met l'accent sur les propriétés réseau. Il n'est cependant pas possible de simuler des systèmes dynamiques où les nœuds peuvent entrer et sortir, NS-3 permet seulement d'ajouter des nœuds et des liens, pas de les supprimer. Par contre, il est possible d'éteindre certaines interfaces de certains nœuds afin de "simuler" une sortie. Comme avec OMNeT++, les nœuds ne peuvent accéder qu'à un seul environnement.
NS-3 permet aux développeurs de créer des tests avec un framework dédié et intégré au simulateur [28].

2) Simulateurs basés agents :

JaCaMo [29] est un framework de programmation multi-agents qui combine trois différentes technologies, Jason pour la gestion des agents, CArtAgO pour l'environnement et Moise pour l'organisation. Il facilite la création de test-nets privés grâce à la possibilité de lancer plusieurs nœuds JaCaMo sur la même machine physique en utilisant différents ports.

JaCaMo est organisé en workspaces où les agents sont créés, ils pourront ensuite à leur tour créer, rejoindre ou quitter d'autres workspaces sur un nœud (qu'il soit local ou distant). Chaque agent a accès aux informations spécifiques aux workspaces auxquels il appartient, il a de cette façon une perception globale de l'environnement dans lequel il évolue.

JaCaMo-web [30] est une API REST qui facilite l'utilisation des agents / environnements tournant sur un nœud JaCaMo. Chaque élément du système est représenté par une ressource Web accessible via URL. Par exemple : "`http://<node>/<mas_name>`" est le répertoire principal d'un MAS (*Multi-Agent System*), "`http://<node>/<mas_name>/main`" représente le workspace principal d'un MAS.

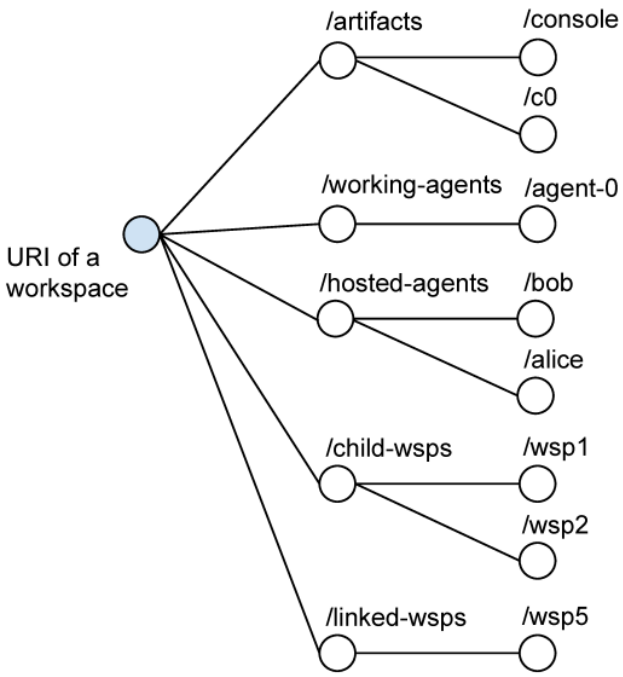

Figure 18 – Organisation JaCaMo-web

Comme le montre la figure 18, à l'intérieur d'un workspace, on a :
— /working-agents, listant les agents liés à ce workspace ;
— /hosted-agents, listant les agents qui ont accédé au workspace via host/home workspace ;
— /child-wsps, listant les sous-workspaces ;
— /linked-wsps, listant les workspaces liés à celui-ci.

L'utilisation de JaCaMo se fait essentiellement via des requêtes GET et POST, voici quelques exemples :
— Pour ajouter un agent : requête POST sur le répertoire /hosted-agents.
— Pour créer un workspace : requête POST sur le répertoire /child-wsps.
— Pour lier 2 workspaces : requête POST sur le répertoire /linked-wsps et spécifier le workspace cible.

3) Autres simulateurs :

— **SimEvents [31]**
SimEvents est une des librairies de Matlab ; qui est un langage haut niveau utilisé essentiellement pour la visualisation et le développement d'applications. Cette librairie permet la planification d'évenements et la simulation réseau.
Elle permet la création d'agents, leur interaction et l'échange de messages entre eux. Pour ce qui est de la dynamicité du système simulé, elle n'est pas garantie mais devrait être facile à implémenter étant donné que c'est basé sur le langage Matlab.
Il existe par exemple une application Matlab qui implémente une *blockchain* où un certain nombre de nœuds pourront être lancés et exécuter un protocole qui consistera à créer des *blocks* et les distribuer [32].

— **Simmer [33]**
Simmer est une librairie de R, qui est un langage de programmation utilisé pour le calcul statistique.
Il s'agit d'un framework génerique qui permet la simulation à évènements discrets et qui exploite un concept nouveau : les trajectoires. Il s'agit d'une "voie à suivre" pour les agents d'un même type, ce qui va définir leur comportement et donc leurs actions. En d'autres mots, une trajectoire est une liste d'actions à la suite qui définissent le comportement d'un agent.

— **Agent Based Modelling in R [34]**
ABM-R est une autre librairie de R, utilisée pour la modélisation basée agent. Dans ABM-R, les modèles sont constitués d'agents pouvant interagir entre eux et avec leur environnement, où ils pourront également se déplacer. Les déplacements et communications des agents sont régies par des régles qui définissent ce qu'un agent a le droit ou non de faire.

7 Conclusion

Cette année passée au sein du LICIA, au CEA-LIST a été une année enrichissante, aussi bien techniquement qu'humainement. J'ai eu la chance de faire partie d'une équipe aussi professionnelle qu'accueillante, ce qui m'a permis de travailler dans un environnement agréable et ainsi donner le meilleur de moi-même.

Au cours de cette alternance passée dans un laboratoire de recherche, j'ai pu en apprendre plus sur le monde de la recherche, mais également sur l'industrie, car en plus de faire de la recherche, le CEA-LIST est également très impliqué dans l'industrie et les technologies nouvelles comme la *blockchain*. Malgré la jeunesse de ce sujet, beaucoup de grandes entreprises s'y intéressent et j'ai eu la chance d'y participer en apportant ma contribution à l'étude et l'amélioration des protocoles les plus en vue.

J'en ai également appris beaucoup sur le monde de l'entreprise, le travail en équipe, ainsi que les différents outils permettant cette cohésion entre les membres d'une même équipe, et qui sont essentiels à l'aboutissement d'un projet. J'ai par exemple été au centre du développement d'un projet, qui en lui-même m'a été très bénéfique et qui m'a également permis d'avoir une meilleure connaissance du monde de l'ingénierie logicielle.

Grâce à cette nouvelle expérience de développement, j'ai pu me former et approfondir mes connaissances en développement orienté objet et en intelligence artificielle. De plus, étant donné que j'ai été confronté au travail en équipe, j'ai également pu en apprendre plus sur les différents outils utilisés en entreprise tel que Gitlab, ou encore Nexus. Cette expérience m'a permis d'acquérir de nouvelles connaissances. Parmi celles-ci, le *testing*, l'intégration et le déploiement continus, ainsi que l'utilisation de Maven, un outil de gestion de dépendances.

L'implémentation du protocole Tendermint, ainsi que toutes les tâches dont j'ai été en charge pour permettre l'aboutissement de mon sujet arrivent comme la conclusion de ma formation. Et ce, avec une mise en pratique des aspects réseaux que j'ai pu étudier au cours de mon cursus, mais également l'utilisation de pratiques nouvelles, ou peu utilisées dans le monde des réseaux. Parmi ces bonnes pratiques, on retrouve la simulation basée agents notamment, l'utilisation de frameworks, ou encore l'étude approfondie d'un protocole pour son implémentation.

Mis à part les outils utilisés et les pratiques que j'ai apprises, le sujet d'alternance en lui-même est très intéressant. En effet, le protocole Tendermint est différent des autres protocoles *blockchain* car il utilise un modèle de création de *blocks* basé sur consensus. C'est une nouvelle façon de faire qu'il est intéressant d'étudier plus en détails, voilà pourquoi mon sujet concerne l'implémentation de ce protocole.

Le simulateur en lui-même est fonctionnel et implémente les différents modèles cités dans ce rapport. Il est donc possible en utilisant MAX, de simuler, d'étudier et d'améliorer Tendermint. De plus il est nécessaire de rappeler qu'à la date de rendu de ce rapport, mon alternance n'est pas encore terminée, puisqu'elle se termine le 30 septembre 2019. Ainsi donc, pendant le mois restant, d'autres améliorations seront apportées au simulateur pour lui permettre d'être encore plus performant. On pourra également améliorer l'implémentation de Tendermint pour la rendre plus réaliste ou encore proposer des améliorations à appliquer au protocole lui-même.

Après mon alternance, il est prévu que je débute une thèse de doctorat avec le même laboratoire où j'ai travaillé pendant cette année, le LICIA. Cette thèse traiterait de "l'étude du potentiel des approches à base de graphes et de "proof of stake" pour les cryptomonnaies avec ou sans permission". Ce sujet nécessitera, entre autres choses, d'avoir recours à des approches expérimentales tel que la simulation, et donc l'utilisation, ainsi que l'amélioration de MAX. Le simulateur sur lequel j'ai travaillé tout au long de cette alternance.

Références

[1] Satoshi Nakamoto. *Bitcoin : A peer-to-peer electronic cash system*. 2009. URL : `http://www.bitcoin.org/bitcoin.pdf`.
Papier original présentant le protocole Bitcoin.

[2] Vitalik Buterin. *Ethereum : A next-generation smart contract and decentralized application platform*. 2014. URL : `https://github.com/ethereum/wiki/wiki/White-Paper`.
Papier original présentant le protocole Ethereum.

[3] Jae Kwon. *Tendermint : Consensus without Mining*. 2014. URL : `https://cdn.relayto.com/media/files/LPgoWO18TCeMIggJVakt_tendermint.pdf`.
Papier original présentant le kit de Développement multi-agents MaDKit.

[4] *Analyse et comparaison des mécanismes de consensus dans la blockchain*. URL : `https://medium.com/@godefroy.galas/analyse-et-comparaison-des-m%C3%A9canismes-de-consensus-dans-la-blockchain-f91aee511ea3`.
Article traitant dees modèles de création de blocks en général.

[5] *PoW vs PoS*. URL : `https://blockgeeks.com/guides/proof-of-work-vs-proof-of-stake/`.
Article traitant des deux modèles de création de blocks les plus connus, PoW et PoS.

[6] *Understanding Blockchain Fundamentals, Part 1 : Byzantine Fault Tolerance*. URL : `https://medium.com/loom-network/understanding-blockchain-fundamentals-part-1-byzantine-fault-tolerance-245f46fe8419`.
Article traitant du modèle de création par consensus, BFT.

[7] *Tendermint protocol*. URL : `https://tendermint.com/docs/introduction/what-is-tendermint.html`.
Documentation de Tendermint fournissant plus détails sur l'algorithme utilisé.

[8] *Eclipse IDE*. 2019. URL : `https://www.eclipse.org`.
Site officiel de l'IDE Eclipse.

[9] *Gitlab*. 2019. URL : `https://about.gitlab.com`.
Site officiel de l'outil DevOps Gitlab.

[10] *JUnit 5 : The new major version of the programmer-friendly testing framework for Java*. 2019. URL : `https://junit.org/junit5/`.
Site officiel de l'outil de test unitaire JUnit 5.

[11] *Apache Maven Project*. 2019. URL : `https://maven.apache.org`.
Site officiel de l'outil de gestion de dépendances Maven.

[12] Thi-Mai-Trang Nguyen. *Simulation, Emulation et Virtualisation*. 2018.
Cours universitaire sur les généralités de la simulation et OMNET++.

[13] Önder Gürcan. "Exploration of Biological Neural Wiring Using Self-Organizing Agents". Thèse de doct. 2013.
Thèse de doctorat traitant de la simulation basée agents.

[14] Önder Gürcan. "Multi-Agent Modelling of Fairness for Users and Miners in Blockchains". In : *2nd Workshop on Block Chain Technologies 4 Multi-Agent Systems (BCT4MAS), co-located with PAAMS 2019, Avila, Spain, June 26-28, 2019*. 2019.
Papier traitant de la simulation basée agents dans les blockchains, ainsi que de l'équité entre utilisateurs dans ces systèmes.

[15] Fabien Michel, Pierre Bommel et Jacques Ferber. "Simulation Distribuée Interactive sous MaDKit". In : 2002.
Papier original présentant le kit de Développement multi-agents MaDKit.

[16] Brian Curran. *What is a Merkle Tree ? Beginner's Guide to this Blockchain Component*. 2018. URL : `https://blockonomi.com/merkle-tree/`.
Article expliquant le MerkleTree, son fonctionnement et son utilité.

[17] Jing-chao Chen. *Dijkstra's Shortest Path Algorithm*. 2003.
Papier original présentant l'algorithme du plus court chemin Dijkstra.

[18] Yoann Pigné et al. "GraphStream : A Tool for bridging the gap between Complex Systems and Dynamic Graphs". In : (2008). URL : `http://dblp.uni-trier.de/db/journals/corr/corr0803.html#abs-0803-2093`.
Article décrivant GraphStream, une librairie Java utilisée pour la modélisation et l'analyse de graphes dynamiques.

[19] Joseph Poon et Thaddeus Dryja. *The Bitcoin Lightning Network :Scalable Off-Chain Instant Payments*. 2016. URL : `https://lightning.network/lightning-network-paper.pdf`.
Papier original présentant le protocole Lightning.

[20] Mathias Bourgoin. *An overview of the Tezos blockchain*.
Papier original présentant le protocole Tezos.

[21] *Tezos Documentation*. URL : `https://tezos.gitlab.io/master/index.html`.
Documentation de Tezos fournissant plus détails sur le protocole.

[22] Yusuke Aoki et al. *SimBlock : A Blockchain Network Simulator*. 2019. URL : `https://arxiv.org/pdf/1901.09777.pdf`.
Papier original présentant l'un des simulateurs existants cités dans l'état de l'art sur les simulateurs blockchain.

[23] Chaitanya Kaligotla et Charles M Macal. "A Generalized Agent Based Framework for Modeling a Blockchain System". In : *Proceedings of the 2018 Winter Simulation Conference*. 2018. URL : `http://dl.acm.org/citation.cfm?id=3320516.3320643`.
Papier original présentant l'un des simulateurs existants cités dans l'état de l'art sur les simulateurs blockchain.

[24] Raheel Memon et al. "Modeling of Blockchain Based Systems Using Queuing Theory Simulation". In : 2018.
Papier original présentant l'un des simulateurs existants cités dans l'état de l'art sur les simulateurs blockchain.

[25] Pierre-Yves Piriou et Jean-Francois Dumas. "Simulation of Stochastic Blockchain Models". In : 2018.
Papier original présentant l'un des simulateurs existants cités dans l'état de l'art sur les simulateurs blockchain.

[26] András Varga et Rudolf Hornig. "An Overview of the OMNeT++ Simulation Environment". In : *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*. 2008. URL : http://dl.acm.org/citation.cfm?id=1416222.1416290.
Papier original présentant le simulateur OMNET++.

[27] George F. Riley et Thomas R. Henderson. "The ns-3 Network Simulator." In : *Modeling and Tools for Network Simulation*. 2010. URL : http://dblp.uni-trier.de/db/books/collections/Wehrle2010.html#RileyH10.
Papier original présentant le simulateur NS-3.

[28] *NS3 Testing Framework*. URL : https://www.nsnam.org/docs/release/3.9/testing.html#TestingFramework.
Outil de test fourni par NS-3.

[29] Olivier Boissier et al. "Multi-agent oriented programming with JaCaMo". In : (2011). URL : https://hal-emse.ccsd.cnrs.fr/emse-00680402.
Papier original présentant le simulateur basé agents JaCaMo.

[30] Alessandro Ricci et al. "Engineering Scalable Distributed Environments and Organizations for MAS". In : *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems*. 2019. URL : https://www.alexandria.unisg.ch/256719/.
Papier original présentant l'interface web créée pour JaCaMo, nommée JaCaMo-web.

[31] *SimEvents*. URL : https://www.mathworks.com/products/simevents.html.
Papier original présentant l'un des simulateurs existants cités dans l'état de l'art sur les simulateurs basé agent.

[32] *Matlab Blockchain Example*. URL : https://fr.mathworks.com/matlabcentral/fileexchange/65419-matlab-blockchain-example.
Exemple de simulateur blockchain créé avec Matlab.

[33] *simmer*. URL : https://r-simmer.org/articles/%20simmer-02-jss.pdf.
Papier original présentant l'un des simulateurs existants cités dans l'état de l'art sur les simulateurs basé agent.

[34] Marco Smolla. *An introduction to Agent-Based Modelling in R*. 2015.
Papier original présentant l'un des simulateurs existants cités dans l'état de l'art sur les simulateurs basé agent.

[35] Tata Communications. *The Cycle of Progress*. 2018. URL : `https://www.tatacommunications.com/wp-content/uploads/2018/12/Cycle_of_Progress_infographic.pdf`.
Article sur l'intérêt des grandes entreprises pour la Blockchain.

A D'autres protocoles modélisés dans MAX

Après avoir créé les modèles abstraits nécessaires à l'implémentation de protocoles. On est passé à la création de modèles plus concrets qui pourront être exécutés et nous permettront donc de simuler les protocoles et ainsi, les étudier.

Le but de mon alternance étant d'implémenter Tendermint dans MAX, nous avons décidé de commencer par d'autres protocoles plus simples, que sont Bitcoin et Bitcoin-Lightning pour me familiariser avec le framework, mais également pour en apprendre plus sur la *blockchain* en général avant de passer à un protocole complexe tel que Tendermint. Il a également fallu implémenter le protocole p2p nécessaire au fonctionnement de ces protocoles.

A.1 Pair-à-pair

Après un bref état de l'art sur les protocoles existants, nous sommes passés à l'implémentation. les points les plus importants que j'ai pris en compte sont :

- **Group :**
  Comme pour chaque modèle, il a fallu créer un groupe qui lui soit propre avec des rôles qui lui sont propres également, à savoir :
  - Scheduler
  - Experimenter
  - Environment
  - User

- **Environment :**
  Les connexions ne sont que des liens logiques, l'envoi de messages se fait via l'environnement qui délivre le message au.x destinataire.s. l'utilité de la *connectionList* est qu'on ne peut envoyer de messages qu'aux agents contenus dans cette liste.

- **Propriétés du protocole :**
  - **Maximum in/out connections :** propriétés du protocole qui définissent le nombre maximal de connexions sortantes et le nombre maximal de connexions entrantes.
  - **Timeout :** le temps d'inactivité à attendre avant de couper une connexion.

- **Messages du protocole :**
  On a mis en place trois types de messages pour pouvoir créer des connexions entre les agents.
  - **CONNECT :** message de demande de connexion.
  - **ACCEPT :** message qui signifie l'acceptation de la connexion.
  - **REJECT :** message qui signifie le rejet de la connexion à cause d'un trop grand nombre de connexions déjà existantes.

— **Connection :**
Représente le lien grâce auquel deux agents peuvent communiquer.

— **Connection list :**
La liste des connexions qu'un agent a, elle regroupe les connexions entrantes / sortantes de l'agent, une nouvelle entrée est créée à l'envoi (par le destinataire de la requête) et à la réception d'un message ACCEPT (par l'emetteur de la requête).

— **Waiting list :**
Une liste de connexions en attente de réponse, afin de limiter l'envoi de requêtes pour pouvoir respecter les propriétés du protocole.

A.2 Bitcoin

Bitcoin [1] est le pionnier des crypto-monnaies, créé en 2008 par Satoshi Nakamoto pour se débarasser de la centralisation du système monétaire où on a besoin d'un tiers de confiance pour chaque transaction, il est le premier protocole à se reposer sur la technologie *blockchain*.

Dans Bitcoin, les utilisateurs signent avec leur clé privée les transactions qu'ils veulent envoyer avant de les diffuser au réseau, le principe est donc basé sur la cryptographie, mais pas seulement pour les transactions. En effet, après avoir diffusé les transactions, celles-ci sont connues du réseaux, mais invalides, c'est au tour des mineurs (créateurs de *block* du protocole Bitcoin) de les valider en les ajoutant dans des *blocks* qu'ils vont lier à la *blockchain*.

La création du *block* se fait sous certaines conditions, pour Bitcoin, le mécanisme de création est le PoW, cela signifie que le mineur doit prouver qu'il a assez travaillé pour créer le *block*, ce qui se traduit par un challenge cryptographique que le mineur va résoudre le plus rapidement possible pour être le premier à valider le *block* et donc recevoir la récompense.

Ce challenge cryptographique consiste simplement à proposer un *block* qui ait un *hash* inférieur à un *hash target* défini au préalable. Cette opération nécessite de modifier le nonce contenu dans le *block* jusqu'à trouver un *hash* qui correspond.

Cela engendre bien évidemment des problèmes, dont les plus connus :

— **Gaspillage énergétique :**

Á cause de cette récompense, tous les mineurs veulent créer le plus de *blocks* possible pour s'enrichir. Malheureusement, chacun utilise une certaine puissance de calcul pour tenter de créer le *block*, et donc une certaine quantité d'energie. Plus le nombre de mineurs qui tentent de créer le même *block* est grand, plus la consommation énergétique est élevée. Étant donné qu'au final, seul un de ces utilisateurs sera récompensé, de l'énergie aura été gaspillée pour rien, ce qui a un impact financier et écologique important.

— **Fork :**

Sous certaines conditions, il peut arriver que deux *blocks* soient créés par deux mineurs différents au même moment. Dans ce cas, la *blockchain* diverge pour former deux branches. Bien évidemment, une de ces branches sera privilégiée à un moment, et l'autre sera supprimée. Ce qui signifiera que les transactions contenues dans cette branche seront annulées, et les mineurs qui ont créé les *blocks* de cette branche ne seront pas récompensés.

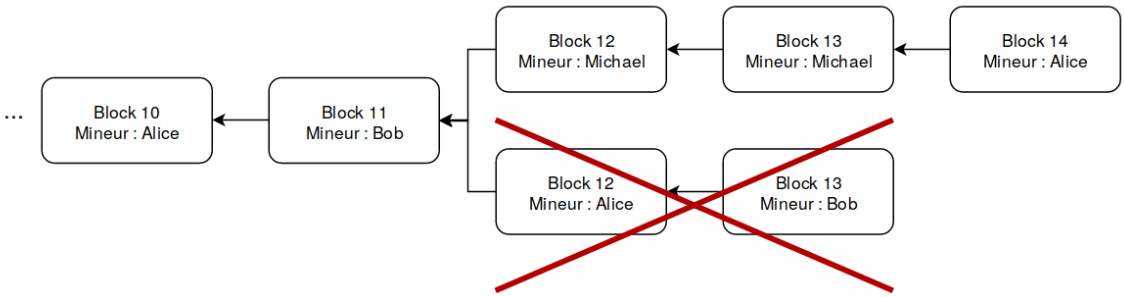

Figure A.1 – Exemple de fork

A.2.1 Propriétés de Bitcoin :

Le réseau bitcoin a des propriétés qui lui sont propres, qui nous permettront de mieux étudier le protocole et donc de proposer des améliorations. Les propriétés les plus importantes sont :

— Intervalle de création des *blocks* : 10 minutes.
— Taille maximale de *block* : 1 MB
— Nombre maximum de transactions par *block* : 4000 txs
— Récompense minimale perçue par un mineur : 12.5 BTC
— Coût d'une transaction : très variable mais diffère selon l'urgence de la transaction, il coûte moins cher d'envoyer une transaction à valider 6 *blocks* plus tard qu'une transaction à valider dans le prochain *block*.

A.2.2 Messages Bitcoin :

Les nœuds Bitcoin interagissent entre eux en échangeant des messages, selon le type et le contenu du message reçu, le comportement d'un nœud va changer, les messages les plus important sont listés comme suit :

— **VER** : annoncer la version du protocole utilisé, le récepteur répond avec sa version. Aucune communication n'est possible sans l'échange de ce message VER.

— **VERACK** : répondre à VER avec le numéro de version qu'utilise le nœud.
— **GETADDR** : demander des informations sur de potentiels nœuds actifs pour s'y connecter et participer au réseau.
— **ADDR** : répondre à GETADDR contenant des informations sur les nœuds actifs connus.
— **INV** : permet à un nœud d'annoncer aux autres qu'il a une nouvelle information, tel qu'une transaction ou un *block*. Ce message peut être reçu spontanèment ou en réponse au message GETBLOCKS.
— **GETDATA** : est utilisé en réponse à INV, pour avoir le contenu de la transaction (ou du *block*) annoncé.e par le message INV.

— **GETBLOCKS** : généralement envoyé à la connexion pour demander la liste des *blocks* que le nœud n'a pas. Ce message contient le *hash* du dernier *block* dont l'emetteur a pris connaissance, et a pour réponse un message INV contenant la liste des *blocks* manquants.
— **TX** : message contenant une transaction, en réponse à GETDATA.
— **BLOCK** : message contenant un *block*, en réponse à GETDATA
— **NOTFOUND** : est une réponse possible à GETDATA, envoyée si la donnée demandée n'est pas disponible.

A.2.3 Modélisation de Bitcoin dans MAX :

Dans notre implémentation, le modèle Bitcoin étend le modèle *blockchain* qui à son tour étend network.p2p, de la même façon que le protocole Bitcoin est une application *blockchain* qui elle-même est distribuée sur un réseau de nœuds interconnectés.

Nous avons essayé de représenter Bitcoin de la manière la plus réaliste possible, mais comme avec toute modélisation, il est indispensable d'omettre certains points qu'on juge peu importants ou sur lesquels on ne voudrait pas se focaliser dans l'immédiat.

Oracle :

La création de *block* dans Bitcoin utilise l'oracle créé dans le modèle Blockchain avec comme mode de création, du PoW exclusivement, comme le stipule le protocole Bitcoin.

Cette méthode prend en compte un paramètre de l'agent "*computational power*" (puissance de calcul) pour décider du créateur de *block*. Ce paramètre est un nombre aléatoire compris entre 0 et 100 attribué à la création de l'agent.

L'oracle a deux modes de fonctionnement en Bitcoin, PoW sans fork et avec fork. Dans l'un, un seul agent est choisi pour créer un *block*, et dans l'autre il y a une certaine probabilité (définie selon les statistiques réelles d'occurence d'un fork dans le réseau Bitcoin) pour que 2 agents créent un *block* en même temps et le diffusent sur le réseau. Ce qui aura comme effet de créer un fork, car tout le monde n'aura pas la même copie de la *blockchain* pendant un instant, on dit donc qu'elle est instable.

Rôles :
Les rôles créés pour les besoins de la simulation de Bitcoin sont entre autres choses les mêmes que dans le modèle que Bitcoin étend, c'est-à-dire Blockchain. Avec un rôle supplémentaire propre à Bitcoin, le rôle "*Miner*", qui lui donne accès aux méthodes de création de *blocks*.

Messages :
Comme avec network.p2p, nous avons mis en place certains types de messages nécessaires au fonctionnement du protocole, mais en omettant certains qui, selon nous, n'influaient pas sur le bon fonctionnement du protocole.
Ces messages sont :
- **INV :** nécessaire pour annoncer aux voisins qu'une nouvelle information est disponible.
- **GETDATA :** nécessaire pour demander une information.
- **TX :** nécessaire pour envoyer une transaction.
- **BLOCK :** nécessaire pour envoyer un *block*.
- **GETBLOCKS :** nécessaire pour demander les *blocks* qui manquent à un agent à sa connexion.

A.2.4 Utilisation du modèle Bitcoin dans MAX :

Comme expliqué au début de la section 4.6.1, l'utilisation de MAX est basée sur des scénarios mettant l'accent sur certains aspects du protocole, voici quelques exemples de scénarios que j'ai créé pour l'étude de Bitcoin :

- **Bitcoin Interactive :**
  Ce scénario a surtout été créé dans un but démonstratif, il permet de montrer l'utilisation du client Bitcoin que j'ai moi même créé, et donc permettre plus d'interactivité avec le simulateur. Cet exemple de scénario utilise un modèle de PoS ne choisissant qu'un seul mineur pour chaque création de *block*, ce modèle a le même fonctionnement que le PoW à l'exception près qu'il n'utilise pas le paramètre "*computational power*", mais plutôt la "*balance*", c'est-à-dire les richesses de l'agent.

- **Bitcoin X Users Y Miners :**
  Ce scénario se concentre sur la taille du système, le but de ce scénario est d'étudier la réaction de Bitcoin face à un grand nombre d'agents évoluant en même temps. Cet exemple de scénario utilise un modèle de PoW ne choisissant qu'un seul mineur pour chaque création de *block*.

- **Bitcoin With Quitting :**
  Ce dernier scénario permet d'étudier la dynamicité du système et l'influence de la déconnexion/reconnexion d'agents sur le fonctionnement de l'application. Ce scénario utilise un modèle de création de *blocks* en PoW avec possibilité de forks.

Voici les résultats d'une simulation de ce dernier scénario "Bitcoin With Quitting" :

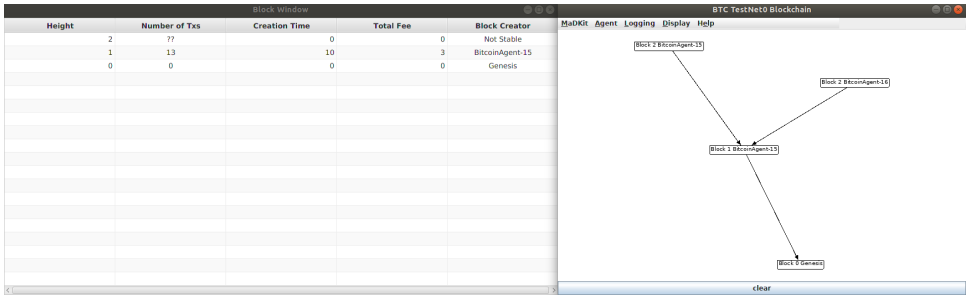

Figure A.2 – Simulation de fork

Sur la figure A.2, on a un exemple de fork avec, à droite une visualisation de la *blocktree* où on voit qu'il y a deux *blocks* "2" créés par des mineurs différents. Á gauche de la figure, une représentation de la "*main chain*" (chaine principale) sous forme de tableau. Dans cet exemple, la *blockchain* est considérée instable car aucune branche n'est plus longue que l'autre, il est donc impossible de choisir une chaine principale.

Étant donné que ce scénario met l'accent sur la dynamicité du système, d'autres graphes témoignent du bon fonctionnement de l'application.

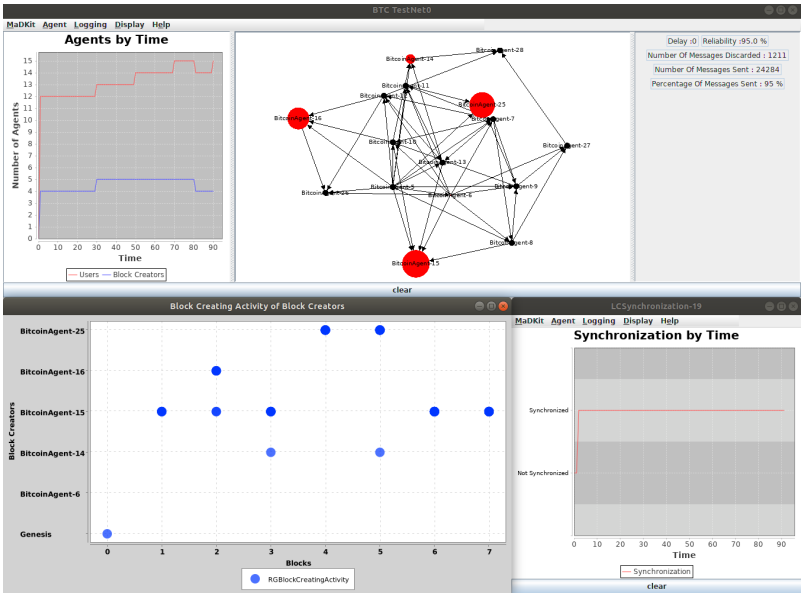

Figure A.3 – Étude de la dynamicité de Bitcoin

La figure A.3 peut être divisée en trois illustrations, la première en haut contient un diagramme de connexion des agents, la topologie du réseau ainsi que les propriétés de l'environnement tel que le délai de transmission.

La seconde partie de la figure montre l'activité de création de *blocks* en fonction des agents, ce qui nous permet de mieux déceler la présence de forks (lorsqu'il y a plus d'un point pour une même abcisse). La dernière partie témoigne de la synchronisation de l'application en fonction du temps.

Cette simulation montre donc une excellente synchronisation malgré les entrées/sorties des agents assez fréquentes, ainsi que la présence de forks. La modification de certains paramètres tel que le nombre d'agents ou le délai de transmission pourrait influer sur les résultats de la simulation, ce qui permet de tirer des conclusions quant aux améliorations possibles.

A.3 BitcoinLightning

Á sa parution en 2008, l'un des problèmes les plus évidents du protocole Bitcoin était l'évolutivité. En effet, avec un *block* toutes les 10 minutes contenant au maximum environ 4000 transactions, on a un débit moyen de 7 transactions par seconde. Étant donné que l'idée est de concurrencer le marché bancaire, Bitcoin est loin de proposer un débit satisfaisant quand on sait qu'en moyenne, Visa tourne sur du 1700 txs/sec.

Une étude publiée par Tata Communications en 2018 [35] montre qu'environ 44% des entreprises qu'elle a sondé pensent à adopter la *blockchain*. Toutefois, le problème de l'évolutivité empêche un grand nombre de sauter le pas.

Dans le but de régler ce problème, des recherches ont été menées et des solutions ont été proposées. Par exemple, augmenter la taille d'un *block* ou réduire la complexité seraient des solutions viables car ce sont là des facteurs importants limitant la capacité à valider des transactions. Dans tous les cas, on se heurte à d'autres problèmes comme la latence du réseau, ce qui rend improbable la compétition avec des institutions tel que Visa.

Toutefois, une toute nouvelle approche appelée Lightning [19] voit le jour en 2016. Elle est basée sur l'idée que "Si un arbre tombe dans la forêt et qu'il n'y a personne autour pour l'entendre, l'arbre a-t-il fait du bruit en tombant ?", ce qui, dans le monde de la *blockchain*, veut dire que si une transaction ne concerne que deux utilisateurs, il n'est pas nécessaire que tout le monde en connaisse les détails.

Dans cette optique, Lightning propose de créer des canaux *lightning* (*Lightning channels*) entre deux utilisateurs. Les couples d'utilisateurs devront créer un *multi-signature wallet*, qui est un *wallet* partagé entre deux ou plusieurs utilisateurs et mettre des fonds dessus sous forme de transactions, qui contrairement à celles passées sur le canal *lightning* devront être validées par la *blockchain*.

Par la suite, les transactions *lightning* ne seront autre qu'un transfert de fonds d'un utilisateur à un autre sur un même *wallet*. Lorsque les utilisateurs voudront fermer ce canal, des transactions contenant ce qui reste de l'acompte de chacun seront envoyées sur la *blockchain*.

De cette façon, deux utilisateurs pourraient échanger une infinité de transactions, alors que sur la *blockchain* n'apparaitront que 2 à 4 transactions. En un sens, cela solutionne le problème de l'évolutivité, car cette solution n'impose aucune limite au débit des transactions.

A.3.1 Protocole Lightning

Le principe du BitcoinLightning consiste en l'utilisation de canaux *lightning* ephémères pour faire des transactions off-chain, ceci se fait en étapes :

— **1- Ouverture du canal :** Afin de créer un canal, certaines conditions doivent être remplies :

Les deux utilisateurs concernés par ce canal doivent faire partie du réseau Bitcoin et avoir un moyen de communication secondaire qui leur permettra d'échanger leurs adresses BTC (en général e-mail, téléphone ...). Par la suite, l'utilisateur A génère sa paire de clés et envoie la clé publique à B, qui fait de même. L'utilisateur A crée alors un *multi-sig wallet* avec sa clé privée et la clé publique de B, qui fait la même chose de son côté.
Les deux utilisateurs vérifient ensuite qu'ils ont bien créé le même canal avec une seule et même adresse en s'échangeant l'adresse du *wallet* créé via leur second moyen de communication.

— **2- Utilisation du canal :**

Afin d'échanger des transactions *lightning*, les deux utilisateurs A et B doivent envoyer une certaine somme sur le *wallet* (*deposit*), ce qui apparait comme une transaction banale sur la *blockchain*.
Á chaque transaction *lightning*, chaque utilisateur crée un nonce appelé "secret" dont il envoie le *hash* à son correspondant. Avant chaque nouvelle transaction, les utilisateurs doivent envoyer le nonce précedent pour valider la précedente transaction (sauf pour la transaction initiale qui définit les sommes déposées).
Par la suite, les deux utilisateurs créent un HTLC (*Hash Time Locked Contract*), un *smart-contract* qui ne peut être déverrouillé qu'avec le secret correspondant au *hash* l'ayant verrouillé, et ce avant que le timeout n'expire. Ce contrat spécifie l'état des balances de chacun et est signé par chacun des utilisateurs.

— **3- Fermeture :**

Dans le meilleur des cas, les deux utilisateurs s'échangent leurs secrets, débloquent le contrat, diffusent la transaction au niveau *blockchain* et reçoivent leurs fonds. Si l'un des deux essaie de transgresser le protocole, les pénalités spécifiées dans le *smart-contract* seront appliquées.

— **4- Lightning Network :**

L'utilité du BitcoinLightning n'est ressentie qu'après avoir dépassé un certain nombre de transactions sur un canal, ce qui nécessite l'utilisation fréquente d'un canal avec une seule personne. Une façon plus utile d'utiliser ces canaux a été proposée, elle consiste à créer un réseau *overlay* à l'aide des canaux déjà crées.
Par exemple, deux utilisateurs A et C, qui sont connectés avec B via un canal *lightning* chacun peuvent utiliser B comme intemédiaire entre eux. Voici un exemple d'utilisation de ce réseau *lightning* :
Pour que A puisse envoyer des BTC à C, ce dernier doit envoyer le *hash* d'un secret à A grâce au protocole *p2p* mis en place par *lightning*, A sait qu'il doit passer par B pour contacter C via *lightning* grâce à ce même protocole.
A crée alors un HTLC qu'il envoie à B spécifiant que les fonds seront versés à B s'il est capable de déverrouiller le contrat avec le secret correspondant. B fait de même avec C qui est le seul détenteur du secret. Il lui renvoie la clé du contract mis en place entre eux. De cette façon, C reçoit son argent et B peut à son tour déverrouiller le contrat le liant à A pour se faire rembourser.

A.3.2 Modélisation de BitcoinLightning dans MAX

Dans notre implémentation, le modèle BitcoinLightning étend le modèle Bitcoin qui à son tour étend le modèle Blockchain, de la même façon que le protocole BitcoinLightning est un niveau parallèle (*overlay*) à Bitcoin.

Nous avons essayé de représenter BitcoinLightning de la manière la plus réaliste possible avec les hypothèses formulées précedemment en 5.2, mais comme avec toute modélisation, il est indispensable d'omettre certains points qu'on juge peu importants ou sur lesquels on ne voudrait pas se focaliser dans l'immédiat.

Rôles :
Les rôles créés pour les besoins de la simulation de BitcoinLightning sont entre autres choses les mêmes que dans le modèle Bitcoin. Avec un rôle supplémentaire propre à BitcoinLightning, ce rôle "*LightningUser*" donne aux agents un accès au niveau *Lightning* pour pouvoir ouvrir des canaux *lightning* et créer des transactions.

Messages :
Comme avec *network.p2p*, nous avons mis en place certains types de messages nécessaires au fonctionnement du protocole, mais en omettant certains qui, selon nous, n'influaient pas sur le bon fonctionnement du protocole. Ces messages sont :
— **OPEN CHANNEL :** nécessaire pour demander l'ouverture d'un canal entre deux utilisateurs.
— **ACCEPT CHANNEL :** en réponse à un demande d'ouverture de canal.
— **SHUTDOWN :** nécessaire pour fermer un canal ouvert entre deux utilisateurs.

Hypothèses :
Pour les besoins de la simulation, nous avons eu recours à des hypothèses supplémentaires qui nous ont permis de simplifier le protocole, celles-ci sont :

- **Signatures :** Dans MAX, les signatures cryptographiques ne sont pas implémentées car on considère qu'il n'y a aucun problème d'authentification. De ce fait, la création d'un *multi-sig wallet* n'a pas besoin de signatures cryptographiques.
- **Nombre de propriétaires d'un *multi-sig wallet* :** En réalité, un *multi-sig wallet* peut avoir plus de deux propriétaires, mais dans le but de simplifier le protocole, le *wallet* implémenté dans MAX n'en accepte que deux.
- **Moyen de communication secondaire :** Aucun moyen de communication secondaire n'a été implementé. Le transfert de l'adresse d'un utilisateur à l'autre se fait via l'interface client créée à cet usage.
- **HTLC :** Les smarts contracts n'étant pas implémentés dans MAX, on suppose les transactions immédiates et les utilisateurs honnêtes.

A.3.3 Utilisation du modèle BitcoinLightning dans MAX

Comme expliqué au début de la section 4.6.1, l'utilisation de MAX est basée sur des scénarios mettant l'accent sur certains aspects du protocole, voici quelques exemples de scénarios que j'ai créé :

- **BitcoinLightning Network :**
  Ce scénario a été créé afin de montrer l'utilisation des canaux *lightning* et montrer leur intérêt. Il permet de montrer les interfaces des clients Lightning que j'ai créé pour ensuite les utiliser afin de créer des canaux et des transactions *lightning*. Au niveau *lightning*, aucune création de *block*, donc les modèles de création n'influencent que le modèle que Lightning étend, c'est à dire Bitcoin.

- **BitcoinLighting With Quitting :**
  Á l'image du scénario du même nom dans Bitcoin, ce scénario permet d'étudier la dynamicité du système et l'influence de la déconnexion/reconnexion d'agents sur le fonctionnement de l'application, et plus précisément sur le fonctionnement du réseau *lightning*. Ce scénario utilise un modèle de création de *blocks* en PoW avec possibilité de forks au niveau Bitcoin.

Voici un exemple de scénario "BitcoinLightning Network" :

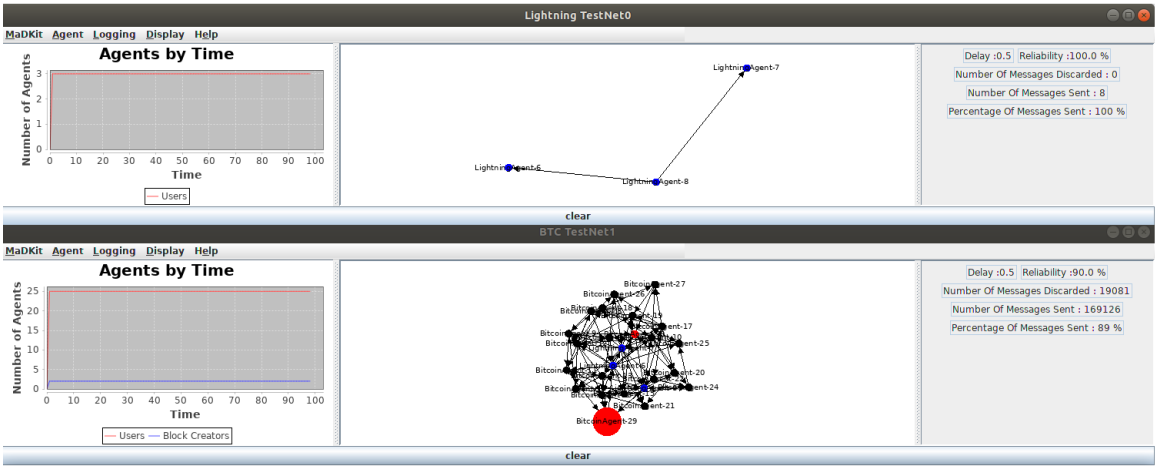

Figure A.4 – Environnement Lightning

Le but du protocole Lightning est de créer une couche parallèle à Bitcoin, un *overlay* où les agents peuvent échanger des transactions *off-blockchain*. La figure A.4 montre cet aspect où on a un environnement peuplé d'agents Bitcoin qui pourront échanger des transactions via la *blockchain*. Un autre environnement est créé dans Lightning où on ne verra que les agents qui ont la capacité de créer des canaux *lightning*. Sur la figure, les canaux déjà créés sont également affichés. Ces canaux serviront de supports aux transactions *lightning* ultérieures. La figure A.5 montre les interfaces client des agents BitcoinLightning qui auront accès aux canaux qu'ils ont déjà créés pour envoyer un *deposit*, échanger des transactions ou encore créer d'autres canaux *lightnings*.

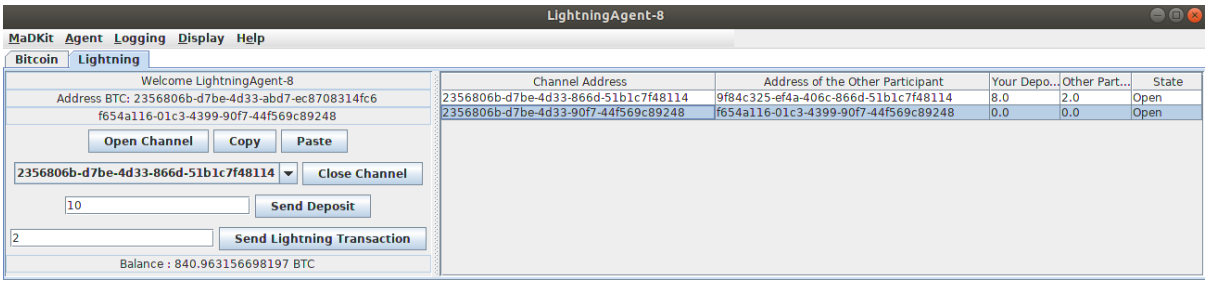

Figure A.5 – Interface Client Lightning

Cet exemple de scénario montre l'utilité du protocole Lightning en termes d'évolutivité et de vitesse de validation d'une transaction. Il permet également la démonstration des interfaces client implémentées et des environnements parallèles créés.

B Diagrammes UML des modèles de MAX

B.1 Network Model

Comme l'illustre la figure B.1, l'agent Network envoie et reçoit des messages via l'environnement qui gère la communication entre agents en fonction de paramètres tel que le délai et la fiabilité de transmission.

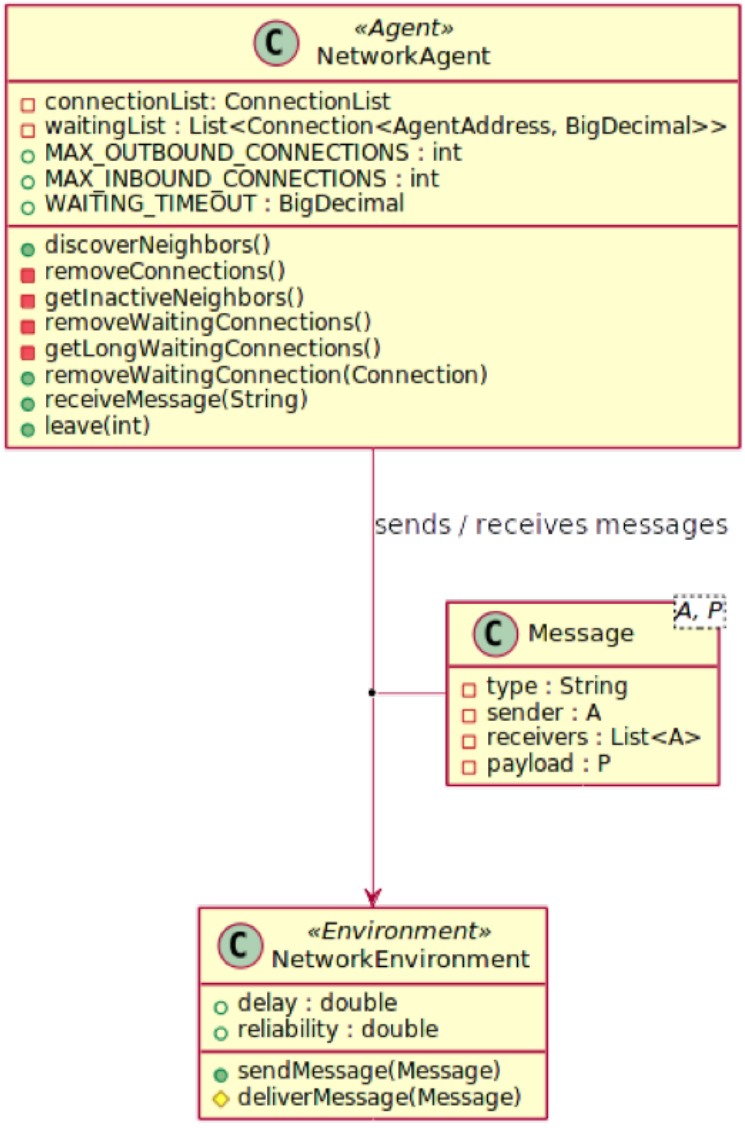

Figure B.1 – Diagramme UML du modèle Network

B.2 Blockchain Model

Comme le montre la figure B.2, l'agent Blockchain est un agent Network qui a accès aux structures *blockchain*. Cet agent a également la capacité de créer des *blocks* en se faisant aider par un nouveau type d'agents "BlockCreationOracle". Les modélisations des protocoles Blockchain tel que Bitcoin et Tendermint étendront le modèle Blockchain.

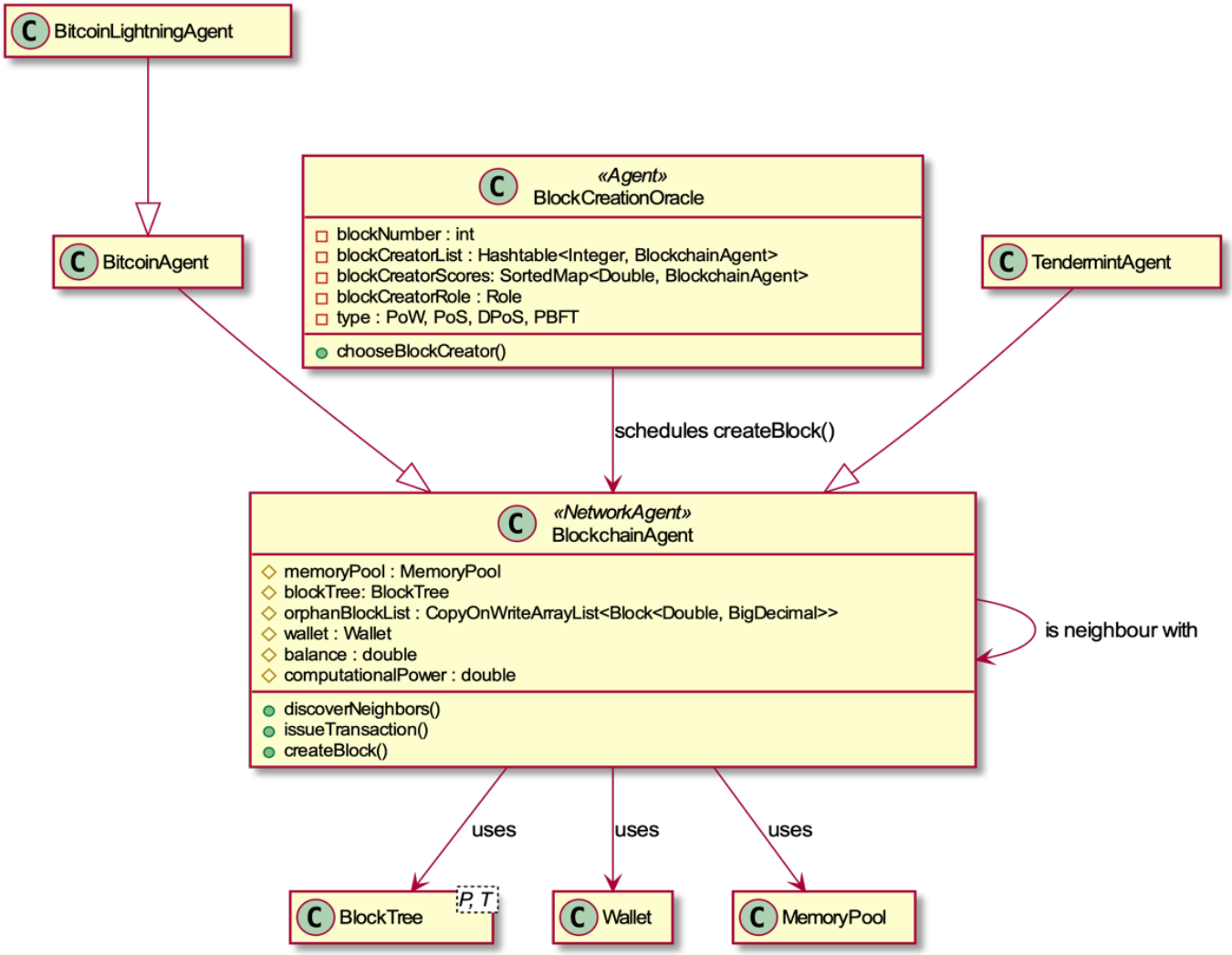

Figure B.2 – Diagramme UML du modèle Blockchain

B.3 Tendermint Model

Dans le modèle Tendermint (voir figure B.3), il existe deux types d'agents. Le premier, TendermintAgentUser étend BlockchainAgent, ce qui lui donne accès aux structures de données *blockchain*. ll peut également envoyer des messages propres au protocole Tendermint.

Le second type, TendermintAgentValidator, étend TendermintAgentUser. Ce type d'agents est éligible au comité, et peut donc participer à la création des *blocks*.

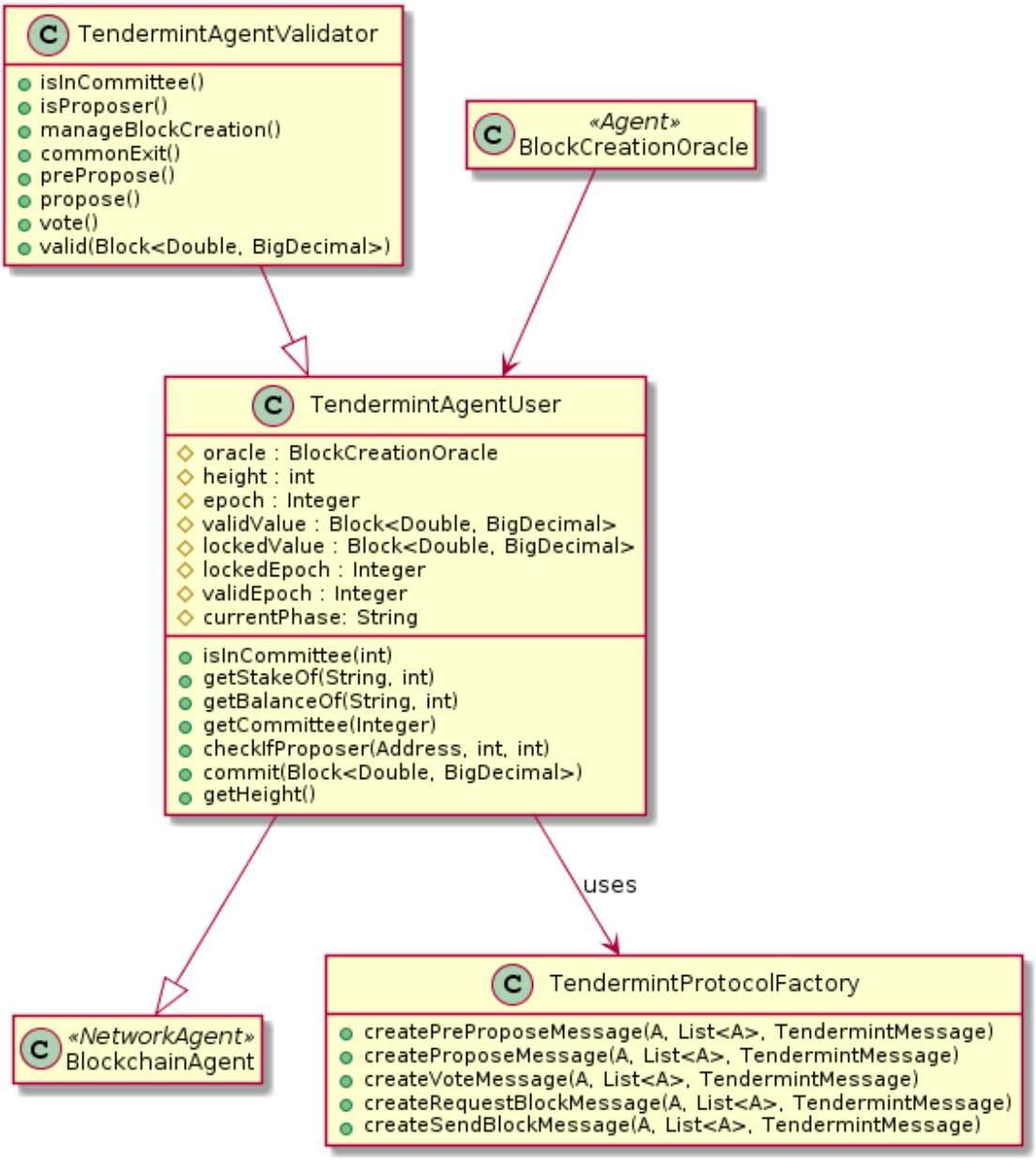

Figure B.3 – Diagramme UML du modèle Tendermint

C Exemple de scénario dans Tendermint

Dans l'exemple de scénario illustré par la figure C.1, on définit le nombre d'utilisateurs comme étant une variable globale qu'on va utiliser lors de la création des utilisateurs. Comme pour tout scénario, on commence par créer le *scheduler*, ainsi que l'environnement qui prendra comme paramètre des valeurs de délai et de fiabilité de transmission. Dans MAX, la création de *blocks* se fait grâce à un oracle, qui est également un agent que l'on doit initialiser avant de passer à la création des utilisateurs.

Une dernière étape facultative consiste à créer des graphes pour une meilleure compréhension de la simulation. Ces graphes ne sont autre que des agents qu'il est nécessaire de lancer. Cette étape apparait ici sous la forme d'un appel à la méthode *launchDiagrams()*.

```
private int numberOfUsers = 10;

@Override
protected void activate() {

        // 1 : launch the scheduler
        TimedScheduler scheduler = new TimedScheduler(TendermintCommunity.getInstance());
        launchAgent(scheduler, true);

        // 2 : launch the environment with a delay of 1 tick and a 100% reliability
        TendermintEnvironment environment = new TendermintEnvironment(BigDecimal.valueOf(1), 1.0);
        environment.setScheduler(scheduler);
        launchAgent(environment, false);

        // get the group
        String group = environment.getGroup();

        // 3 : launch the oracle
        BlockCreationOracle oracle = BlockCreationOracle.getInstance(PoXType.BFT,
                        group,
                        BlockCreationOracle.ROLE_ORACLE);
        oracle.setScheduler(scheduler);
        launchAgent(oracle, false);

        // 4 : launch some simulated agents
        for (int i = 0; i < numberOfUsers; i++) {
                TendermintAgentValidator agent = new TendermintAgentValidator(group);
                agent.setScheduler(scheduler);
                agent.setOracle(oracle);
                launchAgent(agent, false);
        }

        // 5 : lauch the diagrams
        launchDiagrams(scheduler);
}
```

Figure C.1 – Exemple de scénario Tendermint