Rapport du projet

Algorithme distribué

Nguyen Kim Thuat - Nguyen Dang Chuan
Ensimag - ISI3A

Grenoble, 14/01/2013

Table de matières

I. Introduction

Notre application consiste à créer un simulateur simpliste du système distribué. Le simulateur permet aux utilisateurs de simuler la communication entre les machines dans un réseau simple.

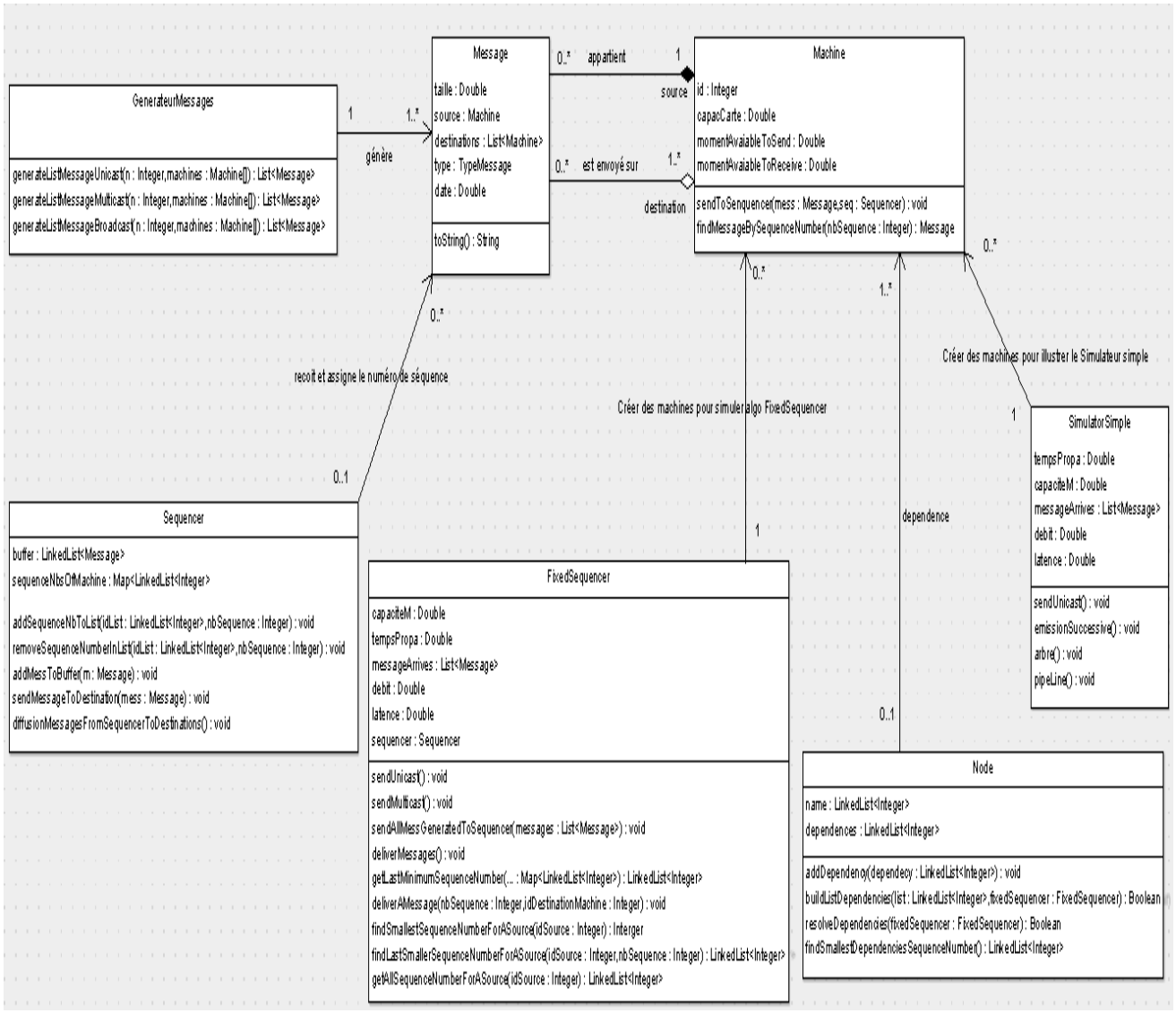
Dans ce document, la notion "*temps de transmission*" fait référence au temps nécessaire qu'un message devrait mettre pour arriver à la destination. De plus, une autre notion "*temps de propagation*" qui est le temps pour que un message puisse propager dans le réseau. Cette valeur est fixe au moment du démarrage du simulateur.

L'utilisateur a la possibilité de créer un nombre de machines quelconques au démarrage. Les machines se communiquent en utilisant des messages de type varié. Le débit et la latence sont les deux indices principaux pour observer l'état du système.

II. Architecture

1. Les composants détaillés

Voici le diagramme de classe qui représente l'architecture générale de notre application.



2. Description des composants

Cette partie liste les différents composants du projet avec une brève description de leurs fonctionnalités.

a. Message

Le simulateur peut traiter les messages de différents types (unicast, multicast ou broadcast). En général, chaque message est représenté par son type, une taille de message, une date (le moment message est envoyé), sa machine source et sa destination (unicast) ou ses destinations (multicast ou broadcast). En outre, un message a aussi un indice qui décrit le moment que le message est délivré par la destination. Ce paramètre est initialisé avec la valeur du date que le message est envoyé. Il sera modifié au moment de

réception aux destinations. De plus, un message dispose également un numéro de séquence qui sert à tracer l'ordre du moment qu'il est généré par une machine source.

b. Machine

Le simulateur se compose d'un certain nombre de machines au démarrage où chaque machine est représentée par une instance de la classe "Machine". Chaque machine possède une carte de réseau *full-duplex* avec un capacité modifiable. C'est à dire, elle peut envoyer et recevoir des messages en même temps. Mais chaque machine ne peut pas envoyer ou recevoir deux messages à la fois. Si elle a un message à émettre quand elle est en train d'envoyer un autre, ce message doit attendre jusqu'au moment que le message précédent termine son envoie. L'attribut "*momentAvaiableToSend*" de la classe "Machine" est donc pour l'objectif de modéliser cette caractéristique (c'est pareil pour recevoir des messages). D'ailleurs, la machine ne traite qu'une message à la fois à la réception. Tous les autres messages arrivés seront mis dans un buffer comme une fille d'attente.

c. Générateur des messages

On a créé une classe "GenerateurMessages" pour faciliter l'expérimentation du simulateur. Il est utilisé particulièrement pour la deuxième version du simulateur plus complexe. Cette classe fournit des méthodes pour générer de manière aléatoire des messages de différents types, sources, destinations. Le moment d'envoie de ces messages sont aussi générés aléatoirement.

d. Simulateur simpliste

Tous les composant ci-dessus a pour but de modéliser un simulateur simpliste qui est illustré par la classe "SimulatorSimple" notamment via l'implémentation de 3 méthodes de dissémination de messages (N émissions successives, arbre et pipe-line) et vérifier les calculs de latence/débit.

e. Simulateur plus complexe

Les autres classes comme "Sequencer", "FixedSequencer" et "Node" est essentiellement utilisées pour implémenter la deuxième version du simulateur qui propose le protocole d'ordre total en basant sur l'algorithme "Fixed Sequencer" et "Communication History" qu'on va détailler dans les parties suivantes du rapport.

III. Simulateur simpliste

1. Objectif

Cette première version de simulateur permet simplement de modéliser les trois algorithmes donnés en cours de disséminations de message et vérifier les calculs de latence/débit.

2. Calcul de latence et débit

Dans ce document, le débit et latence sont définis de façon simple qui montre bien l'objectif proposé avant.

Débit

Le débit du réseau est une mesure de la quantité de données numériques (nombre de messages dans notre cas) transmises par unité de temps. En informatique et dans les communications, le débit est utilisé pour représenter la capacité d'un accès internet (rapide ou lente).

Latence

Dans les réseaux informatiques, la latence désigne le délai entre le moment où une information est envoyée et celui où elle est reçue. En d'autres termes, la latence est l'intervalle de temps entre la fin d'une transmission d'un message et le moment ce dernier est reçu par la destination.

Dans un premier temps, le débit et la latence de notre simulateur simpliste sont modélisés de manière simple. Plus concrètement, une machine source qui a un message à envoyer doit prendre un certain temps pour effectuer son émission à une machine destination. Ce temps là se compose d'un intervalle pour la propagation représentant la latence et d'un autre temps pour la transmission du message désignant le débit.

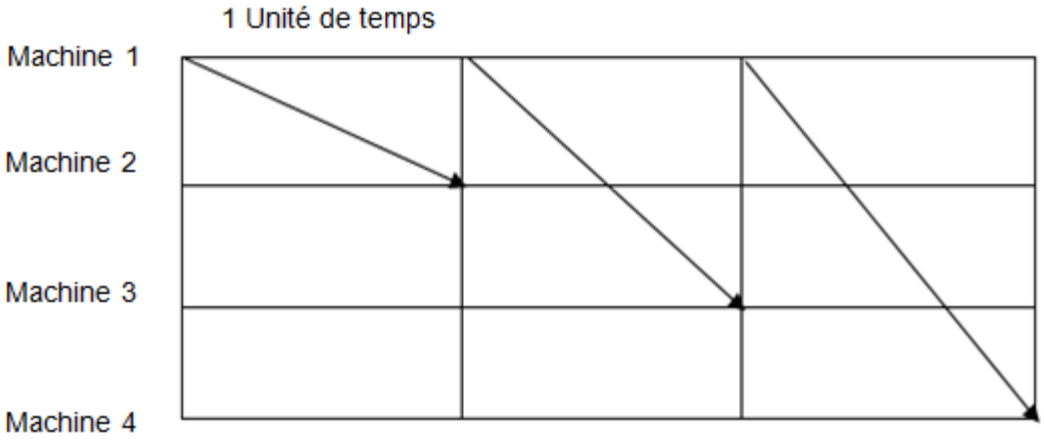
3. Algorithmes données en cours

Cette section tente de décrire notre simulation de trois algorithmes considérés en cours dans le cas de Broadcast d'un message dans le réseau.
Les formules pour calculer la latence et le débit sont communes pour tous ces trois algorithmes:

débit = nombre de messages envoyés / unité de temps
latence = nombres d'unités de temps / message envoyé

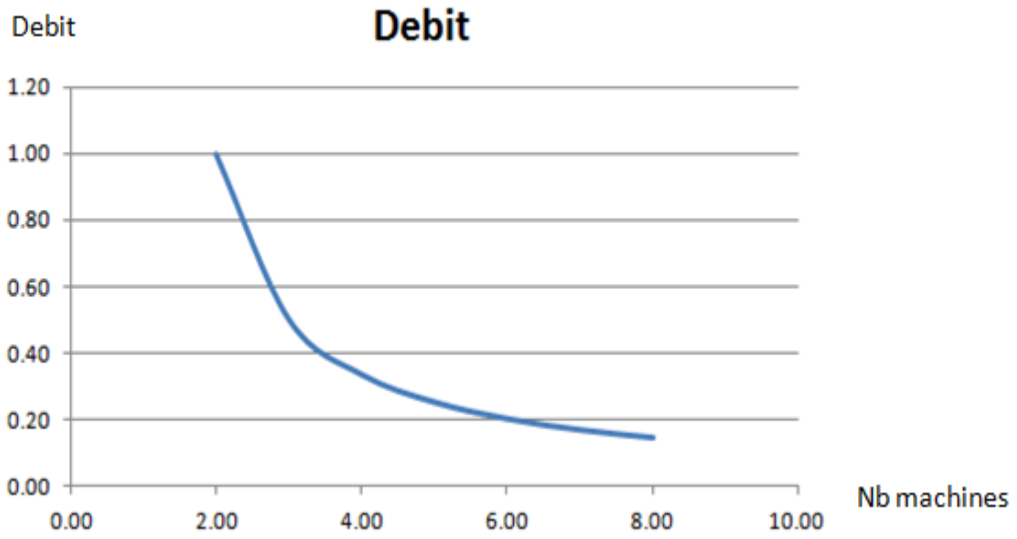
a. N émissions successives

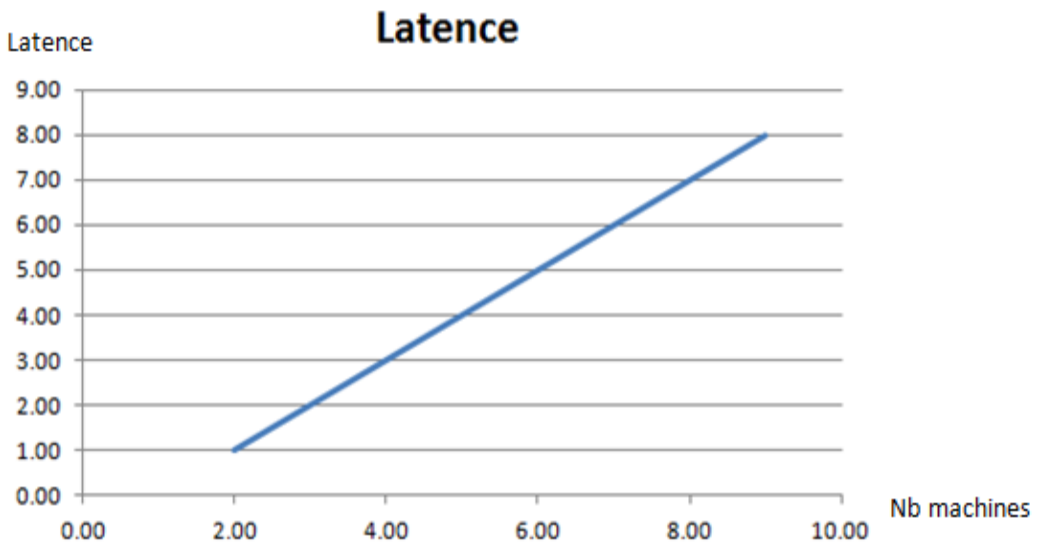
Dans cette mode d'émission, la machine source va envoyer successivement son message à toutes les autres machines du réseau comme dans la figure suivante.



Emission successive

On a modélisé cet algorithme dans le simulateur simpliste pour obtenir les graphes de dépendance du débit et la latence en fonction nombre de machines dans le réseau.

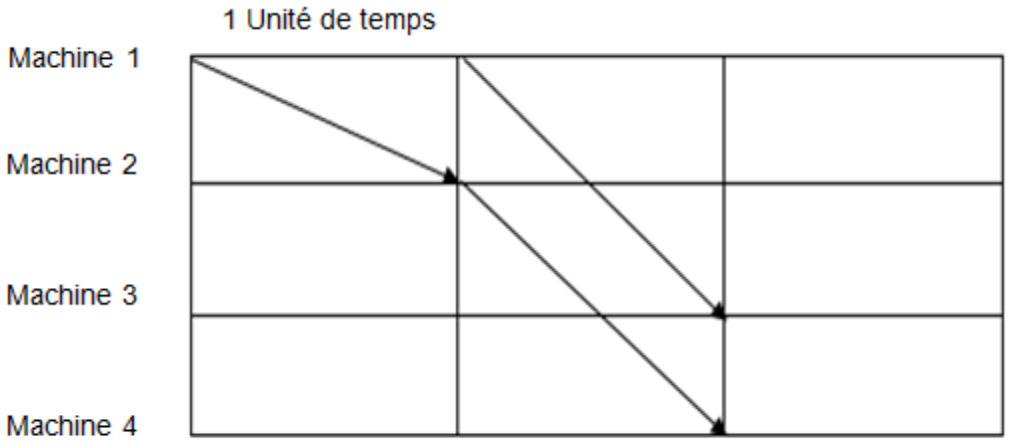




A partir de ces deux graphes, on peut constater bien que le débit du réseau diminue quand nombre de machines augmente, tandis que la latence a la tendance d'accroître linéairement.

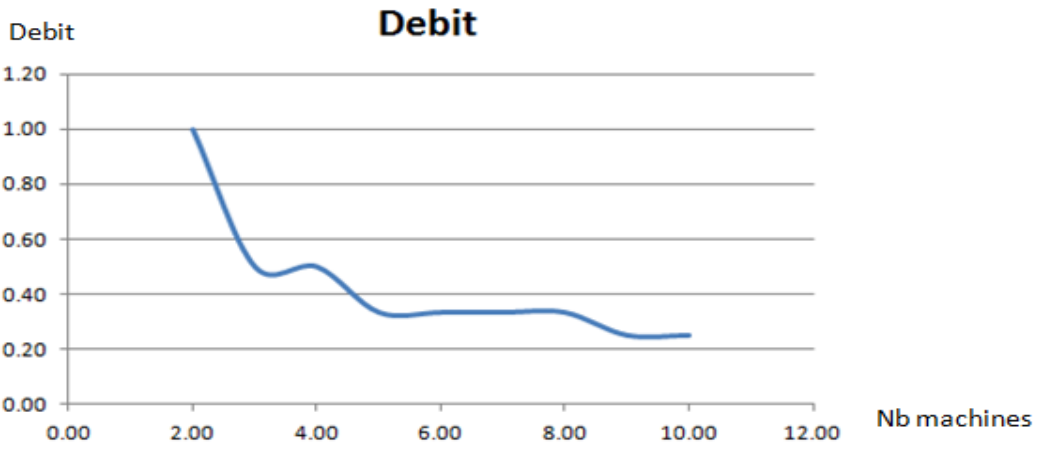
b. Arbre

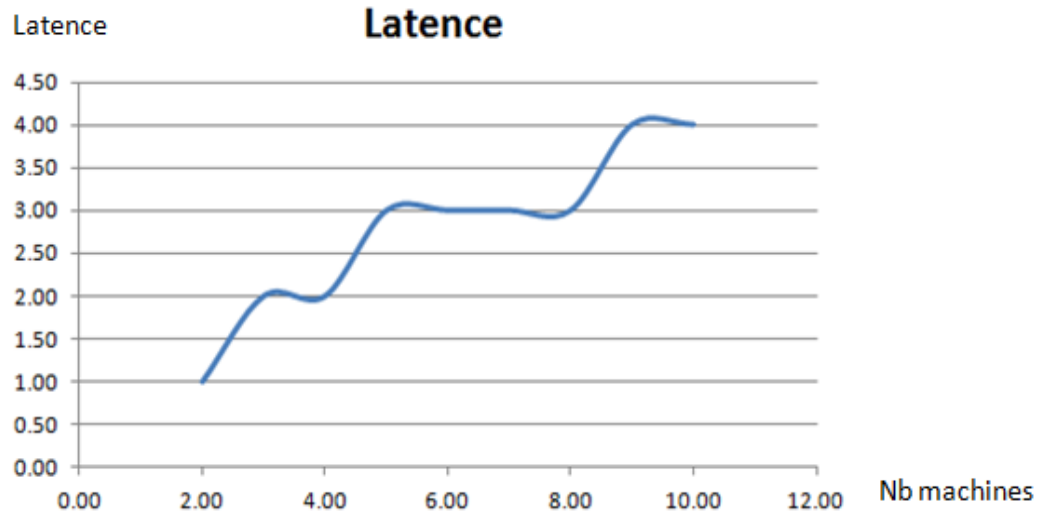
Cet algorithme est une amélioration du mode d'émission précédent par rapport au débit et à la latence. L'idée générale de cet algorithme est que, comme son nom a indiqué, la machine source va envoyer son message dans premier coup, à une machine quelconque. De temps en temps, les machines qui ont déjà connu ce message vont l'envoyer aux autres.
La figure ci-dessous se présente bien cette méthode:



Arbre

On a fait les mêmes tests comme dans l'algorithme précédent pour construire les deux graphes du débit et de la latence suivants.

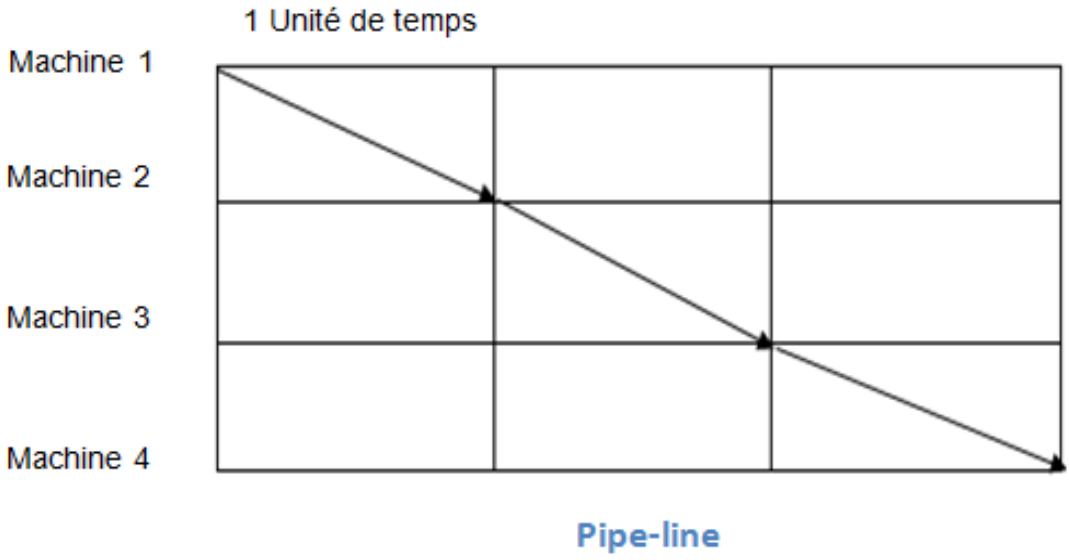




Ces graphes nous montre encore une fois les caractéristiques du réseau comme dans le mode d'émission successive mais avec une petite amélioration (le débit diminue moins vite et la latence accroît mois rapidement quand on augmente le nombre de machines dans le réseau).

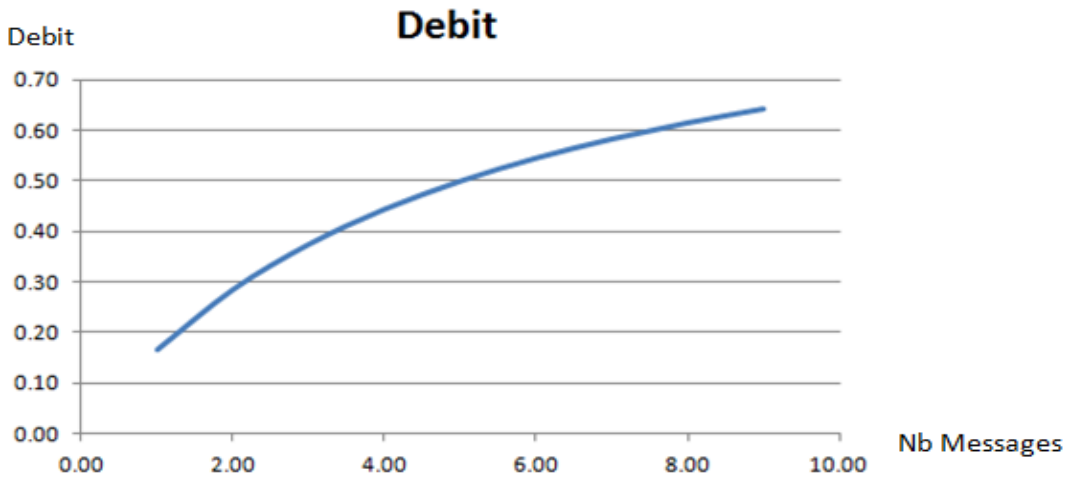
c. Pipe-line

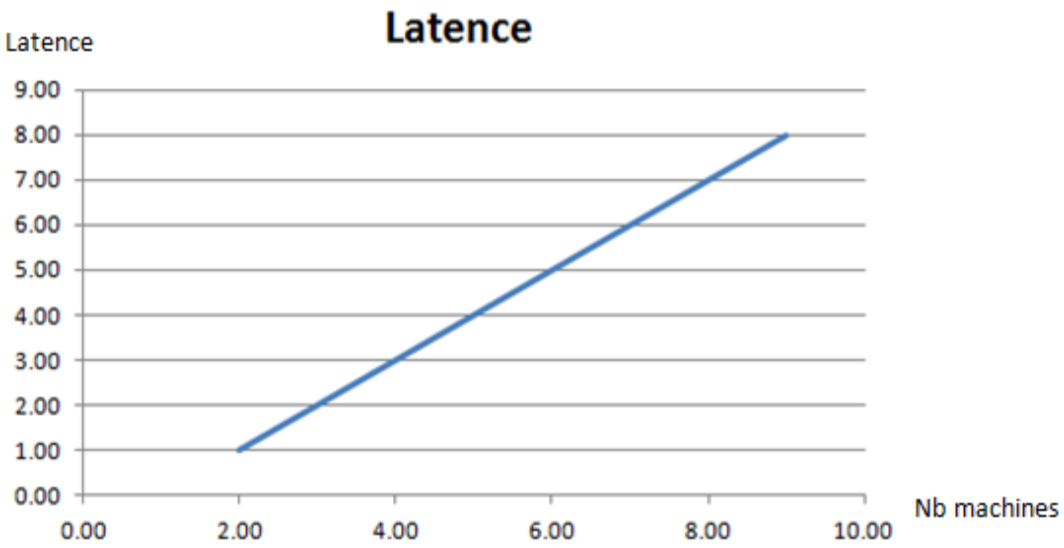
Cette dernière solution apporte une meilleure amélioration du débit du réseau où la machine source va envoyer son message une seul fois (la première fois) à une machine destination. C'est cette dernière qui va jouer ensuite le rôle de la source pour envoyer le message à la destination suivante. Ce processus sera effectué en continu jusqu'à la dernière destination du réseau comme se présente la figure suivante.



Pipe-line

Les expérimentes du cette méthode d'émission de message a vérifie bien que c'est l'algorithme qui porte le meilleur débit par rapport aux autres. Le graphe de dépendance du débit en fonction de nombre de messages (on garde le nombre de machines constant dans ce cas) ci-dessous montre que le débit accroît de manière asymptotique à 1 quand le nombre de messages augmente. Plus les messages sont envoyés, plus claire on reconnaît les avantages de cette méthode.





IV. Simulateur avec l'ordre total

Dans cette section, on essaie d'écrire un protocole qui assure l'ordre total des messages. Inspiré de ce que nous avons étudié à partir du rapport de recherche *"Total Order Broadcast and Multicast Algorithms "*, l'algorithme *Fixed Sequencer* est l'idée principale de notre solution en phase d'implantation.

1. Quelques définitions

Dans le cadre du rapport, on considère l'ordre total est assuré en l'absence des fautes. Concrètement, on s'intéresse au critère *"qui joue le rôle fondamentale"* pour générer les informations nécessaires afin de définir l'ordre total des messages.

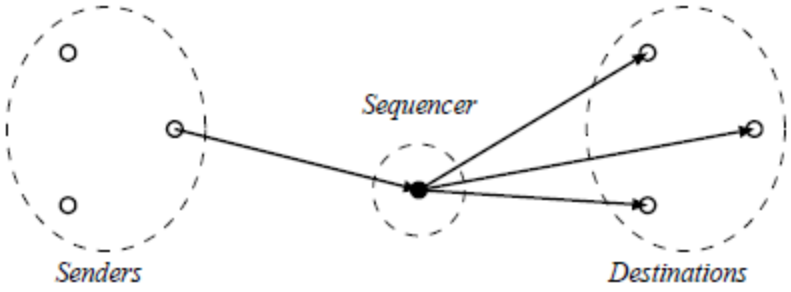
Nous identifions aussi les trois composants principaux dans cet algorithme: le *sender*, la *destination* et le *sequencer*.

a. Sender

Lorsqu'une machine envoie un message de type variable (UNICAST, MULTICAST, BROADCAST) au sequencer, elle est considérée comme un sender.

b. Sequencer

Dans l'algorithme *Fixed Sequencer*, un processus est considéré comme le sequencer et est responsable pour l'ordonnancement des messages dans le réseau. Le sequencer est bien sur unique, et ce n'est pas simplement le transfert des messages.



Le *sequencer* joue un rôle primordiale dans notre implémentions. Il a pour les deux objectifs essentiels. Premièrement, il reçoit les messages qui viennent des *senders*, les trie selon la date d'envoie de chaque message et les stocke dans un buffer avant de les diffuser. Deuxièmement, tous les messages sont assignés un numéro de séquence qui représente leur priorité plus tard. Ce sont effectivement ces numéros qui vont décider l'ordre de délivrerons des messages à l'étape de réception aux destinations.

c. Destination

Une *destination* est un machine qui reçoit un message venant du *sequencer*. Chaque machine possède un buffer qui contient les messages pas encore traités.

2. Composants détaillés

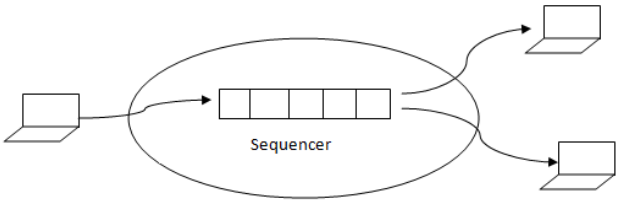
a. Sender/Destination

Cf. section II/2/b.Machine
Chaque sender et destination est considéré comme des machines.

b. Message

Cf. section II/2/a.Message

c. Sequencer



LinkedList<Message> buffer
Map<LinkedList<Integer>, LinkedList<Integer>> sequenceNbsOfMachine;

Le *sequencer* contient simplement un buffer qui stocke les messages envoyés par tous les sources machines. L'ordre des messages dans le buffer est ordonné selon la date que le message est envoyé. Il stocke également une structure de donnée simple pour enregistrer les numéros de séquence selon une source précise. Par exemple, si on donne ici un élément de ce map: [0,1] = [4,7, 8]. Ça veut dire que le machine source avec un identifier 0 a envoyé a la machine destination avec identifier 1, 3 messages avec les numéros de séquence 4,7, 8 relativement. Cette structure de donnée sert a créer la liste de dépendances pour un message selon une source concrète.

Fonctionnalités détaillées

- *addMessToBuffer(List<Message> listMess):* Ajouter des messages au buffer et les trient selon le date envoyé.

- *assignSequenceNumber(LinkedList<Message> buffer)*: Assigner pour chaque message un numéro de séquence selon leur date envoyé.

- *diffusionMessagesFromSequencerToDestinations()*: Envoyer tous les messages dans le buffer aux destionations en regardant le type de chaque message.

- *addSequenceNbToList(LinkedList<Integer> idList, Integer nbSequence)*: Ajouter un numero de sequence du message dans la structure de donnee s*equenceNbsOfMachine*, par exemple: [0,1] = [4,7,8] ->*addSequenceNbToList([0,1], 5)* => nouvelle valeur: [0,1] = [4,5,7,8]

- *removeSequenceNumberInList(LinkedList<Integer> idList, Integer nbSequence)*: Enlever un numéro de séquences dans le map s*equenceNbsOfMachine*, par exemple: [0,1] = [4,5,7,8]-> remove..([0,1], 7) => nouvelle valeur: [0,1]=[4,5,8].

Notes: Les deux derrières méthodes a pour l'objectif de créer la liste des dépendances pour un message (c'est a dire, la liste des messages qu'il doit attendre avant de pouvoir délivrer) qui est utilisé dans la classe *Node*

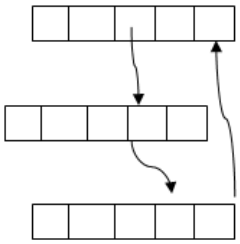
d. Node

Lorsque tous les messages dans le buffer du sequencer sont envoyés, chaque message sera transféré vers une machine destination. Dans le premier temps, tous ces messages sont mis dans le buffer de la destination. Un message peut normalement être délivré s'il satisfait les deux conditions suivante. Premièrement, il doit être à l'en-tête du buffer. Deuxièmement, il n'y a aucun message qui est généré de la même source avec lui et qui a un numéro de séquence inférieure de ce qu'il possède.

L'intérêt est de créer une liste de dépendances pour chaque message selon son numéro de séquence. C'est à dire qu'on va construire une liste chaînée qui décrit l'ordre des messages qui devraient délivrer avant que le message racine peut délivrer.

Chaque node a un nom qui est sous la forme [idSource, idDestination, nbSequence]. Le "idSource" est l'identifier de la machine source. Le "idDestination" est l'identifier de la machine destination. Et le "nbSequence" est le numéro de séquence du node.

De plus, il contient une liste chaînée qui enregistre la liste de dépendances. Par ailleurs, un variable *deadlock* va être mis en *true* s'il détecte un inter-blocage dans la liste des dépendances.



Au cas où l'inter-blocage existerait, le message qui a le numéro de séquence le plus petit sera délivré.

Fonctionnalités détaillées

Un exemple du node dans cette application:

Node (Name:[3, 1, 6] || Dependencies: [[3,1,6],[3, 0, 4]])

On peut observer facilement qu'on cherche à traiter le message avec le numéro de séquence 6, qui est généré de la machine 3 vers la destination 1. Cependant, si on voit dans son liste de dépendance, ce message ne peut qu'être délivré si et seulement si le message avec le numéro de séquence 4, qui est généré aussi de la machine source 3 vers la machine destination 0. Grâce a ce liste, on sait bien quel message doit délivrer avant que le message cible soit délivré.

- *buildListDependencies(....,...)*: Une fonction récursive qui construit la liste chaînée des dépendances d'un message.
- *resolveDependencies(...)*: Délivrer les messages en utilisant la liste chaînée des dépendances construites.

e. FixedSequencer

Cette classe est le composant le plus important dans notre application. Elle gère tous les comportements du simulateur. Tous les indices initiales du simulateur sont initialise par cette classe par exemple: le temps limite pour la simulation, le nombre des machines, l'état initiale de chaque machine, le séquence...

Fonctionnalités détaillées

On cite ici quelques fonctions importantes dans cette classe. En suite, la routine du simulateur sera explique de manière simple

- *initiateEtatMachines(..,..)*: Initialiser l'état initiale des machines dans le réseau.
- *sendUnicast(..,..,..,..,..)*: Envoyer un message de type unicast d'une source vers une destination précise avec une taille définie.
- *sendMulticast(..,..,..,..,..)*: Envoyer un message de type muticast, c'est a dire, une source vers de multiple machin*es*
- *sendBroadcast(..,..,..,..)*: Envoyer un message de type broadcast, c'est a dire, une sources vers tous les autres destinations dans le réseau.

- *getLastMinimumSequenceNumber(..,..)*: Chercher le message qui a le numéro séquence le plus petit dans les buffers des destinations.
- *deliverMessages()*: Délivrer tous les messages arrivant aux destinations selon leur numéro de séquence.
- *EmissionSuccessive(..,..)*: Simulation l'algorithme N émission successives et calcul de latence et débit.
- *pipeLine(..,..)*: Simulation l'algorithme pipeline et calcul de latence et débit
- *Arbre (..,..)*: Simulation l'algorithme arbre et calcul de latence et débit
- *valideOrderSequenceForSource(Integer idSource)*: Valider l'ordre total pour une source précise.
- *valideOrderTotal()*: Valider l'ordre total pour le réseau

Routine du simulateur

a Générer des messages (à la main ou en utilisant le générateur des messages)

b Envoyer les messages au sequencer

c Le sequencer trie les messages selon leur date envoyé et attribue chaque message un numéro de séquence. En suite, tous les messages seront transférés aux destinations.

d Le processus de délivrer les messages aux destinations

while (existe un message qui n'est pas encore délivre et temps <limitTime) {

- Chercher le message qui a le numéro de séquence le plus petit
- Construire la liste de dépendances pour ce message
- Délivrer le message en utilisant la liste construite

}

V. Expérimentation

Dans la plupart des tests, l'ordre total est assuré par cet algorithme. Cependant, le résultat est banal car l'application ne traite pas le faute des envoies des messages. De plus, le sequencer est toujours tolérant aux pannes.

Dans cette partie, on va présenter quelques expérimentes pour montrer que le simulateur assure bien l'ordre total avec des données (messages générés aléatoirement). Pour chaque test, on refait également des calculs de la latence et du débit du réseau se basant sur les définitions et les formulaires abordés dans les sections ci-dessus.

Pour la validation du l'ordre total, on implémente une méthode qui s'appelle "*valideOrderTotal()*" (cf. FixedSequencer classe) qui vérifie l'ordre délivré de chaque message dans une source précise et dans tous les sources en globale. On présente ici une table qui montre quelques tests qu'on a expérimenté avec le calcule de latence et débit.

Test case	Nb machine	Nb Broadcast	Nb Multicast	Nb unicast	Latence	Débit	Ordre total
1	5	10	5	15	2.40	7.08	✓
2	8	8	7	10	11.24	25.41	✓
3	10	9	2	7	3.0	12.59	✓
4	12	4	10	8	4.2	11.56	✓
5	14	15	7	20	5.7	18.97	✓
6	16	5	7	17	4.8	17.72	✓
7	18	12	8	11	6.87	18.68	✓
8	20	14	80	15	6.6	20.25	X
9	22	15	14	10	8.15	19.8	✓
10	24	20	12	60	4.10	19.87	X

L'ordre total est bien assuré par le simulateur. Le débit a la tendance de monter quand on augmente le nombre de machines. Par contre, ce n'est pas le cas pour la latence qui dépend plus tôt à la qualité du réseau (le temps qu'un message a mis pour arriver aux destinations).