UNIVERSITÉ DE SHERBROOKE

DÉPARTEMENT D’INFORMATIQUE

IFT 630 - Processus concurrents et parallélisme

TP02

Travail présenté à

M. Patrice Roy

Par

*Raphaël Cayouette (15 064 430)*

*Jean-Philippe Croteau (15 072 957)*

*Jonathan Martineau (15 059 124)*

*Félix Vigneault (15 079 576)*

Le 7 août 2017

Table des matières

[Contribution des membres de l’équipe 3](#_Toc489868221)

[Description des choix technologiques 3](#_Toc489868222)

[Description des choix de conception 3](#_Toc489868223)

[Protocole de communication 3](#_Toc489868224)

[Acteurs : Patron de méthode 4](#_Toc489868225)

[Stratégies de synchronisation des modules 4](#_Toc489868226)

[Synchronisation de la lecture et de l’envoi des messages par la carte 5](#_Toc489868227)

[Fermeture des processus 6](#_Toc489868228)

[Traitement séquentiel par les chats et les rats 6](#_Toc489868229)

[Problèmes rencontrés et solutions apportées 7](#_Toc489868230)

[Ordonnancement des messages 7](#_Toc489868231)

[Envois à tous 8](#_Toc489868232)

[Recherche de chemin 8](#_Toc489868233)

[Bogues restants 8](#_Toc489868234)

[Explication des fichiers de la remise 9](#_Toc489868235)

[Fichiers binaires compilés 11](#_Toc489868236)

Contribution des membres de l’équipe

Tous les membres de l’équipe ont travaillé de façon équitable durant les quelques séances de travail. Félix a majoritairement concentré ses efforts sur ce qui touche à la carte et la validation des mouvements. Jonathan a travaillé sur la communication MPI ainsi que la génération du log de partie. Enfin Raphaël et Jean-Philippe ont travaillé sur les différents acteurs, soit les chats et les rats, ainsi que sur le calcul du meilleur chemin et la communication avec la carte.

Description des choix technologiques

Le choix technologique principal pour notre travail a été le choix de l’implémentation MPI (*Message Passing Interface*). Pour notre code en C#, nous avons choisi MPI.NET. Cette librairie nous fournit la communication MPI et les méthodes d’envoi (*Send*) et de réception (*Receive*), que nous utilisons dans la carte et les acteurs.

Description des choix de conception

Pour la sauvegarde de la carte dans le registre, nous avons décidé de faire une sauvegarde à tous les X (5) mouvements plutôt qu’à toutes les secondes. Cette façon de faire permet de garder l'interaction entre la carte et le registre séquentiel, ce qui évite l’utilisation massive de verrous dans la carte. Avec des verrous, la carte aurait été encore plus lente pour traiter les demandes des acteurs.

Quant au choix d’algorithme de recherche de chemin, nous avons décidé d’utiliser Dijkstra pour faire une seule recherche de chemin et trouver l’objectif le plus proche. À chaque itération de l’algorithme, nous explorons une case plus loin que l’itération précédente et si une de ces cases est un objectif, nous retournons le chemin entre la case courante et la case objectif (fromages et sorties pour les rats et les rats pour les chats).

Protocole de communication

Pour la communication entre les modules (carte et acteurs), la librairie MPI.NET permettait de passer directement des objets complexes pourvu qu’ils soient sérialisables, ce qui nécessite uniquement l’ajout de l’attribut *Serializable* aux classes et aux classes des membres qu’elles contiennent.

Ce choix fait en sorte que notre implémentation est liée à la plateforme utilisée (.Net sur Windows) et même à la librairie choisie (MPI.NET), mais dans notre cas c’était une limitation que nous étions prêts à accepter pour accélérer le développement et simplifier le protocole de communication. Nous pouvons donc nous concentrer sur la gestion du parallélisme plutôt que la communication entre les modules.

Pour identifier les processus qui nous envoient des messages, en évitant de prioriser certains processus plutôt que d’autres, nous avons décidé d’écouter de toutes les sources en même temps, ce qui fait en sorte qu’on ne peut pas savoir qui nous a envoyé un message (limitation de la librairie MPI.NET). Nous faisons donc confiance aux processus de nous fournir leur rang à l’intérieur de leurs requêtes et de ne pas mentir.

Pour les confirmations et les réponses, la carte fait un envoi à tous (*Broadcast*), ce qui permet aux acteurs de savoir toutes les mises à jour à faire puisqu’ils les reçoivent en même temps que le demandeur (celui qui avait demandé l’action à l’origine). Seuls les signaux de mort (*KillSignal*) sont envoyés directement aux acteurs qui doivent mourir.

Acteurs : Patron de méthode

Pour les acteurs (rats et chats), nous avons utilisé un patron de méthode (*Template method*) pour gérer la logique de base de la communication MPI et de la recherche de chemin. Par exemple, pour la recherche de chemin, les chats et les rats ont le même algorithme de recherche (Dijkstra) implémenté par la classe *Actor*, mais ils implémentent la méthode *IsGoal* qui détermine si une case quelconque est une case favorable pour l’acteur. La même chose est faite pour déterminer de quelle façon les acteurs se déplacent (*GetNeighbors*) ou comment ils réagissent à un miaulement (*ListenMeow*).

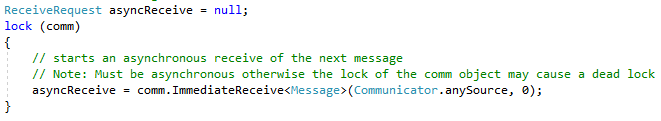
Stratégies de synchronisation des modules

Notre implémentation requiert de la synchronisation pour la réception des messages. Lorsque les acteurs envoient des messages à la carte, cette dernière les place dans une file d’attente bloquante (*BlockingCollection*) afin de les garder ordonnés pour les traiter plus tard. Une tâche (*Task*) est dédiée à la réception de messages pour éviter que le tampon MPI se remplisse, ce qui causerait la réception non ordonnancée des messages. La carte, dans sa boucle de vie, fait simplement lire de façon séquentielle la tête de la file d’attente. De cette façon, nous évitons les problèmes d’accès concurrents et pouvons facilement gérer la cohérence des données.

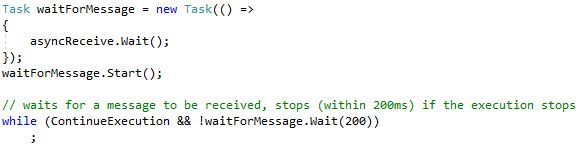
Synchronisation de la lecture et de l’envoi des messages par la carte

Le processus de la Carte était donc le seul processus qui travaillait avec plusieurs fils d’exécution : un pour la lecture des messages et un pour le traitement des demandes et l’envoi de réponse. Ces deux fils d’exécution avaient besoin d’accéder à l’objet commun de communication MPI. Nous étions donc forcés de mettre un verrou (*lock*) lors de l’accès à cet objet.

Pour la réception de message, cela rendait les choses plus compliquées puisque nous ne voulions pas mettre un verrou sur un appel bloquant *Receive*! Nous avons donc utilisé un appel asynchrone à l’aide de *ImmediateReceive* qui retourne un objet permettant de récupérer le message une fois arrivé. Cependant, nous avions un problème avec cet objet puisque nous devions attendre que la requête complète, mais il faut interrompre cette attente pour fermer le fil d’exécution si les autres processus sont tous terminés. Les deux seules méthodes fournies par l’objet sont *Poll* et *Wait*, ce qui ne nous laissait pas beaucoup de choix.



Notre priorité était la réception rapide des messages et la légèreté (sur le processeur) du fil d’exécution qui lit les messages, mais il fallait aussi être en mesure de fermer dans des temps convenables lorsque la partie serait terminée. La solution était donc un délai d’expiration (*timeout*) sur le *Wait*, ce que nous n’avions pas. Nous avons donc encapsulé l’attente à l’intérieur d’une tâche pour utiliser le *Wait(int timeout)* offert par l’interface de la tâche! Nous recevons alors les messages immédiatement, nous ne gaspillons pas de temps de CPU en bouclant inutilement et nous détectons la fin de la partie dans un délai maximal de 200 millisecondes! Tout ça malgré un API incomplet!



Fermeture des processus

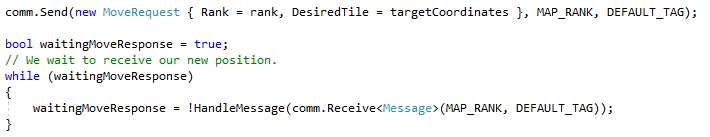
Pour gérer la fermeture des processus, la carte leur envoie un signal de mort (*KillSignal*), ce qui leur indique qu’ils doivent arrêter. Lorsqu’ils ont reçu le message et qu’ils sont prêts à se fermer, les processus répondent à la carte avec une confirmation de mort (*DeathConfirmation*). La carte considère alors que le processus est fermé. Lorsque tous les processus sont fermés, la carte doit aussi se fermer, mais le procédé est un peu plus complexe dû à ses deux fils d’exécutions.

Le procédé se déroule comme suit :

* [Fil d’exécution de lecture] La carte reçoit la dernière confirmation de mort, ne l’analyse pas et la met dans sa file d’attente bloquante pour la traiter plus tard.
* [Fil d’exécution de traitement] La carte traite la dernière confirmation de mort et se rend compte qu’elle doit se fermer. Elle change le fanion *ContinueExecution* à faux pour indiquer que l’exécution ne doit pas continuer.
* [Fil d’exécution de lecture] Après un délai maximum de 200 millisecondes, la carte arrête l’attente de message pour vérifier si l’exécution est terminée. Si c’est le cas, elle indique à la file d’attente bloquante que l’ajout de données est terminé et se ferme.
* [Fil d’exécution de traitement] Reçoit une exception indiquant qu’il n’y aura plus de message à traiter, arrête la boucle de traitement et se ferme.

Traitement séquentiel par les chats et les rats

Du côté des acteurs (chats et rats), nous avons tout traité de façon séquentielle. L’acteur reçoit les informations de la Carte et arrête d’écouter pour commencer sa recherche de chemin. Lorsqu’il a terminé, il envoie sa demande de chemin puis écoute à nouveau. Il peut alors entendre toutes les mises à jour qui ont eu lieu depuis sa dernière écoute et lorsqu’il reçoit la réponse à sa demande, il arrête à nouveau d’écouter pour recommencer son processus de calcul de sa prochaine demande. Cela permet de garder le processus séquentiel, ce qui est également plus simple à gérer. L’écoute des messages qui arrivent pendant le calcul de chemin est peu importante puisque nous ne pouvons pas de toute façon modifier notre carte locale au milieu du calcul. On profite donc du délai de traitement de notre requête à la carte pour faire les mises à jour nécessaires.



Problèmes rencontrés et solutions apportées

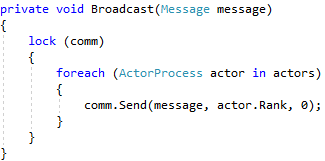
Nous avons eu plusieurs défis à relever pour que la communication par MPI puisse fonctionner correctement et de façon équitable.

Ordonnancement des messages

Initialement, nous avions décidé de nous fier à MPI pour l’ordonnancement des messages reçus afin que la Carte puisse lire un message, le traiter, puis lire le suivant et le traiter et ainsi de suite. Cependant, cela posait problème puisque MPI ne garantissait pas l’ordre des messages. Les deux premiers processus qui réussissaient à communiquer avec la Carte étaient tellement rapides qu’ils avaient le temps de répondre à la carte avant qu’elle n’ait terminé de traiter le message de l’autre, ce qui faisait en sorte que la carte ne recevait que les messages de ces deux processus. La raison de ce comportement nous est inconnue puisqu’il est généré par l’implémentation de MPI. Pour contourner le problème, nous avons utilisé une Tâche (*Task*) dans la Carte qui sert uniquement à recevoir les messages et à les enfiler sur une *BlockingCollection* en ordre de réception. De cette façon, nous sommes en mesure de recevoir assez rapidement les messages pour que tous les processus soient capables de jouer leurs actions et nous les traitons de façon séquentielle par la suite.

Envois à tous

Nous avons aussi eu un problème avec l’envoi de messages à partir de la carte aux différents processus. Initialement, avec l’intention d’être le plus équitable possible, nous utilisions la commande *Broadcast* de MPI.NET pour envoyer les messages à tous. Cela a posé problème quand les processus se fermaient. En effet, avec tous les processus ouverts, les envois à tous se faisaient sans problème, mais dès que des processus se fermaient, certains des processus encore en vie ne recevaient plus les messages envoyés à tous. Cela est probablement un bogue dans la librairie MPI.NET, donc nous avons contourné en utilisant la commande *Send* en direction de chacun des processus de façon séquentielle. Cela a permis de rétablir la stabilité de la communication MPI.



Recherche de chemin

Nous avons également eu un problème de performance et de stabilité en utilisant l’algorithme A-étoile que nous avions vu dans un autre cours. Certains cas limites ne fonctionnaient pas et il y avait beaucoup de calculs qui ralentissaient l’exécution. Nous avons réglé le problème en implémentant plutôt un algorithme de Dijkstra qui est beaucoup plus rapide dans notre contexte.

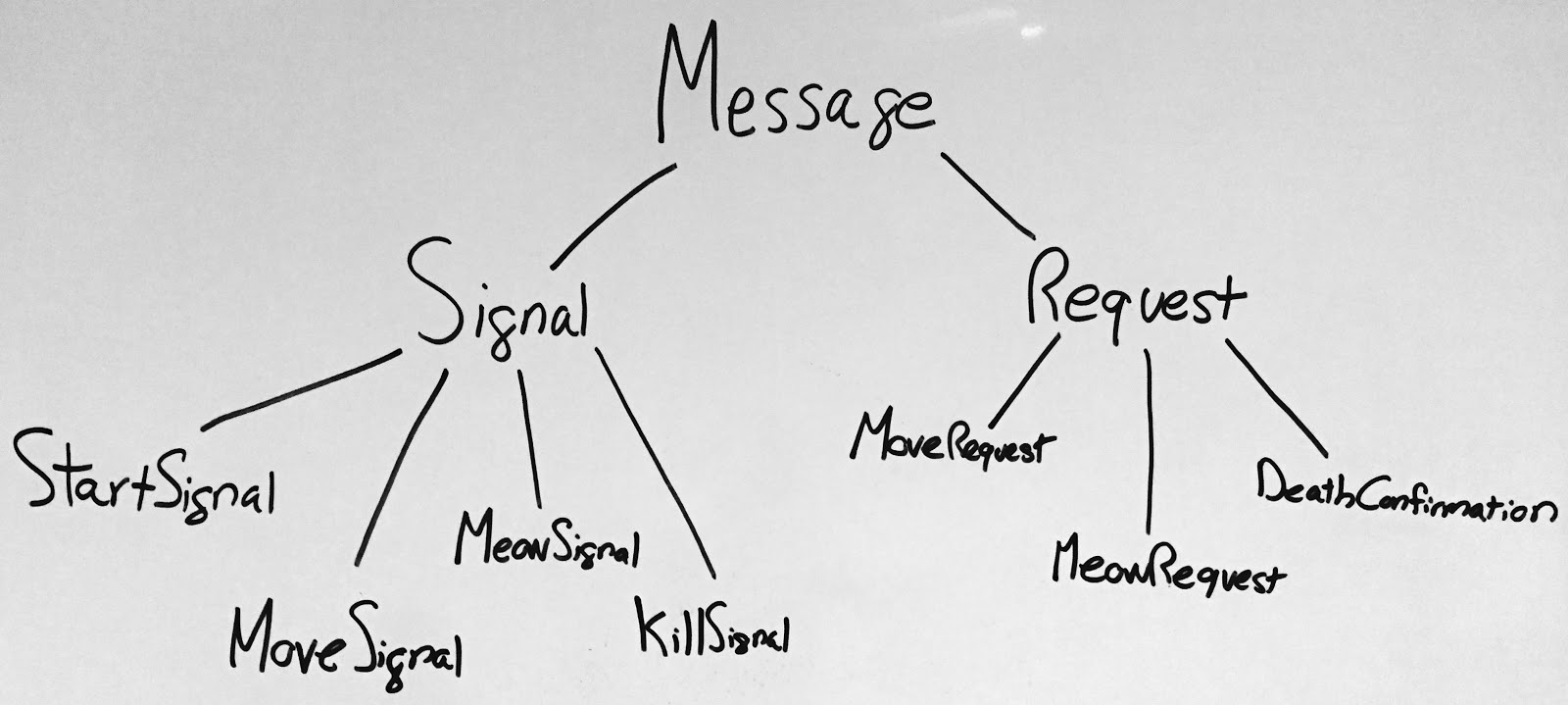
Bogues restants

Il ne reste plus de bogues connus dans notre implémentation.

Il y aurait toutefois de la refactorisation à faire dû à des mauvais choix initiaux sur les structures de données utilisées. En effet, nous avons un tableau 2D pour les tuiles de la carte qui permet l’accès avec des coordonnées y, x aux tuiles, mais ces mêmes coordonnées y, x sont enregistrées dans la tuile, ce qui rend plus difficile la gestion des déplacements et des listes de positions des acteurs. Les changements requis étaient simplement trop longs à faire pour le temps restant.

Explication des fichiers de la remise

Voici une explication sommaire des différents fichiers présents dans notre remise.

* Execution : dossier contenant les fichiers compilés finals de notre solution.
  + *Log.txt* contient les logs de la partie.
  + *Map.txt* contient le contenu de la map.
  + *Rongeurville.exe* contient le code généré en release.
  + *RunRongeurvilleMPI.bat* contient l’exécution de Rongeurville à l’aide de MPI.
  + Autres fichiers compilés nécessaires à l’exécution du programme.
* Rongeurville : dossier contenant notre solution.
  + *Program.cs* contient le code qui est initialement appelé par MPI et qui démarre les différents processus selon leur rang.
  + *MapManager.cs* est le processus qui gère la carte, les vérifications de mouvements et la communication avec les acteurs via MPI.
  + *Map.cs* contient la logique des accès à la carte, dont la création à partir d’un fichier, l’application des mouvements et la vérification des conséquences sur les mouvements, par exemple un chat qui attrape un rat sur la même case que lui.
  + *Coordinates.cs* représente les coordonnées d’un point et contient une position en X et en Y.
  + *TileContent* est une énumération qui représente les différents contenus possibles dans une case, soit : Vide, Mur, Fromage, Chat et Rat.
  + *Tile.cs* représente une tuile du la carte. Cette tuile contient une *Coordinates* et un *TileContent*.
  + *Loggers.cs* contient la logique de génération de statistiques sur la partie.
  + *Cat.cs* est la classe de l’acteur chat qui explicite certaines caractéristiques et logiques propres au chat.
  + *Rat.cs* est la classe de l’acteur rat qui explicite certaines caractéristiques et logiques propres au rat.
  + *Actor.cs* est la classe parent de *Cat* et *Rat* et contient les fondements des interactions avec les deux acteurs et le calcul du meilleur chemin vers une tuile quelconque sur la carte. Elle gère également la communication avec la carte via MPI.
  + *ProcessType.cs* contient une énumération, comme son nom l’indique, des différents types de processus possibles, soit *Map*, *Cat* et *Rat*.
  + *ActorsDivider.cs* est une classe qui gère le choix du type de processus en fonction du rang, du nombre de rats et du nombre de chats.
  + *PathTile.cs* est une classe utilisée par l’algorithme de calcul du meilleur chemin.
  + Le dossier *Communication* contient les classes qui représentent tous les types de messages que nous pouvons envoyer par MPI entre la carte et les acteurs. Nous avons utilisé l’héritage afin de réduire la duplication de code et de faciliter le classement des différents messages reçus. La partie *Signal* représente ce qui est envoyé de la carte vers les acteurs. La partie *Request* représente ce qui est envoyé des acteurs vers la carte. Voici le diagramme présentant la hiérarchie.
  + 
* MSMpiSetup.exe : fichier d’installation de MS-MPI qui ajoute la commande mpiexec à l’invite de commande Windows.

Fichiers binaires compilés

Pour exécuter un programme MPI sur Windows 10, il faut d’abord installer MS-MPI qui ajoute la commande mpiexec à l’invite de commande Windows.

Lien d’installation : <https://www.microsoft.com/en-us/download/details.aspx?id=55494>

ou directement à la racine de la remise sous le nom *MSMpiSetup.exe*

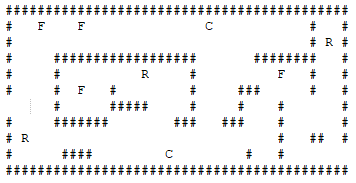
La commande suivante peut ensuite être réalisée dans le répertoire contenant Rongeurville.exe pour partir le programme :

mpiexec -n %NOMBRE\_DE\_PROCESSUS% Rongeurville.exe %CHEMIN\_DE\_LA\_CARTE% %NOMBRE\_DE\_RATS% %NOMBRE\_DE\_CHATS%

Description des paramètres :

* NOMBRE\_DE\_PROCESSUS : nombre de processus partis par MPI, doit être égal à NOMBRE\_DE\_RATS + NOMBRE\_DE\_CHATS + 1 pour avoir suffisamment de processus pour la carte et les acteurs;
* CHEMIN\_DE\_LA\_CARTE : chemin d’accès complet au fichier de la carte;
* NOMBRE\_DE\_RATS : nombre de rats présents dans la carte, doit représenter le nombre exact de rats dans la carte fournie;
* NOMBRE\_DE\_CHATS : nombre de chats présents dans la carte, doit représenter le nombre exact de chats dans la carte fournie.

Par exemple, pour partir la simulation à partir du fichier map.txt contenant la carte suivante :



Il faudrait faire la commande :

mpiexec -n 6 Rongeurville.exe map.txt 3 2

Le résultat de la partie, contenant tous les logs, est exporté dans le fichier Log.txt.