Compte Rendu
-
Simulation de réseaux de files d'attente

ADOUANE Cylia
ALI AHMEDI Mycipssa
BONNET Ludivine
DOUBI Dylan
HAMMAD Amir
HELLAL Ouiza
LAMOUR Lauriane
MOLINER Emma

Mercredi 20 mai 2020

Table des matières

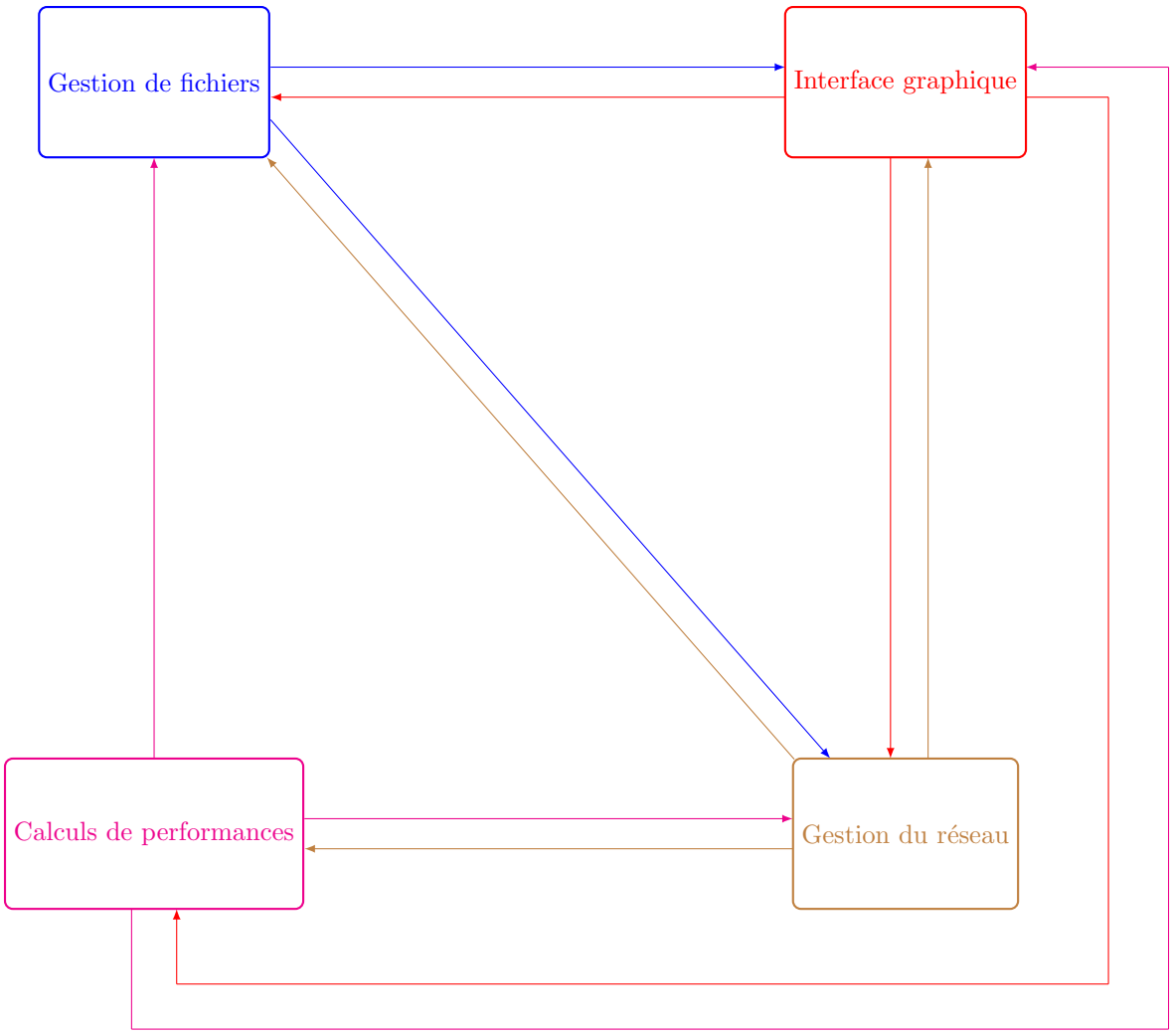
1 Introduction

Un réseau de files d'attentes est un ensemble de files d'attentes inter-connectées. Elles peuvent être vues de manière générale comme un regroupement de clients pouvant être des individus ou des requêtes informatiques, attendant de manière organisée leur tour pour un service fourni par un serveur. Elles résultent d'une demande supérieure à la capacité d'écoulement d'une offre.

Pour notre projet de fin de licence d'informatique, nous devons réalisé une application simulant des réseaux de files d'attente. Cette simulation doit se faire de façon discrète par l'utilisation d'un échéancier exécutant des évènements faisant évoluer la simulation. Les événements peuvent être des tâches tel que l'arrivée d'un client dans une file, l'arrivée d'un client dans un serveur ou la sortie d'un client du serveur. Cette simulation se résulte par un calcul des performances du réseau permettant de l'évaluer.

L'application nommée "MODALLEC" est destinée aux professionnels souhaitant améliorer leur réseau et aux particuliers souhaitant en apprendre davantage sur les réseaux de files d'attentes et leur fonctionnement.

2 Explication de l'architecture



Organigramme

L'étude des files d'attente nous a mené vers cet organigramme. Il permet de visualiser les étapes afin de réaliser une simulation d'un réseau de files d'attente.

Le module *Interface Graphique* propose à l'utilisateur de créer son propre réseau en ajoutant des stations (files et serveurs) et des classes et en les configurant. Il a également la possibilité de charger un réseau grâce au module *Gestion de fichiers*. Une fois le réseau crée, il peut lancer la simulation sur ce réseau qui sera affiché par l'interface graphique.

Le module *Gestion du Réseau* et le module *Interface Graphique* s'échange constamment des informations afin de sauvegarder en mémoire et afficher en temps voulu les informations du réseau de files d'attente.

Un calcul des performances est réalisée par le module *Calcul de performances* à la demande de l'utilisateur par l'appui sur le bouton *Results* de la fenêtre de simulation. Ce dernier permet d'évaluer le réseau construit par l'utilisateur. Ainsi l'utilisateur peut connaître les performances de son réseau et tenter de l'optimiser.

3 Description du fonctionnement

MODALLEC est une application portable, elle peut être lancée sous Linux sans avoir besoin d'effectuer une installation au préalable. Nous avons fait le choix de développer notre application en anglais dans le but qu'elle soit accessible à tous. Une fois que l'application est lancée une interface s'affiche. Nous avons décidé d'afficher des boutons en haut de la fenêtre qui permettent d'agir directement sur le réseau (ouvrir un fichier existant, sauvegarder, lancer/arrêter /pause/continuer la simulation).

A droite de cette même fenêtre, des boutons permettent la construction du réseau de files d'attente (ajouter/supprimer/modifier une file d'attente et son serveur, ajouter/supprimer une classe de clients). Une fois le réseau de files d'attente construit, il est possible de lancer la simulation et choisir sa durée de simulation. Durant la simulation, il est possible d'afficher les performances du réseau et des indications sont affichées pour suivre en temps réel l'avancée de la simulation.

3.1 Fonctionnement des modules

3.1.1 Interface graphique

Ce module est le lien direct entre l'utilisateur et le reste de l'application, c'est-à-dire à travers les boutons qui se trouvent sur l'interface graphique, l'utilisateur peut réaliser la conception et gérer le déroulement de la simulation de son réseau de files d'attente. Nous avons mis en place des boîtes de dialogues afin de faciliter l'interaction entre utilisateur et notre application.

3.1.2 Gestion du réseau

Ce module est situé à l'arrière-plan dans notre application, mais il est tout autant important que l'interface graphique. Il s'occupe de la simulation à événements discrets sous forme d'un échéancier du réseau créé/choisi par l'utilisateur. L'échéancier qui est l'élément fondamental de ce module, d'ailleurs c'est lui qui donne le rythme en traitant les événements suivants un ordre chronologique.

3.1.3 Calculs de performances

Ce module s'occupe essentiellement d'effectuer les calculs concernant les performances du réseau.

3.1.4 Gestion de fichiers

Ce module permet de sauvegarder le réseau et les performances du réseau dans un fichier. Il permet également de charger un réseau sauvegardé p réalablement. L'application est dotée d'une méthode de sauvegarde de l'historique de toutes les performances calculées auparavant.

4 Changements techniques

4.1 Module Interface graphique

4.1.1 FenetreInfoReseau.hh

```
#include <qt4/QtGui/QScrollArea>
```

Cette bibliothèque contient la classe permettant de définir un périmètre qui aura une barre de défilement si les éléments affichés dans la fenêtre venaient à dépasser la taille de la fenêtre.

4.1.2 DialogAddClass.hh

```
#include <qt4/QtCore/QString>
```

Cet include a été ajouté pour pouvoir convertir les données récupérées des classes en string et les insérer dans les labels correspondant.

4.1.3 DialogConfigClass.hh

```
#include <qt4/QtCore/QString>
#include <string>
#include "ClasseClient.hh"
```

Les includes <qt4/QtCore/QString> et <string> permettent pouvoir convertir les données récupérées de la classe en attribut, en string et les insérer dans les labels correspondant. L'include "ClasseClient.hh" a été ajouté car la classe prend en argument un objet de cette classe.

4.1.4 DialogConfigFile.hh

```
#include <string>
```

Cet include permet de convertir en string les données récupérées de la file en attribut de la classe et les insérer dans les labels correspondant.

4.1.5 FenetreCreation.hh

```
#include <qt4/QtGui/QMessageBox>
#include <qt4/QtGui/QInputDialog>
#include <qt4/QtGui/QFileDialog>
#include <qt4/QtGui/QMenuBar>
#include <qt4/QtGui/QGridLayout>
```

Les includes <qt4/QtGui/QMessageBox>, <qt4/QtGui/QInputDialog> et <qt4/QtGui/QFileDialog> ont été ajoutés pour pouvoir interagir avec l'utilisateur à travers des boîtes de dialogue. L'include <qt4/QtGui/QGridLayout> permet d'avoir une boîte de rangement sous forme d'une grille pour ordonner les objets dans la fenêtre de création. L'include <qt4/QtGui/QMenuBar> a été ajouté pour pouvoir ranger les objets QMenu de la classe dans un menu.

```
QLabel * infos_reseau;
```

Cet attribut permet d'afficher le nombre de files et de classes dans le réseau. Cela permet d'apporter des informations supplémentaire sur le réseau pour l'utilisateur.

```
QGridLayout * layout_principal;
```

Ce nouvel attribut permet de ranger plus facilement les objets dans la fenêtre de création. Il était nécessaire pour avoir une interface claire.

```
QMenuBar * menuBarFichier;
QMenuBar * menuBarPropreReseau;
QMenuBar * menuBarClass;
QMenuBar * menuBarQueue;
```

Ces nouveaux attributs permettent de ranger les attributs de type QMenu de la classe sous forme d'un menu déroulant.

```
QPushButton * change_window;
```

Ce nouvel attribut permet de switcher entre la fenêtre de simulation et la fenêtre de création afin de faciliter les accès aux fenêtres pour l'utilisateur.

```
QPushButton * voir_routage;
```

Ce nouvel attribut permet d'activer le slot aff_routage_classe().

```
bool verif_pos_clic_file(const int x, const int y);
bool verif_proximite_clic(const int x, const int y);
```

Ces deux méthodes ont été ajoutées pour réduire la répétition de code. Ils permettent de déterminer si lorsque l'utilisateur clique sur la zone de dessin, une file, l'entrée ou la sortie s'y trouve.

```
void aff_routage_classe();
```

Ce nouvel attribut permet d'afficher le routage de la classe indiquée par l'utilisateur. Il a été ajoutée pour éclaircir l'affichage du réseau qui devenait illisible à l'affichage du routage des toutes les classes.

4.1.6 FenetreInfoReseau.hh

```
QVBoxLayout * rangement_scroll;
QScrollArea * scrollArea;
QWidget * fenetre_affichage;
```

Ces nouveaux attributs ont été ajoutés pour créer un environnement pour permettre à la fenêtre d'être défiler. Les en-têtes correspondant à ces objets ont été également ajoutés.

4.1.7 FentreSimulationRF.hh

```
#include "SimulationRF.hh"
```

Cette bibliothèque contient la classe FenetreSimulationRF qui permet d'utiliser une fenêtre de simulation d'un réseau fermé. Cette fenêtre est liée à la fenêtre de création.

```
HistoriqueRF historique;
```

Cet attribut permet d'accéder à la classe Historique qui manipule la structure PerformancesRF.

```
SimulationRF& getSimulation();
```

Cette méthode permet l'utilisation de la structure SimulationRF.

```
HistoriqueRF& getHistorique();
```

Cette méthode a été ajouté pour pouvoir récupérer l'historique et les sauvegarder.

```
void stop();
```

Cette méthode a été ajouté pour pouvoir arrêter la simulation.

4.1.8 FenetreSimulationRO.hh

```
#include "SimulationRO.hh"
```

Cette bibliothèque contient la classe FenetreSimulationRO qui permet d'utiliser une fenêtre de simulation d'un réseau fermé. Cette fenêtre est liée à la fenêtre de création.

```
HistoriqueRO historique;
```

Cet attribut permet d'accéder à la classe Historique qui manipule la structure.

```
SimulationRO& getSimulation();
```

Cette méthode permet l'utilisation de la structure SimulationRO.

```
HistoriqueRO& getHistorique();
```

Cette méthode a été ajouté pour pouvoir récupérer l'historique et les sauvegarder.

```
void stop();
```

Cette méthode a été ajouté pour pouvoir arrêter la simulation.

4.1.9 Vue.hh

```
void remise_a_zero();
```

Cette méthode remet à zéro les coordonnées du dernier clic sur la vue. Il est nécessaire pour assurer que le clic peut-être traiter dans les fonctionnalités nécessitant de récupérer la position du clic.

```
void clicked();
```

Ce signal a été créé pour permettre de savoir si un clic a été émis sur la vue. Il est nécessaire pour pouvoir récupérer les coordonnées du clic.

4.2 Gestion de fichiers

4.2.1 PerfsFile.hh

```
float taux_perte
```

Cet attribut est passé du type int au type float pour permettre une plus grand précision du résultat du calcul du taux de perte.

```
File *file
```

Cet attribut était statique et a été changé en pointeur. La file passé en paramètre du constructeur de la classe devait être copié. Un problème survenait à la suppression d'un objet de cette classe car elle entraînait la suppression de cet attribut qui possédait une copie de la liste des serveurs de la file passé en paramètre du constructeur qui est une liste de pointeurs. Les serveurs étaient donc supprimer.

4.2.2 HistoriqueRO.hh

```
typedef struct performancesRO{
    PerfsRO* perfReseau;
    list <PerfsFile*> ListePerfsFiles;
}PerformancesRO;
```

La structure a été modifiée par le changement des champs en pointeur. La liste contient des pointeurs car elle ne pouvait se construire avec des éléments statiques de part l'absence d'un constructeur par défaut PerfsFile. Le champ perfReseau a été passé en pointeur pour pouvoir créer l'élément de manière dynamique et gérer sa suppression.

```
void ajouterPerfsFile (File& f,const int duree_simulation);
```

Le paramètre *duree_simulation* a été ajouté pour permettre le calcul du taux d'utilisation des serveurs de la file en paramètre.

4.2.3 HistoriqueRF.hh

```
struct performancesRF{
    PerfsRF* perfReseau;
    list <PerfsFile*> ListePerfsFiles;
};
```

La structure a été modifiée par le changement des champs en pointeur. La liste contient des pointeurs car elle ne pouvait se construit avec des éléments statiques de part l'absence d'un constructeur par défaut PerfsFile. Le champ perfReseau a été passé en pointeur pour pouvoir créer l'élément de manière dynamique et gérer sa suppression.

```
void ajouterPerfsFile (File& f,const int duree_simulation);
```

Le paramètre *duree_simulation* a été ajouté pour permettre le calcul du taux d'utilisation des serveurs de la file en paramètre.

4.3 Gestion du Réseau

4.3.1 ClasseClient.hh

```
vector <int> routage;
vector <int> getRoutage ();
```

La vecteur de routage contenant des pointeurs sur des objets *File* a été modifié pour contenir des entiers car un problème d'accès aux files existait. Son accesseur a été également modifié.

4.3.2 Client.hh

```
Class ClasseClient
Class File
```

Une déclaration partielle des classes ClasseClient et File a été faite pour gérer les références croisées avec *File* et avec *ClasseClient*.

```
int getFileActuelle ();
```

Cette méthode renvoie l'identifiant de la file actuelle. Elle renvoyait le pointeur sur l'objet *File* mais un problème d'accès aux files existait.

4.3.3 Serveur.hh

```
Class Client;
```

Une déclaration partielle de la classe Client a été faite pour gérer la référence croisée.

4.3.4 File.hh

```
#include <list>
```

Cette include a été ajoutée car un objet de la classe *File* possède un ensemble de listes.

```
float getTpsServiceMoy ();
```

Le type de retour de cette méthode est passé de *int* à *float pour une meilleure précision du résultat.*

```
float getTpsAttenteMoy ();
```

Le type de retour de cette méthode est passé de *int* à *float pour une meilleure précision du résultat.*

```
void setTempsAttenteMoyen( float temps );
```

Le paramètre de cette méthode est désormais un *float* car *tps_attente_moyen* est un float.

```
void setTempsServiceMoyen( float temps )
```

Le paramètre de cette méthode est désormais un *float* car *tps_service_moyen* est un float.

```
void trier_priorite_file() ;
```

Cette méthode a été ajoutée pour trier la file lors de la configuration de l'ordonnancement par priorité.

4.3.5 Reseau.hh

```
File* chercherFile(int IDFile) ;
```

Cette méthode a été pour récupérer une file dans le réseau en fonction de son identifiant.

4.3.6 Echeancier.hh

```
struct Compare_evenement{
    Bool operator()
}
```

Cette structure a été ajoutée pour pouvoir trier la liste des événements à l'aide de la méthode *sort()* de la classe *list*.

4.3.7 DonneesCalculsRO.hh

```
struct TSClient{
 int numero client ;
int date sortie ;
int date entree ;
int tps sejour ;
struct TSClient * suivant ;
}TSClient ;
```

a été changé en :

```
struct TSClient{
 int numero client ;
int date sortie ;
int date entree ;
int tps sejour ;
}TSClient ;
```

La classe *list* est utilisée. Il n'est donc plus nécessaire de conserver le pointeur sur l'élément suivant.

4.3.8 Simulation.hh

```
Struct Poisson {
Int nombre ;
Double proba ;
}Poisson ;
```

Cette structure a été ajoutée pour conserver un nombre et sa probabilité.

```
Void setDureeSimulation(int duree) ;
Void setNbClientsTot(int nombre) ;
Void setNbFilesTot(int nombre) ;
Void setNbClassesTot(int nombre) ;
Void setNbServeursTot(int nombre) ;
Void incrementer_nbClientsTot() ;
Void incrementer_nbFilesTot() ;
Void incrementer_nbClassesTot() ;
Void incrementer_nbServeursTot() ;
```

Ces méthodes ont été ajoutées afin d'incrémenter et d'affecter des valeurs aux variables de *Simulation*.

```
Unsigned long long facto(int n);
Double calcul_proba_poisson(int k, int lambda) ;
```

Ces méthodes ont été ajoutées afin de calculer une probabilité suivant une loi de Poisson de paramètre lambda.

4.3.9 SimulationRF.hh

```
#include  DonneesCalculsRF.hh
```

Cette include a été ajoutée car la classe possède un attribut de type DonneesCalculsRO.

4.3.10 SimulationRO.hh

```
#include DonneesCalculsRO.hh
```

Cette include a été ajoutée car la classe possède un attribut de type DonneesCalculsRO.

5 Description des points délicats

```
void routage_clients();
```

Cet algorithme va distribuer les clients dans les files. Tout d'abord, on va router les clients multi-classes dans les files correspondant à leur position dans le routage. Ensuite, on va router les clients mono-classes. On va itérer sur les files dans le réseau. A chaque file qu'on traite, on va réduire un compteur correspondant au nombre de file dans le réseau. On s'occupe d'abord des files ayant une distribution d'arrivée constante, on envoie autant de client que le nombre défini par la file. Ensuite, si la file a une distribution d'arrivée suivant une loi de Poisson, on va calculer,pour chaque file, un taux d'arrivée moyen lambda et calculer à l'aide de la méthode *int calculer_taux_arrivee_poisson(const int lambda)* un taux d'arrivée aléatoire. On envoie autant de client que le nombre défini par ce taux. Si le compteur est nul et qu'il reste des clients, on distribue un client à chaque file et on recommence jusqu'à ce qu'il n'y ait plus de clients à router. A la fin, on supprime les clients qui n'ont pas réussi à entrer dans une file car celle-ci était pleine.

```
void distribution_initiale(const int nb_clients_initial);
```

Dans cet algorithme,un nombre de clients équivalent à la variable en paramètre est créé. Ensuite, le même procédé que dans *routage_clients()* est effectué.

Un point très délicat à gérer était la libération de la mémoire lors du déroulement de la simulation. Il était nécessaire de gérer les suppressions dans les listes qui engendraient un grand nombre de problème de par l'impossibilité de supprimer un élément durant leurs défilements. Les algorithmes de *Gestion du réseau* ont donc été adaptés pour gérer ces problèmes.

6 Comparaison avec l'estimation des coûts

Notre estimation des coûts s'est avéré fausse.
Pour le module Gestion de Fichier, nous avons obtenu 300 lignes sur 500 estimés.
Pour le module Mesures de performances nous avons obtenu 1100 lignes sur 600 estimés.
Pour le module Gestion de Réseau, nous avons obtenu 3000 lignes sur 2500 estimés.
Pour le module Interface Graphique, nous avons obtenu 7000 lignes sur 3000 estimés.

7 Conclusion

MODALLEC est une application conçue dans un but précis qui est de réaliser une simulation de réseaux de files d'attente, grâce à une interface graphique qui facilite l'interaction avec l'utilisateur.

Nous avons principalement utilisé des pointeurs, ainsi un grand nombre de suppressions mémoire était nécessaire pour le bon fonctionnement de notre application. Cependant, il nous reste quelques erreurs à propos de ces suppressions.

La fonctionnalité *Stop* n'est pas fonctionnelle à cause de problèmes de suppressions mémoire. Elle n'a donc plus le même fonctionnement qu'indiquer dans le cahier des charges. Lors de son enclenchement, elle termine la simulation et efface l'ensemble de la simulation.

La fonctionnalité *Zoom (in et out)* n'est pas comme nous l'avons réfléchie. La taille de la surface des files ne s'agrandit pas au fur et à mesure d'ajout des files. La taille de la zone de dessin est fixée, le zoom permet seulement d'agrandir et de rétrécir la représentation des files d'attente.

Travailler en équipe était difficile mais enrichissant à la fois. Nous avons acquis de maintes compétences (mathématiques et algorithmiques) et nous avons également appris à échanger sur différentes plateformes de communication. Nous avons rencontré quelques soucis d'organisation dû à des problèmes d'emploi du temps et du mode de vie des différents membres du groupe. Mais au fil des semaines, nous avons appris à mieux nous organiser pour faire avancer ensemble le projet.