Projet Kayak

IFT3051

Rapport Final

Présenté à :

Yann-Gaël Guéhéneuc
et
Houari Sahraoui

par :

Antoine Tremblay

LE 28 JANVIER 2005
UNIVERSITÉ DE MONTRÉAL

Table des matières

1. Introduction

Cette dernière année, BitTorrent c'est imposé comme un outil efficace pour la distribution de données volumineuses très en demande à un temps précis. Désengorgeant ansi plusieurs réseaux de distribution par ftp qui fléchissaient en période de forte demande.

Son principal avantage par rapport à ces réseau étant de faire participer efficacement les usagers au canal de distribution, BitTorrent a ainsi permis à ces réseaux d'économiser leur précieuse bande passante. De plus, nous ne sommes plus pris avec des problèmes de ftp pleins ou de vitesse dérisoire lorsqu'on nous voulons télécharger par exemple, la dernière copie de notre distribution en ISO.

1.1 Problème

Cependant, les clients actuels bien que souvent très bien faits ont tous une épine dans leur pied, de l'utilisation abusive de ressources à la disponibilité sous Windows seulement. Bref, il est facile de ne pas se satisfaire des clients existants.

1.2 Solution

Un nouveau client.

1.3 Objectifs

Le but ultime de ce projet est de créer un client convivial, simple et rapide pour Linux, Windows et Mac dont les principaux objectifs sont :

1. Le transfer d'information par BitTorrent

2. La recherche et le listing des torrents disponibles à partir de plusieurs sources

3. Offir une interface simple à la création et à la publication de torrents

Dans le cadre du cours ift3051 par contre, il est impossible de réaliser les objectifs au complet compte tenu du temps limité. Ces objectifs sont donc modifiés pour le cours.

De plus, la façon de réaliser le projet sera particulière pour le cours, il est demandé de réaliser une premiere version du logiciel à mi-temps dans laquelle on ne tiendra pas compte des patrons de conception et ensuite de réécrire l'application en tenant compte de ces patrons.

Il s'agit donc de réaliser une partie de l'objectif 1. soit de pouvoir télécharger un fichier avec BitTorrent à partir d'un seed. Et de résaliser cela de deux manières.

2. Tour d'horizon

Kayak est un client simple pour le téléchargement de fichiers par BitTorrent. Il comporte un GUI minimal d'où l'ont peut ouvrir un .torrent pour le télécharger. Le GUI affiche les informations sur le torrent et sa progression.

Kayak est composé de 43 classes et comporte environ 11 000 lignes au total (toutes lignes incluses). Il a été réalisé en environ 360 heures du 7 Septembre 2004 au 26 Janvier 2005.

2.1 BitTorrent

2.1.1 Qu'est-ce que BitTorrent ?

BitTorrent est un protocol peer-to-peer qui permet l'échange de fichiers volumineux et en très forte demande de façon assez efficace.

Pour ce faire, BitTorrent sépare le fichier en petites parties appelées pièces. Ces parties sont ensuite échangées entre les peers.

Par contre, à l'instar de plusieurs réseaux peer-to-peer , BitTorrent n'est pas complètement distribué, alors que la pluspart des protocoles utilisent une méthode ou une autre pour détecter leurs peers, BitTorrent utilise un Tracker qui permet à tout les peers qui téléchargent un fichier donné de se connaitre très facilement.

Ceci donne des masses de peers disponibles pour le téléchargement d'un fichier précis beaucoup plus grandes que dans les autres réseaux et contribue énormément au succès du protocol.

De plus, l'échange des pièces est très bien fait, dès que vous avez une pièce du fichier vous pouvez commencer à l'envoyer aux autres peers. Ainsi un nouvel arrivant dans le réseau commence très tôt à distrubuer le peu qu'il a aux autres. Ceci fini par faire l'effet d'une vague donnant de plus en plus de bande passante disponible à mesure que des peers s'ajoutent au réseau aulieu de faire l'effet contraire.

2.1.2 Les composantes de BitTorrent

1) Le fichier .torrent

Bien souvent tout commence par un fichier .torrent, ce fichier contient les informations nécessaires pour le téléchargement du fichier en question. Il contient entre autres l'adresse du Tracker, le nombre de pièces du fichier , la longeur du fichier, la longeur des pièces et le SHA-1 de chaque pièce.

2) Le Tracker

Le Tracker sert à rassembler les peers, un peer peut donc demander au tracker de lui donner la liste des autres peers qui sont en train de télécharger le fichier.

3) Le client BitTorrent

Le client rassemble touts les composants ensembles pour le téléchargement du fichier.

2.1.3 Le bencodage

Le bencodage est une partie très importante de BitTorrent, en effet, le .torrent par exemple est bencodé. C'est à dire que son information est formaté dans le format de bencodage de BitTorrent. Les interactions avec le Tracker sont elles aussi bencodées.

Le bencodage supporte quatre type de données : Des chaines de caractères, des nombres, des listes et des dictionaires.

Voici une description rapide du format de bencodage :

1) Les chaines de caratères : <longeur de la chaine en décimal ASCII>:<chaine>

Example : 7:bonjour pour la chaine "bonjour"

2) Les nombres : i<nombre en décimal>e

Example : i55e pour le nombre 55

3) Les listes : l<type bencodé>e

Example : l4:allo3:toie est la liste de chaines "allo, toi"

4) Les dictionaires : d<chaine bencodée><élément bencodé>e

Par exemple : d4:fichier8:truc.avie représente le dictionaire : '' fichier -> truc.avi "

2.2 Fonctionalitées

Kayak permet :

1. L'ouverture d'un .torrent et son décodage à partir d'un format bencodé.

2. Le transfer d'un torrent contenant un fichier simple à partir d'un simple seed.

À noter que Kayak n'est pas encore un peer valide, il ne peut s'occuper de plus d'un seed ou de plusieurs peers et les peers ne peuvent lui demander des pièces d'information. Il reste encore un mois de dévelopement pour atteindre cet objectif.

2.3 Environement requis

Kayak demande d'avoir wxGTK 2.4.2 d'installé sous linux avec le support pour les sockets et la librarie boost.

3. Système en détails

3.1 Classes

1. BDecoder :

Le BDecoder est la pière angulaire pour le décodage des .torrents il permet de décoder un fichier .torrent ou une string en format bencodé

2. BDictionary , BInteger, BList , BString :

Ces classes composites contiennent un élément du format bencodé et sont capable d'encoder leur élément selon le format bencodé.

3. BValue

Cette classe à pour but de pouvoir traiter n'importe quel classe composite (voir point 2.) de la même manière. On peut donc appeler des méthodes sur une BValue sans avoir besoin de savoir si c'est un BString ou un BList ou autre. Ceci viens implémenter le patron de conception composite.

4. Protocol

Protocol sert de classe de base pour tout ce qui est intéraction avec les sockets, elle permet de : créer un protocol client , serveur ou les deux à la fois, d'écrire sur le socket et de lire sur le socket avec les buffers apropriés. Cette classe sert aussi d'abstration dans le cas d'un évenement socket, en effet comme cette classe contient des méthodes virtuelles pour chaque évenement socket le SocketEvtHandler n'a pas besoin de savoir de quel protocol il s'agit, il ne fait qu'appeler les méthodes du pointeur de type Protocol. (Voir 3.2 Les sockets dans Kayak)

4. BitTorrentClient

BitTorrentClient dérive de Protocol et implémente le protocol BitTorrent. Elle permet de répondre à des événements sockets selon le protocol et d'envoyer des messages corrects. Chaque instance de BitTorrentClient représente en fait une connection avec un peer. (Voir 3.3 Le protocol BitTorrent dans Kayak)

5. BitTorrentClientEvent

Cette classe représente un événement envoyé par le BitTorrentClient elle peut contenir de l'information utilie pour le traitement de l'évenement.

6. BitTorrentClientEvents

Ceci est une classe abstraite qui contient les méthodes des événements pour le BitTorrentClient.

7. BitTorrentContainer

Cette classe contient une liste de BitTorrentClient, elle est utilisée par le PeersCtrl pour pouvoir maintenir à jour la liste des clients actifs.

8. BitTorrentServer

Cette classe n'est pas encore implémentée mais, elle est là dans le but de recevoir des connections de peers sur BitTorrent.

9. DebugPanelCtrl

Cette classe permet d'afficher de l'information dans le tab Debug. Elle était d'abord prévue à cause de l'utilisation des threads qui rendaient les autres fonctions de debug non sécuritaires. Elle est resté pour le futur.

10. DiskioCtrl

Cette classe controle l'accès au disque en mode lecture ou écriture, seulement l'écriture est implémenté en ce moment.

11. DownloadsListCtrl

Ce controleur, controle l'affichage de la liste de downloads. C'est le tab principal de l'application.

12. DownloadsListCtrlEvent

Cette classe représente un événement produit par DownloadsListCtrl.

13. DownloadsListCtrlEvents

Cette classe abstraite contient les méthodes des événements du DownloadsListCtrl.

14. EvtHandlerContainer

Ceci est un template qui permet de contenir des "listeners" , de les enlever, et d'aviser les listeners des évenements produit par la classe à laquelle il apartient. Il est important de noter ici qu'on peut ajouter ces listeners dans deux listes différentes une liste UI et une liste normale. La différence réside dans le fait que la liste UI n'est pas nécessairement exécuté tout de suite, elle est ajoutée à la file d'événements de wxWidgets pour être exécuté plus tard. Ce qui n'est pas le cas de la liste normale qui est exécuté tout de suite. (voir 3.4 Les listeners)

15. Hasher

Cette classe permet de faire le hash SHA-1 d'une string.

16. Http

Cette classe implémente une partie du protocole Http pour la communication avec le tracker.

17. HttpEvent

Cette classe représente un événement produit par Http.

18. HttpEvents

Cette classe abstraite contient les méthodes des événements de Http.

19. Main_frame

Cette classe gère la fenaitre principale, elle contient entre autres les autres controles UI comme le DownloadsListCtrl.

20. Main_frameEvent

Cette classe représente un événement produit par Main_frameEvent.

21. Main_frameEvents

Cette classe abstraite contient les méthodes des événements de Main_frame.

22. Peer

Cette classe contient l'information d'un peer comme son peer_id et son adresse ip en format d'adresse IPV4 de wxWidgets.

23. PeersCtrl

Cette classe gère les interactions entres les peers et avec le tracker. (voir 3.3 Le Protocol BitTorrent dans Kayak)

24. PeersCtrlEvent

Cette classe représente un événement produit par PeersCtrlEvent.

25. PeersCtrlEvents

Cette classe abstraite contient les méthodes des événements de PeersCtrl.

26. PeersList

Cette classe contient la liste de peers tel que reçu du tracker, lorsque des peers sont reçus ils sont ajouté à la liste si ils n'y sont pas déja présents.

27. Piece

Cette classe contient l'information d'une pièce d'un fichier, elle contient par exemple son offset dans le fichier.

28. PiecePicker

Cette classe permet de dire au BitTorrentClient quel pièce il doit demander à son peer. (voir 3.3 Le Protocol BitTorrent dans Kayak)

29. SocketEvtCtrl

Cette classe s'occupe de gérer les événements sockets de wxWidgets et de les envoyer à l'instance de Protocol qui correspond au socket d'où viens l'évenement. (voir 3.3 Le Protocol BitTorrent dans Kayak)

30. SocketList

Contient la liste des sockets et leur instance de protocol associé.

31. Torrent

Contient toute l'information relative à un torrent ainsi que tout les controleurs associé.

32. TorrentCtrl

Contient la liste des torrents et controle les actions sur les torrents.

33. TorrentCtrlEvent

Cette classe représente un événement produit par TorrentCtrlEvent.

34. TorrentCtrlEvents

Cette classe abstraite contient les méthodes des événements de TorrentCtrl.

35. TorrentinfoPanelEvtCtrl

Controle l'affichage du tab d'information sur un torrent.

36. TorrentinfoPanelEvent

Cette classe représente un événement produit par TorrentinfoPanelEvent.

37. TorrentinfoPanelEvents

Cette classe abstraite contient les méthodes des événements de TorrentinfoPanelEvtCtrl.

38. Tracker

Controle les interactions avec le tracker et contient toute l'information relative au tracker.

39. TrackerEvent

Cette classe représente un événement produit par TrackerEvent.

40. TrackerEvents

Cette classe abstraite contient les méthodes des événements de Tracker.

3.2 Les sockets dans kayak

Comme kayak est une application peer-to-peer les sockets sont un élément essentiel de l'application. Aussi une bonne conception du système de sockets est de vigeur.

Voyons d'abord les contraintes auquel le logiciel doit s'adapter :

1. Les sockets ne doivent pas blocquer l'application.

2. La prise en charge des événements sockets doit être très rapide.

3. Comme Kayak utilise http pour la communication avec le tracker, on doit pouvoir utiliser facilement plusieurs protocols

wxWidgets offre plusieurs options pour la gestion des sockets : utiliser les sockets bloquants, utiliser les threads avec les sockets bloquants ou utiliser les sockets non-bloquants.

Pour répondre aux contraintes j'avais d'abord choisi une approche avec les threads et les sockets bloquants mais cette aproche ne fonctionait pas bien dans wxWidgets on se retrouvais facilement avec des socket io errors pour des raisons mystérieuses interne à wxWidgets.

J'ai donc prévilégié l'utilisation des sockets non-bloquants. Voici comment ceci fonctionne :

L'application contient :

1) Une SocketList :

Qui contient une hashmap avec l'adresse du socket comme hash et le pointeur de type protocol comme lien.

2) Un SocketEvtHandler

Qui réparti les événements sockets de wxWidgets vers l'instance de protocol à laquelle appartient le socket.

3) Une classe abstraite Protocol

Qui s'occupe d'abstraire les protocoles dérivé à la gestion des sockets, lors d'un événement socket on peut donc appeler les méthodes virtuelles de Protocol sans ce soucier de quel protocol dérivé il s'agit.

Voici un sénario :

1. Un socket non bloquant est crée dans la classe Protocol.

2. Dans le constructeur de la classe dérivée de Protocol (ex : Http) ce socket est ajouté dans la liste des sockets de l'application soit la SocketList.

3. Un événement socket de wxWidgets arrive, par exemple, une connection est effectuée. L'événement est donc envoiyé au SocketEvtHandler qui cherche dans la table de hashage de SocketList pour le socket et son instance de Protocol. Une fois trouvé il appele la méthode correspondante à la bonne instance de Protocol.

4. L'événement socket est traité par l'instance dérivée de Protocol.

3.3 Le Protocol BitTorrent dans Kayak

Pour implémenter le protocol Kayak a recours aux élements suivants :

À noter que la pluralité est en fonction d'un torrent.
Donc, il y'a Des BitTorrentClients pour le transfer d'un fichier par bittorrent mais il n'y a qu'Un PeersCtrl.

1) Des BitTorrentClient

Un BitTorrentClient gère une connection avec un peer.

2) Des DiskioCtrl

Qui gèrent les accès de lecture et d'écriture au disque. Chaque BitTorrentClient possède un DiskioCtrl.

3) Un PiecePicker

Qui tiens à jour l'état de toutes les pièces d'un torrent par rapport à leur peer. C'est à dire qu'il contient la liste des pièces intéressantes que possède chaque peer. C'est à lui que les BitTorrentClient demandent la prochaine pièce à télécharger.

La sélection de la pièce est faite en fonction de 3 algorithmes selon le cas :

1. Si une pièce est déja en téléchargement on prévilégie cette piece.
2. Si toutes les pièces sont présentes de façon égale sur le réseau on choisi la pièce de façon aléatoire.
3. Enfin si les pièces sont réparties de façon inégales, on prend la pièce la plus rare en premier.

4) Un PeersCtrl

Qui gère les interactions entre les BitTorrentClients et le tracker. Ainsi que les événements à envoyer pour l'affichage de la progression et du statut du torrent.

5) Un Tracker

Le Tracker gère l'interaction avec le tracker du torrent, en particulier il doit aviser le tracker d'événements comme le début du téléchargement ou sa fin. Il doit aussi télécharger la liste des peers à partir du tracker.

Voici un sénario :

1) On click sur Start

2) Le Tracker envoie au tracker un message Start , et demande de lui transmettre les peers disponibles.

3) Une fois les peers reçus il envoie un message au PeersCtrl pour qu'il initie les connections aux différents peers.

4) Le PeersCtrl réalise cela en créant de nouvelles instances de BitTorrentClient

5) Une fois une instance de BitTorrentClient crée elle initie la connection avec le peer en lui envoyant un message de handshake.

6) Une fois le handshake accepté, si le peer a des pièces dans lesquelles nous sommes intéressées, le BitTorrentClient envoie un message "interessé".

7) Une fois ce message envoyé il demande au PiecePicker une pièce à télécharger et envoie un message de request pour un block de cette pièce au peer.

8) Le peer envoie un message contenant la pièce au BitTorrentClient qui l'écrit dans le fichier à l'aide de son DiskioCtrl.

9) les étapes se répetent 7 et 8 se répetent jusqu'à la fin du fichier ou que le peer n'ai plus de pièces que nous n'avons pas déja.

3.4 Les Listeners

Les Listeners sont un élément très important dans l'application, en effet ils permettent une communication claire entre les classes.

Voici une discussion sur l'implémentation choisie du patron de conception des listeners dans kayak :

Dans ce patron de conception on crée une interface qui correspond aux events générés par un type,

- TypeListener :

    - virtual type_event1(Event) = 0
    - virtual type_event2(Event) = 0

Ensuite cette interface est implémentée dans toutes les classes qui doivent écouter cet event par example :

- Type (ex :Tracker) : public TypeListener

    - virtual type_event1(Event)
    - virtual type_event2(Event)

Ainsi Type est donc capable de gérer les events du Type de TypeListener,

Aussi, dans Type , on crée des objects de type TypeListener = new Tracker() par exemple donc on déspécialise nos classes pour pouvoir appeler les méthodes de TypeListener , sans ce soucier de la classe de controle qui en est dérivée.

De plus, Type doit implémenter des méthodes pour ajouter des TypeListeners les enlever et les garder en mémore (array), comme Type est dérivé de la classe abstraite TypeListener , ceci se fait bien avec un seul type de type TypeListener

```
addTypeLister(TypeListener)
removeTypeListener(Typelistener)
        -TypeListenerArray;
```

Ce qui donne par exemple;

```
class Type : public TypeListener , public Type2Listener

public :
Type() {
        TypeListener1 = new Tracker();
        TypeListener2 = new PeerControl();
}
```

À noter ici que je décris les méthodes d'événements comme des récepteurs d'événements et des producteurs d'événements donc, type_event1 recoit un événement de type Event mais aussi crée un événement event.

```
	//events derivés de TypeListener
	type_event1(Event) {
		...
		...
		TypeEvent event;
		for all TypeListener* ->type_event1(event);
	}


	type_event2(Event) {
		...
		...
		TypeEvent event;
		for all TypeListener* ->type_event1(event);
	}

	// events dérivés de Type2Listner

	type2_event1(Event)  {
		...
		...
		TypeEvent event;
		for all TypeListener* ->type2_event1(event);
	}

class TypeListener {


	addTypeLister(TypeListener)
	removeTypeListener(Typelistener)


	virtual on_event1() = 0
	virtual on_event2() = 0

	private:

	TypeListenerArray

}
```

Problèmes de cette façon de faire :

1. addTypeListener , removeTypeListener et la structure TypeListeneArray sont répétées dans chaque classes Type pour leur types respectifs

2. L'appel TypeListener->type_event1() crée une incompatiblité avec le modèle d'events de wxwidgets (celui-ci ferait wxPostEvent(TypeListener*, Event);

3. La classe Event est déja bien définie avec wxEvent déja existante dans wxwidgets la refaire serait superflue.

Solutions :

Ici j'ai décidé de créer un template : EvtHandlerContainer qui contient les méthodes add, add_ui remove , remove_ui, notify , notify_ui et notifyall et qui prend dans son template le type de l'event et la classe abstraite qui contient les méthodes des événements.

Par exemple : EvtHandlerContainer<BitTorrentClientEvents, BitTorrentClientEvent> evthandler_container;

Déclare un EvthandlerContainer pour les listeners qui peuvent écouter des événements de type BitTorrentClientEvents

Au sujet des différents notify , add et remove :

Il y a 2 types pour ces méthodes soit : le type normal et le type ui. Ceci est du au fait qu'il y a 2 façon de propager un événement dans wxWidgets soit par : ProcessEvent(event) ou par AddPendingEvent(event).

Dans le cas d'un event ajouté par add et propagé avec notify, on utilise la méthode ProcessEvent pour propager l'événement. Cette méthode exécutera alors immédiatement l'évenement en appelant la fonction requise tout de suite et en retournant ensuite.

Mais dans le cas d'un event ajouté par add_ui et propagé avec notify_ui , on utilise la méthode AddPendingEvent. Cette méthode ajoutera l'événement à la liste d'événements à executer de wxWidgets. Elle retournera donc tout de suite sans avoir executé l'événement, l'exécution se fera dans l'ordre des événements dans la file d'exécution de wxWidgets.

Notifyall propage l'événement à tout les listeners par leur méthode respective.

À noter donc, que un listener ajouté avec add ne peut recevoir des événements par notify_ui et vice versa.

L'appel ici est donc remplacé par la manière de wxWidgets combiné au template afin de garder la cohérance du programme.

Soit :

```
type_event2(Event) {
        ...
        ...
        Event event;
        event.set_data(x);
        for all TypeListener* ->type_event1(event);
}
```

Remplacé par

```
type_event2(wxCustumEvent) {
        ...
        ...
        TypeEvent event(EventID);
        event.set_data(x);

  evthandler_contaier->notifyall(event);
}
```

Et Dans la classe dérivé de TypeListener

```
EVENT_TABLE()
EVT_TYPE(ID, type_event1)
END EVENT_TABLE()

type_event1(wxCustumEvent) {

}
```

aulieu de simplement

```
type_event1(Event) {

}
```

Conclusion:

Après réflextion le patron de conception des Listeners s'applique très bien et serait probablement préférable à la façon de wxWidgets comme lorsqu'on utilise wxWidgets on doit gérer en plus des fonctions un set d'IDs et y faire correspondre les bonnes fonctions.

Cependant, en procédant ce cette manière on ne brise pas le style wxWidgets du logiciel, et nous n'avons pas à faire la correspondance entre le système d'events du programme et celui de wxWidgets et comme on retrouve le système de wxWidgets dans presque toute les classes de base (soit wxThread, wxSocket, wxFileInputStream, wxUI* ,) cette interface peut devenir problématique.

D'un autre coté cette interface pourrait nous permettre de produire un "core" plus indépendant de wxWidgets, mais il n'est pas dans l'intention du logiciel d'etre portable de cette manière et seulement le système d'events serait portable, ce qui serait insuffisant.

Donc, après mure réflextion j'ai implémenté le patron de conception des listeners à la sauce wxWidgets et optimisé avec le template EvtHandlerContainer comme défini dans la section Solutions.

4. Détails et notes

4.1 Conventions de codage

1) Chaque classe a son fichier .h nommé par le nom de la classe, similairement à Java. Sauf exception lorsque de très petites classes sont définies par exemple pour un table de hashage wxWidgets.

2) Les variables de classe privées commence avec un "_" example : _tracker

3) Les variables de classe protégées commence avec un "p_" example : p_tracker

4) Les variables de classe publiques sont simplement leur nom ex : tracker

5) Les .h ne contienent que le minimum pour la défénition de la classe.

4.2 Environnement wxWidgets

Kayak à été testé sur wxWidgets wxGTK 2.4.2 avec comme configuration :

./configure --enable-gui --enable-gtk2 --enable-xrc --with-libpng --with-libjpeg --with-libxpm --enable-debug --enable-debug_flag --enable-debug_info --enable-debug_gdb --enable-debug_cntxt --enable-sockets --enable-threads

4.3 Code source

Le code source de la version présenté par ce rapport est disponible par CVS en utilisant la commande :

cvs -d :pserver:anonymous@kayaksoft.com:/var/src checkout -r rel-0-1-1 kayak