Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Michał Ślizak
Nr albumu: 201104

An implementation of a distributed flow control library for peer-to-peer systems

**Praca magisterska
na kierunku INFORMATYKA**

Praca wykonana pod kierunkiem
dr Janiny Mincer-Daszkiewicz
Instytut Informatyki

Październik 2006

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora pracy

Abstract

Distributed Flow Control aims at ensuring optimal usage of resources in a P2P system. This is crucial in high-performance applications, where users expect responsiveness and efficiency of the system. Currently available systems use simple forms of control flow, usually based on static user-supplied limits. This paper presents a version of a distributed flow control which uses only locally available performance information, based on the assumption that the number of requests destined for each host is approximately equal. It doesn't require any data exchange with other peers in the network, and therefore doesn't add to the load which it is trying to control.

Słowa kluczowe

flow control, distributed flow control, load handling, hotspot, denial-of-service mitigation

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

C. Computer Systems Organization
C.2. Computer-communication Networks
C.2.4. Distributed systems

Tytuł pracy w języku angielskim

An implementation of a distributed flow control library for peer-to-peer systems

Contents

List of Figures

Chapter 1

Introduction

Peer-to-peer systems are currently on the rise, mostly due to low cost of upkeep, high resilience, and easy expansion. While P2P systems gained a somewhat notorious reputation in recent years due to illegal file sharing, they are now being introduced in many other, legal domains. These include (but are not limited to): high-performance computing, mass data storage and internet telephony. Each of these applications requires some sort of control over the amount of time each request is processed by a system node.

Every time-constrained peer-to-peer system design must deal with the problem of resource exhaustion. When a peer receives requests faster than it can handle them, the entire system can be affected. The mechanism which comes to mind when thinking about this issue is *Distributed Flow Control*. It provides a P2P system the means to determine if a specified request should be accepted or rejected. In order to work properly, distributed flow control should take into account performance of the individual node, as well as that of the entire system. If the flow control mechanism is to work without impending too much performance slowdown, or introducing the assumption that some nodes are error-free, the system must not use the centralized resource-manager approach.

By their very nature, nodes in a P2P system are error-prone, and can disappear randomly. This also has to be considered when designing such a system. It practically eliminates the possibility of using a token-based solution, for fear of a failing node never returning a token, unless some mechanism of token recovery is introduced.

This paper presents a distributed flow control library which uses only locally available information. Unlike other solutions, it doesn't require data exchange with other peers in the network. This has many advantages: firstly, it avoids introducing a single point-of-failure, and secondly it doesn't add to the load which it is trying to control. Of course any distributed flow control mechanism can only provide a best-effort approach, since at any time any number of nodes can fail, thus timeouting requests. The solution presented here aims at reducing the number of timeouted requests without impending too much slowdown on the system itself.

Key issues each P2P control flow system design addresses are: scalability, extensibility, overhead, portability and flexibility. It is important for the system to detect both local and remote slowdowns (for example caused by other processes starting on the server, or a remote server crash). We will present our solution in all these aspects, and compare it with other algorithms.

This work focuses on providing programmers with a well-tested, useful library they can incorporate into their P2P solutions. As each of them is usage-specific, it may have different characteristics when it comes to dealing with load. Therefore, the library is created with configurability in mind.

This paper is divided into several parts as follows:

1. Description of several P2P systems and their flow control mechanisms are presented for reference and comparison purposes in chapter 2.
2. The basics of our solution, definitions used throughout the rest of the document, assumptions made are given in chapter 3.
3. System design is described in chapter 4.
4. Implementation is briefly described in chapter 5.
5. Description of a framework we used to test our solution, and test results are given in chapter 6.
6. Conclusions and ideas for future work are outlined in chapter 7.

Chapter 2

Related work

The issue of control flow is quite commonly raised in communication. Specifically, whenever we have two communicating systems, we need to make sure that neither of them will overflow the other with requests. There are several communication protocols which utilize control flow and are, therefore, interesting from our point of view.

For this paper, we decided to describe the following protocols:

- TCP — due to its widespread usage, and quite simple, yet effective, control flow mechanism,
- eDonkey protocol — it is one of the most widespread P2P systems used for sharing huge amounts of data,
- BitTorrent — due to its widespread use in the P2P community, and focus on efficiently spreading data instead of high download speeds.

There are many other solutions to ditributed flow control. The Tor project ([Din04]), for example, uses data throttling. There were also attempts to predict web flash crowds using statistical information ([Bar05]). We will not focus on these results, however, since they are not generic enough to implement as an all-purpose library.

2.1. TCP window mechanism

The TCP protocol has been designed over 20 years ago, and since then, had been a subject of study for many scientists. It has been described in [Koz05].

TCP is a stream-oriented protocol, and sends data in *segments*. Data is split into them, and a complicated confirmation mechanism, which is beyond the scope of this paper, is used to make sure every byte has been delivered. Suffice to say, data is retransmitted, if it hasn't been acknowledged within a certain time.

Data in the stream is split into 4 categories (as shown in figure 2.1):

- category 1: sent, acknowledged,
- category 2: sent, not acknowledged,
- category 3: bytes to send, for which the recipient is ready,
- category 4: bytes to send, for which the recipient is not ready.

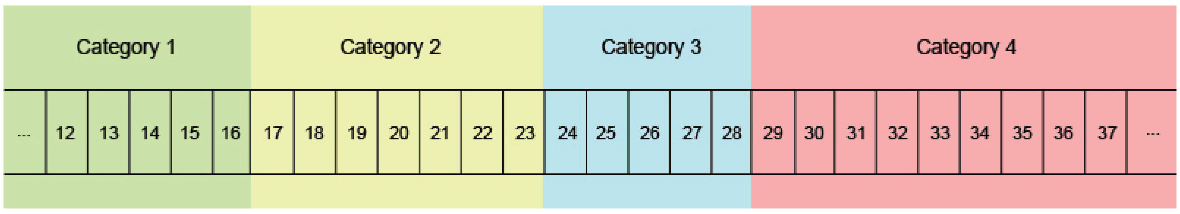


Figure 2.1: Here we see the data stream split into several categories

TCP incorporates a simple flow control mechanism, which could be viewed as an alternative to our solution. This mechanism is called ”the sliding window”, because it is based on the concept that neither the sender, nor the receiver can send data from outside a given range.

This mechanism not only limits data flow rate, but also ensures that lost packets are retransmitted.

2.1.1. The problem

The main problem all ”send-and-forget” transmission protocols face, is the fact that data can be lost (as shown on diagram 2.2). While in some applications (streaming media, audio, internet telephony) this might not have a huge impact on the overall quality of the service, in general we would like to have a protocol which is reliable, and doesn’t loose data.

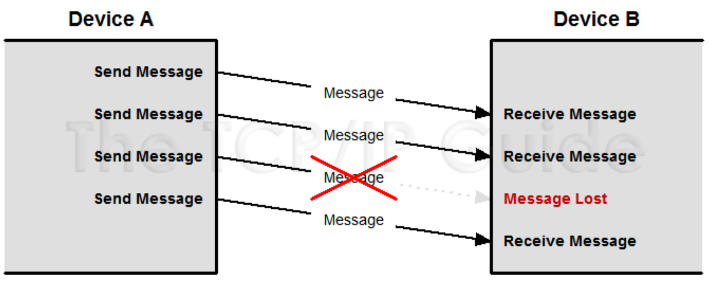


Figure 2.2: In this example we see the problem of losing data when confirmations are not required (source: [Koz05])

The second issue is that the sender does not know (and, theoreticaly, it is impossible for him to know exactly) how fast the receiver can parse incoming data. This is caused by communication latency, and the only way to solve this problem is some sort of a binding ’contract’ between two sides of communication.

2.1.2. The sliding window mechanism

The sliding window is the solution to both of these problems. It acts as a buffer between a sender and a receiver. The receiver notifies the sender of its ”send window size” (which is simply a number of octets) via special *window announcements*. The sender then knows it is allowed to send no more than a given amount of octets before getting confirmation. This is depicted in figure 2.3.

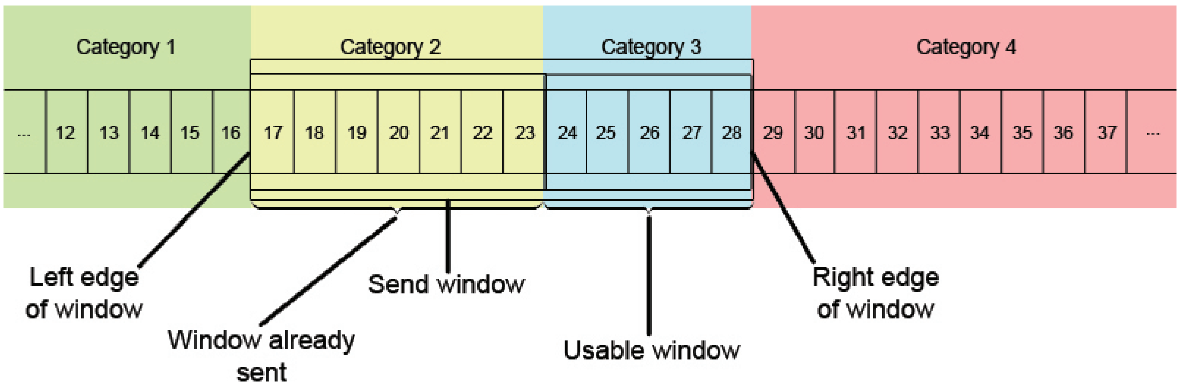


Figure 2.3: An illustration of the sliding window concept

A sender always can send data from its usable window. When the usable window is zero-sized, the sender must wait for confirmation of previously sent data. This confirmation moves the left side of the window, making the usable window grow. Figure 2.4 shows window movement.

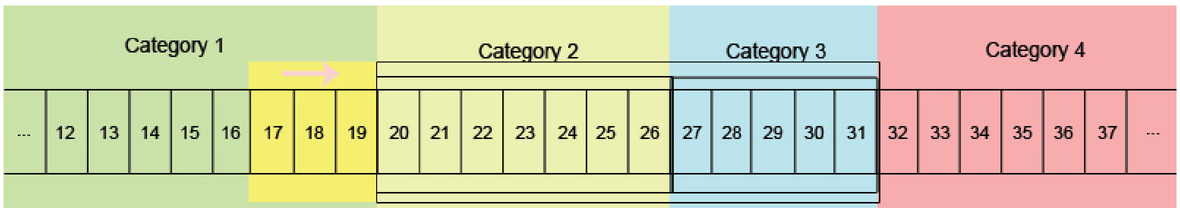


Figure 2.4: Movement of the sliding window during data transfer

2.1.3. Sliding window and flow control

The sliding window mechanism can be used for flow control. Window size changes can be sent along with confirmations to speed up or slow down transmission, as shown on the diagram 2.5.

Assume that we have a sender, receiver pair. The receiver is currently heavily loaded. Here's the typical sender choking scenario:

1. The sender sends the first 140 bytes to the receiver.
2. The sender moves the left edge of the usable window 140 bytes to the right.
3. The receiver accepts data, and stores it in its incoming buffer. The left edge of the buffer's receive window moves 140 bytes to the right. The right edge moves 40 bytes to the right (meaning 40 bytes are passed to the application layer).
4. The receiver sends an acknowledgement message to the sender, and propagates the new size of the window – 260 bytes.
5. The sender receives acknowledgement, and sets the window size accordingly (moving left edge by 140 bytes, and the right by only 40 bytes).

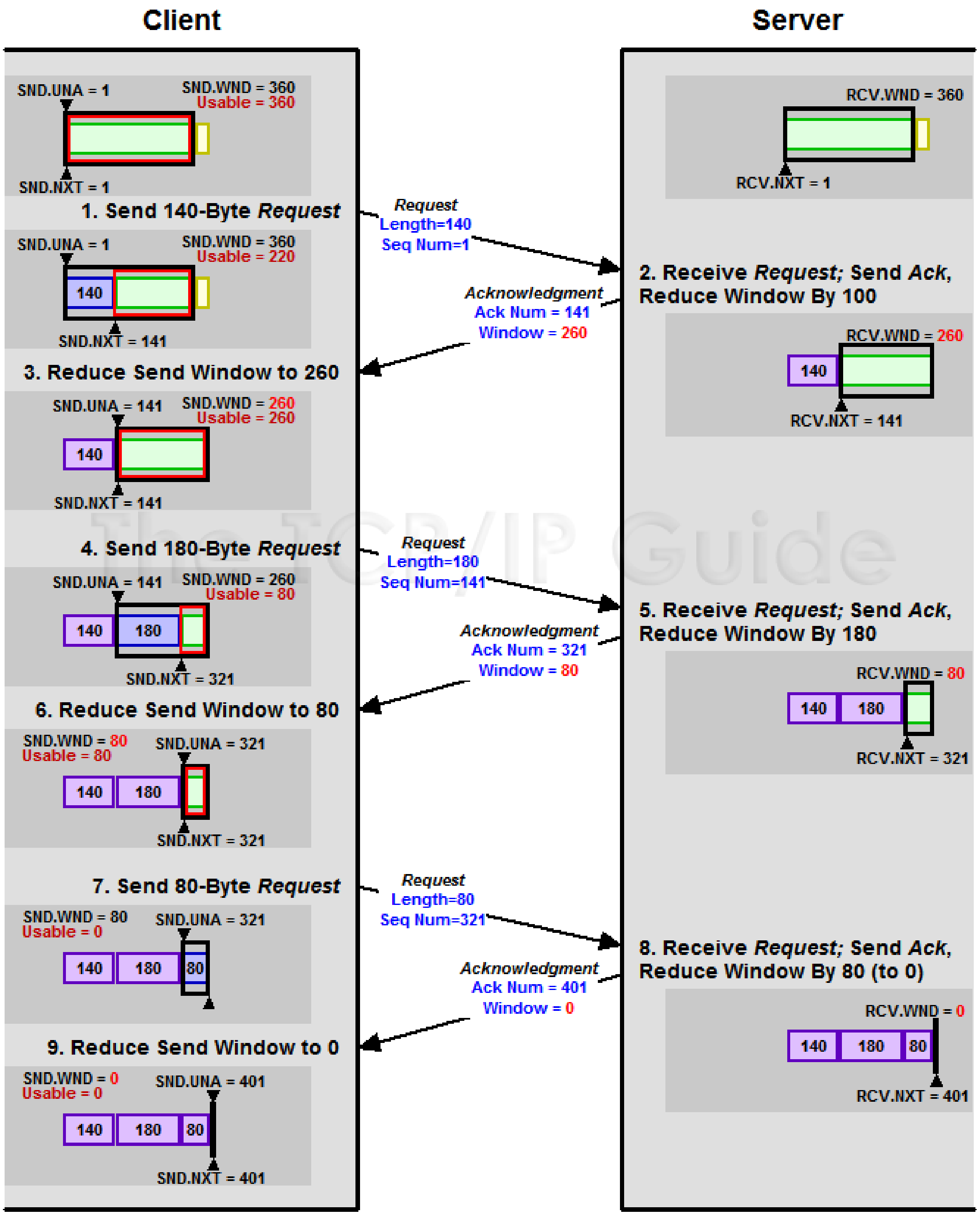


Figure 2.5: Resizing the TCP window during overloading (source: [Koz05])

6. With the new reduced window, the sender sends 180 bytes.
7. The sender moves the left edge of the usable window.
8. The receiver accepts 180 bytes, stores it, and resizes the receive window again, this time moving the left edge by 180 bytes.
9. The receiver sends an acknowledgement with the new window size (80 bytes).
10. The sender receives acknowledgement, and resizes his window, moving the left edge 180 bytes forward.
11. The sender sends last 80 bytes.
12. The sender moves the usable window 80 bytes to the right, leaving the window zero-sized.
13. The receiver accepts 80 bytes, stores them, resizes the window, moving the left edge 80 bytes to the right. The buffer is now full.
14. The receiver sends an acknowledgement with the window size 0. This tells the sender to pause.

At this point, the sender must wait for a new window size to be sent. Once this happens, the right edge of the window will move forward, and transmission can continue.

2.1.4. Problems

While the idea behing sliding windows may be simple, implementing flow control can be very tricky. One of the issues to consider is shrinking vs. reducing the window in case of overloading.

Let's assume that the server is very overloaded, and is trying to cut down the speed at which it is receiving data as much as possible. All it can do is to leave the right edge of the window where it was whenever data comes in. Moving it *to the left* could be problematic, because that might lead to a situation where the client has already sent some data, and suddenly that portion of data falls out of the usable window. This operation (moving the right edge of the window to the left) is called *shrinking the window*. Resizing the window when the right edge doesn't move or moves forward is called *reducing*.

TCP explicitly disallows window shrinking, only reducing is allowed.

So what can a server do to prevent window shrinking? It has to be very careful about assigning window sizes, since it can be seen as a binding contract. When overloading does happen, it has to be patient and reduce the window as data comes in.

2.2. eDonkey network

The eDonkey network ([Kul05], [Eng05], [Wik06]) is a popular platform for peer-to-peer file sharing. Its widespread use and the huge amount of traffic it generates (current research [Cac05] shows that BitTorrent and eDonkey are the two leading peer-to-peer networks with regard to data volume) makes is a very interesting protocol from our perspective.

There are many programs which implement the eDonkey protocol (originally developed by the MetaMachine company). One of the most popular is eMule, which I have chosen to describe as an example of eDonkey flow control. eMule builds upon the original eDonkey protocol, adding several new features.

2.2.1. Overview

The eMule network is populated with several hundred eMule servers, and millions of eMule clients. Each client connects to a server in order to perform searches, and this connection is open as long as the client is in the system. It is also used by the client to report the resources (files) it wishes to share with other clients. Each server keeps a registry of files from all of its connected clients. Servers do not communicate with other servers.

Every client receives from its server a list of other clients who have the files it is searching for. Each of these clients can now be contacted for different parts of the file.

2.2.2. Data interchange

Every client maintains its own upload queue for each file it shares. Downloading clients register in the queue, joining it from the bottom and gradually moving upwards, until they reach the top of the queue and start downloading. A client can register to several queues, asking for different parts of the same file. A client can upload parts of a file, even if it doesn't have the complete file yet.

The place in the waiting queue is determined by a factor called *ranking*. Downloading clients can be preempted by a waiting client with higher ranking. During the first 15 minuts of downloading, the downloading client's ranking is raised to prevent thrashing.

The download is initiated by the uploading client, which opens a connection to the downloader and uses a push mechanism to send him his requested file parts.

A credit system is used to encourage uploads and reciprocation. The more files client A uploads to client B, the more credits it will have on client B, and will move to the top of B's upload queue faster. eMule uses 16-byte user IDs to identify users between sessions, so even if a connection to B is broken, the next time client A connects to B it will use the credits available previously. RSA public/private key pairs are used to avoid impersonation.

The credit system

When client A uploads data to client B, the downloading client updates the credit related to the uploader according to the amount of data received. This means that the credit system is not global, it will be taken into account only when downloading specifically from client B.

Credit is calculated as a minimum of:

- $uploaded\_total * 2/downloaded\_total$, with a maximum value of 10. If $downloaded\_total$ is zero, this expression evaluates to 10.
- $\sqrt{uploaded\_total + 2})$, when $uploaded\_total$ is less than 1 MB, this expression evaluates to 1.

Both upload and download amounts are calculated in megabyte units. Credit is always in the $[1, 10]$ range.

Upload queue management

Each client registered in an upload queue is assigned a priority, based on its rating and the time spent in the queue. The queue is sorted by *score*, which is calculated using the following formula:

$score := (rating * seconds\_in\_the\_queue)/100$ or ∞ for clients defined as friends.

The rating's initial value of 100 can be modified by the client's credits (from 1 to 10), and by the uploaded file priority (0.2 – 1.8) which is set by the uploading client. A client which has just started downloading has a boosted rating of 200 for the first 15 minutes of downloading.

The download will stop if:

- the uploading client was terminated,
- the downloading client got all the parts of the file,
- the downloading client was preempted by another downloading client with a higher score.

An uploading client will serve several (typically 4) downloading clients with currently highest score.

File parts

Each file is divided into 9.28 megabyte *parts* and each part is further divided to 180 KB *blocks*. The downloading client decides in what order the file parts will be downloaded, by sending requests for different blocks. A client can download one part from each source at a time, and all blocks that are requested from the same source reside in the same part. The following rules are used when choosing parts (in this order):

1. frequency of the parts — rare parts are downloaded first to create new sources,
2. parts used for preview — first and last part,
3. request state — ask all sources for blocks from different parts, spread the requests between sources,
4. completion — parts for which some blocks have already been received should be completed before beginning another part.

2.3. BitTorrent

BitTorrent (referred to as 'BT' later in this document) is a P2P system which is focused on efficient distribution of large contents. The specification can be found in [Bit06] and [Eng05], while [Iza04], [Leg05A], [Leg05B], [Qiu04] and [She04] provide some insight into theoretical performance of this protocol. It is based on the idea of a 'horde' or 'swarm'. A .torrent file contains the address of a *tracker*, a server which counts the number of downloads and uploads, and can be queried for information when searching for new peers to share data with. The idea is for the tracker to host meta-information about the download, and for the downloaders to share pieces of the file between themselves. This of course assumes there initially exists at least one *seeder*, a peer which shares the entire file. The connection process is shown on diagram 2.6

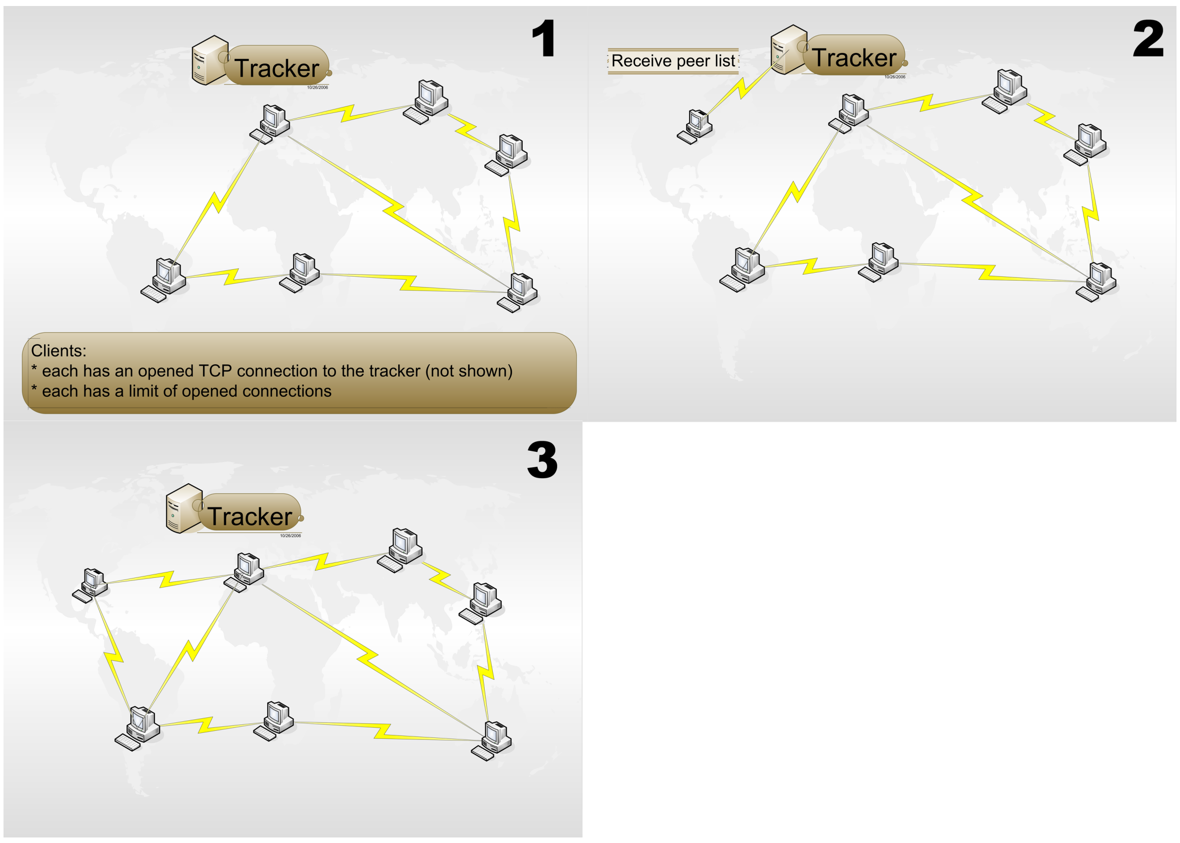

1
Tracker
Clients:
* each has an opened TCP connection to the tracker (not shown)
* each has a limit of opened connections
2
Receive peer list
Tracker
3
Tracker

Figure 2.6: Connecting a new client to the network

2.3.1. The problem

One of the main problems P2P systems face is 'free-riding' [Kri04], which means downloading data without providing any youself. Some networks limit the amount of resources available to a client ([Tsu03A] and [Tsu03B]), while others discourage this kind of behaviour by using algorithms which reward uploaders ([Bur03]). BitTorrent uses a 'tit-for-tat' flow control mechanism ([Coh03], [Tam03], [Tho05]), which is used on top of the TCP flow control to regulate traffic on multiple TCP connections.

When limited to one connection only, the TCP control flow performs quite well. Unfortunately most P2P applications use multiple connections, sometimes even opening many connections to one remote client. This is when TCP flow control starts to fail, because it treats all these connections separately. This opens up a potential for abuse, since a 'free-loader' could open multiple connections for different parts of the same file (most P2P applications allow this), and each of them would get the same share of the server's bandwidth. The outcome is that the free-loader gets more bandwidth than other, 'honest' clients.

2.3.2. Solution

A simple solution is to somehow slow down clients which do not upload any data. This slowdown must be significant enough so that it's not worth opening new connections.

A good algorithm should:

- limit the number of simultaneous connections for good TCP performance,
- avoid slowing down and speeding up clients quickly (also known as 'fibrillation'),
- reciprocate to peers who let it download,
- try out unused connections to see if they could be better than currently used ones.

BitTorrent uses a 'choking' mechanism to slow down leechers. All connections in BT are two-way (meaning each client is also a server). Every connection has a set of flags associated with each endpoint. An example state with regard to these flags is shown on diagram 2.7.

- choked — Set by the sender, describes a peer to whom the server refuses to send pieces. This can happen in 2 situations:
  - the receiver is a *seed* (meaning it already has the complete file),
  - the sender reached its maximum upload capacity (bandwidth or a number of open upload connections).
- interested — describes the downloader, who would like to obtain a piece of the file the client has.

The protocol ensures that in order to achieve good download speed, one must also upload at a considerable rate. This is caused by using the choked flag to mark all but the "best" 4 uploaders. This way, fast uploaders are rewarded with high download speeds. This rating is refreshed every round (which usually lasts 10 seconds). Every 3 rounds, the client will perform an "optimistic unchoke", choking the worst of the currenly "best" clients, and unchoking one other. This might help find better partners, since the unchoked peer might offer good download speeds now that it is receiving data from us. Seeders implement the same strategy, but based on download, not upload speeds.

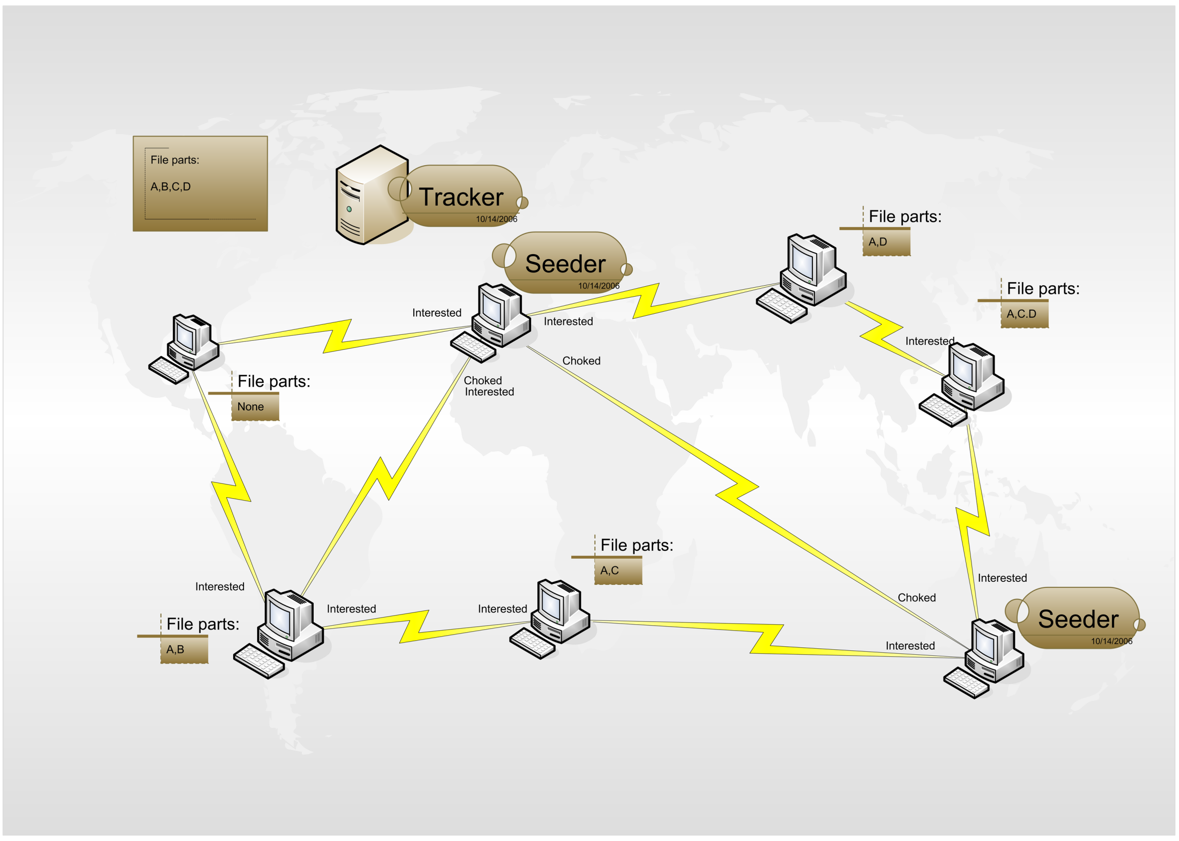


Figure 2.7: Flags used in the BitTorrent network

Another important aspect is the file chunks we ask for. Peers essentially try to maximize the entropy of each chunk, picking the one which is least available in their view of the network. This makes sure that all chunks should be equally available in the torrent (at least locally). There exists an exception to this rule, however. New clients do not have any data, so the only way for them to get any is through optimistic unchoke. It follows that new clients should not pick rare chunks, since they are available on few peers, and waiting for them to unchoke us might take too long. Instead, fresh clients use a 'random first' policy, and after receiving some data, switch to 'rarest first' policy.

Experiments [Iza04] shown that BitTorrent clients are 'altruistic', meaning they actively send data to other nodes, both as leechers and seeds. Free-riders will be served with very low-priority (through optimistic unchoke only). Additionally, BitTorrent deals well with the problem of 'flash-mobs', by quickly creating a large number of seeds.

While it has been shown ([Hal05], [Qiu04], [Tho05]) that the BitTorrent tit-for-tat implementation is optimal only if one assumes certain parameters, which are unlikely to be found in the real world (for example equal maximum upload speed for all nodes), it can be seen as a step in the right direction with regard to flow control.

2.4. Summary of other solutions

The three solutions presented in this chapter were the models I studied when preparing my own control flow implementation. As will be evident in next chapters, all algorithms mentioned above had a great impact on my design.

Chapter 3

Flow control library - requirements

The control flow library which I am about to introduce is a generic solution which can easily be added to any peer-to-peer system to control resource usage.

Before continuing, let's take a look on the vocabulary we are going to use. This section may prove very useful later on.

3.1. Definitions

I have added the definitions list here, and not at the beginning of the documents, because documents regarding previously mentioned solutions use these definitions in slightly different context. In order to avoid confusion, let's assume the following definitions will be used in the next chapters:

Definition 3.1.1. A *peer-to-peer network* is a communications environment that allows all computers in the network to share their resources.

Definition 3.1.2. A single computer participating in a peer-to-peer network is called a *node.*

Definition 3.1.3. A *request* is an atomic unit of work to be performed by a peer-to-peer network. Depending on the desired function of the system, it can contain a description of some calculations to be performed, a request to read/write some amount of data, or perform any other function.

Definition 3.1.4. *Request confirmation* is done when the request in question has been processed (either successfully or unsuccessfully).

Definition 3.1.5. A request which has been started, but hasn't yet finished is called an *outstanding request.*

Definition 3.1.6. *Request latency* is the amount of time required to complete processing of a particular request, meaning the amount of real-time between the moment the request was accepted by one of the nodes, and its completion.

Definition 3.1.7. A computer or program that can submit requests to the network is called a *client.* It can, but doesn't have to be, one of the nodes in the network.

Definition 3.1.8. A *slot* is a hipothetical unit of resources, the posession of which allows a client to submit a request.

Definition 3.1.9. The amount of slots available on a particular node is called a *total slot pool.*

Definition 3.1.10. A *slot share* is a floating-point value between 0 and 1 assigned to each client, corresponding to the fraction of slots this client has been assigned from the total slot pool.

Definition 3.1.11. Time on each node is divided into units called *rounds.*

3.2. Requirements and assumptions

Every project should start by collecting requirements. After doing a bit of research on the subject, I found that a common view on distributed flow control is hard to establish. I have, however, managed to extract several basic ideas which seem to outline the concept, and present them here. I also have to point out the assumptions, both implicit and explicit, of our design.

3.2.1. Requirements

The basic requirements for a distributed flow control library are as follows:

Performance control

Probably the most important aspect of flow control is measurement and control of current (and maximum) system performance. This information allows us to determine if more load can be allowed, or if requests should be denied. Possible properties that we might wish to measure are: latency of requests (responsiveness), overall throughput (speed), resource consumption (memory, for example). Of course, it is important to balance these properties.

The purpose of this library is to make the average latency of the entire system to be approximately equal to a user-specified value.

Low overhead

The mechanism must not consume too many resources. While some applications may allow the introduction of a centralized flow control server, it might not be a good idea to use this approach blindly. A centralized server would, of course, be able to select an optimal strategy, thanks to its complete knowledge about the system, but it would also be a single point-of-failure, endangering the stability of the system. The central server could also become a bottleneck as the network grows.

Fairness

This is an obvious concept. When many clients use a server concurrently, the flow control mechanism must assure that client starvation cannot occur. The important concept here is the type of "fairness" we are trying to achieve. There are many possibilities:

- Let's assume we have many clients connected to the server, and K slots in the total slot pool. Client i is capable of submitting s_i requests per second. Then, for client i the amount of requests it is allowed to submit should be approximately $\frac{s_i}{\sum_{j=1}^{n} s_j} * K$
- Every client gets the same amount of slots regardless of their usage.

Portability

Our solution will be used in many different applications and in various computing environments. Therefore the library must be flexible enough to allow the user to include estimations (for example different types of slot reduction functions, averaging functions, etc.) that would be applicable to the problem at hand.

Stability

A failure of one of the P2P nodes should not stop or crash the system.

3.2.2. Assumptions

Relationship between number of started requests and latency

Our model assumes that there is a relationship between the number of requests which are started on each node, and the latency of their execution. This relationship has to be known to the library's user.

Measurable latency

Our solution requires request latencies to be measured in order to recalculate the size of the total slot pool.

Time quantization

Time is quantized into rounds. At the beginning of each round, we will calculate the size of the total slot pool, and assign slots to all current clients. During the round, clients will be able to use the assigned slots to submit requests.

Slot reservation

The server has a specified number of slots to use. This number corresponds to its performance, and will be automatically adjusted by the library. Slots will be distributed between multiple clients.

Before a request is accepted, a slot must be reserved for it. Slots are used to control the number of outstanding requests.

Nodes can fail

Nodes in the system can fail without warning.

Requests are either replied to, or timeouted

When a request finishes, the slot used by it is returned to the server. The same happens when a request times out.

Chapter 4

Flow control library - design

Our design is strongly influenced by the TCP window sliding mechanism. Many ideas were taken and adapted from the TCP design notes.

4.1. General overview

Our design is based on the idea that each request should reserve a slot in the control flow reservation mechanism. Each server will have its own reservation mechanism, and the total number of slots will be recalculated at the beginning of each round based on the number and the latency in the previous round. This total slot pool will then be distributed between clients according to their needs.

4.1.1. Environment

The control flow mechanism is to be incorporated in two critical sections of a peer-to-peer network:

- within the entry point to the network itself (on a node participating in the network, called a *backend node*),
- on a *proxy node*, whose functionality is to multiplex requests to many backend nodes.

This is illustrated on diagram 4.1.

4.2. Definitions

Definition 4.2.1. A request which is pending for more than a configurable amount of time is called a *high latency request.*

Definition 4.2.2. An *active client* is a client which has used at least a certain configurable percentage of its slots.

4.3. Slot management

As mentioned previously, slots are distibuted amongst clients. Some slots may be left unassigned, and used for new clients. The details of the algorithm are described in subsection 4.3.1.

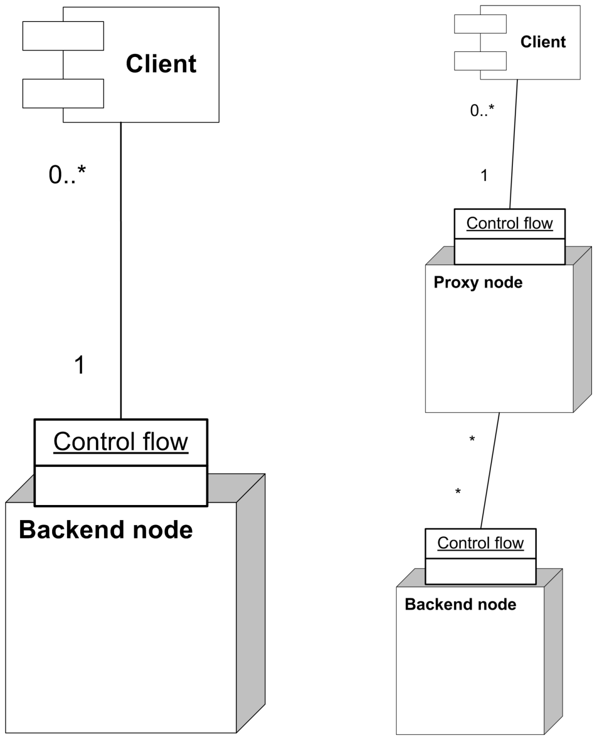


Figure 4.1: Control flow environment

4.3.1. Total slot pool size change

The mechanism assumes that several data sources may exist.

Four implementations were provided:

- *Constant strategy* was implemented at the beginning for testing purposes,
- *Timeout-based strategy* was implemented as a first step towards load balancing,
- *Reported-pool strategy* was implemented for use on proxy,
- *Latency-based strategy* was implemented as a refinement of the timeout-based strategy.

Each environmental context uses a different strategy:

- On backend, we use a *timeout-based strategy*,
- On proxy, we use a *reported-pool strategy.*

At the beginning of each round, the object is requested for an estimation of the current total slot count.

Constant strategy

This strategy always returns a constant, predefined amount of slots.

Timeout-based strategy

This strategy uses pending requests to assess node performance. It is used on backend nodes.

Check if there was a high latency request in the previous round:

- Yes: decrease the total number of slots by a configurable percentage.
- No:
  - If there was an active client in the previous round, increase the total number of slots by a configurable percentage.
  - If no active client exists, the total slot count re4mains constant.

The goal of this strategy was to keep request latencies at a specified level. The timeout at which a pending request is considered a high latency request defines the desired latency of the system. On the other hand, slots should only be created when they are needed, because we don't want to over-estimate our resources. This explains the reasoning behind the 'active client' check when adding new slots.

This strategy has the following characteristics:

- for an overloaded system — decreases the total slot count to a level at which latency is close to expected,
- for an underloaded system — increases the total slot count to a level at which latency is shorter than expected. The definition of an 'active client' influences the amount of extra unused slots this strategy will create. These empty slots could act as a buffer when one of our clients suddently starts sending more requests, or a new client joins the system.

Reported-pool strategy

This strategy is used by proxy, and communicates with an external object to obtain information about the number of slots in the system. This external object collects information about slot count from all backend slots proxy is connected to, and runs algorithms to estimate performance of the entire network. This is an example of an external data source.

The following algorithm was used for performance estimation of the entire network:

- assume P_i is the number of slots on node i,
- assume $R_i \in [0, 1]$ defines the fraction of total load that node i receives,
- assume k is the index of a node which has the smallest amount of slots available $(\forall_{i \in \{1,2,\dots,n\}} P_i \leq P_k)$.

The total slot count estimation for the entire system is equal to $\frac{P_k}{R_k}$.

Latency-based strategy

This is a generalization of the timeout-based strategy. It uses a user-defined *"slot reduction function"* to calculate the factor by which the size of the total slot pool must be multiplied. A slot reduction function takes a single latency value as a parameter, and returns a floating-point value. Such functions should decrease the number of slots when latency is too large, and increase it when it's too small.

Several functions were implemented and tested:

- LinearSlotReductionFunction — This slot reduction function is parametrized by 4 points – low, desired, high and critical latency. Assume $A = \frac{1}{desired - critical}$ and $B = \frac{critical}{critical - desired}$. This function has the following value:

  1. $\forall_{x<low} f(x) = A * x + B$,
  2. $\forall_{low \leq x < high} f(x) = 1$,
  3. $\forall_{high \leq x < critical} f(x) = A * x + B$,
  4. $\forall_{critical \leq x} f(x) = 0$.

  This function was the first one to be tested, because I suspected that the relationship between latency and the number of requests will be linear (meaning that doubling the slot count will increase the latency by a factor of 2). The low and high latency values were added later as a refinement in order to avoid constant changing of the slot count.

  The problem with this function is that such changes are still possible. A value slightly less than low latency may increase the number of slot so much that latency will raise above high latency. The next function doesn't have this problem.

- QuadraticSplineSlotReductionFunction — The quadratic spline function consists of two quadratic ($f(x) = A * x^2 + B * x + C$) functions. This strategy is parametrized by the following values:

  - desired latency,
  - critical latency.

  This function has the following properties:

  1. $f(desired) = 1$,
  2. $f(critical) = 0$,
  3. $\frac{\partial f}{\partial x}(desired) = 0$,
  4. $\forall_{y \neq desired} \frac{\partial f}{\partial x}(y) < 0$,
  5. $\forall_{x<desired} f(x) > 1$,
  6. $\forall_{x>desired} f(x) < 1$.

Research is currently performed to find better slot reduction functions, and their optimal parameters.

4.4. Assigning slots to clients

This mechanism is the same on both backend and proxy. We calculate a vector of slot shares $< F_1, F_2, ..., F_n >$, where:

1. $\forall_{i \in \{1,2,...,n\}} F_i \in [0, 1]$,
2. $\sum_{i=1}^{n} F_i \leq 1$.

Currently, only one implementation is available (see 4.4.1), investigating different splitting strategies is in the scope of further research.

4.4.1. History-based splitting strategy

This strategy tries to predict client needs by observing the amount of submit attempts in the previous round.

It uses 4 configurable values:

1. penalty-trigger level $DecCut \in [0, 1]$ — the level of slot usage which, if not met, causes a penalty to be inflicted for the duration of the next round,
2. bonus-trigger level $IncCut \in [0, 1]$ — the level of slot usage which, if exceeded, causes some bonus slots to be assigned if available.
3. penalty buffer value $buffer \in [0, 1]$ — added to penalized clients in order to leave them some room for improving their speed,
4. maximum bonus value $max\_bonus \in [0, 1]$ — a limit on the bonus factor fast clients can receive.

This strategy works in several phases:

1. Start with equal share for all clients.
2. Remove slots from clients which used less than $DecCut$ fraction of their assigned slots.
    - Assume that client C used U_{old} fraction of it's assigned slots in the previous round. We need to calculate the fraction A_{new} of all slots it can be assigned now. There are N clients.
    - $U_{old} < DecCut$.
    - $A_{new} := min(\frac{U_{old}+buffer}{N}, \frac{1}{N})$.
3. Add slots to clients which used at least $IncCut$ fraction of their slots. Sequentially:
    - Assume that at this moment K fraction of the total slot pool is unassigned and that client C used U_{old} fraction of it's assigned slots in the previous round. We need to calculate the fraction A_{new} of all slots it can be assigned now. There are N clients.
    - $U_{old} \geq IncCut$
    - $A_{new} := \frac{1}{N} + \frac{min(K, max\_bonus)}{N}$
    - $K := K - \frac{min(K, max\_bonus)}{N}$

Assumptions defined in 4.4 are fulfilled:

- assume there are N clients in total.
- assume P of them were penalized, freeing up F fraction of slots ($F \in [0, 1]$).
- assume B of them were given bonus slots.

The algorithm assures that:

- each penalized client can have at most a $\frac{1}{N}$ fraction of all slots assigned,

- each client with a bonus can have no more than a $\frac{1+min(F,max\_bonus)}{N}$ fraction of all slots assigned ($F < 1$, $N = 1 \Rightarrow F = 0$),
- all other clients have $\frac{1}{N}$ of all slots assigned,
- the total bonus assigned to all clients is not greater than F (because when adding a bonus to a client, all previously assigned bonuses have already decreased the available unassigned fraction K).

4.5. New clients

When a new client connects, its slot share will be calculated automatically based on the currently unused part of the total slot pool (with a maximum initial fraction configurable during initialization). The assigned amount will then be modified in next rounds, so its initial value shouldn't have too much influence on the system as a whole.

4.6. Client operation

The design is based on the assumption that the client will resubmit a request if it is rejected by the flow control mechanism. This resubmitted request can be directed to a different node (a client may be connected to many servers at once, and try to find the first one which accepts a request), or on the node where it was rejected previously (due to some requests finishing, or new slots being created). This is illustrated on diagram 4.2.

4.7. Request lifecycle

Diagram 4.3 shows the request lifecycle as it travels through the external system, reaches the control flow library, and continues its way through the system.

1. client tries to obtain a slot from the pool,
2. pool returns TRUE if request can be accepted, FALSE otherwise,
3. if pool returned TRUE, then:
   (a) client starts the request when it is actually passed for processing,
   (b) when reply is returned, finish is called,
4. if pool returned FALSE, then client is responsible for the request.

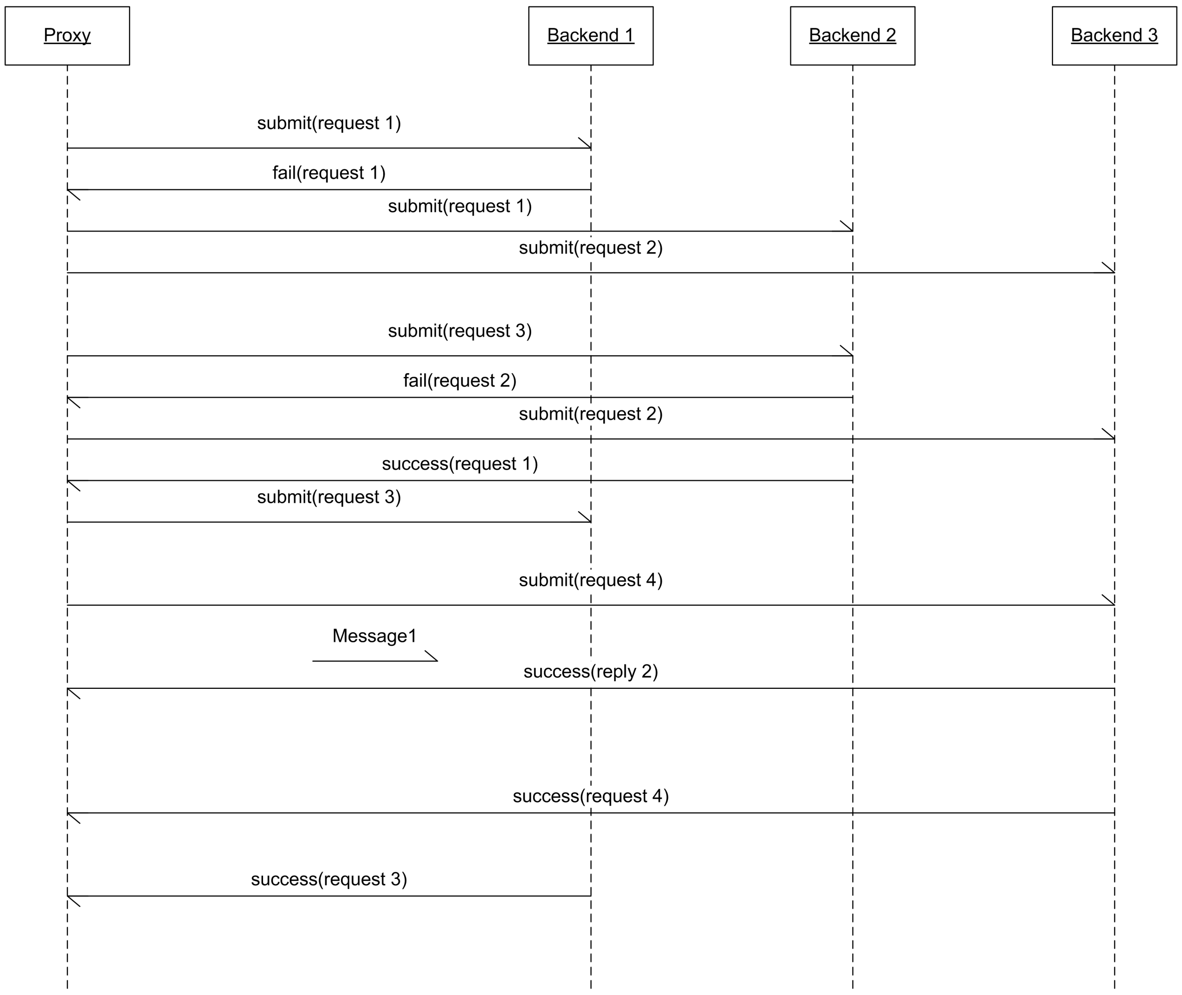


Figure 4.2: Control flow resubmission

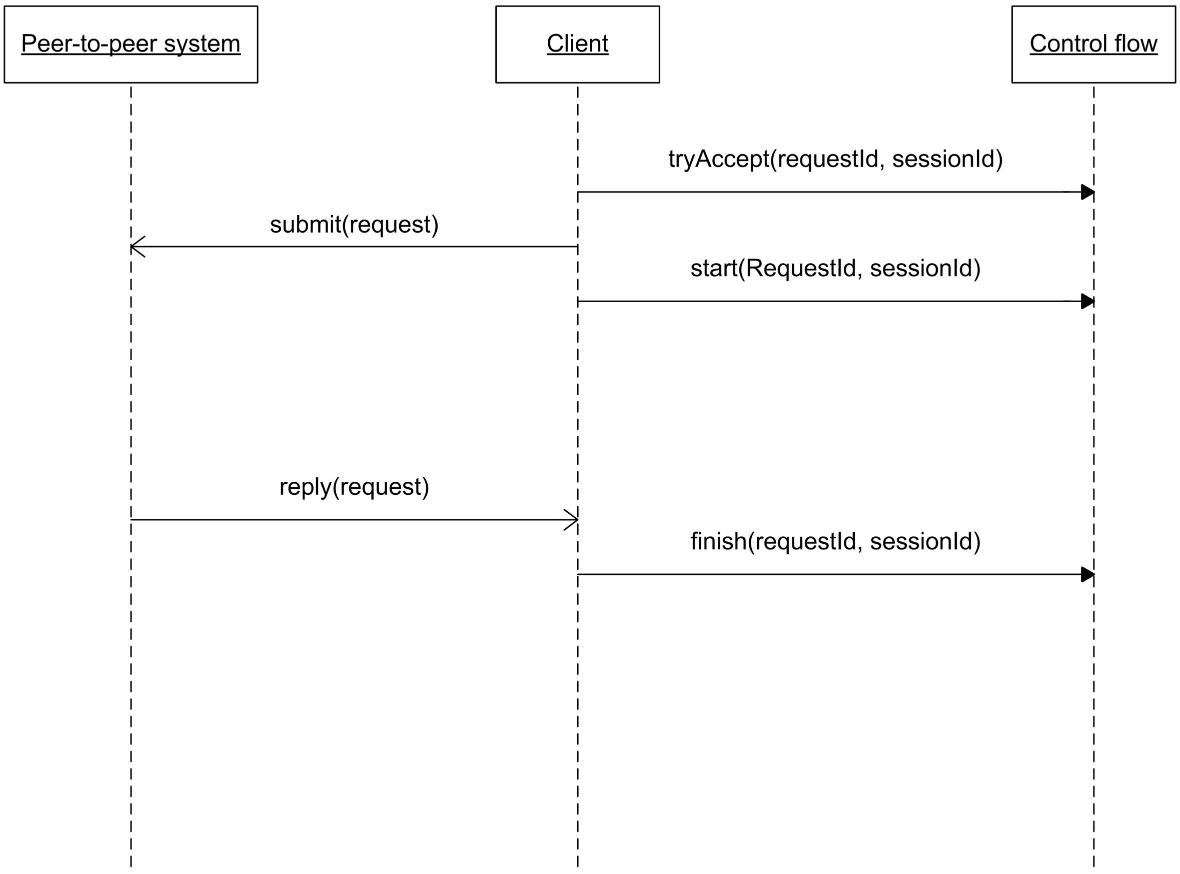

Peer-to-peer system
Client
Control flow
tryAccept(requestId, sessionId)
submit(request)
start(RequestId, sessionId)
reply(request)
finish(requestId, sessionId)

Figure 4.3: Operation sequence

Chapter 5

Flow control library - implementation

5.1. Overview

The entire library consists of several classes implemented in C++. These classes provide core functionality, and interfaces required for tuning the library to specific needs.

5.2. Class hierarchy

Diagram 5.1 illustrates the class hierarchy.

5.3. Class responsibilities

5.3.1. OutstandingPool

This is the main class in the library. It keeps track of slot usage per client and is responsible for dividing the total slot pool amongst clients. New client must register prior to using other methods of this class. It is thread-safe, all public methods are protected by a mutex.

5.3.2. LatencyTable

This is the generic interface for a placeholder for request start timestamps and durations. It keeps start time and request id pairs for pending requests, and start times and durations for requests which have already finished. It is used by OutstandingPool, which can insert and remove requests on demand. It doesn't need to be thread-safe, as synchronization is already done by OutstandingPool.

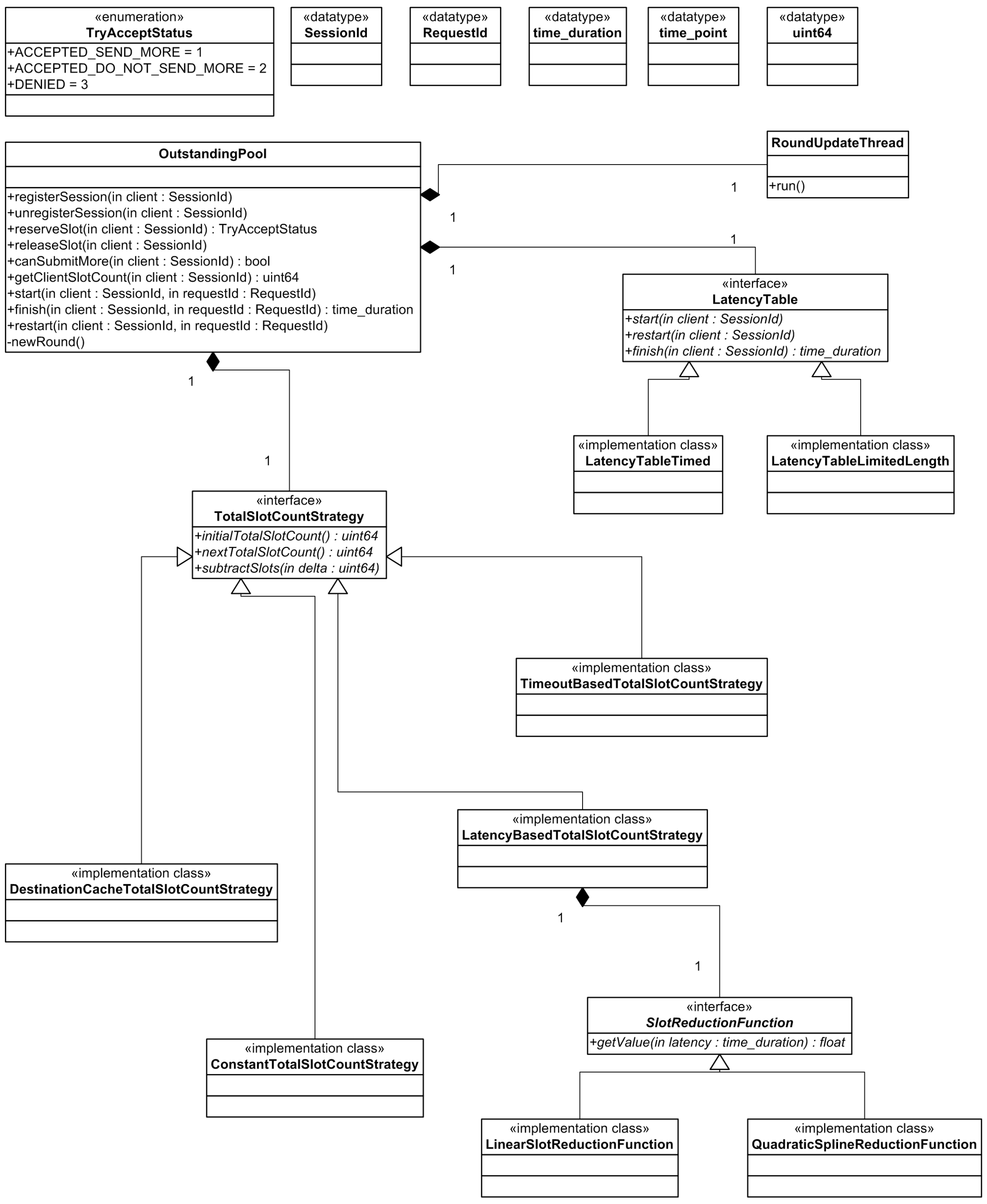
There are currently two implementations of this class.

LatencyTableTimed

Removes finished requests which are older than the specified amount of time.

LatencyTableLimitedLength

Keeps at most a specified number of finished requests in memory.



«enumeration»
TryAcceptStatus
+ACCEPTED_SEND_MORE = 1
+ACCEPTED_DO_NOT_SEND_MORE = 2
+DENIED = 3
«datatype» SessionId
«datatype» RequestId
«datatype» time_duration
«datatype» time_point
«datatype» uint64
OutstandingPool
+registerSession(in client : SessionId)
+unregisterSession(in client : SessionId)
+reserveSlot(in client : SessionId) : TryAcceptStatus
+releaseSlot(in client : SessionId)
+canSubmitMore(in client : SessionId) : bool
+getClientSlotCount(in client : SessionId) : uint64
+start(in client : SessionId, in requestId : RequestId)
+finish(in client : SessionId, in requestId : RequestId) : time_duration
+restart(in client : SessionId, in requestId : RequestId)
-newRound()
RoundUpdateThread
+run()
«interface» LatencyTable
+start(in client : SessionId)
+restart(in client : SessionId)
+finish(in client : SessionId) : time_duration
«implementation class» LatencyTableTimed
«implementation class» LatencyTableLimitedLength
«interface» TotalSlotCountStrategy
+initialTotalSlotCount() : uint64
+nextTotalSlotCount() : uint64
+subtractSlots(in delta : uint64)
«implementation class» TimeoutBasedTotalSlotCountStrategy
«implementation class» LatencyBasedTotalSlotCountStrategy
«implementation class» DestinationCacheTotalSlotCountStrategy
«implementation class» ConstantTotalSlotCountStrategy
«interface» SlotReductionFunction
+getValue(in latency : time_duration) : float
«implementation class» LinearSlotReductionFunction
«implementation class» QuadraticSplineReductionFunction
1
1
1
1
1
1
1
1

Figure 5.1: Class hierarchy

5.3.3. RoundUpdateThread

This class is responsible for periodic calls to OutstandingPool::newRound() method. Calling this method signals a start of a new round, and causes slot assignment recalculations.

This class is defined as a friend class in OutstandingPool, therefore it can call the newRound() method.

5.3.4. TotalSlotCountStrategy

This is an interface for a function used to recalculate the size of the total slot pool at the beginning of each round.

Four implementations have been provided.

- ConstantTotalSlotCountStrategy — obvious,
- DestinationCacheTotalSlotCountStrategy — is the equivalent of the reported-pool strategy mentioned in section 4.3.1,
- TimeoutBasedTotalSlotCountStrategy — is the equivalent of the timeout-based strategy mentioned in section 4.3.1,
- LatencyBasedTotalSlotCount — is the equivalent of the latency-based strategy mentioned in section 4.3.1.

5.4. Summary

The biggest problem I have encountered was designing slot latency functions. There are endless possibilities and picking the best solution may be complicated. A good mathematical model must be created and analyzed during further research in this field.

Chapter 6

Performance tests

6.1. Definitions used in tests

The following additional definitions are necessary in order to explain the testing environment:

Definition 6.1.1. *Pipeline length* means the maximum amount of requests which were processed in parallel at any time. When referring to '100 requests in pipeline' I mean 100 requsts were submitted for parallel processing. Submission doesn't necessarily mean a request has been accepted, it's possible it is waiting for a free slot.

Definition 6.1.2. *Test attribute* — the measure of performance reported by the test. For example: bandwidth, latency (maximum, minimum and average), etc.

6.2. Test setup

This library has been incorporated in a peer-to-peer software designed as a distibuted data storage network. The network allows parallel processing of many requests at once. The STAF framework [Sta06] was used to implement testing scripts. This framework allows parallel execution of tasks on multiple machines, and uses an engine called STAX to run XML-based testing scripts. It was used to test various software pieces, including the Linux kernel.

6.2.1. Hardware

Backend nodes and proxy are identical machines, each consisting of:

- processors: two Dual Core AMD Opteron 265 processors running at 1.8GHz,
- motherboard: TYAN High-End Dual AMD Opteron, S2882,
- hard drives: six WD Caviar drives – SATA, 250GB, 7200rpm, 16MB buffer,
- memory: 4GB physical, 3GB swap,
- network: two Broadcom Corporation NetXtreme BCM5704 Gigabit Ethernet cards working in trunking mode.

There are 4 backend machines and 1 proxy machine.

6.2.2. Software

Two versions of the same software were compiled, one with and one without control flow.

Software was compiled with all debugging statements in order to get full information about latencies, speed and request results in the log. This, unfortunately, greatly decreased performance (by a factor of 10). The change, however, equally impacts both versions, so test results remain valid.

The following settings were used during tests:

- The timeout value above which a request is considered a high latency request, used by the timeout-based total slot count strategy, was set to 15 seconds.
- Round length was set to 1 second.
- Penalty level for slow clients (*DecCut*) was set to 0.8.
- Bonus-trigger level for fast clients (*IncCut*) was set to 0.95.
- Maximum bonus for active clients was set to 0.1.
- Penalty buffer value was set to 0.05.
- Initial assumed total slot count was 250.

6.2.3. Test scenarios

The following tests were implemented and executed:

- Comparative data load tests:
  - With a client trying to submit 10000 requests for parallel processing,
  - With a client trying to submit 5000 requests for parallel processing,
  - With a client trying to submit 2000 requests for parallel processing,
  - With a client trying to submit 1000 requests for parallel processing.
- Slot reduction tests:
  - With a client trying to submit 2000 requests for parallel processing,
  - With a client trying to submit 1000 requests for parallel processing,
  - With a client trying to submit 500 requests for parallel processing.

Comparative data test

Comparative data test compares bandwidth and latency with and without the control flow mechanism.

The client driver in this scenario is loaded on a proxy machine connected to 4 backend nodes. It is trying to write two gigabytes of data, in 64 KB blocks.

The client is very simple, it is trying to submit as many requests to proxy as it can. When submission return value contains information from the proxy not to submit more data, it waits for a wake-up callback call. This is illustrated on diagram 6.1.

If the client doesn't receive a reply within 90 seconds, it resubmits a request to the network (not necessarily to the same node).

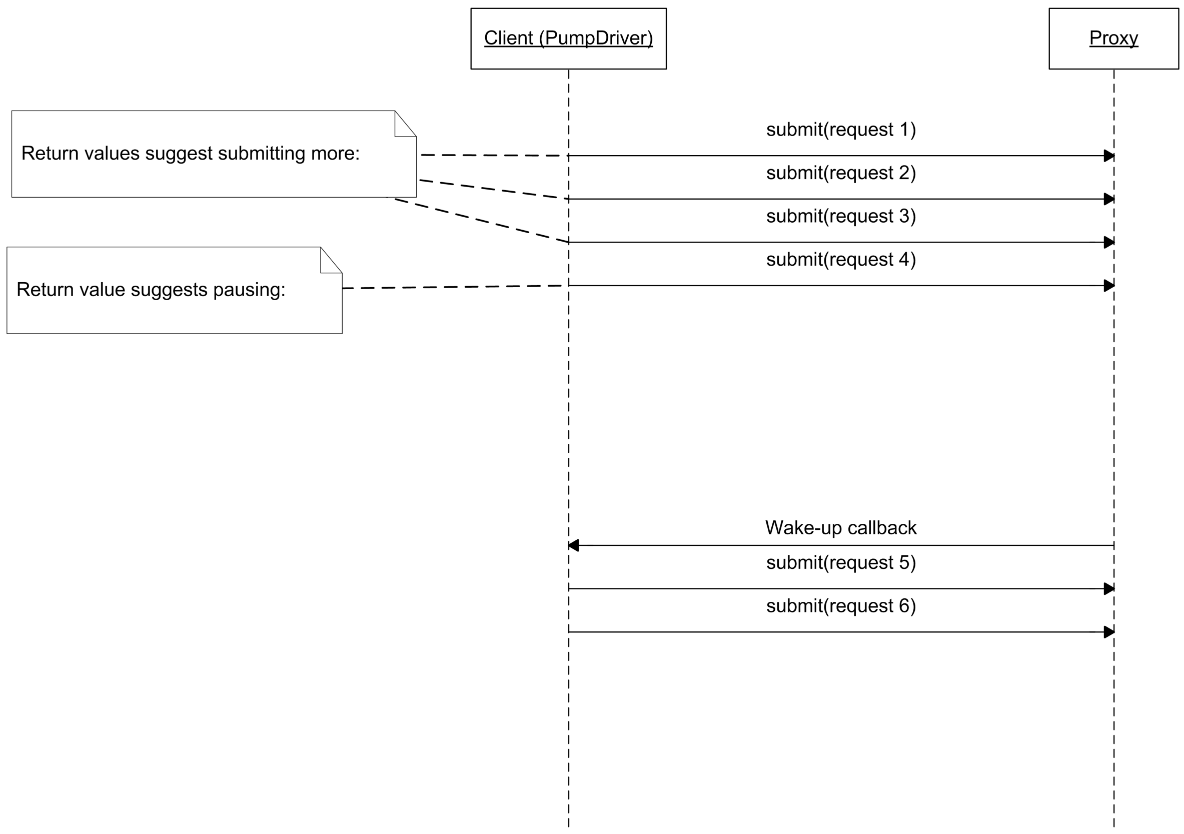


Figure 6.1: Pump driver operation sequence

Timeouts are reported by backend nodes when a request takes more than 30 seconds to complete. In such case, backend will send a reply to proxy with a failure status. Proxy will resubmit timeouted requests, perhaps to another node.

After writing data, the client tries to read it.

Slot reduction test

In this scenario the client driver is also loaded on a proxy connected to 4 backend nodes, but during writing one of the nodes fails. This is simulated by manually killing the peer-to-peer server process. This test only writes data.

6.3. Results

Each test scenario was executed 7 times, the highest and lowest value of each test attribute was discarded and an average value was calculated from the remaining 5.

It is important to note that the latencies reported here are measured by proxy. They contain network transmission latency, as well as latency on backend.

6.3.1. Comparative data test

Write latencies

All latencies and standard deviation described here were measured in seconds. In all tables CF means 'with control flow' and no CF 'without control flow'.

Requests in pipeline	Version	Latency			Standard deviation
		Maximum	Minimum	Average	
10000	CF	21.82	6.89	16.82	1.04
	No CF	138.59	15.64	40.28	2.67
5000	CF	22.86	9.84	15.48	0.61
	No CF	23.39	8.02	17.88	1.45
2000	CF	10.67	2.7	7.2	0.37
	No CF	9.74	3.95	7.2	0.87
1000	CF	4.93	2.13	3.71	0.23
	No CF	4.48	1.85	3.67	0.57

It is obvious that the control flow mechanism is very good at controlling latency. The test scenario with 10000 requests in pipeline shows 2.39 times better average latency with control flow enabled. The test scenario with a pipeline length of 5000 requests also shows that this mechanism is good at reducing latency. The control flow overhead starts to show with 1000 requests in pipeline, adding an average 40 milliseconds to each request.

Write bandwidth and number of timeouts

Bandwidth is reported in MB/sec.

Requests in pipeline	Version	Bandwidth	No. timeouts
10000	CF	15.1	0
	No CF	15.2	1481
5000	CF	15.5	0
	No CF	17.1	0
2000	CF	16.2	0
	No CF	16.2	0
1000	CF	15.8	0
	No CF	15.9	0

This result shows the overhead of the control flow mechanism. As can be seen from these results, the overhead is acceptable (maximum observed difference is 9%). It is worth noting that only one scenario shows such a difference, all other scenarios show either minimal differences, or even improvements in speed, probably caused by better latency and lack of retransmissions.

The difference in the scenarios with a request pipeline length set to 5000, upon closer examination of the logs, turned out to be easy to explain. The value at which a request is considered a 'high latency' request was set to 15 seconds on backend. This latency was observed on backend with approximately a thousand requests pending, so with 4 backends to use, the proxy was accepting 4000 requests from the pipeline. The version without control flow was pushing requests out faster, because it was keeping 5000 requests pending at a time.

This can be seen as a problem with configuration. The 15 second value seems quite low, perhaps setting it to a higher, more optimal value, would further decrease the difference between both versions. Such tests will be conducted in the future.

Read latencies

Requests in pipeline	Version	Latency			Standard deviation
		Maximum	Minimum	Average	
10000	CF	42.78	18.33	23.96	1.57
	No CF	56.47	14.04	30.24	2.86
5000	CF	21.46	2.67	11.07	0.98
	No CF	21.99	1.76	11.79	1.4
2000	CF	9.21	0.41	4.82	0.55
	No CF	8.88	0.35	4.9	0.76
1000	CF	5.33	0.4	2.47	0.29
	No CF	5.35	0.4	2.45	0.31

Again, we see the latency with control flow enabled is much better than without it.

Read bandwidth and number of timeouts

Requests in pipeline	Version	Bandwidth	No. timeouts
10000	CF	20.1	0
	No CF	15.2	521
5000	CF	20	0
	No CF	22.4	0
2000	CF	22.3	0
	No CF	22.3	0
1000	CF	22.5	0
	No CF	22.6	0

As could be expected, the bandwidth loss caused by control flow is minimal. At 10000 requests in pipeline, we see that the bandwidth differences are the largest, but this seems offset by the huge loss in bandwidth when control flow is enabled in the "5000 requests in pipeline" test. This unexpected drop is still undergoing investigation.

6.3.2. Slot reduction test

Write latencies

Requests in pipeline	Version	Latency			Standard deviation
		Maximum	Minimum	Average	
2000	CF	73,45	9.89	13.82	1.45
	No CF	183.59	35.64	68.28	4.58
1000	CF	68.86	7.84	9.48	1.13
	No CF	78.39	6.02	10.88	2.78
500	CF	63.67	2.7	8.2	0.97
	No CF	22.74	0.95	7.8	1.34

Again, we can observe that for fast clients the control flow enabled version performs much better, latency-wise, than the version without it. The large maximum latency value with 2000 outstanding requests is caused by a large number of retransmissions a request had to go through to be stored. The test scenario with 500 outstanding requests shows an opposite trend, with the flow control version of our software performing worse than the one without. Upon closer examination of the logs, I found that this latency was caused by a single request not being sent due to lack of slots assigned on backend. Future work will concentrate on improving the configuration settings to handle such scenarios better.

Write bandwidth and number of timeouts

Requests in pipeline	Version	Bandwidth	No. timeouts
2000	CF	6.5	500
	No CF	4.8	3956
1000	CF	6	250
	No CF	6.4	1658
500	CF	6	125
	No CF	7.2	876

A similar connection between outstanding requests and speed can be noticed in this table. This suggests that there is a certain limit of resubmissions which seems to be acceptable and actually improves performance. Future work will focus on finding this limit and setting configuration values to take advantage of it.

6.3.3. Test summary

The tests clearly show that my control flow library is very good at controlling latency. Unfortunately the relation between latency and bandwidth was not as straightforward as I initially suspected. In several tests the version without control flow performed better, bandwidth-wise, than the version with control flow enabled. These few scenarios will have to undergo further study in order to improve performance.

Chapter 7

Summary

After careful study of some of the most widely used control flow implementations, I believe I have managed to create a viable framework which can be used in many P2P applications. During my research I managed to identify and solve several problems regarding fair resource sharing. I have also shown that global knowledge isn't necessary in order to provide a near-optimal solution to distributed flow control.

I have separated the problem into several independent fields of research, such as: calculation of the total slot pool, fair sharing of the slot pool between clients and storing historical data. I believe this step will encourage other researchers to search for even better algorithms to solve these problems.

I have chosen latency as the main characteristic of a P2P network which this library was to improve. While bandwidth and latency weren't as strongly coupled as I initially suspected, there was enough correlation between these two parameters to show significant improvement in both of these areas.

Tests show that this algoritm is efficient with regard to controlling latency, while having minimal bandwidth overhead. It copes very well with heavy loads, and allows creating a system which is self-adapting to network traffic and local resource usage. No network communication is used, making it easy to avoid problems with failing nodes. This seems to be a good solution to be implemented on peer-to-peer network entry points.

The purpose of my master thesis was the design and implementation of a distributed control library, and initial testing. This has been done, and these initial results are presented here. However, this library is a part of a bigger commercial project, and therefore will have to undergo additional improvements and performance tuning. This is, however, beyond the scope of this master thesis.

7.1. Future work

I hope this work will be an impulse to test new control flow methodologies based on input throttling, as this seems a very powerful approach. Future work which still needs to be done includes adding new data sources to the library and improving the total slot pool splitting algorithm. The eMule upload queue sharing algorithm could be adopted in order to share the total slot queue, with "best" clients chosen at the beginning of each round.

While the timeout-based total slot size strategy was very simple to implement, it is in my opinion the main source of problems. I believe careful study will have to be performed in order to create a viable alternative to that strategy, based on request timeouts and user-defined slot reduction functions. The framework for research has already been implemented, which opens up a whole new area of research.

Bibliography

[Bar05] Yuliy Baryshnikov, Ed Coffman, Guillaume Pierre, Dan Rubenstein, Mark Squillante, Teddy Yimwadsana *Predictability of Web-Server Traffic Congestion*, 2005, Columbia University

[Bit06] *BitTorrent documentation*, `http://www.bittorrent.org`, 2006

[Bur03] Chiranjeeb Buragohain, Divyakant Agrawal, Subhash Suri, *A Game Theoretic Framework for Incentives in P2P Systems*, 2003, Computer Science Department, University of California

[Cac05] *Peer-to-Peer in 2005*, `http://www.cachelogic.com/home/pages/research/p2p2005.php`, 2005

[Coh03] Bram Cohen, *Incentives Build Robustness in BitTorrent*, May 2003, Berkeley, CA

[Din04] Roger Dingledine, Nick Mathewson, Paul Syverson, *Tor: The Second-Generation Onion Router*, 2004

[Eng05] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, Steven Lim, *A Survey and Comparison of Peer-to-Peer Overlay Network Schemes*, 2005, University of Cambridge

[Hal05] David Hales, Simon Patarin, *How to cheat BitTorrent and why nobody does*, May 2005, University of Bologna

[Iza04] M. Izal, G. Urvoy-Keller, E.W. Biersack, P.A. Felber, A. Al Hamra, and L.Garces-Erice, *Dissecting BitTorrent: Five Months in a Torrents Lifetime*, 2004, Institut Eurecom, Sophia-Antipolis, France

[Koz05] Charles Kozierok, *The TCP/IP Guide*, `http://www.tcpipguide.com`, 2005

[Kri04] Ramayya Krishnan, Michael D. Smith, Zhulei Tang, Rahul Telang, *The Impact of Free-Riding on Peer-to-Peer Networks*, 2004, H. John Heinz III School of Public Policy and Management, Carnegie Mellon University

[Kul05] Yoram Kulbak, Danny Bickson, *The eMule Protocol Specification*, January 2005, Distributed Algorithms, Networking and Secure Systems Lab, School of Computer Science and Engineering, The Hebrew University of Jerusalem

[Leg05A] Arnaud Legout, *Understanding BitTorrent: An Experimental Perspective*, November 2005, Institut Eurecom, Sophia-Antipolis, France

[Leg05B] Arnauld Legout, *Rarest First and Choke Algorithms Are Enough*, June 2006, Institut Eurecom, Sophia-Antipolis, France

[Qiu04] Dongyu Qiu, R. Srikant, *Modeling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks*, 2004, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign

[She04] Rob Sherwood, Ryan Braud, Bobby Bhattacharjee, *Slurpie: A Cooperative Bulk Data Transfer Protocol*, 2004, Department of Computer Science, University of Maryland, College Park, Maryland, USA

[Sta06] *STAF: Software Testing Automation Framework*, `http://staf.sourceforge.net`, 2006

[Tam03] Karthik Tamilmani, *Robustness of the BitTorrent protocol*, October 2003, Stony Brook University

[Tho05] Richard Thommes, Mark Coates, *BitTorrent Fairness: Analysis and Improvements*, 2005, Department of Electrical and Computer Engineering, McGill University

[Tsu03A] Tsuen-Wan Ngan, Dan S. Wallach, Peter Druschel, *Enforcing Fair Sharing of Peer-to-Peer Resources*, 2003, Department of Computer Science, Rice University

[Tsu03B] Tsuen-Wan Ngan, Animesh Nandi, Atul Singh, *Fair Bandwidth and Storage Sharing in Peer-to-Peer Networks*, 2003, Department of Computer Science, Rice University

[Wik06] *Wikipedia*, `http://en.wikipedia.org`, 2006

Appendix A

CD contents

The CD attached contains:

- `mslizak.pdf` – This document in PDF format,
- `mslizak.tex` – Source for this document,
- `pracamgr.cls` – The class for this document,
- `diagrams` – Figures used in this document (and sources),
- `sources` – Other documents (related work).