NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE

DEPARTMENT OF INFORMATICS AND TELECOMMUNICATION

BSc THESIS

Implementation of a BitTorrent client in Go

Loukas E. Litsos

SUPERVISOR: Mema Roussopoulos, Assistant Professor NKUA

ATHENS

MARCH 2020

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση BitTorrent προγράμματος-πελάτη στην Go

Λουκάς Η. Λίτσος

Επιβλέπουσα: Μέμα Ρουσσοπούλου, Αναπληρώτρια Καθηγήτρια ΕΚΠΑ

ΑΘΗΝΑ

ΜΑΡΤΙΟΣ 2020

BSc THESIS

Implementation of a BitTorrent client in Go

Loukas E. Litsos

S.N.: 1115201500082

SUPERVISOR: Mema Roussopoulos, Assistant Professor NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Υλοποίηση BitTorrent προγράμματος-πελάτη στην Go

Λουκάς Η. Λίτσος

Α.Μ: 1115201500082

Επιβλέπουσα: Μέμα Ρουσσοπούλου, Αναπληρώτρια Καθηγήτρια ΕΚΠΑ

ABSTRACT

Currently software using peer-to-peer networking solutions has become increasingly popular. Compared to the more common server-client solution, a peer-to-peer approach has several advantages including decentralization, increased robustness and resource providing, such as bandwidth, storage space and computing power, by peers. One area where robustness and utilization of resources are important is file distribution, especially of large files.

In the past, many protocols for file sharing have been presented but the one that is currently the most widespread is BitTorrent. The protocol aims for decentralization and simplicity and these were some of the keys of its success.

Because of its simplicity, a lot of developers have implemented BitTorrent clients or libraries. Most of the implementations, like the initial one, follow the event-driven programming paradigm. That allows developers not to deal with the synchronization of processes and data like in multi-threaded applications because usually, parallelism is deferred in I/O operations.

However, event-driven programs have their limitations too because developers do not fully control the level of parallelism in their applications. In this thesis, we developed Charo, a highly concurrent/parallel BitTorrent client to fully leverage a machine’s multiple cores. The implementation was written in the Go programming language which was one of the most suited candidates.

SUBJECT AREA: Peer-to-Peer File Sharing
KEYWORDS: bittorrent, p2p, concurrency, tracker, dht, software library

ΠΕΡΙΛΗΨΗ

Επί του παρόντος, το λογισμικό που χρησιμοποιεί λύσεις δικτύωσης ομότιμων χρηστών (P2P) έχει γίνει όλο και πιο δημοφιλές. Σε σύγκριση με την πιο κοινή λύση διακομιστή -πελάτη, η P2P προσέγγιση έχει πολλά πλεονεκτήματα, συμπεριλαμβανομένης της αποκέντρωσης, της αυξημένης ευρωστίας και της διαθεσιμότητας πόρων, όπως το εύρος ζώνης, ο χώρος αποθήκευσης και η υπολογιστική ισχύς, από τους ομότιμους. Ένας τομέας όπου η αξιοπιστία και η αξιοποίηση των πόρων είναι σημαντικές είναι η διανομή αρκετά μεγάλων αρχείων.

Στο παρελθόν, έχουν εισαχθεί πολλά πρωτόκολλα κοινής χρήσης αρχείων, αλλά αυτό που είναι σήμερα το πιο διαδεδομένο είναι το BitTorrent. Το πρωτόκολλο αποσκοπεί στην αποκέντρωση και την απλότητα και αυτά είναι μερικά από τα κλειδιά της επιτυχίας του.

Λόγω της απλότητάς του, πολλοί προγραμματιστές έχουν υλοποιήσει προγράμματα-πελάτες ή βιβλιοθήκες BitTorrent. Οι περισσότερες από τις εφαρμογές, όπως και η αρχική, ακολουθούν το παράδειγμα προγραμματισμού που βασίζεται σε γεγονότα (event-driven). Αυτό επιτρέπει στους προγραμματιστές να μην ασχολούνται με τον συγχρονισμό διαδικασιών και δεδομένων όπως σε εφαρμογές πολλαπλών νημάτων, επειδή συνήθως, ο παραλληλισμός μετατίθεται στις λειτουργίες Εισόδου/Εξόδου (I/O).

Ωστόσο, τα προγράμματα που βασίζονται σε γεγονότα έχουν τους περιορισμούς τους εξίσου επειδή οι προγραμματιστές δεν μπορουν να ελέγχουν πλήρως το επίπεδο παραλληλισμού στις εφαρμογές τους. Σε αυτή τη διπλωματική εργασία, αναπτύξαμε το Charo, έναν πελάτη BitTorrent που αξιοποιεί σε μεγάλο βαθμό τους πολλαπλούς πυρήνες ενός μηχανήματος. Η εφαρμογή γράφτηκε στη γλώσσα προγραμματισμού Go, η οποία ήταν ένας από τους κατάλληλους υποψηφίους.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Δικτύωση Ομότιμων Χρηστών για Διαμοιρασμό Αρχείων
ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: bittorrent, p2p, concurrency, tracker, dht, software library

CONTENTS

LIST OF FIGURES

LIST OF IMAGES

PREFACE

This project was developed in Athens, Greece between September 2019 and March 2020. At its initial stages, it was important to rigorously read the BitTorrent protocol specification in order to understand every detail of it. Consequently, it was essential to understand the BitTorrent algorithms which were not part of the official specification. Yet, the most important part was to build the actual BT client and test it.

1. INTRODUCTION

A peer-to-peer network is designed around the notion of equal peers/nodes simultaneously functioning as both clients and servers to the other nodes on the network. This model of network arrangement differs from the client-server model where communication is usually to and from a central server.

In P2P networks, clients both provide and use resources. This means that unlike client -server systems, the content-serving capacity of peer-to-peer networks can actually increase as more users begin to access the content. That makes P2P networks ideal for file sharing applications.

In this thesis we present Charo, a library implementing the popular peer-to-peer (P2P) protocol BitTorrent [1] which is nowadays considered the most used network for file sharing. The implementation was written in Go (Golang) which offers some concurrency primitives which differ from other popular programming languages (e.g Java, Javascript, Python). BitTorrent requires that many connections with different peers should be simultaneously maintained, so it was one of the perfect candidates to explore Go concurrency capabilities.

The rest of this thesis is organized as follows:

1. In Chapter 2 we give an overview of what BitTorrent is and describe the different entities involved.
2. In Chapter 3 we give an overview of the Go concurrency model.
3. In Chapter 4 we describe Charo’s architecture which is different from
4. most other BitTorrent clients.
5. In Chapter 5 we present the algorithms that Charo utilizes.
6. In Chapter 6 we provide the documentation of the Charo library.
7. In Chapter 7 we present the Charo CLI application.
8. In Chapter 8 we summarize our conclusions.

2. BITTORRENT

BitTorrent is a communication protocol for peer-to-peer file sharing (P2P) which is used to distribute data and files over the Internet. It can be used to reduce the server and network impact of distributing large files. Rather than downloading a file from a single source server, the BitTorrent protocol allows users to join a swarm of hosts to upload to/download from each other simultaneously. The protocol is an alternative to the older single source, multiple mirror sources technique for distributing data, and can work effectively over networks with lower bandwidth. Using the BitTorrent protocol, several basic computers, such as home computers, can replace large servers while efficiently distributing files to many recipients.

Files transferred using BitTorrent are split in pieces, and each piece is split in blocks. Blocks are the transmission unit on the network, but the protocol only accounts for transferred pieces.

The BitTorrent architecture normally consists of the following entities:

1. Peer

A peer is typically a user running a BitTorrent client that wants to transfer data by joining a swarm.A peer has two states:

ii) the leecher state, when it is downloading a content, but does not have yet all pieces

ii) the seed state when the peer has all the pieces of the content.

For short, we can say that a peer is a leecher when it is in leecher state and a seed when it is in seed state.

2. Torrent file

In the BitTorrent file distribution system, a torrent file or metainfo is a computer file that contains metadata about files and folders to be distributed, and usually also a list of the network locations of trackers, which are computers that help participants in the system find each other and form efficient distribution groups called swarms. A torrent file does not contain the content to be distributed; it only contains information about those files, such as their names, sizes, folder structure, and cryptographic hash values (SHA-1) for verifying file integrity. Each torrent file is identified by a unique number called the Infohash.

3. Tracker

A BitTorrent tracker is a special type of server that assists in the communication between peers using the BitTorrent protocol. The tracker server keeps track of where file copies reside on peer machines, which ones are available at time of the client request, and helps coordinate efficient transmission and reassembly of the copied file. Clients that have already begun downloading a file communicate with the tracker periodically to negotiate faster file transfer with new peers, and provide network performance statistics; however, after the initial peer-to-peer file download is started, peer-to-peer communication can continue without the connection to a tracker.A tracker is the only centralized entity in the BitTorrent network.

However, because the protocol thrives for decentralization, another technology was introduced called DHT (Distributed Hash Table) [2]. Any peer implementing the DHT essentially becomes a tracker and stores lists of other nodes/peers which can be used to locate new peers.

Note: To avoid confusion,from now on we will call 'client' the BitTorrent peer that is running on the local machine,whereas every other remote peer will be called 'peer'. Readers of this thesis may choose to think of themselves as the client which connects to numerous peers

A client joins an existing torrent swarm by downloading a torrent file usually from a Web server. As stated above, the torrent file contains the url of some trackers so the client tries to connect to them and eventually they respond with the IP addresses of others peers in the swarm. Then, the client tries to open new TCP connections with the addresses responded by the tracker and finally can exchange blocks of data with the rest of the swarm.

For every connection established with a peer,a client must maintain state information for each connection that it has with a remote peer:

i) choked: whether or not the remote peer has choked this client. When a peer chokes the client, it is a notification that no requests will be answered until the client is unchoked. The client should not attempt to send requests for blocks, and it should consider all pending (unanswered) requests to be discarded by the remote peer. Choking is done for several reasons. TCP congestion control behaves very poorly when sending over many connections at once. Also, choking lets each peer use a tit-for-tat-ish algorithm to ensure that they get a consistent download rate.

ii) interested: Whether or not the remote peer is interested in something this client has to offer. This is a notification that the remote peer will begin requesting blocks when the client unchokes them.

3. GO CONCURRENCY DESIGN

Go is a statically typed, compiled programming language. It is syntactically similar to C, but with memory safety, garbage collection, structural typing and CSP-style [3] concurrency. The Go language has built-in facilities, as well as library support, for writing concurrent programs.

The primary concurrency construct is the goroutine, a type of light-weight thread. A function call prefixed with the **go** keyword starts a function in a new goroutine.

The language specification does not specify how goroutines should be implemented, but current implementations multiplex a Go process's goroutines onto a smaller set of operating-system threads, similar to the scheduling performed in Erlang programming language.

While a standard library package featuring most of the classical concurrency control structures (mutex locks, etc.) is available, idiomatic concurrent programs instead prefer channels, which provide send messages between goroutines. Optional buffers store messages in FIFO order and allow sending goroutines to proceed before their messages are received.

Channels are typed, so that a channel of type `chan` *`T`* can only be used to transfer messages of type *`T`*. Special syntax is used to operate on them; **`<-ch`** is an expression that causes the executing goroutine to block until a value comes in over the channel `ch`, while **`ch <- x`** sends the value `x` (possibly blocking until another goroutine receives the value). The built-in `switch`-like **`select`** statement can be used to implement non-blocking communication on multiple channels. Go has a memory model describing how goroutines must use channels or other operations to safely share data.

4. ARCHITECTURE OF CHARO

The basic objects of the library we provide are Client and Torrent. Torrent maintains state about a given torrent and Client manages multiple Torrents. The Client also runs the TCP listener for accepting new incoming connections and the DHT server.

Most implementations of BitTorrent, for example libtorrent [4], have adopted an event driven architecture for building BT clients. A single thread polls each peer connection for every torrent and handles the received messages. Multiple threads are mainly utilized when doing disk I/O operations. That is in contrast with Charo (the library we provide), which handles each connection in a separate goroutine. In essence, every process mentioned in the BT protocol that requires disk or network I/O is managed by a separate goroutine.So, for each torrent there are four goroutines:

i) 'conn' goroutine: exchanges messages with peers (requests pieces from the remote peer and uploads to him).

ii) 'verifier' goroutine: responsible for checking the integrity of the downloaded pieces.

iii) 'tracker announcer' goroutine: its job is to make requests to trackers when it’s notified.

iv) 'torrent' goroutine: it is responsible for managing the state of each connection, that is if the client is interested in what the remote peer offers and if the client chokes the remote peer (choking algorithm described below). Also, it holds some information about the state of the torrent (if the client is leecher or seed,etc),some statistics and keeps track of what pieces of the torrent the client has downloaded. Moreover, it notifies the 'tracker' goroutine to make requests at regular intervals and finally,it is responsible for queueing pieces for an integrity check to 'verifier' goroutine.

The communication happening between the different goroutines can be visualized with the following figure:

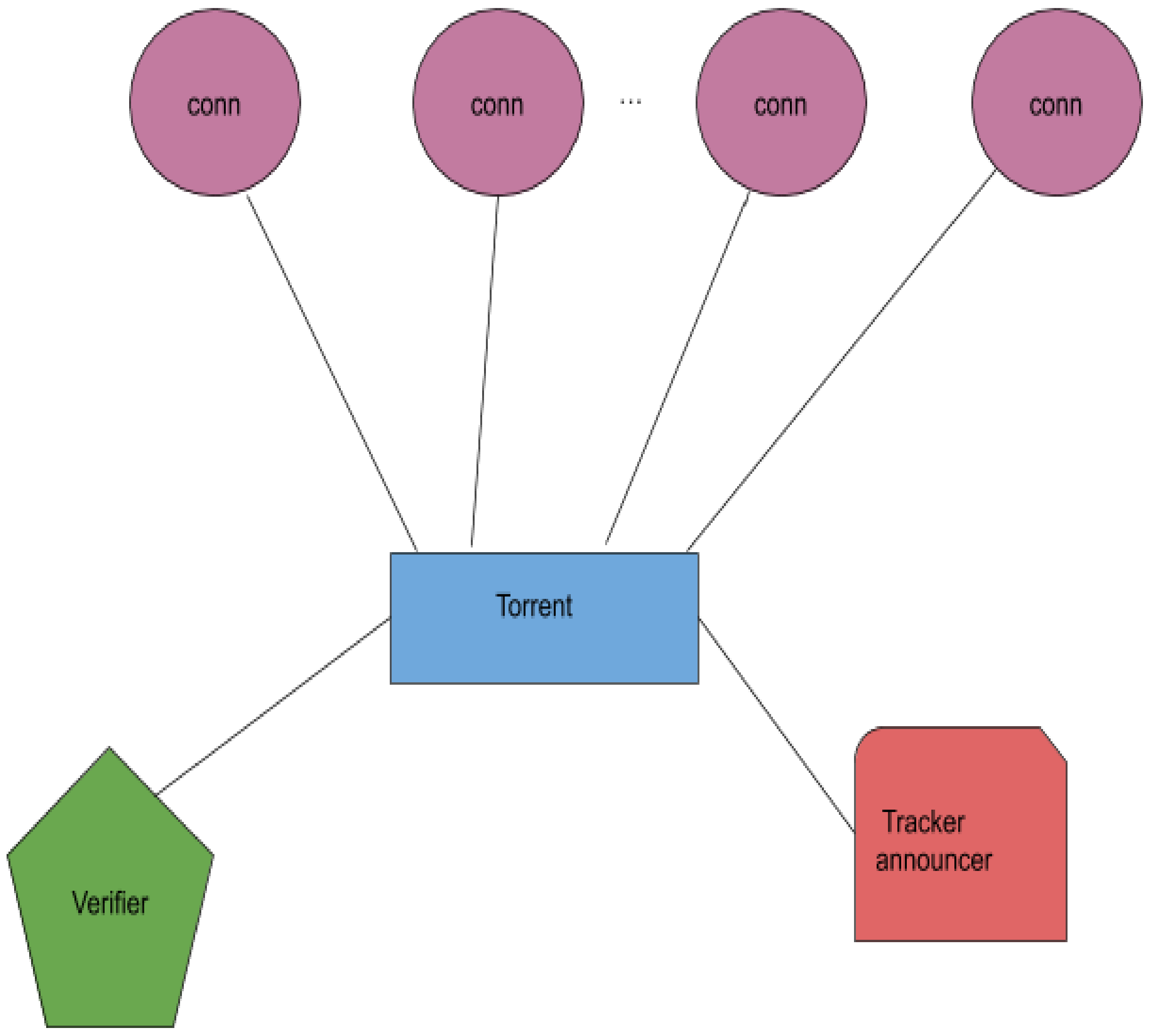


Figure 1: Torrent Goroutines

The overall picture includes the Client which manages multiple Torrents.The Client doesn't run in a separate goroutine but the DHT server and TCP listener which are managed by him are running in their own goroutines.

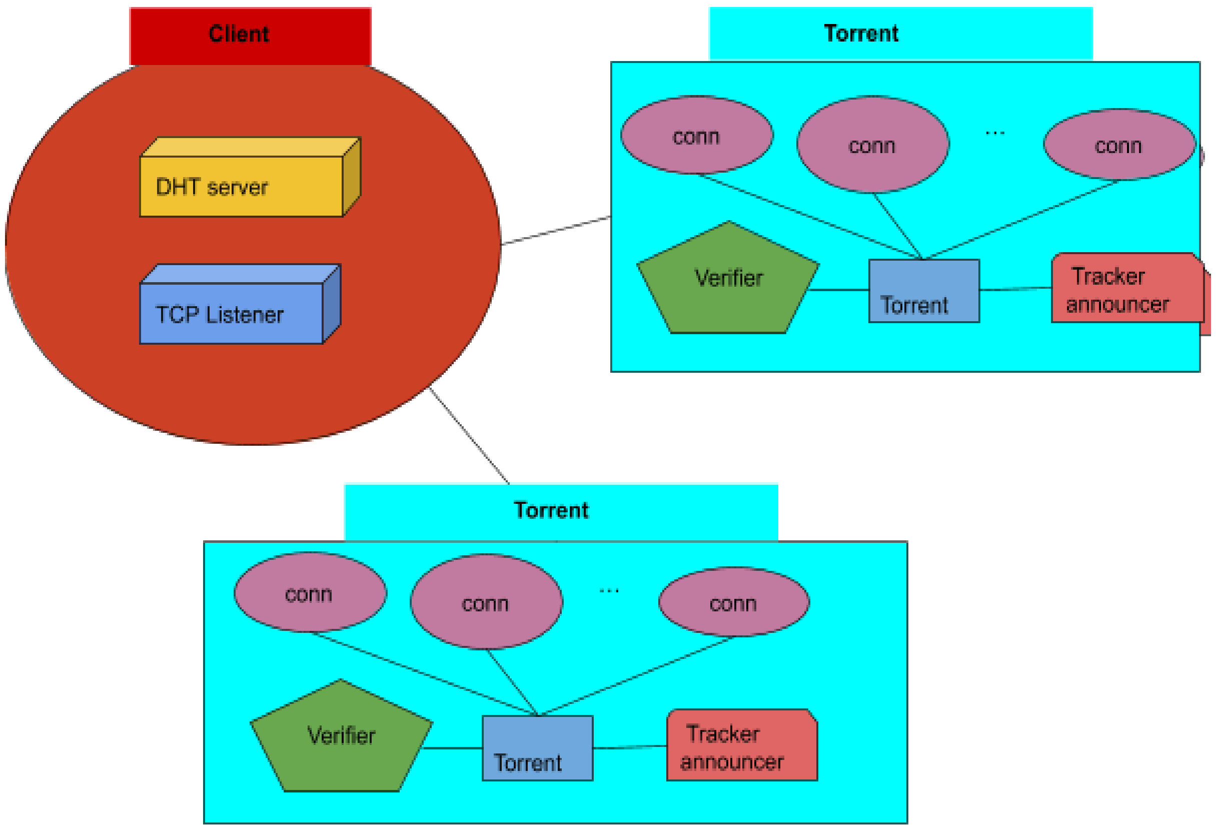


Figure 2: Charo Architecture

5. CHARO ALGORITHMS

Below, we mention some of the main algorithms deployed in Charo. These algorithms are not part of the original BitTorrent specification but they are proposed by the BitTorrent creators and community [5] [6]. Most other BT clients have adopted them,as did Charo.

5.1 Queueing

The client keeps a few unfulfilled requests on each connection. This is done because otherwise a full round trip is required from the download of one block to beginning the download of a new block. On links with high BDP (bandwidth-delay-product, high latency or high bandwidth), this can result in a substantial performance loss.

5.2 Piece Selection

Selecting pieces to download in a good order is very important for good performance. A poor piece selection algorithm can result in having all the pieces which are currently on offer or, on the flip side, not having any pieces to upload to peers you wish to.

5.2.1 Strict Policy

Charo’s first policy for piece selection is that once a single sub-piece has been requested, the remaining sub-pieces from that particular piece are requested before sub-pieces from any other piece. This does a good job of getting complete pieces as quickly as possible.

5.2.2 Rarest First

When selecting which piece to start downloading next, charo downloads pieces which the fewest peers of the swarm have, a technique we refer to as ’rarest first’. This technique does a good job of making sure that peers have pieces which all of their peers want, so uploading can be done when wanted. It also makes sure that pieces which are more common are left for later, so the likelihood that a peer which currently is offering upload will later not have anything of interest is reduced. Information theory dictates that no downloaders can complete until every part of the file has been uploaded by the seed. For deployments with a single seed whose upload capacity is considerably less than that of many downloaders, performance is much better if different downloaders get different pieces from the seed, since redundant downloads waste the opportunity for the seed to get more information out. Rarest first does a good job of only downloading new pieces from the seed, since downloaders will be able to see that their other peers have pieces the seed has uploaded already. For some deployments the original seed is eventually taken down for cost reasons, leaving only current downloaders to upload. This leads to a very significant risk of a particular piece no longer being available from any current downloaders. Rarest first again handles this well, by replicating the rarest pieces as quickly as possible thus reducing the risk of them getting completely lost as current peers stop uploading.

5.2.3 Random First Pieces

An exception to rarest first is when downloading starts. At that time, the client has nothing to upload, so it’s important to get a complete piece as quickly as possible. Rare pieces are generally only present on one peer, so they would be downloaded slower than pieces which are present on multiple peers for which it’s possible to download sub-pieces from different places. For this reason, pieces to download are selected at random until the first complete piece is assembled, and then the strategy changes to rarest first.

5.2.4 End game Mode

Sometimes a piece will be requested from a peer with very slow transfer rates. This isn’t a problem in the middle of a download, but could potentially delay a download’s finish. To keep that from happening, once all sub-pieces which the client doesn’t have are actively being requested it sends requests for all sub-pieces to all peers. Cancels are sent for sub-pieces which arrive to keep too much bandwidth from being wasted on redundant sends. In practice not much bandwidth is wasted this way, since the endgame period is very short, and the end of the file(s) is always downloaded quickly.

5.3 Choking Algorithm

BitTorrent does not have a central resource allocation. Each peer is responsible for attempting to maximize its own download rate. Peers do this by downloading from whoever they can and deciding which peers to upload to via a variant of tit-for-tat. To cooperate, peers upload, and to not cooperate they ’choke’ peers. Choking is a temporary refusal to upload; It stops uploading, but downloading can still happen and the connection doesn’t need to be renegotiated when choking stops. The choking is necessary because TCP congestion control behaves very poorly when sending over many connections at once.

On a technical level, the client always unchokes a fixed number of peers (the default is four) so the issue becomes which peers to unchoke. The decision is made based on the download rates. Every peer is sorted based on their upload rate and the ones that are interested and have the higher rate get unchoked. We will refer to these peers as downloaders. Peers which have a better upload rate (compared to downloaders) but aren't interested get unchoked. If they become interested, the *downloader* with the worst upload rate gets choked. Charo recalculates who they want to choke once every ten seconds,when an unchoked peer becomes interested and when a peer connects or gets disconnected with the client. If the client is a seed, it uses its upload rate rather than its download rate to decide which peers to unchoke.

5.3.1 Optimistic Unchoking

Simply uploading to the peers which provide the best download rate would suffer from having no method of discovering if currently unused connections are better than the ones being used. To fix this, at all times a BitTorrent peer has a single ‘optimistic unchoke’, which is unchoked regardless of the current download rate from it. Which peer is the optimistic unchoke is rotated every third rechoke period (30 seconds). 30 seconds is enough time for the upload to get to full capacity, the download to reciprocate, and the download to get to full capacity.

5.3.2 Anti-Snubbing

Occasionally a peer will be choked by all peers which it was formerly downloading from. In such cases it will usually continue to get poor download rates until the optimistic unchoke finds better peers. To mitigate this problem, when over a minute goes by without getting any piece data while downloading from a peer, Charo assumes it is "snubbed" by that peer and doesn't upload to it except as an optimistic unchoke.

6. LIBRARY DOCS

Below we provide the documentation of the Charo library. The official and maybe more up to date documentation can be found here. The project repository is hosted on Github [7] and can be found here.

Package torrent implements the BitTorrent protocol. It is designed to be simple and easy to use. A common workflow is to create a Client, add a torrent and then download it.

```
cl, _ := torrent.NewClient(nil)
t, _ := cl.AddFromFile("example.torrent")
t.Download()
fmt.Println("torrent downloaded!")
```

type Client

```
type Client struct {
    // contains filtered or unexported fields
}
```
Client manages multiple torrents

func NewClient

```
func NewClient(cfg *Config) (*Client, error)
```
NewClient creates a new Client with the provided configuration. Use `NewClient(nil)` for the default configuration.

func (*Client) AddFromFile

```
func (cl *Client) AddFromFile(filename string) (*Torrent, error)
```
AddFromFile creates a torrent based on the contents of filename. The Torrent returned may be in seeding mode if all the data is already downloaded.

func (*Client) AddFromInfoHash

```
func (cl *Client) AddFromInfoHash(infohash [20]byte) (*Torrent, error)
```
AddFromInfoHash creates a torrent based on it's infohash (not implemented yet)

func (*Client) AddFromMagnet

```
func (cl *Client) AddFromMagnet(uri string) (*Torrent, error)
```
AddFromMagnet creates a torrent based on the magnet link provided (not implemented yet)

func (*Client) Close

```
func (cl *Client) Close()
```
Close calls torrent.Close for all the torrents managed by the client.

func (*Client) ID

```
func (cl *Client) ID() []byte
```
ID returns the Client's random ID

func (*Client) ListenPort

```
func (cl *Client) ListenPort() int
```
ListenPort returns the port that the client listens for new connections

func (*Client) Torrents

```
func (cl *Client) Torrents() []*Torrent
```
Torrents returns all torrents that the client manages.

type Config

```
type Config struct {
    //Max outstanding requests per connection we allow for a peer to have
    MaxOnFlightReqs int
    //Max active/established connections per torrent
    MaxEstablishedConns int
    //This option disables DHT also.
    RejectIncomingConnections bool
    DisableTrackers           bool
    DisableDHT                bool
    //Directory to store the data
    BaseDir string
    //Function to open the storage.Provide your own for a custom storage
implementation
    OpenStorage storage.Open
    //Dials for new connections will fail after this duration
    DialTimeout time.Duration
    //BitTorrent handshakes will fail after this duration
    HandshakeTiemout time.Duration
}
```
Config provides configuration for a Client.

func DefaultConfig

```
func DefaultConfig() (*Config, error)
```
DefaultConfig returns the default configuration for a client

type Peer

```
type Peer struct {
    P      tracker.Peer
    Source PeerSource
}
```
Holds basic information about a peer

type PeerSource

```
type PeerSource byte
```

Which source informed us about that peer

```
const (
    //The user manually added this peer
    SourceUser PeerSource = iota
    //It was an incoming connection
    SourceIncoming
    //The peer was given to us by DHT
    SourceDHT
    //The peer was given to us by a tracker
    SourceTracker
)
```

type Stats

```
type Stats struct {
    //Number of blocks/chunks downloaded (not necessarily verified)
    BlocksDownloaded int
    //Number of blocks/chunks uploaded
    BlocksUploaded int
    //Remainings bytes to download (bytes that are downloaded but not
verified are not included)
    BytesLeft int
    //Number of verified bytes we have downloaded
    BytesDownloaded int
    //Number of bytes we have uploaded
    BytesUploaded int
}
```

Stats contains statistics about a Torrent

func (*Stats) String

```
func (s *Stats) String() string
```

type Torrent

```
type Torrent struct {
    // contains filtered or unexported fields
}
```

Torrent represents a torrent and maintains state about it.Multiple goroutines may invoke methods on a Torrent simultaneously.

func (*Torrent) AddPeers

```
func (t *Torrent) AddPeers(peers ...Peer) error
```

AddPeers adds peers to the Torrent and (if needed) tries to establish connections with them. Returns error if the torrent is closed.

func (*Torrent) Close

```
func (t *Torrent) Close()
```

Close removes the torrent from the Client and closes all connections with peers. Close is safe to be called multiple times on the same torrent.

func (*Torrent) Closed

```
func (t *Torrent) Closed() bool
```

Closed returns whether the torrent is closed or not.

func (*Torrent) Download

```
func (t *Torrent) Download() error
```

Download downloads all the torrent's data. It requires the info first. After the download is complete, the Torrent transitions in seeding mode (i.e altruistically upload) until it's closed. If the data is already there,Download returns immediately and Torrent transits in seeding mode.

func (*Torrent) HaveAllPieces

```
func (t *Torrent) HaveAllPieces() bool
```

HaveAllPieces returns whether all torrent's data has been downloaded

func (*Torrent) Metainfo

```
func (t *Torrent) Metainfo() *metainfo.MetaInfo
```

Metainfo returns the metainfo of the torrent or nil of its not available

func (*Torrent) Seeding

```
func (t *Torrent) Seeding() bool
```

Seeding returns true if `HaveAllPieces` returns true and a call to `Download` has been made for this torrent

func (*Torrent) Stats

```
func (t *Torrent) Stats() Stats
```

func (*Torrent) Swarm

```
func (t *Torrent) Swarm() []Peer
```

Swarm returns all known peers associated with this torrent.

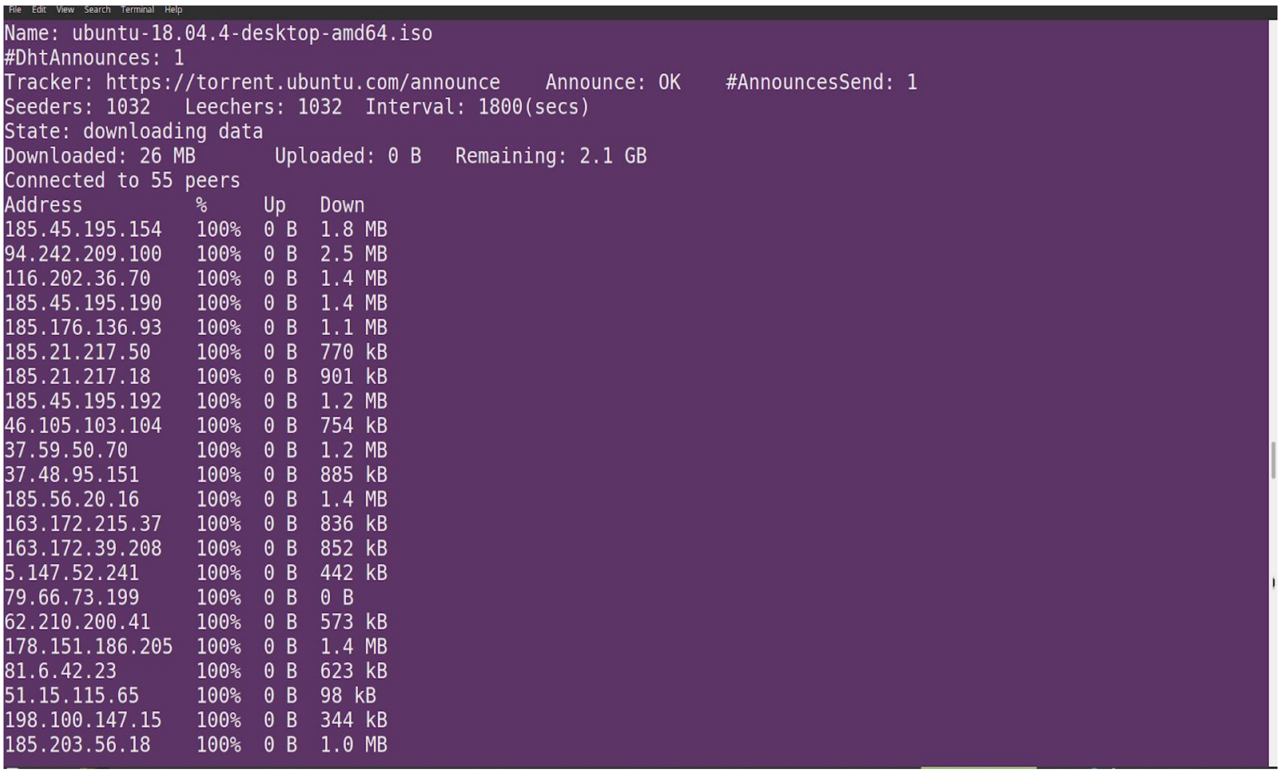
func (*Torrent) WriteStatus

```
func (t *Torrent) WriteStatus(w io.Writer)
```

WriteStatus writes to w a human readable message about the status of the Torrent.

7. CHARO CLIENT

A simple client has been built utilizing the Charo library. Here's how it looks in action:



```
Name: ubuntu-18.04.4-desktop-amd64.iso
#DhtAnnounces: 1
Tracker: https://torrent.ubuntu.com/announce    Announce: OK    #AnnouncesSend: 1
Seeders: 1032   Leechers: 1032  Interval: 1800(secs)
State: downloading data
Downloaded: 26 MB       Uploaded: 0 B   Remaining: 2.1 GB
Connected to 55 peers
Address          %     Up   Down
185.45.195.154   100%  0 B  1.8 MB
94.242.209.100   100%  0 B  2.5 MB
116.202.36.70    100%  0 B  1.4 MB
185.45.195.190   100%  0 B  1.4 MB
185.176.136.93   100%  0 B  1.1 MB
185.21.217.50    100%  0 B  770 kB
185.21.217.18    100%  0 B  901 kB
185.45.195.192   100%  0 B  1.2 MB
46.105.103.104   100%  0 B  754 kB
37.59.50.70      100%  0 B  1.2 MB
37.48.95.151     100%  0 B  885 kB
185.56.20.16     100%  0 B  1.4 MB
163.172.215.37   100%  0 B  836 kB
163.172.39.208   100%  0 B  852 kB
5.147.52.241     100%  0 B  442 kB
79.66.73.199     100%  0 B  0 B
62.210.200.41    100%  0 B  573 kB
178.151.186.205  100%  0 B  1.4 MB
81.6.42.23       100%  0 B  623 kB
51.15.115.65     100%  0 B  98 kB
198.100.147.15   100%  0 B  344 kB
185.203.56.18    100%  0 B  1.0 MB
```

Image 1: Charo client in action

On the first line it's displayed the name of the file we are downloading or in case we are downloading multiple files the name of the base directory of all these files. In this particular example we are downloading a single file. On the second line, we can see how many times we have made effort to obtain new peers via the DHT (aka Announce) . On the third and the fourth one, we can see information about the tracker we requested and his response (how many seeds and leechers this torrent has - this information is typically outdated - and how much we have to wait until the next announcement - Interval field). Below, we can learn some information about the state of the torrent and about the progress that has been made. The 'Downloaded' refers to the total number of bytes we have downloaded and verified (i.e made the integrity check based on the hash contained in the metainfo file), the 'Uploaded' refers to how many bytes we have uploaded to remote peers in total and 'Remaining' = TotalBytesToDownload - Downloaded. Next, information is provided about the peers we are currently connected to. At the first column we can see the IP addresses of the peers. In the second column, we see the percentage of the pieces a peer has. For example, if a torrent consists of 100 pieces and a peer 50%, then it means that the peer has half the pieces (50). In the snapshot provided, it is clear that all peers are seeds. The third and the fourth column show how many bytes we have uploaded and downloaded respectively from each peer.

8. CONCLUSION

The BitTorrent protocol has been around for quite a long time. Although the protocol is mature enough, many of its functionalities are not standardized (see Algorithms Section). As a result, many of the clients out there provide custom implementations for algorithms such as the Choking Algorithm, which if not designed well may harm and even eliminate torrent swarms.

Moreover, many BitTorrent libraries expose APIs that don’t encourage seeding. In contrast with this, Charo offers the blocking ‘Download’ method which when returns makes the client seed automatically in the background until the torrent is closed.

Aside from the protocol, the development of Charo was not trivial due to the synchronization needed for all those goroutines. In an event driven architecture, that's not a big problem since developers haven’t to deal with these aspects but in the case of Charo, having to synchronize many goroutines with different functionality each was difficult. Although Go offers an elegant concurrency model, the language does not guarantee thread safety in any ways.

However, what we achieved by building this highly concurrent client is scalability. As mentioned before, each torrent that the client manages runs on its own goroutine so performance does not downgrade as we add torrents.

ACRONYMS AND ABBREVIATIONS

DHT	Distributed Hash Table
BT	BitTorrent
TCP	Transmission Control Protocol
CSP	Communicating Sequential Processes

REFERENCES

[1] “BitTorrent specification” [Online]
Available: https://www.bittorrent.org/beps/bep_0003.html

[2] Kademlia: A Peer-to-Peer Information System Based on the XOR Metric

[3] Hoare C.A.R. "Communicating sequential processes". 1978

[4] “libtorrent” [Online]
Available: https://www.libtorrent.org

[5] Bram Cohen. Incentives Build Robustness in BitTorrent. 2003

[6] Arnaud Legout, Guillaume Urvoy-Keller, Pietro Michiardi. Understanding BitTorrent: An Experi-mental Perspective. 2005

[7] “GitHub” [Online]
Available: https://github.com