BitTorrent distributed generation of rainbow tables

Advanced Networking and Future Internet

Students: Arnaud Durand, Mikael Gasparian, Thomas Rouvinez
Professors: Prof. Dr. Torsten Braun, Dr. Tuan Anh Trinh, Dr. I. Aad
06 January, 2014

University of Bern [1]
Msc. Computer Science • 3012 Bern • Switzerland

1. University of , www.unibe.ch

Table of Contents

Abstract

In this paper, we discuss the possibility to use the BitTorrent protocol as a way to distribute a workload to be processed by multiple nodes. We create our own implementation and use the rainbow tables generation process as a test application. We present our results, the torrent library we chose to enhance as well as the modifications we operated on it.

1 Introduction

Moore's law applies to our current state of technologies with periodic improvements of the computing power available in a single processing unit. Along side this evolution goes the always growing need for more processing power as scientists can count on large improvements on the hardware level within a couple of years. Since we expect such an improvement to happen, we tend to design algorithms with higher accuracy and number of input parameters to process. Unfortunately a single processing unit is still far from providing enough processing power to solve such problems, whereas distributing the work on many computers can.

Following this spirit we can consider distributed computing as a solution that comes with a price : the efficiency / performances ratio. For a workload to be efficiently processed, we need a work that is splitable in sub-tasks, each computable independently (no precedence of computation issues). Another liability remains in the way to gather back all computations. This is where the BitTorrent protocol can be helpful : a torrent file is split in a given number of files, each split in a given number of pieces. Each piece can already be downloaded or not.Provided that the two conditions explained above for distributed computing are granted, we could transform a torrent that downloads the rarest pieces into a torrent that generates the rarest pieces.

BitTorrent is often considered in distribued computing, but only as a mean to store large amounts of data. Our goal for this project is to actively use the BitTorrent protocol to transform each servent node into a computing node that would make the pieces it computed available to other nodes for download. An application requiring a significant amount of processing power and that can easily be split is rainbow tables generation, which we will use as a proof of concept application.

In this report, we will explore what are rainbow tables and how they can be generated on multiple machines. We will also present the modifications we operated on the existing BitTorrent protocol to allow us to piggyback computed data from the nodes. Finally, we will present the results of our implementation.

2 Tools and libraries

We chose to use Java as main programming language for this project since it is adequate to develop a prototype quickly. Moreover there are already many existing libraries that handle the generic BitTorrent protocol written in Java.

To develop this project, we used the following programs :

- Eclipse IDE
- RTGen (RainbowCrack [1])
- Bencode editor v0710u [2]
- GitHub
- LaTeX

1. Source : http://www.project-rainbowcrack.com/
2. Source : https://sites.google.com/site/ultimasites/bencode-editor

We chose to start with "**Turn's BitTorrent**" [3] library because it is an open-source project (Apache License 2.0) and is easy-to-embed in any program. Ttorrent has also a very clean code and provides a tracker / client source code already.

3 Distributed computation with BitTorrent

The BitTorrent protocol is used to download files from a network made of nodes which are both servers and clients. Servers because they have a list of peers who share the same files and clients because they are looking to download these same files. The initial information is contained in the .torrent file (tracker URI, files, piece size, etc). Let us denote by **RTTorrent** a torrent file that holds specific additional information for rainbow tables generation.

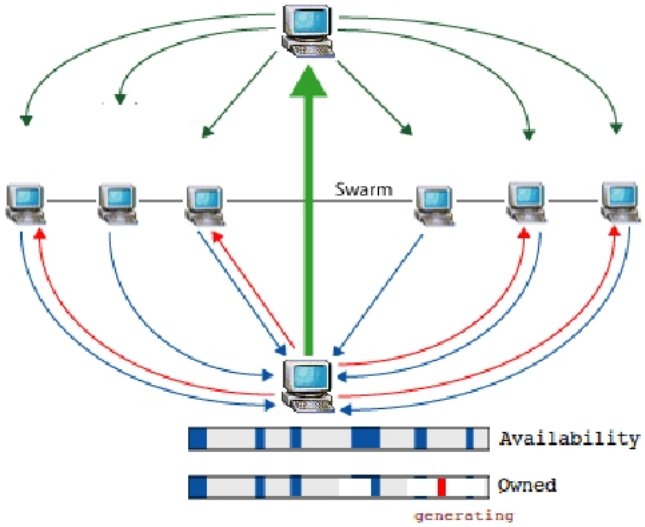

FIGURE 1 – RTTorrent pinciple

Each time a new torrent is launched, the client will map its neighbors thanks to the tracker and compute the availability of each piece of the files to download. The rarest piece is selected and downloaded, then marked as "Have" to let other know this piece is available from our client now. If no peer has a piece, we have to wait until a new peer connects and has the piece we are looking for. With RTTorrent, the difference is that if a piece is not available we can generate it by launching an instance of RTGen on the node (cf. figure 1). Once the piece is computed, the peers can download it (no need to generate it).

4 Modifications to Turn's library

This section describes all the modifications we had to operate to tranform Turn's library into a rainbow tables torrent library. Note that we have specified in the code which part we have altered to enable support for the new features.

4.1 Torrent file specification

To enable a torrent to carry the required information to compute hash tables, we had to modify the specification of the .torrent file. We added the following fields to the already existing BitTorrent fields :

3. Source : http://mpetazzoni.github.io/ttorrent

Field	Description
chainLength	The rainbow chain length. Longer chains store more plaintexts (longer to generate)
plaintextLenMin	These two parameters limit the plaintext length range of the rainbow table
plaintextLenMax	These two parameters limit the plaintext length range of the rainbow table
pieceLength	The number of bytes in each piece
charset	The charset includes all possible characters for the plaintext
hashAlgorithm	Rainbow table is hash algorithm specific

To enable the torrent to recognize these fields that are not specified in the original BitTorrent protocol, we had to modify the parsing of the .torrent files. In addition to the previously mentioned fields, we added a "table index" field in each file that the torrent is supposed to download. This parameter selects the reduction function. Rainbow table with different table index parameter uses different reduction function.

4.2 Rainbow tables generation

Each time a piece is unavailable for download among all the connected peers, the RTTorrent starts generating one. We had to implement a new thread for the generation aside the regular download threads. If a piece is missing, we randomly select one of the remaining pieces (see chapter 4.3) to download and start generating the related rainbow table.

To generate a piece, we use the RainbowCrack Project RTGen that we can call from a dedicated thread on any client node. The piece to compute is selected at random from the array of unavailable pieces. Then we provide RTGen with the information on the hash algorithm, charset, plain text minimum and maximum length and the chain length, all previously stored in the .torrent file.

Once the required tables are generated, they are stored in the data byte array of a new piece, which in turn is recorded in the general files of the torrent. Once the piece is secured we can treat it as a completed download by tagging that piece as "complete" and by sending to all connected peers a "Have" message so they are aware that this piece is available for download.

4.3 Piece selection and validation

In a regular Torrent, the selection of the next piece to download is made upon a set of rarest pieces among the peers connected to a given node. Each time a piece is downloaded and validated, this piece is removed from the rarest list and the next one starts downloading. As we don't have the problem of having nonexistent pieces with RTTorrent (since we can generate them), we have to find new strategies to select the next piece to generate :

1. Choose first piece from k pieces where k is unavailable. This translates into picking the first missing piece in the vector of missing pieces. This means that almost every peer will select the same piece, thus yielding a nearly 100% collision rate.
2. Choose random piece from k pieces. When not optimal, it drastically reduced the collision rate at the beginning of the process, but the more complete the RTTorrent, the more collisions will happen.
3. Choose a piece k based on peers agreement using communication. This synchronizes the generation by keeping an internal tracking of the currently generated pieces. This is the most efficient solution, but it induces more traffic and validations for each node.
4. Choose piece from k based on neighbors peer IDs. This achieves relatively the same as strategy 3, but no communication is required because we compute wich peer ID compared to our own ID computes which piece. There are no additional communications induced.

This project being a proof of concept, we chose strategy number 2 for its ease of implementation compared to its performances. The best solution would be strategy number 4 though for optimal performances and robustness.

Compared to a common Torrent, the RTTorrent cannot operate a regular validation on the chunks downloaded for two reasons : first the pieces that do not exist are generated, which means we do not know their hash to validate them. Second validating received chunks means recomputing them, which defeats the purpose of distributing the workload. Therefore we set valid field of each piece downloaded to true.

4.4 Torrent client

With all the previously mentioned modifications brought to the code, the regular torrent client could not work anymore without adaptations :

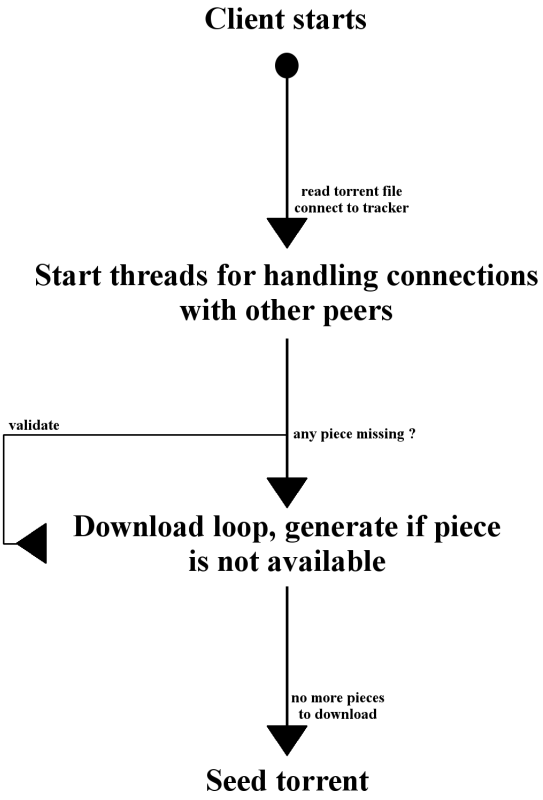

FIGURE 2 – Client workflow

In the original Turn's library, the client starts by initializing a "ShareTorrent" object from a torrent meta-info source (either a file or a byte array, see com.turn.ttorrent.SharedTorrent). Then, it instantiates the Client object with this SharedTorrent and downloads the torrent and continues seeding for the given amount of time after the download completes.

For this project we had to modify the download process so that instead of waiting for a piece that is unavailable, we start generating it via a callable execution.

TO COMPLETE

5 Experiments and results

To test our library, we created a custom torrent containing two file, each made of 32 pieces. Each piece weighs one MB. We chose to use small pieces with a small charset length for the purpose of demonstration : with such parameters, we can generate a new piece roughly every 15 seconds.

No tit-for-that

6 Improvements

Being a proof of concept, this project's main focus is to provide a working RTTorrent library on which various improvements such as the following ones can be made :

Interrupt exception : it may happen that a piece becomes available while we are generating it. This would occur if a given piece is not downloadable at time X so that the node starts generating it and finishes at time Z, but another node finishes to generate that same piece at time Y. To avoid such waste of processing power, we could implement an interrupt exception in case we receive a message "Have" from another peer on a piece we are currently generating.

Pause/resume state : Torrent networks often get peers connecting and disconnecting, not always in a gentle manner. Considering that generating a piece takes more time than downloading the generated piece, it would be better to be able to resume state on application reload.

Piece selection : as discussed in chapter 4.3, we are currently selecting randomly which will be the next piece to generate on a node. This strategy loses performances the closer we are to the end of the torrent's download. Strategy number 3 or 4 should be considered instead, depending on what are the needs for the final application.

Piece validation : currently no verification can be operated since the time required to verify a piece is greater than the generation time of that same piece. A solution could be to download randomly pieces and re-generate them, then compare with the downloaded ones. Also some pieces could be asked from multiple different peers and compared, which spares the regeneration time waste. The each peer could tag a bad peer so others stop downloading from it.

7 Conclusions

Through this project we have proven that the BitTorrent protocol, given the right modifications, can accommodate distributed computations. Not only the workload is shared among the peers, but the original purpose of BitTorrent allows to share and store quickly the generated chunks together. It is also interesting to notice that a regular tracker can provide the bootstrap for RTTorrents and BitTorrents alike. Only the client is different since it has to be able to read the extra information contained in an RTTorrent and dedicate the generation process to an external software.

From a more conceptual perspective RTTorrents are an example of how distributed computations can be achieved via BitTorrent. Nevertheless all types of computations may not apply correctly to this process since we need a way to generate pieces resulting from the computation that have the same length.

Must trust the clients.