b
UNIVERSITÄT
BERN

Distributed generation of rainbow tables using a P2P network

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 a P2P protocol (specifically BitTorrent) 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 adapt 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 content is 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, it is possible to generate pieces and then share them using the BitTorrent protocol.

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 peer 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 gather computed data from the nodes. Finally, we will present the results of our implementation.

Please note that another project on distributed rainbow table exists : distrrtgen (http://boinc.freerainbowtables.com/distrrtgen/). This project is based on BOINC, a middleware system for grid computing. Unlike this project which relies on P2P, distrrtgen uses client-server model.

2 Tools and libraries

We chose to use Java as main programming language for this project since the team is already experienced with this language. Moreover there are already many existing libraries that handle the generic BitTorrent protocol written in Java.

To develop this project, we used the following tools :

- 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 well commented clean code and provides a tracker / client source code already.

3 Distributed computation with BitTorrent

3.1 Principles

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 **RTorrent** a torrent file that holds specific additional information for rainbow tables generation.

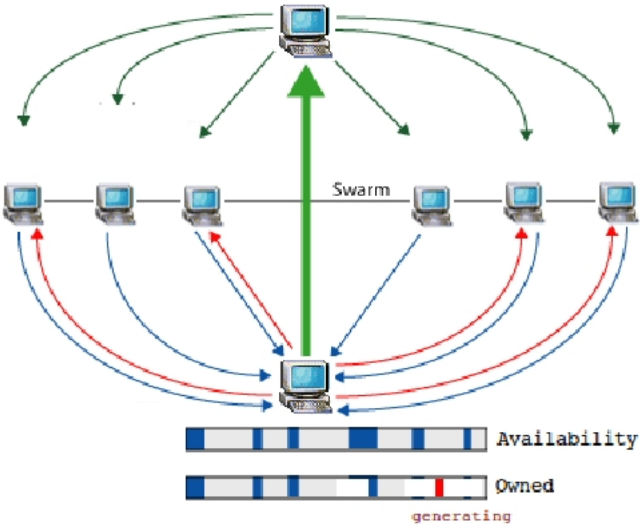


FIGURE 1 – RTorrent principle

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 RTorrent, 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 2). Once the piece is computed, the peers can share it (no need to generate it).

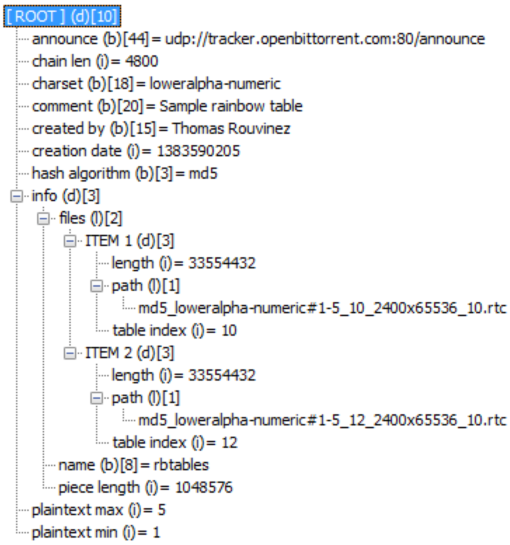
3.2 Torrent file specification

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 :

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

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

In addition to this, pieces hashes were removed due to the fact that they are not known in advance. This is not possible to verify downloaded pieces anymore without regenerating them.



```
[ ROOT ] (d)[10]
  announce (b)[44] = udp://tracker.openbittorrent.com:80/announce
  chain len (i)= 4800
  charset (b)[18] = loweralpha-numeric
  comment (b)[20] = Sample rainbow table
  created by (b)[15] = Thomas Rouvinez
  creation date (i)= 1383590205
  hash algorithm (b)[3] = md5
  info (d)[3]
    files (l)[2]
      ITEM 1 (d)[3]
        length (i)= 33554432
        path (l)[1]
          md5_loweralpha-numeric#1-5_10_2400x65536_10.rtc
        table index (i)= 10
      ITEM 2 (d)[3]
        length (i)= 33554432
        path (l)[1]
          md5_loweralpha-numeric#1-5_12_2400x65536_10.rtc
        table index (i)= 12
    name (b)[8] = rbtables
    piece length (i)= 1048576
  plaintext max (i)= 5
  plaintext min (i)= 1
```

FIGURE 2 – RTorrent sample file content

The complete specification is available on project wiki : http://github.com/DurandA/drttor/wiki/Rtorrent-file-specification

4 Implementation

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.

Complete source code is available here : http://github.com/DurandA/drttor/

4.1 Rainbow tables generation

Each time a piece is unavailable for download among all the connected peers, the RTTorrent starts generating one (see chapter 4.3).

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

4.2 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 and next one is selected then starts downloading. RTorrents don't have the problem of having nonexistent pieces (since we can generate them). RTorrents clients (like our own) have to use specific strategies to select the next piece to generate. Here are some example strategies :

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. Very naive implementation. The more complete the RTorrent, 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. Using mathematical function based on ordering neighbours peer IDs and comparing own ID, it is possible to avoid collisions without specific communications. However this does not apply to larger pools as the so created connectivity structure would not complete.

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 3 though for optimal performances and robustness.

4.3 Java client implementation

For this project we had to modify the library workflow so that instead of waiting for pieces that are unavailable, we start generating them as soon as the torrent is initialized via a dedicated thread :

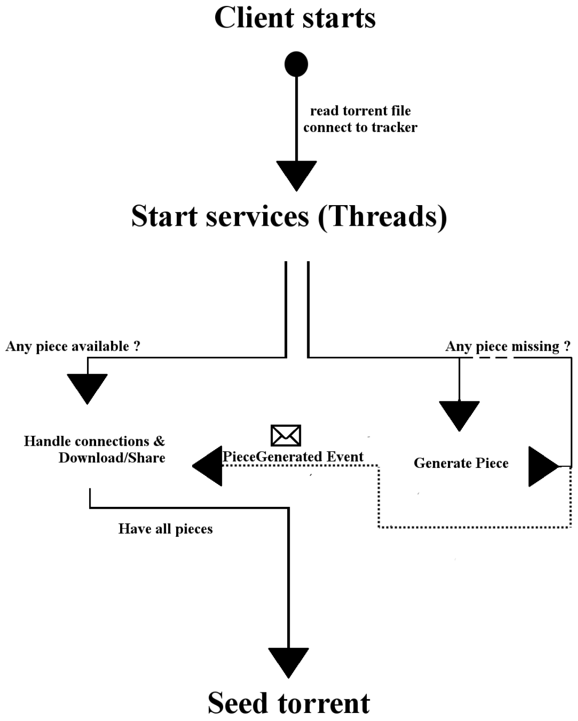


FIGURE 3 – Client workflow

In the original Turn's library, the client starts by initializing a torrent `T<SharedTorrent>` 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 from the associated `SharedTorrent`.

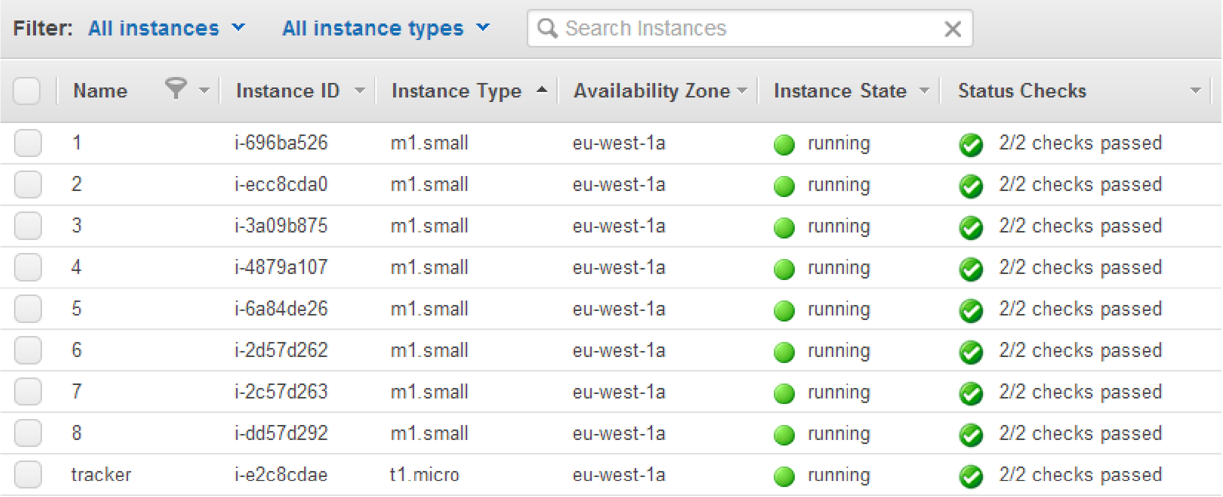
Client object contains the main logic of the application. Constructor starts threads for handling connection with other peers (service `T<ConnectionHandler>`) and with tracker (announce `T<Announce>`. It was modified to start generator T<RTGenerator> after the who mentioned services.

RTGenerator class was not part of the original library and generates pieces using random piece from getUnavailablePieces() available from the SharedTorrent instance. The generator is executed until there is no more pieces unavailable on the Torrent swarm. Generating pieces is done with external process call (equivalent to exec) to rtgen using parameters relative to the specific piece. After having generated a piece, the generator writes the piece on the associated chunk of the rainbow table file and then calls firePieceCompleted() to notice observer(s) (which is the client) that a piece generation is completed.

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.

To test our application we have used Amazon EC2 instances[4]. To simulate a torrent-like network, we used 9 EC2 instances : one micro instance for the bittorrent tracker and up to eight small instances for the peers that generate and share the pieces of the table. We have tested with 2, 4, and 8 peers to follow a logarithmic growth. Figure 4 shows the 8 peer instances running under Windows Server 2012, each with CPU capacity of one EC2 Compute Unit. We choose settings for the generation that suited well our hardware configuration : 60 pieces each 1 MB small. The time needed to generate it on one machine is 70 minutes.



□	Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks
□	1	i-696ba526	m1.small	eu-west-1a	● running	✓ 2/2 checks passed
□	2	i-ecc8cda0	m1.small	eu-west-1a	● running	✓ 2/2 checks passed
□	3	i-3a09b875	m1.small	eu-west-1a	● running	✓ 2/2 checks passed
□	4	i-4879a107	m1.small	eu-west-1a	● running	✓ 2/2 checks passed
□	5	i-6a84de26	m1.small	eu-west-1a	● running	✓ 2/2 checks passed
□	6	i-2d57d262	m1.small	eu-west-1a	● running	✓ 2/2 checks passed
□	7	i-2c57d263	m1.small	eu-west-1a	● running	✓ 2/2 checks passed
□	8	i-dd57d292	m1.small	eu-west-1a	● running	✓ 2/2 checks passed
□	tracker	i-e2c8cdae	t1.micro	eu-west-1a	● running	✓ 2/2 checks passed

FIGURE 4 – EC2 instances

Figure 5 depicts the time of completion (combination of download and generation) of the same torrent. We can see that it takes 70 minutes with a single machine and for example 12 minutes with 8 machines. We draw the conclusions that the generation time is significantly decreased with each added machine :

[4] See http://aws.amazon.com/ec2/instance-types/

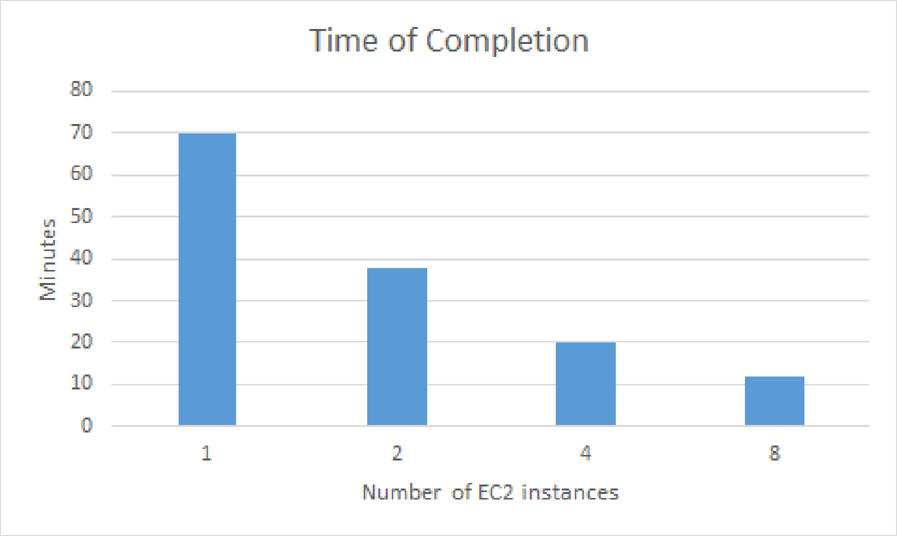


FIGURE 5 – Computation time improvements

Figure 6 renders the theoretical maximum acceleration with each added machine compared to the real EC2 experience. With random piece selection technique (blue line) we see a growing gap because each machine choose randomly which currently unavailable piece to generate and at the same time an other may also select the same piece. This yields duplicates. With 8 machines, instead of decreasing the total generation time by a factor of 8 it is divided by a factor closer to 6.

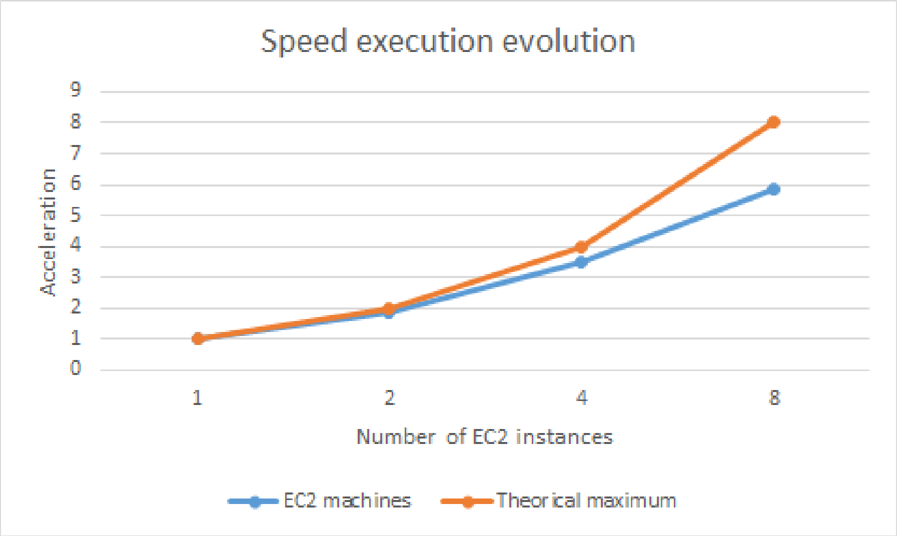


FIGURE 6 – Execution speed improvements

6 Drawbacks

A few drawbacks are inherent to P2P distributed computing and sharing mechanisms :

No chunk verification : Verifying chunks is as costly as generating them. It is possible to measure trustfulness statistically by randomly analysing incoming chunks but it is very costly and does not ensure 100% uncorrupted data. Another approach would be to have a voting system between peers to trust other peers. But it would be easy to cheat by having a lot of peers connected to the network.

Tit-for-that inefficiency : Tit-for-tat mechanism is suited for bandwith fairness and does not care about dedicated computing power. There is not tracking about who generated a chunk. Enabling this mechanism for RTorrents distribution is inefficient.

7 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.

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.

Chunk 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.

8 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. The closest work we found on this topic was DistrRTgen[5] which distributively computes rainbow tables in a classical manner and then upload their generated content to a server. This is the main advantage of our implementation as we combine generation and sharing, thus automating file management and easing work distribution. It is also interesting to notice that neither the BitTorrent exchange protocol nor the BitTorrent tracker protocol required changes to accomodate our RTorrent. This enables our client to use any generic tracker.

From a more conceptual perspective RTorrents 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 pre-computed chunk sizes to make it work. With the actual computation dedicated to a callable program it is possible to adapt this library to other types of computations than rainbow tables generation. Given a cluster of servers with high processing power and direct links between them, RTorrents performances and inherent data sharing can improve performances and reduce the data gathering overhead. But this application could become handy when it comes to crowd processing power use for massive computations.

[5] See https://www.freerainbowtables.com/