Vrije Universiteit Amsterdam
Faculteit der Exacte Wetenschappen

# Efficient and non-intrusive data distribution for web application clusters

Master Thesis by Michal Tekeľ

Supervisors: Dr.-Ing. habil. Thilo Kielmann, Ramon van Alteren

Second reader: Guillaume Pierre

Department of Computer Science,
Parallel and Distributed Computer Systems

2011-08-31,
Amsterdam

# Acknowledgements

I wish to express my sincere gratitude and appreciation to my supervisors, *dr. Thilo Kielmann*, for accepting this role even though he was already very busy with other students and his work duties, for his guidance, suggestions and feedback he provided, and to *Ramon van Alteren*, who was supervising me at Hyves. I would like to thank him for his thoughtful guidance, his valuable suggestions and comments during our discussions, his response to my questions and feedback he provided. I would also like to give him my appreciation for providing me with his support and creating the environment necessary for completing this thesis and for giving me the opportunity, together with *Bo Roos Lindgreen*, do to my internship at Hyves.

I would also like to thank *Guillaume Pierre*, my second reader, for his prompt reactions to my questions, his feedback, suggestions and willingness to read and comment various drafts of this thesis.

I would like to give my thanks to *Nick Loeve*, *Jeffrey Lensen* and all other colleagues at Hyves, who have been willing to cooperate and have helped me to solve and overcome various unexpected issues when I was developing the file transfer framework, as well as for their various suggestions and mainly the great working environment they all contributed to.

My last thanks go to *dr. Mathijs den Burger*, whose PhD thesis has provided me with valuable information and high quality research done in the area very closely related to my own work and has given me valuable guidance.

# Contents

Title: Efficient and non-intrusive data distribution for web application clusters
Author: *Michal Tekeľ*
Supervisors: *dr. Thilo Kielmann, Ramon van Alteren*
Second reader: *Guillaume Pierre*
Department: Computer Science, Parallel and Distributed Computer Systems

Abstract:

Various current large scale on-line applications and web sites which run on large clusters/clouds require to efficiently distribute data (usually) from one source to the groups of/all nodes. This concerns usually various application content updates, software and security patches deploying, backups and similar. It is very important that such data distribution does not interfere with the application responsiveness and performance, so that it remains transparent to the external clients. One possibility to achieve this is to simply slow the transfers down to some safe limit. This solution doesn't however scale and it can then take several hours to deploy even smaller update/security patch, which is too long and unacceptable.

In this thesis we have developed a scaleable and cluster environment sensitive file transfer framework, which brings down such transfers to just a couple of minutes, while still being non-intrusive. We have achieved this via collecting bandwidth and resource consumption feedback from single nodes as well as network routers. We have also taken additional care to use the available bandwidth efficiently and thus to further reduce the data delivery times. To achieve this, we have modified bit torrent p2p client and also extended the protocol itself. We have developed a distributed tracker which creates a dynamical, available bandwidth and network topology aware overlay, which is then used to determine the data transfer paths. Together with advanced techniques like work stealing, we have achieved global workload distribution among the recipient set of nodes. The framework itself is fault tolerant and is able to handle and react to node failures in real time. We evaluate and compare our framework against other file transfer methods for clusters.

Keywords:
p2p, distributed tracker, network bandwidth and topology aware, environment feedback

# 1 Introduction

Information technology has been improving our lives for quite some time. Initially via increasing our productivity, but also enabling us to do new things, like space flights. Leisure and entertainment followed soon. In the late 20th century we have seen huge improvement in communication – internet and mobile phones. Today in the web 2.0 age, we see large spread of social networks – people want to be in touch, see how others are doing and maybe also show off a little. While IT wasn't always perfect, user experience has vastly improved in the recent times – people are used to and expect things to work. However those who have insight into IT realize the complexity of these systems and know that high quality is not so easy to achieve. Let's take for example these social networks: millions of concurrently connected users are chatting, viewing and uploading pictures or videos, playing games against each other, posting new messages etc. simultaneously. Long gone are the times of the 4 second rule [1], today everything has to happen instantly – the computers and networks have become fast enough to provide and learn user to expect fast responses. Therefore it is very important that such experience is also delivered by those who run these social sites in their datacenters. This is not a trivial task. The easy solution – throwing more hardware at it simply doesn't work – there are not enough resources available: most social networks are for free and their income comes mainly from selling advertising space. This leads to the necessity of being very effective – starting with hardware, through datacenter and network design, continuing with backend software and services to the front-end web page outline itself. A lot of research and new solutions have been developed in all of these areas recently, however there's not enough space to cover all, or even just a few of them in this text. The focus of this thesis is on the area of the back end services, specifically on the file transfers within and among the clusters.

## 1.1 Internal data transfers in web clusters

The data that's being transferred in the clusters has two origins – the primary are the users themselves – web page requests and responses, picture/video up/downloads, chat messages etc. The second source of data is the internal backend services – on-line backups, database replication, deployment of new content to the servers and similar. Whereas the primary "front-end" data should always have higher priority, also backend services often require to distribute content in a timely manner – e.g. when deploying new code with some important security fixes. It is also necessary that the network infrastructure itself is well designed and has enough capacity. Here again the efficiency has more priority over the (costly) raw bandwidth: if you don't distribute your services among racks and locations well, you can easily create a hot-spot and overload the shared links. As the traffic grows, packets are getting slowly congested (even before 100% link utilization, figure 1) and spend more and more time waiting in the queues in the routers. This introduces the unacceptable additional latency, or even chokes the connection altogether for some time – when the queues run out of capacity and routers start to drop the packets. This results in the interruption of the service. Such an overload of the network infrastructure can easily happen when you need to do urgent, high priority file transfer to most of the servers in

your deployment. Fortunately in comparison with the primary source of traffic, you have much more control over this process. There are many methods and ways how to such transfers, they differ in where from, where to and when they transfer the data and in the way they utilize bandwidth. It's relatively easy to do a fast transfer, but overloading the network capacity at the same time. It is also not so complicated to transfer files non intrusively (without introducing any additional latency) – by slowing the transfers down, up to the point where the whole process is unacceptably slow and thus useless. The main focus of this thesis is to provide a solution for file transfers within and among clusters which are running latency sensitive applications. We aim for very high efficiency – meaning achieving shortest possible transfer times, but still having no negative impact on the other data transferred in the network. We will first have a brief look on the current ways and methods for file transfers and then describe our approach and proposed solution.

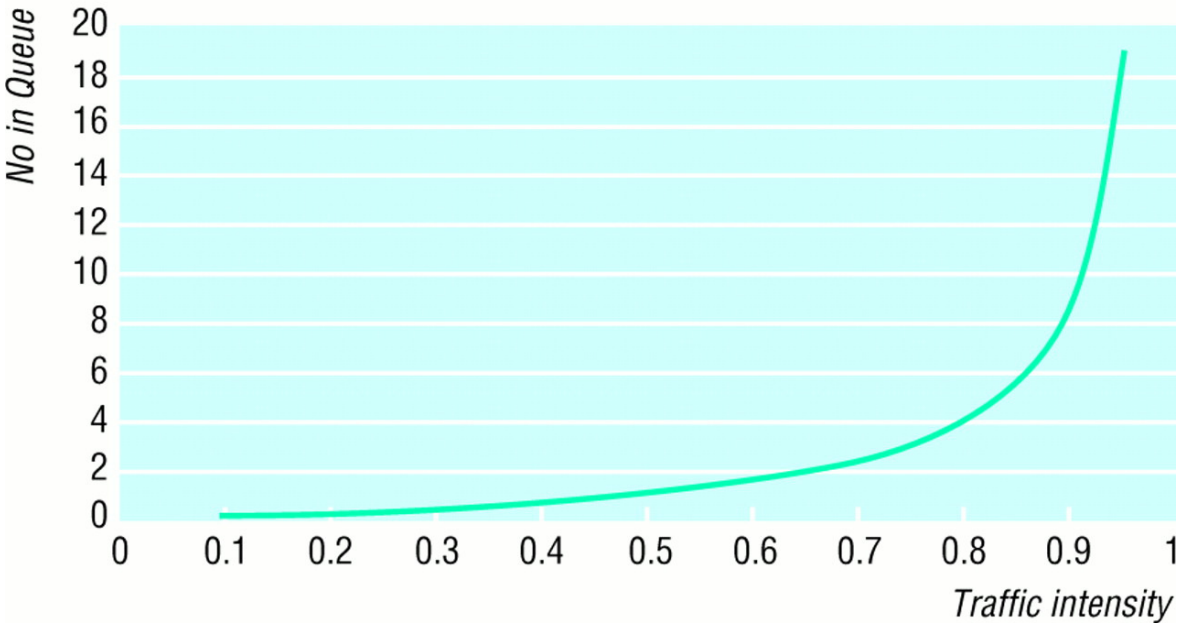



Figure 1 – number of queued packets depending on link utilization

## 1.2 **Problem description**

We have developed this project with cooperation, and specifically for the needs of Hyves.nl social network, which is one of the largest websites in the Netherlands. Currently, the site has over 10 million members and reaches more than 55% of the Dutch population. Features include: profiles, photo's, blogs, messages, chat, schools, companies and music. Hyves can be used on a browser, a mobile phone and a desktop client. The Hyves platform is characterized by high-volume traffic and cutting-edge web 2.0 technology. More than 3000 Linux servers, located in 3 datacenters around Amsterdam, generate 8.3 billion page views per month. Currently, the Hyves platform stores more than 350 million photos and

videos, and each month 30 million more are uploaded. Where other companies focus on configuration of the systems, Hyves focuses on ensuring optimal performance of all available systems and their integration. To realize that in a scalable manner, Hyves makes extensive use of cutting edge open source technologies.

One of the most common activities within their technology department is deploying code and/or content to server environments. This deals with deploying website source-code to groups of target servers in OTAP environments and deploying (database) content over database servers. In all cases this usually involves distributing one or more large files from a single originator to several (groups of) target nodes, the amount of target-nodes can vary between 1-700+ nodes. All this of course has to happen while the site is serving the content and without influencing its performance by putting too high load on any of the components, mainly the network infrastructure. Considering that deployment of code & content (payload) is such a regular activity they would like to use a method that spreads the payload of the target nodes. They expect this would reduce the total time required to deliver the payload to target nodes which is beneficiary for agile testing, rolling out critical (security) bug fixes and would reduce any downtime required for deployment of new releases. In addition they also require having capability to control the bandwidth and other resource consumption of the file transfer both at the source(s) and target(s). They would like the solution to be aware of network and geographical topology and possibly use it to its advantage. The solution has to be open-source or public domain and preferably unencumbered by patents and Linux based, so that it can be easily integrated into their existing environment. Also content and sender/receiver verification and authentication is expected. Hyves would like the solution to be flexible enough, so that it would be applicable to a larger set of use-cases than the main one mentioned above, they strongly prefer an implementation that is pluggable and can be reused in different environments.

In addition to the requirements mentioned above they would like the end-product to have ability to trigger transfer on the receiving side and lowest possible system footprint to allow inclusion in our install-framework. Possibility to create policy-based rules which deal with the availability of content in a group of participating nodes is seen as very useful. If certain content fails to meet the policy requirements, additional nodes should be added automatically. They also see possibilities to apply the requested solution to the few more other problem domains within system engineering, which we were explicitly requested to consider:

- Installation of OS-images on bare-metal or virtual nodes
- Backup and restoring of (database) content
- Redundancy-requirements in the media storage system
- Seeding database groups with content

## 1.3 **Our approach and the solution**

Based on the requirements of Hyves, we did research which methods would be best suited for their needs. This is covered in chapter 2, where we characterize selected solutions. The selection is made so that each of them shows one important feature or approach, or to show one major drawback. The second chapter thus doesn't cover all the research we did, but gives the reader the necessary overview and understanding of the

applicable transfer methods. Initially we wanted to base our solution on the advanced algorithms presented in [5], where the implementation needed to add few more features, like robustness and some more resiliency related to local environment changes (e.g. a node application crashes permanently; or is restarted but the local environment and transfer situation has changed a lot since last running state). After further discussion with Hyves we have decided to take different approach. The features of robustness, agility and resiliency were so important for them, that they preferred to start with solution that already has these and only add new features and characteristics to it, which improve its performance and reduce environment impact. Another drawback of [5] was that the implementation depended on some services, like IBIS [16], which would have to be introduced into Hyves environment and presented another possible point of failure.

In the end we have decided to base our solution on the peer-to-peer approach, more specifically on the Bit Torrent protocol [14]. We did not implement our own client from scratch, instead we chose to use one of the existing and extend it with functionality specific to our solution. We discuss the client selection in section 3.2. Also we couldn't easily just deploy the existing unmodified client, because this would easily flood the network – with very bad impact on performance. As [15] shows, even few nodes running bit torrent transfer already have influence on web page load times, whereas running the transfer on all nodes (one of our use case scenarios) made the times jump from few milliseconds without transfer to 3+ seconds while doing transfer. In case of Hyves, given how much more traffic there is internally, such a "flooding" transfer would mean entirely disrupting the operations. Therefore not only local, but most importantly global network utilization control was a must with our solution. We have tried several approaches of controlling the bandwidth utilization at the client side, but in the end we have created a distributed network topology and bandwidth aware tracker – an application which controls which peers connect to which others and how much bandwidth they can use for such connections. More details about that in chapter 4. This includes complete description of this concept and comparison to some other possibilities of controlling the global bandwidth. We also discuss the reasons of several properties of our solution here. Chapter 3 instead contains only information connected to the Bit Torrent protocol. This includes the technical description of the protocol specification – not too broad, but detailed enough for the reader to understand the principles of operation. We also describe enhancements we did to the protocol and the client itself here. In chapter 5 we evaluate our solution and finally, in chapter 6 we conclude and present possibilities for further improvements.

# 2 Current solutions of data distribution

The current methods range from plain point to point transfers, via creating simpler distribution/replication networks, computing optimal distribution trees, to most recent peer to peer based solutions. We are concerning only transfer of files in this thesis, not any replication of databases or other sort of communication between applications themselves, although some concepts we will present are general and flexible enough to accommodate these uses as well. Let's consider now a basic scenario, where we want to distribute a file from one (or few) source(s) to many recipients. In practice this can be for example restoration of a binary database image to the DB cluster (large file, relatively few recipients), a deployment of a new static content to the web farm (medium file, numerous recipients), or a quick security patch for the media storage servers (small file, lots of recipients). In general, we can divide solutions into **pushed** (initiator of the transfer pushes the data to the destinations), **pulled** (recipients download data from the source) **and other**, where participants pull the data in some phases of the transfer, and push in others, or do both pushing and pulling at the same time – this will be described in further detail in each relevant sub-chapter:

## 2.1 Point to point transfers

Point to point transfers present the simplest way to deliver the data. Most of the times there's one source machine from which the data is simply copied by **scp** [2], netcat [3] or similar utilities. **Rsync** [4] represents much more advanced tool to "sync" files, which is using delta-transfer algorithm, which reduces the amount of data sent over the network by **sending only the differences** between the source files and the existing files in the destination. All these and alike utilities can be used both in push and pull configuration – depending on how is the transfer executed/scripted. The most easy way, copy from one source to destinations one by one, is not as advantageous as to copy to (or from) all destinations at the same time. In both cases the bottleneck is presented by the source's upload capacity. However in the first case destinations experience one by one intensive transfer for a short time. This can lead to a (minor) service interruption. Also if one of the recipients is not able to download data as fast as the source can upload them, or even crashes/times out during the transfer, then it is delaying data delivery for the following recipients. Whereas if data is uploaded to (or downloaded by) all recipients at the same time, this presents only small load on each recipient (and network link) and will not likely cause any service interference. Also the broken recipients don't influence or slow down transfer for others. And in the end, most of the recipients tend to finish the transfer around the same time – this can be advantageous if we want to do some synchronized deploys or updates. Also usual means of HTTP or FTP servers can be used to deliver the data. However this time only in pulled manner, most likely having many concurrent downloaders, which as we have shown is rather a preferable choice.

Disadvantage of point to point transfers is that the data is duplicated only in the source, therefore the source has to upload the whole file (or the difference in data if we are using Rsync) n-times. The obvious bottleneck is source's upload capacity, which can be

somewhat compensated by setting up load-balanced clusters of HTTP or FTP servers – but bringing only linear speedup. In the end, if we have too many recipients (let's say more than 100), this solution is very slow and thus unusable. Setting up source clusters brings additional complication, resource (servers) consumption and configurational complexity, negating the main advantage of point to point transfers – simplicity.

## 2.2 **Distribution trees**

Distribution trees determine the way how data "flows" from the source to the recipients. The most simple form can be seen as setting up your own "Rsync replication network" – you have one (or few) master servers, then a layer or two of slaves and then the recipients. Each of them periodically runs a cron-job to sync the data with upper layer. Because of this implementation, this solution is more suitable for low priority background transfers. Even though the file has to be pulled via several hosts before it reaches its destination, this can be already faster than point-to-point transfers – if the involved nodes number is large enough. The advantage is that the data duplication is happening at more that one node and the cumulative upload bandwidth of involved slaves is much larger than that of sole master.

The more advanced approaches **split the data into multiple smaller pieces**, so that the intermediate nodes can start forwarding and duplicating as soon as possible. This way the **data is streamed from source to recipients, being replicated on the way**. The distribution tree is built according to every solution's rules and algorithms. The resulting trees can have various shapes:

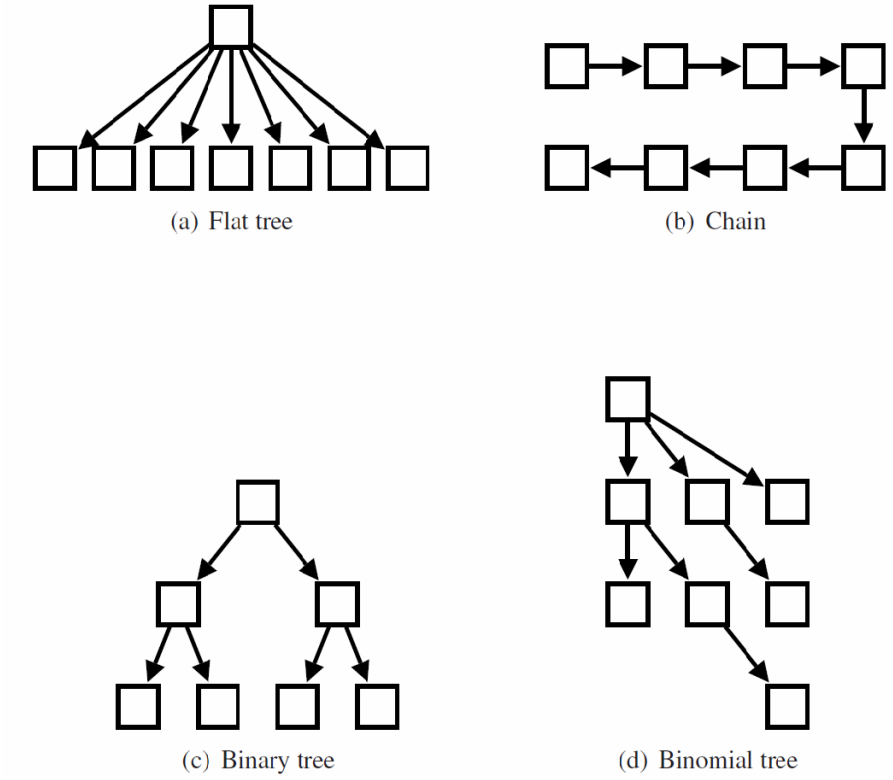



Figure 2 – Common static multicast tree shapes [fig. 3.1 in 5]

All the nodes in the given layer can get the data only as fast as the upper layer and links to its nodes can provide. Each new layer of the tree also represents an opportunity to grow the number of nodes that will serve as sources for the next layer. Thus if the link speed between the nodes is always the same, the layer that adds nodes will receive data slower (since upper layer has less uploaders), but the next layer will then have more sources available, so it will get the data faster. From this point of view, chain topology (b) will never lose (or gain) any speed. In reality this tree is most disadvantageous – links don't have equal or even stable speeds, slow link slows delivery for all the following nodes and one failed link or node will break the chain. The flat tree (a) represents the situation we have suggested in the point-to-point transfers – one source is pushing data to many recipients. The other two trees (c) and (d) represent rather a theoretical example – in reality a good distribution tree's shape is determined more by the nodes' and links' bandwidth capacities. However if we only need to broadcast some simpler short messages, with such frequency that they never interfere with other traffic, then these tree shapes are still suitable to determine communication routing.

One of the possible methods to spread the same data to multiple nodes is to use **IP multicast – a protocol level method** which **requires support of the network infrastructure** (routers and switches) to work. The whole path from the source to recipient has to be IP multicast enabled. The advantage with is that this method is very efficient: data duplication happens at the network devices and the distribution tree is the same as the network topology. However, there is no per-link speed control. The data is duplicated to all the links which lead to recipients at the same speed the data source node is broadcasting. If one of the links has more traffic, it can become overloaded. Also, all the recipients have to be able to accept the data at the initiator's upload rate. If they are slower, they will miss the data. If we want to ensure that no links will be overloaded and that all the hosts are able to accept the data, we have to broadcast at the slowest common link speed and host download rates. This can mean a considerable slowdown – one overloaded connection or node is hindering performance for everyone. Even worse, if one of the nodes joins the transfer too late, or crashes during transfer and after restart wants to re-join, it will miss the data and the whole transfer has to be done again. Because of these reasons IP multicast is used in practice only for distributing such data, where it doesn't matter if you miss some: audio and video broadcasts. Another disadvantage is that because the network devices have to do the data duplication, this increases their workload and can possibly introduce some latencies for live traffic.

## 2.3 **Optimized multicasting trees**

There is whole range of algorithms and solutions that are constructing optimized multicast trees based on the link and nodes capacities. The introduction in chapter 3 of [5] gives a good overview: In Overcast [6] nodes use probes to measure bandwidth to their siblings, based on which they then construct a single bandwidth optimized overlay tree. M-RTP [7] creates the tree based on graph of the network using Real Time Transport protocol streams. Also work presented in [8] is dynamically creating and optimizing multicast tree based on each node's capacity, but not taking link speeds into account, as it considers only node capacity to be a possible bottleneck.

The second class of approaches start to take network topology into account and build a multi layered solutions: MagPIE [9] distinguishes between intra and extra cluster communication, MPICH-G2 library [10] takes WAN, LAN and local communication differences into account, both solutions yielding significant gains over topology aware-less approaches.

All these solutions share the need to find an optimal distribution tree, spanning all the nodes. This task is not always trivial: for finding maximal bottleneck tree we can use for example Jarník's algorithm [11], but this doesn't take local node's capacity into account. If any of the nodes in the tree is not able to receive and transmit at the capacity of the edges tree was computed in mind with, then this node will create a new bottleneck, lowering overall throughput. If we want to compute with consideration of local node's capacities, then we face an NP complete problem, equivalent to finding a Hamiltonian cycle. Some approaches [7] use heuristics to solve this, achieving good results.

Single (max. bottleneck) multicast tree will however often not use all the available bandwidth. The overall throughput can be increased by using multiple distribution trees simultaneously. The FPFR [12] tool uses depth first search to find a tree covering all nodes. Links with no bandwidth left are then removed from graph and the algorithm continues finding alternative spanning trees, until it can't find any. More advanced approaches use linear programming to find an optimal set of trees. Yet these computations are quite complex and can be expensive (considering that we are optimizing not one tree, but looking for a best set of trees). If we want to have a real-time applicable solution, we would need to rely on heuristics. FPFR has another disadvantage – it also doesn't consider local node capacity. All the computed trees get the same share of the source's upload capacity. This often results in less than optimal performance, where fast multicast trees don't saturate their capacity. To solve this problem, [5] presents **Balanced Multicasting** algorithm.

This solution tries to achieve the maximal multicast speed by using linear programming to determine bandwidth allocation of the source to several concurrent distribution paths – trees. If we want to find the optimal solution, we have to consider every possible combination of all the possible trees. The problem is that there exists $n^{n-2}$ of such trees between n hosts (Cayley's formula [13]). Therefore some heuristics have to be used in order to make this approach usable in real world and for larger sets of nodes. The Balanced Multicasting thus uses the same set of trees that are generated by FPFR algorithm. These are already optimal in case that bottleneck is purely in the network speed (and not in local node capacity), or if node capacity instead is much smaller than network speed (then already almost any existing tree saturates node's output speed). In the cases in-between, FPFR still generates trees with low fan-out (out degree), lowering the duplication need and thus possibility of saturating local upload speed. Using this approach, **B**al. **M**ulticast achieves throughput close to the theoretical maximum. Another optimization used for transfers between distant clusters is to split the whole operation into three phases – initially data is transferred quickly over LAN inside source cluster. Then the clusters are abstracted as nodes and BM is used again to determine optimal distribution paths between these. Once the data gets into the destination cluster, BM is used again to provide fastest distribution inside that cluster.

The evaluation of BM in [5] shows that it usually achieves results very close to the possible maximum. Yet this solution also has its limitations. First, it relies on complete and

exact network capacity information, including available bandwidths between all hosts. This is rarely available, moreover requires $n^2$ network characteristics for n hosts. Another disadvantage is that even if you have this information available, this all-to-all graph doesn't capture real world situation where some links are shared and utilization of bandwidth between some hosts influences available bandwidth between other hosts. This of course changes the situation and requires re-computation of paths to achieve the best performance. Taking background traffic, but even other concurrent transfers using BM into account leads us to the conclusion that the initial graph the distribution paths were computed upon was actual only at the time of (and shortly after) the computation. This leads to the need of perpetual recomputation as soon as new network situation snapshot is available. This also means increased computational load for the source node. Also in the time between these complete snapshots, if there's more bandwidth available for some time, it will be not utilized. On the other side, if there's less bandwidth available as previously reported, overall multicast speed will take hit.

To sum things up, the presented solutions which try to achieve best multicast performance by using distribution trees are fast enough to be usable, yet have their own share of shortcomings. The main problem lies in the principle itself – if we want to compute an optimal tree, we need a network model first. Because of the high computational complexity, which doesn't scale well with growing node numbers, good solutions have to use heuristics. Some solutions try to simplify the network model by not taking local node capacities into account, yet this can yield sub-optimal results. Instead, considering real network topology and node locations leads to improved performance. Not all presented algorithms handle node failures well. Adding new nodes during the transfer is also usually not supported. In the best case, this only means a change in the network model and recomputation of new distribution paths. If the solution is implemented well, transfer can then be resumed. In general, multicast tree solutions perform well in stale environments, with little changes during the transfer itself. Given that our solution has to work in a cluster with constantly changing background traffic, we will not base it solely on computing optimal distribution paths. Nevertheless we will utilize some of the ideas used by these algorithms, albeit in a bit different manner.

### 2.4 **Decentralized solutions**

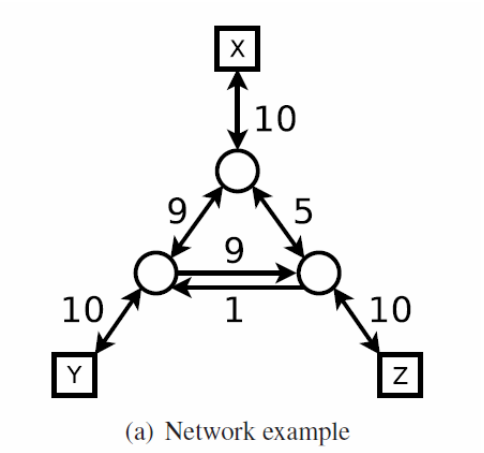
To overcome main disadvantage of the tree optimizing solutions – dependency on the accurate network model, [5] presents **A**daptive **M**ulticasting approach. AM doesn't rely on the network link capacity information, instead it uses backpressure from nodes to determine distribution path. Backpressure from a node is a feedback letting the sender know, that the receiver has reached its forwarding/multiplying capacity (by saturating all it's uplinks to further receivers) and doesn't want to increase incoming bandwidth. This feedback might cause that the sender itself has also saturated its uplinks, so it should look for more receivers, or if it can't find any and further extend its outgoing rate, it should create backpressure to its source. This way the actual distribution path is created in a distributed manner by nodes themselves, trying to saturate their uplinks, thus reaching the maximal possible multicast rate. The path is changed dynamically via backpressure feedback, as the situation changes at different nodes. This unfortunately means that a set of

problems related to message routing have to be solved. Deadlocks can occur when all the possible ways to a given node are stalled. This can be solved by adding new connections, thus creating new paths. Yet the biggest issue is message duplication: when a message arrives at some node and it still has some more recipients left to reach, if that node has two possible uplinks to forward the message to so that it can reach the remaining recipients, it will do so. In the end, both these duplicates of the message (or even more if the above described situations occur again on the remaining path) reach the destination. This is wasting the available bandwidth and making the multicast slower. The solution is to split the remaining destinations of the message into disjoint subsets and send the message to each of them. This way the message will never go from one subset to another. However if we go on and apply the same on the subsets, we are actually in fact creating a multicast tree, since we pre-define the distribution path by restricting where the message can go, up to a per-node level. The main problem however lies in determining to which subsets split the destinations of the message. This is analogous to computing the optimal distribution tree, yet using only local backpressure information, instead of having complete global network map. In order to resolve which subsets yield the best results, all the possible combinations have to be tried – that's $2^M$ for M remaining destinations. This makes this solution either not so effective (duplicate messages) or computationally heavy (all combinations) – both slowing down the multicast process.

All of the so-far described shortcomings of the discussed approaches come from the fact that information is "pushed" from the source to the destination. Because of this pushing, the "pushers" have to make decisions what to send where – and this then creates the mentioned difficulties. If we revert from pushing to pulling, where each node explicitly asks for the given messages, we'll arrive at a different set of solutions. With this approach the nodes have only information about their local part of network, yet if we organize the whole multicast operation well, we can achieve very good results, close to the theoretically maximal throughput rate. We also avoid nodes receiving the same data multiple times – a node asks only for pieces it is missing. The path each pieces is distributed via is determined at run-time by requests for it by nodes. Thus one multicast uses if fact multiple concurrent distribution trees. The whole organization in pulled (receiver-initiated) multicast lies in choosing which nodes to pull the data from – and what data to request from them. For that, the nodes have to first know which node has which pieces available. Each node informing each other about every piece in the whole set of recipients doesn't scale well. Therefore we still have to somehow organize which nodes will be communicating with which others. Random neighbor selection with some fixed limit (e.g. 50) gives quite acceptable results. We just have to make sure we don't create two disjoint node groups. The second part that needs organizing is piece selection – what do we try to download first from the set of pieces available at our neighbors. Again random selection gives best results – if all nodes selected pieces sequentially, they would quickly sync-up to the same state. This would limit the available piece selection. This is even more so important, because there's some round trip time latency, thus if nodes want to achieve reasonable download speeds, they need to request pieces ahead – and that's only possible if there's enough pieces to request. Otherwise some nodes might not be able to saturate their download rates. A partial solution to this problem is to make the pieces larger, but this limits pipelining – every node in the path has to download whole piece before it can forward it, if the piece is large, this can take some time. Another property of this type of solutions is that they try to use all the

available bandwidth – they flood the network. This is however undesirable for our needs, chapter 1.3 describes how we solve this problem. In general we can call the above described class of the solutions as peer-to-peer, since with random neighbor selection, there is no hierarchy, no master-slave, all peers are equal.

Let's consider a "naïve" implementation and scenario on figure 3. The worst case for speed is when nodes Z and Y will both download 5MB/s from node X. Because Z can upload only 1MB/s to Y, Y's download speed would be just 6MBs – and that only in case they don't both download exactly the same pieces from X. Therefore a good p2p algorithm should consider not only neighbor and piece selection, but also some overall behavior. A good example of this is **B**it **T**orrent [14], which prefers to allocate higher upload bandwidth to peers which themselves have high download bandwidth using so-called choking algorithm. Moreover tit-for-tat scheme and e.g. super-seeding mode prefer peers that have higher upload rates. This way the fig. 3 scenario would soon stabilize in the situation where node Y is downloading at 8+MB/s rate and node Z only remaining rest. Also default piece selection in BT implementation tends to put preference on rare pieces. This way Y and Z would select different pieces to download from X and therefore could use their mutual connection to share data between themselves. Because of this and some other properties of Bit Torrent, we have decided to build our solution on its basis. More (technical) details about BT in chapter 3.




(a) Network example

Figure 3 – unfavorable p2p scenario [fig 4.2 in 5]

## 2.5 **Advanced peer to peer based solutions**

[5] presents two important extensions to p2p solutions. Both exploit the network topology knowledge to achieve fast throughput rates. In the first method, nodes in the same cluster form a collective, called **MOB**, where each node has to download a certain part of the whole file from other collectives (clusters). Thus all nodes within one cluster divide the file retrieval work by equal shares and download concurrently. They exchange the retrieved pieces then on the fast local network. The algorithm expects the nodes to know their location – thus to know to which MOB they belong. After a node completes download of its share, it keeps sending pieces to all other nodes that request them, until everyone is finished. Because each peer in MOB has distinct "work set", each piece will be retrieved from other collective only once. This doesn't waste shared downlink capacity. Combined with maximal possible utilization (all peers download concurrently), this achieves the highest possible speed on this link. Because external connections are usually much slower than local ones, this method then achieves almost the best possible transfer speeds. However it can happen that some peers are slower than the others – either because their local connectivity is limited, or they are overloaded doing some other tasks. In that case, because workload division is static, the overall performance is hurt, because other nodes in the collective have to wait for the slow one to finish its share of the work. This also slows down piece spread to other collectives, since they might have to get some pieces from the slow collective.

To address these shortcomings, [5] has extended MOB with dynamic load balancing. This is realized as work stealing – whenever a node is idle, it can take over some work from slower nodes which still didn't finish their work share. Thus the name Robber. The work stealing is coordinated in such way that at any time, an external piece is desired only by one peer, thus it would not be downloaded twice. As [5] presents, Robber is a very effective solution, usually achieving download rates very close to the theoretical maximum. What is more important is that it is also very resilient and can adapt well to even vastly changing environment, such as link capacities, still achieving rates close to maximal possible.

Our solution takes inspiration from both these approaches, mainly the work division of MOB and random work stealing of Robber. The "view" of the world in our case is a bit more complex than my group – other groups as in the above solutions. The peer selection and organization is also different, chapter 4 contains more details on that. Both MOB and Robber utilize IBIS [16] grid environment and rely on some of its services. The implementation as it was done in [5] doesn't expect any nodes to join/leave while the transfer is in progress. Because of these reasons we have decided that starting with a solution that is already robust and independent and only adding these new features to it afterwards will be a better choice for us. Therefore we based our solution on and existing bit torrent client, by extending its functionality and building peer management and environment feedback software around it. We elaborate on the client selection, design decisions and choices made in the chapters 3 and 4.

## 2.6 **Other Bit Torrent enhancing solutions**

Apart from the specialized solutions we have presented, there's whole class of various enhancements done in the "bit torrent world". Most of these modifications don't alter the protocol, but rather focus on the client behavior. The main goal usually is to improve the transfer speeds, but the ways to achieve this differ. A notable example can be Bit Tyrrant [17]. This client was built specifically to demonstrate that the default bit torrent tit-for-tat behavior can be tricked and exploited. Bit Tyrrant does a strategic peer selection – it prefers to upload pieces to peers which give the highest download rate in return. This can be seen as a form of "resource management" – bit tyrant just maximizes its download rate, based on its maximal upload speed. This however works well only when other peers use standard/other than tyrrant client and contribute upload bandwidth "altruistically". Once all the peers in the transfer use tyrrant client in selfish mode (don't contribute any available excess upload rate if it doesn't further improve your download speed), then the actual download times increase – simply because downloads can be only as fast as there is available upload capacity – and that was decreased due to selfishness. Yet the tyrrant client is a very good example how much can peer behavior influence overall transfer dynamics.

Another group of solutions also focuses on peer selection, yet doesn't try to exploit any of the shortcomings of either BT protocol, or other peer's behavior. The simple idea behind most of them is that the closer the peer the better. The closer can mean geographically, by latency, by hop count, by same subnet/ISP etc. The better usually means the faster downloads, because the closer the peer is, the faster its connection should be. But it can be also better for the peer's service provider – preferring local sources saves costly interconnect bandwidth.

[18] proposes selecting peers that use the same ISP. This is achieved by modifying the tracker /see chapter 3 for definition/ and the client. The tracker uses internet topology map, or AS mappings to identify the ISP and then returns a set of peers that are within the same ISP. This could be also done by ISP's traffic shaping devices, which do deep traffic inspection, identify client's request to the tracker and add local peers to the tracker's response. In the evaluation of [18], significantly reduced redundant downloads from outside to the ISP can be observed, while download rates remain the same. Yet this assumes that there are enough peers participating in the same transfer within the same ISP. The more peers there are, the better results this solution has. P4P [19] goes even further and proposes explicit information system implemented at ISPs (iTracker), which would help to guide p2p applications by providing them with the recent network-layer information. Ono [20] uses DNS resolution of major content delivery networks to determine if peers are close by (peers exchange their "CDN coordinates"). This approach also works well, most of the peers end up being close to ourselves. Disadvantage of this class of solutions is that they rely on some third party involvement – either ISPs themselves, which e.g. would be not so happy about exposing their internal network structure, as proposed in [18], because of privacy and security reasons. Ono then relies on the CDN services and puts unwanted load on their DNS servers.

[21] presents a solution that doesn't need ISP involvement. Also client implementations don't have to be modified. The "latency driven" tracked uses system of landmarks to determine the client location in the network coordinates system. The response to request for peers then contains a selection of peers where most are near-by to the

requester. [22] adds hop count information and selects portion of peers in the tracker's response based on their low hop distance. Both approaches achieve lower download times and reduced long distance traffic. The interesting observation they present is, that we need to include some portion of randomly selected peers to the results, so that we don't portion the swarm /see chapter 3 for definition/ into small badly interconnected islands. This can be also seen on the figure 4, which is from Ono [20] evaluation. It represents a scenario where local peers have ADSL connection, which has much higher download rate than the upload rate. If we select those peers, then they will not have fast enough combined capacity to saturate our download link, thus the whole transfer would be slower compared to random peer selection. Because of practical implementation, where ISPs rent the last mile to the client, having local peers means quite high probability that the other peer could be using some other ISP. Fortunately due to lower DSL upload speeds, this won't generate too much intra-ISP traffic, yet it demonstrates shortcomings of local-only approaches. Considering that such ADSL is quite common type of connection and is used by large amounts of P2P users, we arrive at the conclusion that a closer peer is not always a better (faster) peer.

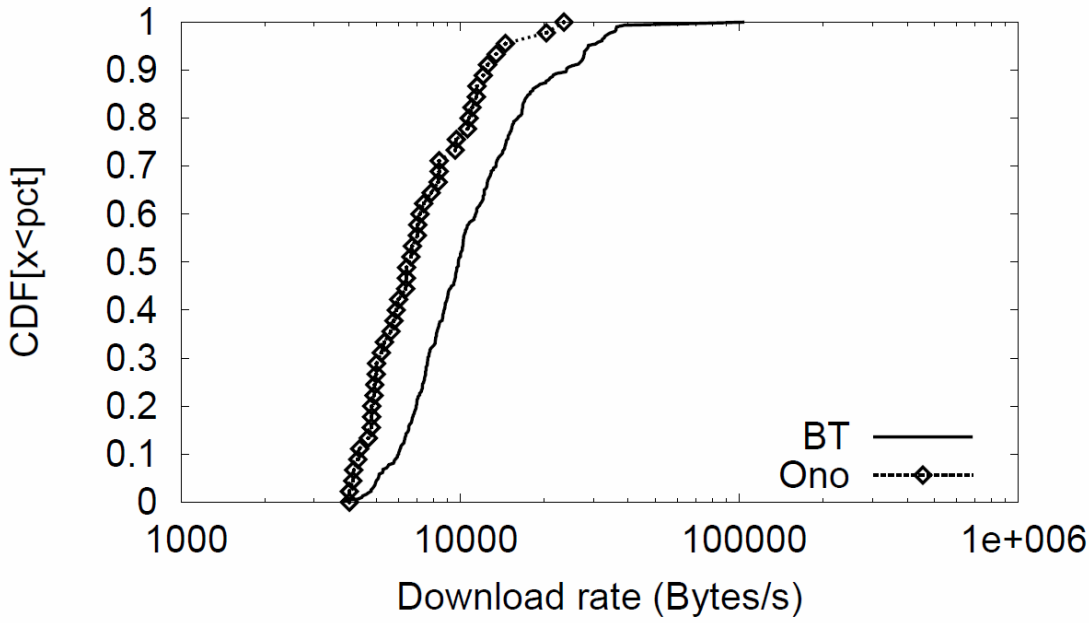



Figure 4 [fig. 8 in 20] – "ADSL scenario" – performance with local peer preference

TopBT [22] realizes this and in addition to AS and hop count proximity, it also considers peer's speed to be a factor in the peer selection. This is implemented in the topBT's unchoking algorithm /see chapter 3 for definition/, thus requires no tracker, protocol or other peer's client modifications. Interestingly, in the evaluation TopBT achieves speedups and traffic reductions comparable with [20] and [21].

The class of the solutions we have described in the last two sub-chapters are based on the observation that in the original BT client, the overlay (which peers connect to which) and the underlay (the real network which transfers the packets) are two distinct unconnected worlds. The underlay is presented only partially in the tit-for-tat scheme where we choose to or not to upload to a given peer based on its upload rate to us. This has lead to long-distance traffic usage. Given that BT is the current major global consumer of the all internet traffic, these solutions try to reduce this unoptimized resource utilization by preferring the local peers. As we have shown, the closer peer doesn't always mean that the transfers would be faster. However in general, all these solutions bring improvements in the transfer rates and traffic reduction. Yet clever peer selection is not the only possibility to improve transfers efficiency by better organization. As MOB and Robber show, further peer cooperation on piece selection can significantly improve performance and adds additional robustness and resiliency.

Based on all these observations, we have decided that in addition to being aware of network topology and available bandwidth, **we will focus not only on the peer selection policies, but we will also use some advanced piece selection algorithms**, specifically work division and work stealing in our solution. We will also change the client's behavior, so it can take larger advantage of these new features.

# 3 **Enhancing Bit Torrent**

Bit Torrent [14] is a peer to peer file transfer protocol, but also a client of the same name, used to do such transfers using this protocol. It was developed by Bram Cohen and first released in 2001. Since then it has become a de-facto standard p2p protocol, currently responsible for the most of the internet traffic. Currently there are about 100 million users. The protocol specification is open and well known, so anyone can implement his own client. There are a lot of extensions and additions done to the protocol and clients, most of them are described on the official bittorrent.org web site in a form of BEP documents. Because the protocol is quite simple and contains all the basics needed for peer-to-peer communication, there is no need to create any other p2p protocols and any specific functionality can be done as an addition to it. This has also contributed to its popularity: many of the important features were implemented first in one of the non-mainstream clients and once they have proven useful and popular, they have been also adopted in the official client and added to the protocol specification. As we mentioned in the previous chapter, we have decided to base our solution on this protocol, adding some specific enhancements. We will first describe some protocol and client basics, so that reader can later understand how we did these enhancements and why have we chosen the given way.

## 3.1 **Bit Torrent**

The initial bit torrent design had several parts and some more were added later. We will start with the most simple scenario. To distribute a file via Bit Torrent, the provider of the file divides the file into small *chunks* (also called *pieces*, usually 256KB in size), and generates another file containing meta-information about the data file (called the *torrent* file). If some peer wants to download the primary file, he first needs to get the *torrent* file first. This is usually done by downloading from web site, but any other means are possible. For completeness of information, one torrent file can contain meta-data about more files, so a peer can transfer multiple files at once. One piece can span multiple files, so there's no overhead in the data transferred. After a peer has this file, it reads the *tracker* address. A *tracker* is a central server which tracks which peers are currently doing the transfer. After a request, it will provide our peer with a list of peers to connect to and get the chunks from. Usually the selection of peers is random, but as we previously shown, there's space for optimization by returning specific set of peers. The download can begin then. Because file is split into chunks, peers can start exchanging parts of the file already when download a chunk. There's no need to wait for the whole file transfer to complete. This is very important – usually the user's interest is just to get the file as fast as possible. As it was learned by experience times and times again, after this is done, very few people continue to share the downloaded file afterwards. If so, then usually for very short time. This causes the downloads to be slow – because there's not enough upload capacity and combined download speed can be only as fast as the combined upload speed. Bit torrent changed this by so called *tit-for-tat* scheme: this means that a peer from which we want to get data, will send us chunks, only if we ourselves are willing to upload chunks to other peers. This way the peers are "forced" (or shall we say motivated) to upload, contributing to the overall

performance. This approach scales very well and enables fast downloads even with thousands of users.

Let's now dive a bit deeper into details. The torrent file actually contains the list of files (full name, also with directories), addresses of trackers, user added comments and possibly some more meta-data. The format is open and if needed, user can add some more information, yet you can't expect 100% client compatibility – that depends on the other clients' implementations. The most important part of the torrent file however is piece size (practically (client compatibility) from 32KB to 32MB, implemented as integer number) and then 20 bit SHA1 hashes of all the pieces. The piece is claimed to be downloaded by a client only after is passes the hash verification against the torrent file. This is to ensure that data won't get corrupted and also prevent some malevolent seed from providing broken chunks. The amount of peers tracker returns is usually up to 50 (number can be specified in the client's request). That should be enough to reach fast speeds. Also clients themselves have some limitations how many peers to connect to (some permit user setting). This is because for each chunk a client downloads, it has to inform all its peers that it has it available for sharing. If a *swarm* (a group of all peers downloading the given torrent) has say 1000 peers, it's not feasible to send this message to all peers for every chunk. In our scenario we use different criteria to select peers, so none of these limits apply.

Another important piece of the client is a so called *choking algorithm*: the peer doesn't upload the chunks it has to all the peers, but only to those which are *unchoked*. Those are the peers that are willing to share and upload their chunks to our peer. Also for performance and efficiency, it is not reasonable to upload with rates lower than about 5KB/s – the chunk delivery times become then quite large. Thus if we have limited upload bandwidth, it makes sense to upload only to such amount of peers, that each receives at least this 5KB/s bandwidth. Other peers become then *choked*. When a peer joins the transfer, it initially has no pieces, thus all other peers would choke it and won't upload any data – our peer has nothing to exchange in the tit-for-tat scheme. Because of this, BT uses so called *optimistic unchoke* – which is altruistic offer/upload of some data to the peer to help it start the transfer, so it has some data to share with others. The exact amount of unchoked peers, frequency of optimistic unchokes and some other properties depend on how exactly is the given peer implemented (let's call this a *peer behavior*). As we have shown, Bit Tyrrant [17] and TopBT [23] implement specific strategies here to get better download rates.

The last important aspect is the piece selection. This happens when our peer has possibility to choose from different pieces from its peers. The most simple and quite robust solution is just to select the piece randomly – we need all of them anyway, but it doesn't matter on the order in which we get them. The other possibility is to select the rarest piece. The rarest meaning the one that is least present among our peers (e.g. only 2 of 38 peers have it). The logic here is that this piece is hard to get, thus we should put our effort here, because once the given peer goes offline, we might have to wait till some of our other peers get it. In our case this is not a good piece selection strategy, as we will demonstrate later. Our client will implement very specific piece selection strategy – based on "work stealing" and cooperation with other peers.

All these aspects (tracker response – peer selection, choking algorithm, piece selection) influence overall transfer dynamics. There has been quite some research done in this area and there exist several simulators. All that is used to see which settings give the

best performance, robustness and resilience. As it turns out, the default bit torrent settings from 2001 (5 unchoked peers, random peer selection and random piece selection) provide quite good results and give good robustness (random peer selection avoids creating isles in the swarm graph of peer interconnection). The enhancements and alterations to these settings thus don't provide extreme gains and improvements. It then becomes a bit harder to justify them if they are more complicated. Yet given that BT transfers are the major source of internet traffic, every improvement is welcome, especially if there is no simpler way to gain more efficiency and speed. During 10 years of its use, BT has adopted a lot of enhancements, to name a few – utilization of distributed hash tables (kademlia) for peer discovery (avoids tracker begin SPOF and bottleneck), peer exchange, magnet links etc. We'll explain these (and much more) terms in the following sub-chapter. Currently it looks like Bit Torrent will be here with us for another while, so we can expect further enhancements and extensions. We believe the solution we present here has potential to bring some of its ideas to mainstream, or at least inspire other enhancements – not only for bit torrent, but also for other P2P solutions.

## 3.2 **Terms and nomenclature**

In this sub-chapter we will describe the meaning of some not yet mentioned terms used in the "BT world". The aim here is not only to describe what these mean, but also why they exist and what they are used for. This chapter can be seen as a continuation of the bit torrent description.

***block***

The smallest transfer unit. Peers actually don't send requests for chunks, but for blocks. Thus one chunk can be concurrently downloaded from more peers. The most compatible size is 16KB, some clients close connection if you request larger size.

***piece***

Also called chunk – a part into which the file is divided. There is quite some naming confusion – in the initial protocol specification, piece meant block. This is also present in the protocol's command names. Yet currently, and also in this thesis, the piece is a chunk, and only block is a block. The piece size can vary, usually between 32 and 4096KB. This is to keep the size of the .torrent file down – if we have a large transfer (several GB), having small pieces would mean many pieces. Each piece means at least 20B in the torrent file. Some clients have compatibility problems with transfers larger than 128GB and torrent files with more than 65535 pieces.

***peer exchange***

An extension that enables peers to exchange information about their other peers. This is important if we want to discover other peers than only those returned by tracker – it adds robustness and ability to function without tracker (after bootstrapping).

***distributed hash tables***

A form of finding peers: currently bit torrent clients use kademlia DHT to find more peers. The .torrent file contains a list of bootstrap nodes. Each .torrent file contains a HASH that identifies this torrent. If we look up this torrent in the DHT, we will get at most 8 peers participating in this transfer. Using peer exchange with these we can find more peers. This way we don't rely on a tracker, thus having completely distributed solution. Torrents which don't have a tracker are also called **trackerless**. The hash of this torrent can be also identified by the **magnet link**. The peer then looks up other peers in the DHT and gets the .torrent file from them. After that it can start download.

***local peer discovery***

A method of discovering local peers: HTTP-like messages are sent to a local (organizational scope) multicast address (thus should reach all the multicast enabled peers in the local network). Given that local peers should reside on a fast network, it is very beneficial to discover them. As we will show later, it is very beneficial for local peers to cooperate, e.g. using work stealing. There is also BEP 26 – zeroconf peer discovery extension, which we can say is just another form of local peer discovery. There is quite possibility that there will be further development in this area, as it seems like Bit Torrent Inc. themselves do recognize importance of local peering [24].

***private torrent***

A flag inside torrent file telling the client that it should obtain peers for this transfer only from the tracker (and not use any other methods of peer discovery). This ensures that only peers verified by tracker will be participating in this transfer. We use this setting as default and as a form of security/authentication, and to ensure that given peer won't connect to anyone else than we tell it to.

## 3.3 **Protocol details**

Bit Torrent uses mainly TCP connections for communication. There is also possibility to use UDP, but this is an extension. There are two main communication channels – with tracker and with peers. Because our solution implements a very specific way of communicating with peers, we do not use tracker protocol at all, thus we will not describe it here. The peer protocol is used for exchanging the pieces and other transfer and state related information. Peer exchange, DHT, local peer discovery and other extensions have their own specific interfaces and we will also not describe them, since our solution doesn't need these extensions. The main peer protocol is the essential data exchange link that has to be implemented by each client to do transfers at all. The used messages are: `choke`, `unchoke`, `interested`, `not interested`, `have`, `bitfield`, `request`, `piece`, `cancel`. `Choke` and `unchoke` are used by choking algorithm to communicate the choking state to other peer. `Interested` and `not interested` inform the other peer if we are interested (or not) in downloading any pieces from him. `Have` message is sent every time we complete download of a chunk to all our peers, informing them about the availability of this piece at us. The payload of `have` message is the piece number. `Bitfield` message is used only once, when initiating connection with a peer – it

gives us a true/false bit map of pieces available at that peer. `Request` message is used to request a block of data, parameters are piece number, offset and block size. `Cancel` message carries piece number and is used to cancel the `request` for that piece. The `piece` message is a reply to `request` and contains the piece number, offset and the block of actual data. There is also `keepalive` message to prevent connections from timing out. The `port` message let's the client know on which port is our DHT node listening. This message is not a part of the original protocol and is only used by clients using DHT (which means almost all clients today). In our modifications we will add one new message to help peers coordinate their piece selection strategy.

### 3.4 **Advanced functionality**

The advanced functionality consists mainly of features and extensions built atop the basic protocol. This functionality consists mainly of a behavior induced by trying to achieve some specific goal.

***end game scenarios***

The "end of the game" is finishing of the download. Once the client approaches the end of the download, it changes the piece requesting strategy. The issue is that when finishing the download, all the easily available "fast" pieces have already been downloaded and it takes a while to download the remaining pieces from slow peers (slow because they haven't been retrieved so far). To cut this waiting time, clients enter so called **aggressive mode** where they send piece request to multiple peers having that piece. Once the piece is retrieved a cancel message is sent to other peers the piece was requested from. In our evaluation we have experimented with disabling this mode, as it seems wasteful (by the time we send cancel, we might have received some data multiple times already), but we didn't see any noticeable difference. Moreover this mode adds robustness (when one peer fails, we still have requests pending at the other), so we decided to leave it unchanged.

There is another meaning of "end game" in our solution – and that is finishing of the whole transfer to all the peers, description how we handle this is described in chapter 4.

***super seeding***

Super seeding is a method how to efficiently distribute the file data initially, when the transfer is just starting. The peer pretends to be just a normal downloader who has no data yet. Then it sends out `have` messages, one for each piece, to different peers. This way he can ensure that it will not receive more than one request for the given piece, thus would not upload data multiple times into the swarm. If the bottleneck is formed by the initial seed's upload bandwidth, this can help speed things up, since otherwise there would be quite some possibility that multiple peers would request and download the same piece multiple times. The downside is if a piece gets to a slow peer, which is not able then to upload/spread it quickly – this can then hinter overall swarm performance. In a good implementation, peers should switch to normal seeding (uploading) mode after they finish super seeding phase. We see some potential in further enhancements of this mode, more discussion in the chapter 4.

***pipelining***

Because there is undeniable RTT latency in the communication between peers, it is necessary to pipeline – or request ahead the blocks we want to download. Otherwise we might not be able to achieve maximal possible transfer speeds. It is important to realize this fact when considering piece selection strategies. By the time we select/decide to request another block, we already have quite some requests pending at our peers. Thus if we were using the rarest piece strategy, what was valid when we requested the given block (which can be still pending) might not be valid now and some other choice might seem better. Thus if we want to have good piece selection approach, we need to consider that our choices have to be good not only in the moment we send the request, but until we actually receive responses. We'll discuss this matter again when we'll describe our enhancements in the piece selection strategy area.

***broadcatching***

A technique called Broadcatching combines RSS with the BitTorrent protocol to create a content delivery system, further simplifying and automating content distribution. In short, a specialized client supporting RSS feeds gets the data from these. The data contains location of torrent file, or other information (magnet link) to do the transfer. Our approach implements somewhat similar solution, where peers are subscribed to a redis [25] channel through which they receive information about the upcoming transfer.

***(initial) seeder***

A peer that has already finished download (or had the file initially) and is now only contributing the upload bandwidth.

## 3.5 **Existing clients and libraries**

Given the popularity of bit torrent protocol, there exist quite some clients. The most popular are Vuze (Azureus), mainline bit torrent (the original client, discontinued), µTorrent, bitComent, bitSpirit, Transmission, rTorrent and plenty others – many times based one on another. There are also several libraries available, if you decide to create your own client, or implement some of the BT transfer capabilities in your software. The naming here is again confusing: there are two libtorrents – a libtorrent by Rasterbar software, which is using boost library to gain platform independence. There's a lot of software based on this library, see [26]. The other libTorrent [27] by Jari Sundell, "Rakshasa", is a BitTorrent library written in C++ for *nix, with a focus on high performance and good code. The library differentiates itself from other implementations by transferring directly from file pages to the network stack. On high-bandwidth connections it is able to seed at 3 times the speed of the official client. The main client using this library is rTorrent from the same author.

Currently the most popular is µTorrent, which is now also an official client, after Bit Torrent Inc. bought it in 2006. This software is pioneer of the new features (along with Vuze and some others). In the BT world, if something wants to make it into the standard specification, it has to prove itself worthy. Thus distributed hash tables were first

implemented in Vuze, and only after so many people were using them and it was obvious that this feature brings undeniable benefits, they were adopted to the standard client as well, albeit in a non-compatible way. Later Vuze started using also this mainstream DHT, so now there's only one standard. Most recently µTorrent added µTP – and UDP transfer protocol, with automatic speed throttling when connection is over utilized. Nevertheless, µTorrent is available only for windows. Currently it has about 100 million users and Bit Torrent Inc. is trying to figure out ways how to monetize on this. Most recently they added ability to run applications inside µTorrent. Things are thus getting a bit political and it's quite hard to get any of the enhancements not done by the BT inc. recognized. The highest form of recognition seems to be re-implementation of the same feature by BT inc. in a non-compatible way. Another notable feature of µTorrent was ability to stream videos as they were downloaded: the piece selection was altered in a such way, that the client was trying to download the file in a somewhat linear matter, where it would focus mainly on getting the pieces that are needed for showing the video in the following few minutes. The video was then streamed as a standard video stream source and could be watched using e.g. VLC media player. This feature worked quite well, albeit it required some specific conditions – mainly good file availability (high combined upload bandwidth of all peers). It is quite interesting to see this feature implemented here, as some large companies like Adobe have mentioned adding p2p functionality to their flash video formats a few years ago, but I am not sure if there was any progress. At least one huge user (YouTube) would really appreciate this functionality. Let's now go back to the client selection:

With so many clients available, there was not an obvious choice which one should we pick for implementing our enhancements. We needed something that was open source, so we can access the code at all, and the preference was for something that performs well, but also doesn't consume a lot of other resources (memory, CPU time) – since the transfers would be done on live system and we didn't want to interfere with the main running application in any way. In the end we have chosen the rTorrent, mainly due to its very high configurability and external control interfaces. We have also further extended these to enable communication with our distributed tracker software.

### 3.6 **rTorrent**

We have chosen the rTorrent client because of its extensive configurability. The user has ability to control even low level settings, like send/receive buffers size, polling facility (select/epoll based on what is the OS we run the client in supporting), limit maximal memory usage, hash checking algorithm read-ahead and check frequency. In addition to that, the standard settings themselves are quite exhaustive: hostnames, IP addresses and ports to bind, download and upload rates – limits, control of various BT enhancements (Peer exchange, DHT) and a scheduler, where we can define some periodic actions, like watching over a directory and automatically starting downloads if there are any new torrent files. The scheduler is universal and there are around 450 commands we can use. We have also added some new commands to control some of the new features we have implemented in the client. One very important function we utilize is group throttling: user can define a custom group of peers and give them one common shared speed limit (all peers together from that group will not exceed that limit). Our distributed tracker uses this

function to control peers speed based on which location they are at. Most importantly, all these commands and functions can be used in real time via XML-RPC interface. We use this to feed the client in the real-time with the most recent transfer data and control it's speed limits and peers it connects to. In fact, in our solution the modified rTorrent client is just a "blind workhorse" which just does the transfers, but all the other logic is implemented in the distributed tracker.

The rTorrent client is using libTorrent library – we have implemented our enhancements also here – depending where the functionality we needed to change or add was implemented (in the client itself or in the library). In the end we think rTorrent was quite good choice, even though we were not able to implement every single feature we wanted. This might have been easier with some other clients, but on the other side we would also have to implement some other rTorrent functionality we used that is not present in these other clients (mainly group throttles). The rTorrent version we were modifying was 0.8.6, libtorrent version 0.12.6.

## 3.7 **Enhancing the protocol and the client**

As we have mentioned in several places already, our main aim was to implement work stealing and work division into the rTorrent client. This functionality cannot be done elsewhere, since we need to alter piece selection algorithm. Besides this, we also modified various internal (hard coded) settings in order to create so-called low-latency client.

### 3.7.1 **Work distribution**

The idea behind work distribution is that local peers divide among themselves the task of downloading all the transfer's pieces from outside. In order to know which peer should download which pieces, it either needs to be aware of all the other local peers (in practice this can be done by locating these via local/zeroconf peer discovery), or it can be explicitly told which pieces to get. We have implemented the second possibility and added a new rTorrent command `d.update_ID <HASH, myID/peerNO>`. The parameters are compulsory. This tells the peer that in the given transfer, identified by `HASH`, what is his ID and how many other peers are there. Based on this, the peer will *prefer* to download pieces from IDth slice of all chunks. The slice size is #of pieces/#of peers. E.g. if there are 1280 pieces and 4 peers and our ID is 2, then our peer will *prefer* to download pieces 640 – 959. The word *prefer* is important: we have implemented work distribution in a "soft" way. In the piece selection algorithm, the peer will first try to download pieces from his assigned work share – these **pieces are simply given high priority**. But if there are none of these pieces available at any of his peers yet, our peer will select some other pieces to download. This is to keep the transfer always going, the peer needs to get all the pieces anyway. Usually those other pieces will be downloaded from a peer whose work share they represented. Another reason for "soft" implementation is robustness and resiliency – if any of the other peers dies in the process of transfer, his work share would be downloaded by other peers anyway, even without recognizing that the other peer is gone and that we need to re-divide the work shares. That happens only when a peer joins or leaves a transfer in a

proper way – managed by our distributed tracker. The basic, underlying piece selection algorithm is random – also for selecting among our work share pieces.

### 3.7.2 **Work stealing**

We could also call this technique peer coordination. The original idea of work stealing in Robber [5] was used for claiming parts of work-shares of slow nodes. In our implementation we have decided to take slightly different approach. Our peers steal work by one piece and they steal it from everyone at once. The idea behind is simply to let the other peers know that right now I have requested this piece and if you can, select something else to transfer, so that we can avoid duplicate downloads. This way all the peers in the given location de-facto coordinate their piece selection strategies in the real-time. Again, the implementation is "soft" – a peer can download what others have requested already. The main reason is again robustness – if the initial "requester" dies, it doesn't matter much, the piece would be retrieved anyway, albeit maybe a bit later (as all other non-requested pieces will have preference before the other peers start downloading what our deceased has already informed them he requested). Another reason is that in some specific scenarios, like the ADSL on fig. 4, it is desirable that the given piece is downloaded from outside multiple times, even concurrently, simply because local peer's upload capacity is not high enough to saturate the combined download speed. This way we avoid performance penalty, while still being as effective as possible: as long as there are enough pieces to select from, peers would prefer to download different ones. After all the chunks are already present inside our location, peers would still continue to download from outside, but because all the chunks are present locally – and more specifically, different peers have different pieces, they can all contribute their upload bandwidths to the full extent.

To implement this "work stealing" we have added a new `START` message to the protocol. The syntax is the same as with `HAVE`, except message ID is 16. Every time a peer sends a request for a block from a new chunk he has not requested any blocks from any peers yet, it will inform all its peers that he started download of this chunk by sending them `START <piece number>` message. Peers remember which messages are `START`ed (and how many times) by their peers and use this when choosing a piece to request. After peers finish the piece download, they send `HAVE <piece #>` message to their peers to inform them about that piece availability. That's when we realize that this piece is not `START`ed anymore (actually, we only decrease `START` count, if the count is 0, that means no one has currently any requests for this piece pending). When we then select a piece to request from our peers, we choose first from pieces that are not `START`ed yet, and only then if there is no other choice, we also select from pieces that are `START`ed already.

### 3.7.3 **Modified piece selection preference**

We have implemented two new features in work division and work stealing, that bring major changes to the piece selection preference. In fact, peers don't download chunks, they request for blocks. Therefore to get one piece, we usually have to request for

more blocks. We can get these from multiple peers. In rTorrent we have thus modified block selection strategy to (this is the preference order):

- a block from a chunk that is already being transferred from this peer – this puts preference on finishing transfers of pieces, so that we can share them ASAP – this is also important that we have something to trade in the tit-for-tat scheme
- a block from already in-transfer high priority piece
- a block from new high priority piece
- a block from already in-transfer normal priority piece
- a block from new normal priority piece

When searching for new high priority or standard piece, a chunk that is not `START`ed yet is selected first, only if none such exists, also `START`ed pieces are taken into consideration.

### 3.7.4 **Low latency client**

The un-modified original rTorrent puts emphasis on efficiency. Because the application is event driven, many times things don't happen immediately, but are rater enqueued and processed only if there's some other related processing to be done (e.g. message sending is done by concatenating multiple messages into a buffer which is then sent (flushed) at once to the peer). This saves on the various content switches and interrupts. However, such max efficiency implementation means that there are some unnecessary latencies. Because we need as much up-to-date information as possible for our work stealing implementation to work efficiently, we have modified the client by removing/disabling these optimizations. We do this either by reducing/removing the hard-coded timeouts and various time intervals, but also by explicit calls to flush pending events every time we need to e.g. send `START` message.

# 4 Controlling the transfer

The main core of our work lies in the transfer management. This includes all the "wrapping" activities like creating the torrent file, starting transfer on all the peers that are supposed to download the data, keeping track of the transfer, doing post transfer checks and finally reporting some statistics and success to the user. The most important however is controlling the overlay topology during the transfer. This determines which peers connect to which others and how fast they are allowed to transfer the data between them. By this quite strict control we aim to ensure non-intrusiveness as well as high efficiency and thus transfer speeds. The overlay structure counts with utilization of the new features we have implemented (work division, work stealing) and is created and maintained in a form that allows to expose them well. We will start with general design description and will dive into further details in the following sub-chapters.

## 4.1 The design overview

As we mentioned earlier in the chapter 3, we have decided to split the logic in our solution into two parts. The modified rTorrent client only deals with the actual transfer itself, it is not aware of all the peers, nor doesn't try to anyhow discover them. It only does what it is explicitly told to. The second part in our solution is a distributed tracker, let's call it dTrack. dTrack client runs on every node that should participate in the transfers. dTrack and rTorrent are running all the time (daemons) – this was specifically requested by Hyves, who view the p2p transfers capability as an service that should be always available. The dTrack client takes care of finding the appropriate peers, adding them to correct throttle groups and setting the correct speed limits. How exactly is this done will be described in the following sub-chapters.

All the dTrack clients are subscribed to "Transfers" redis channel. This can be hosted by single or multiple hosts – DNS name is used for connecting. In our implementation, one host is easily able to handle the 3000+ servers of Hyves. In practice, if the host providing the channel functionality was overloaded, it would only result in some peers receiving information about the transfer few seconds later. The usual capacity of an ordinary node is about 100 thousand announcements per second. In our implementation, the transfer initiator announces the transfer (this can be done in multiple ways, more details in 4.5). But before that, it "pushes" the torrent file via scp to all the peers that are supposed to join the transfer. This can be easily changed to tell peers to "pull" the file instead. The rTorrent 0.8.6 didn't support magnet links yet, however since 0.8.7 the support was added. This looks like a perfect way to announce the start of a transfer to the peers (and it really is close to it). Yet we need to realize that even with magnet links support, all the peers have to download the torrent file anyway. If we push the torrent file to the peers, we can use multiple hosts to do the work. In fact, if the torrent file is quite large, we can use *this* (our) p2p framework to transfer it, since the torrent file for that torrent file would be much smaller. We have decided to make it easy to change the way of getting the torrent file in our implementation, since we wanted to make it easily adjustable to the specific conditions of the deployment it would run in. One more remark to magnet links – they require DHT to

work (so that they can look up the peers for the torrent – the magnet link contains only the HASH of the torrent). But if we enable DHT, rTorrent would find peers on its own, breaking unintrusiveness and getting a bit out of control. Therefore in order to enable magnet links for our solution, further rTorrent modifications would be required.

After the transfer is started, the transfer initiator is running redis where the actual overlay information is stored. The transfer initiator is a single point of failure in our design. Yet we should realize that the initiator is usually the only source of the data, thus being SPOF already. Our solution of course supports multiple seeds, yet if we do a bit more thinking, we realize that the data has to somehow get to those seeders before we start the transfer. And the most effective way is to do it via our p2p framework. If we think further, we'll realize that we don't need to do this in 2 steps (1. get data to the seeders, 2. do the actual transfer), but we can start the global transfer immediately. It would be faster because the phase 1 seeders would be able to upload the data already while they are getting the file. Therefore in the end, we usually really have only one seeder.

After the transfer starts, dTrack feeds rTorrent with information about which peers it should connect to and which speed limits to impose. This is determined by local peer's position in the overlay. This position is not static and can change over time, based on other circumstances. The speed limits are dynamically updated. If the transfer initiator hosting overlay communication manages to die during the transfer, no further speed or overlay changes would be applied. However because the overlay is created in the initial few seconds of the transfer, this would only mean that the transfers can possibly exhaust capacity on some of the links (or not really exhaust, just keep using the old bandwidth amount, whereas it could have been lowered later due to other application using more bandwidth). If all the data has been transmitted by the initial seeder, usually the peers would be able to finish the transfer. If a peer or seed dies anytime during the transfer, it can be just restarted. It will join the transfer again. The bit torrent clients check already downloaded pieces for consistency by default, so all the data that was retrieved already before crash and is not corrupt won't be downloaded again.

When a peer finishes download, it keeps on seeding until all other peers are finished. This is to still contribute its upload bandwidth, so the remaining peers can finish faster. The transfer initiator waits until the last peer has finished download. That means it will possibly be waiting forever, if any of the peers died during the transfer. This is however desired behavior: when we started the transfer, we wanted to get the file to the all specified peers. If any of them dies, there might be further trouble with that host that would require manual human intervention. Yet as long as the initial seeder doesn't finish the transfer, you can just fix the broken peers and upon restart they will resume the transfer. Once all are fixed and finished, the initiator will report finished transfer too.

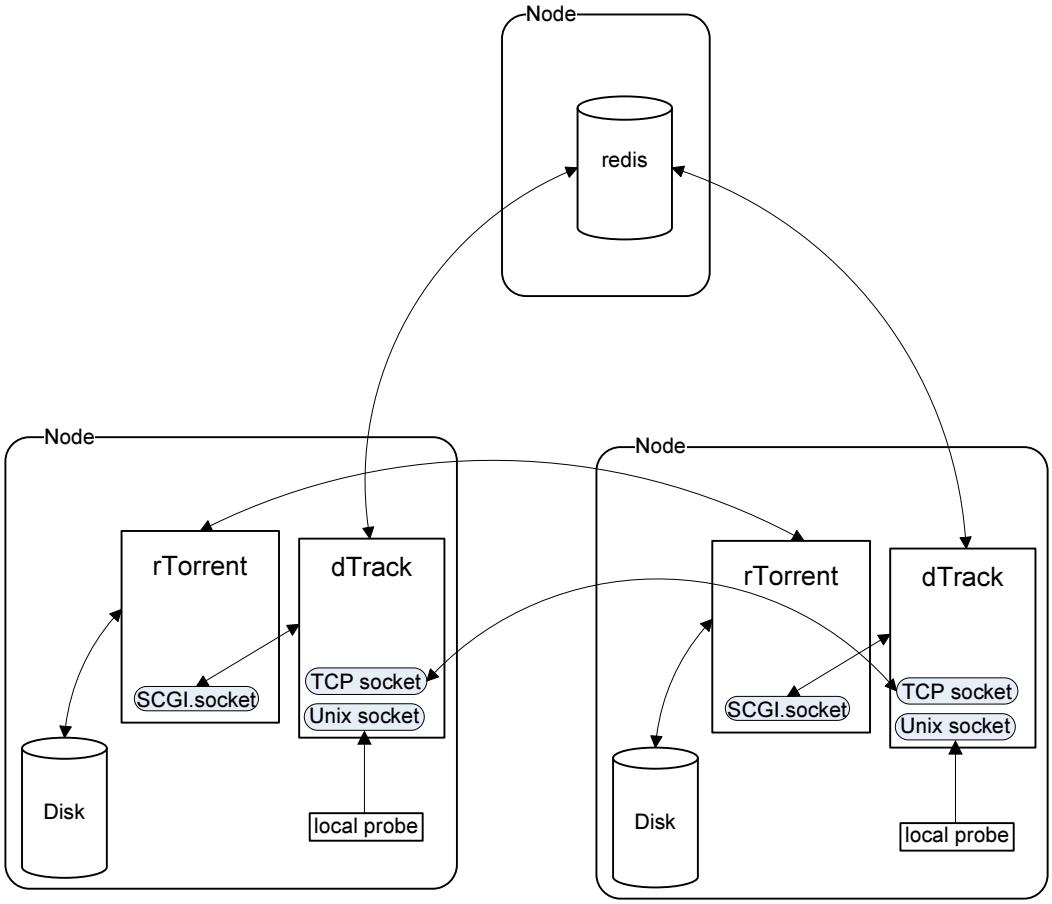



Figure 5 – schema of the solution, the node hosting redis is usually also a peer, but doesn't have to be

## 4.2 **The overlay**

The main contribution of this thesis is the new point of view on overlays. As we have shown in the chapter 2, many solutions (not p2p only) gain efficiency and speed once they start considering real network topology in their overlays. With some of the solutions, like MOB and Robber, the nodes only have notion of their location and of locations of other peers. With solutions for BT, which try to peer with local peers (be it either via modified tracker returning local peers, or like in TopBT by the discovery of the client itself), there is only proximity awareness (that the peers are close), but the underlying topology is unknown, or irrelevant for their point of view. The multicast tree optimizing solutions rely on accurate "network" model, yet they only consider possible connections between the peers that are in the graphs they are provided with. These don't always contain all the possible connections (e.g. once a peer can connect out to internet, it can probably connect to all the hosts directly – thus distance of all nodes from root can be 1 – the replication tree would then be a flat one). Also, if they don't relate to the real network topology well, it is possible then that some of the in-graph independent links end up having

part of their real-network path on a shared physical link. Thus capacity of one graph link would depend on utilization of the other link, whereas the optimizing algorithm expects them to be independent and counts with utilizing both links to the full speed. This is also hard to measure before transfer – any point to point probing techniques would return full speed for both links, because in the time of the measurement there are no transfers yet and full speed is thus available.

The most important issue however is, that these topology graphs only contain the participating hosts and nodes, but not any actual network components, like switches or routers. It is then non-trivial to create an overlay that has the same shape and properties as the underlying network, and we are not sure if any of the solutions actually attempt to do so. Nevertheless, in our case, we know the exact network topology. For each node we can tell exactly in which rack and datacenter it is located. Given that the network parts usually have star topology, e.g. an end switch to which the servers (leaves in graph) connect, the speed to every node is usually the same as the speed of the router/switch to the upper network layer (outside of rack). This means that if we have n peers in the rack, we have n-times more connectivity inside (packets doing only one pass through the switch) than connectivity to outside of the rack. We can easily identify this to be the bottleneck in our transfers. And because of this, we have to make sure that we use this shared bottleneck as efficiently as possible, as this would be the main factor determining the speed of our multicast. The algorithms presented by MOB and Robber are well applicable here – and we do make use of them. Yet they consider only one level hierarchy – our location vs. other locations. In our environment we could map this to our rack, other rack. Nevertheless we have one more layer of connections and locations – one datacenter vs. the other ones. This means that there is the same situation as in the racks with switches: each rack has full link speed to the router, but router that connects to the other DC has only few times faster link speed. E.g. 96 racks have 1Gbit each, but inter-DC speed is only 10 gbit. Therefore there is almost 10 times more local connectivity. We then of course need to utilize these links efficiently as well, also because there's usually some fee associated with amount of traffic going in and out the DC. Yet this time, the nodes that should be using robber algorithm are not the peers, but the rack switches – which of course don't support anything like that. In our solution we therefore pick two peers (how and why two we'll explain in following sub-chapters), which are responsible for sending the data from and to rack. We will call these peers **the representants** (of that rack). The other peers inside rack connect all-to-all, but only to these representants are allowed to connect outside of rack. Now that we have the rack **R**s, we can easily abstract the rack's switch by these, as no other local peer would be visible outside that rack. This way we "convert" the layer of switches to a layer of peers, which can now run work division and work stealing algorithms in order to cooperate on efficient utilization of the inter-DC link. In the Hyves network setup, the DC routers don't connect to other datacenters directly, yet they are connected to the main Amsterdam's peering centre. This means that we will again pick some of the peers representing the racks to create a group representing the DC. Thus in our overlay we will have same three level topology as the real network. Moreover our peers are aware of this and cooperate specifically on utilizing the critical bottleneck links efficiently. However, this is not the whole solution. The logical overlay is the same as the physical network, yet the peers are only in the leaves of the real topology. This means that when a piece is transferred to a peer representing some DC in the highest (let's call it root) group, it is in fact transferred

directly to it, going through the DC router and the rack switch, utilizing all the links on the way. If we want to keep our real network topology representation accurate, we need to include this peer in our logical topology in the each place representing the link it utilizes. This means that this peer has to be present not only on the highest layer it represents, but on all the layers below. This creates an interesting situation, where we in fact have "worm holes" in our graph – the same peer can be present multiple times in different places. This is OK and our solution expects this and uses the stored information correctly.

We implement this overlay information as a per-location and per-peer class sets in the redis store. Therefore every location is represented by a set of normal peers, set of representants and set of seeders. The peers that are in the representants set for the given location (rack), will be also present in the location one level above (DC), where they might be normal peers, or representants again. In the highest root group, all the peers are normal, there's no need for representants. As we can see, for this implementation it doesn't really matter if the same peer is present more times in different locations – it is the peer himself (actually, peer's dTrack client) who alters his behavior appropriately to the role the peer has. Another advantage of seeing the world via this per-location point of view is, that if there are in reality more physical links connecting our location to the upper one, it doesn't really matter – the representants in the given location will use work stealing for all these links, since we can easily abstract them as one link with combined capacity.

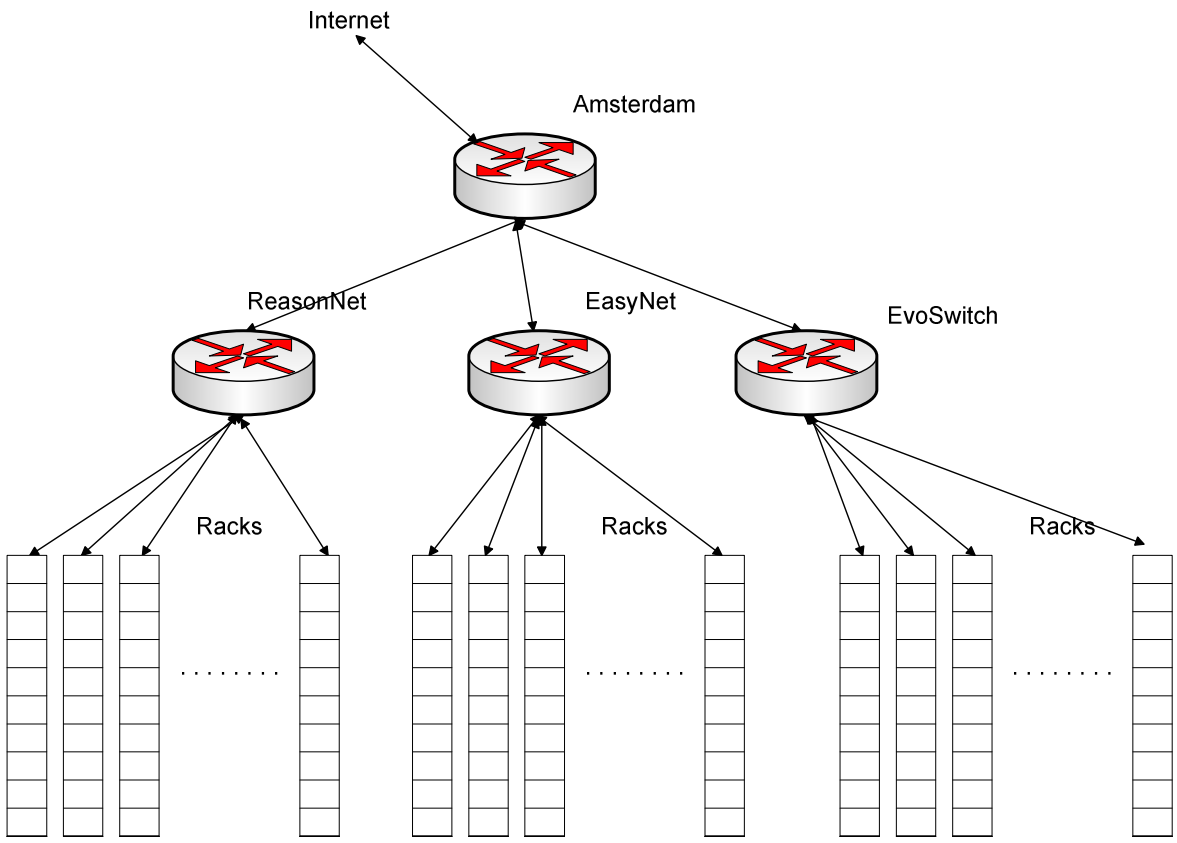



Figure 6 – simple view of real physical network topology

To make this description more clear, figure 6 shows the real network topology and figure 7 shows how is that represented in the overlay we use. The grey rectangles are the location groups. In the figure 6 we don't include all the redundant backup network elements and links.

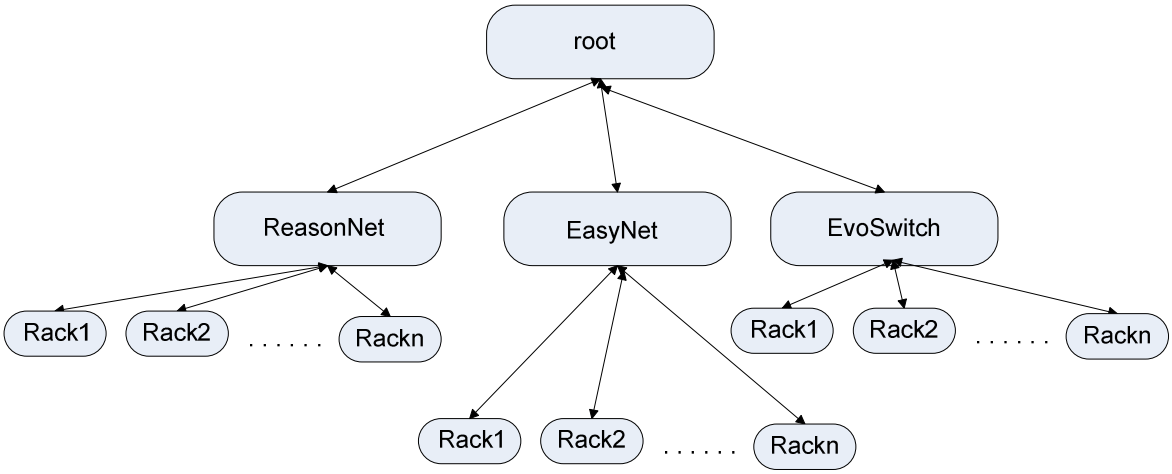



Figure 7 – representation of the fig. 6 network in our topology

## 4.3 **Bandwidth awareness**

In our case obtaining bandwidth information is quite easy. The switches and routers used by Hyves have a SNMP interface, where you can ask for each port utilization. The other possibility for controlling our transfers is to set some speed limit manually. However taking non-intrusiveness into consideration, we never use all the available bandwidth for the transfers, but always leave some reserve. For 1gbit link that can be 200mbit. The reason is that the packets will start getting queued (and thus delayed) already at 80% link utilization – see figure 1. We are "aware" of both download and upload capacities of the link and we treat them separately (which is also not a norm, many other solutions consider the bandwidth to be symmetric).

We store the information about the available bandwidth in the redis, basically in the same location where we store all other information about the given location. In our current implementation, all the representants (from now on Rs) get equal bandwidth share. Yet advanced bandwidth allocation algorithms (e.g. giving more bandwidth to the fastest peer) can be implemented easily, just by altering the formula that computes the R's available bandwidth in the dTrack client. The egalitarian bandwidth allocation yields good results, but that is because we always have at least two Rs for the rack and 5 Rs for the DC.

In addition to these "external" link speed readings, every peer runs local CPU and network utilization probe. The network link utilization is used to give us the free bandwidth from our node to the switch. This information is more up-to-date than the one in the switch (for performance reasons, the port statistics in the switches are updated only each 10 seconds). Moreover we want to relieve switches from any more workload, so we gather this information locally. Calculating the link speed minus used bandwidth minus (performance/non-intrusiveness) reserve gives us the remaining available bandwidth that we can use for the transfers. This is fed to the rTorrent by setting the global speed throttle. We also run CPU utilization probe – more specifically we check for idle time. If this time is below the "safety" threshold (default 20%) then we also start throttling. The reason is that CPU utilization in Linux doesn't only count used CPU cycles, but also time application spent waiting for disk to finish operation (`IO_WAIT`). Therefore if there's something disk intensive in progress on the given node, CPU idle time would drop to 0 if we overload the (already busy) disk with our transfer – because `IO_WAIT` time would grow and consume all the idle. If there's less than 20% idle time, then we set the global throttle to `($idle_time / $treshold) * $available_bandwidth`. Thus e.g. if there was 5% idle time, the global speed throttle would be set to 25% of the available bandwidth. The 100% available bandwidth would be set once CPU idle time rises above the threshold.

One more important remark, all our bandwidth management is set towards maintaining state of at least reserve bandwidth being available. If some other application starts using more bandwidth, we compute exactly how much we need to set our throttles to, in order that we maintain the state where all the other apps use as much bandwidth as they need and our transfers use only what's left (minus the reserve of course). We can compute these throttles exactly, because we know how much bandwidth we are using. In our current implementation, bandwidth usage feedback (followed by adjustment of the throttles) is collected every 10 seconds.

### 4.4 **Creating and managing the overlay**

The overlay as we described it in 4.2 is created from bottom up. This means that after being told to join a transfer (+ given a pointer to the overlay "storage" host – a host where all the overlay related information are stored during the transfer. For storage we use redis [25].), the peer joins his location first. Each peer knows where he is located – this is given to the dTrack at startup as a compulsory parameter. Thus a peer joins a set of normal peers for his rack, by adding itself to this set. It also subscribes to a redis pub-sub channel, where it would communicate with other local peers. Implementation details are in the following sub-chapter. After the peer "joins" the rack, it obtains the set of all the peers currently present and adds them to its rack throttle group in rTorrent. This is to ensure that if any of them connects, the connection speed to him would have appropriate setting. Speaking of which – speed management is a separate thread in dTrack, which reads the limits and based on our utilization, role and position in overlay, computes and applies correct per-group throttle limits. By default, when transfer starts, each throttle group (myRack, myDC, root) has low, safe default limit of 1MB/s. Back to our peer: after adding local peers to the myRack throttle group, peer announces its existence on the rack channel.

This even wakes up overlay management threads in the dTracks of other peers, which would see that there is a new peer. They would then add this peer to the throttle group and tell their rTorrent to connect to that new peer. At this point peers can already start exchanging the data.

The peer then sleeps for a short random time, after that it checks if there are enough representants (at least two for rack – explained in the following sub-chapter). If not, then he promotes himself to the rack representant. He announces this to the rack group, so that other representants can adjust their throttles (now they have to share the rack <-> DC bandwidth with one more R). The R then joins higher level group - adds the new peers from the DC location to its myDC group and connects to them. Once this is done, the peer will again check if it should become R and possibly join higher level group. Thus in the beginning, when peers are joining the network, they try to get to the highest possible location. This ensures that the torso of the overlay is created ASAP when the transfer starts and the data can start flowing immediately. Creating the overlay for about 1200 peers takes just around 2 seconds.

Because every change in the overlay (new peer, new representant etc.) requires all the peers in the given group to react, possibly triggering another changes, we prefer that the group membership doesn't change too much. Because of this, once a peer becomes a representant, it never steps down (this would trigger a race among the other peers to replace him).

After the peer has finished rising in the overlay, he has to act his highest role. That means that the work division – high priority pieces selection – is done according to number of peers and his ID in the highest group. The same is true for work stealing. You might ask for which link the peers cooperate in the highest group – the answer is, for the uplink of the initial seeder. While the peer is in the overlay, it listens for events – those might be new peers joining the location (this means not only new peers to transfer data with, but also new work division. If that peer is a new representant from our rack, this means we have to share bandwidth with him = re-adjustment of the throttles). The same actions apply when a peer finishes a download. It stops being a normal peer and becomes a seeder. Seeder doesn't consume download bandwidth, thus other representants can adjust their throttles. The peer remains being seeder until all the peers have finished the transfer. This is because in the moment he finishes download, he usually still has some other peers connected and downloading from him. We don't want to interrupt any running transfers, simply because the combined download speed of all peers cannot be higher than the combined upload bandwidth of all seeders and peers. This represents the first bottleneck, the second one is then the speed of the links. Important thing to realize is that methods like work division and stealing not only efficiently use the link, but also give different peers different pieces, thus enable them to better utilize their upload bandwidth (everyone has something = I can download from everyone). A practical remark – once the peer becomes a seeder, the download is complete for him. Therefore any further actions that should be made after the transfer, can already start happening. The only condition is that the file remains intact (so that it can be read). If we need to move the file to a different location, this can be temporarily solved by symlinking it there and doing the actual move after the complete transfer has finished.

Important aspect in this type of topology is that the peer is concurrently present in multiple locations. This is necessary to depict both real topology in a "virtual way", but

also the real fact that the peer is always a leaf, thus if he downloads something in his higher level group, bandwidth would be consumed on all the physical links on the way down to the peer. To reflect this, the peer has to adopt specific bandwidth management. As we have described already, peer adds all the other peers to throttle groups, based on which location they are in. In our implementation, the groups are `myRack`, `myDC` and `outside`. If I am doing some transfers with peers in the `outside` group, this means that I am using that bandwidth also in the `myDC` and `myRack` groups. The groups max speed is determined by my max. throttle on the link leading to that group, e.g. speed to the `myDC` group is determined by how big share I get on the link from rack to DC. However the same link leads not only to DC, but further also outside. Therefore the bandwidth I can use for communicating with peers in `myDC` is `myDC_max – Outside_utilization`. And the same applies for the link leading from my host to the rack switch: if I am already doing transfers to outside and DC, then in the rack I can utilize only what's still left free on this link. This way I ensure that the combined bandwidth of communication to all these groups would never exceed the available bandwidth on any of the links. The priority of bandwidth allocation is from the highest group to the lowest: the task of being representant means utilizing the link efficiently and that means using my full share.

If our peer is initial seeder, then there are few little differences. First, the initial seeder goes always to the highest level group. This is to ensure better "spread": in the situation where init. seeder would stay in the rack, the first ones to get the pieces would be the peers that are the closest – in it's own rack. Then the peers in it's DC, and only then peers in the other DC – and only then the "lowest" rack peers in the other DC. Whereas if the init seeder is in the root group, all the peers from all the DCs get equal (workload) share on the init peers upload link. Init seeder is of course seeder from the beginning and doesn't consume any download bandwidth. In our implementation, init seeder is also the one to check if the transfer has finished, and if so, to decompose the overlay.

### 4.5 **Design decisions and implementation details**

So far we have been describing our concept abstractly, but in this chapter we will show which design decisions did we make and also describe some important implementation details. Because we did this solution for Hyves, we had to respect their specific requirements. One of these was that they want a working solution. They also wanted us to take safe approach: start with something that already works and only then improve it and add features. This meant that we never did any big jumps, and were just adding functionality one after another. Many times this meant building a very simplistic base design, just to see if the given improvement works. After it did, some other functionality depending on it was added. The original base design has many times proven itself to be good enough (performance wise) for the specifics of the Hyves environment. For example, representant selection algorithm (will be explained in this sub-chapter). However, if we wanted to deploy our solution in real-world scenario, or some other environment with different specifics, we probably would need to improve or alter some of the algorithms and solutions, mainly the representant selection, but also e.g. bandwidth management. We give our suggestions and some outlook of possible future development in last 5.4 sub-chapter and in the final part of the conclusion.

Let us explain our choice of redis as a store of overlay information. Redis is a very high performance key-value store. The initial aim was something like persistent memcached (thus high focus on performance). Yet in it's current form, redis is rather a data structures server. It supports many more types than just key-value, e.g. lists, sets etc. Most importantly, it also provides operations on those data structures – thus you have complete sets functionality (joins, difference, substractions etc.). Moreover, still being performance focused, these are implemented almost always in the optimal way. Ordinary computer can do about 100 thousand operations per second. Therefore in our case, one host is powerful enough (with quite large reserve) to store information for an overlay of about 1200 peers and cope with dynamical updates. These happen mainly in the beginning when the overlay is created – this however takes only about two seconds. The other "high activity time" is the end phase of transfer: since our solution works quite well, we do get multicast behavior – most of the peers finish the transfer in almost same time. Rs are then becoming S(eeders), this triggers further reorganization. To save on unnecessary operations, we employed some heuristics. For example, if a peer is 95% done with the download, he won't try to become R – soon he would finish the download anyway and then he would become S and someone else would then try to replace him. The only exception is when there are no other Rs – that is to keep the overlay connected (so that the peer can still seed into higher layer groups).

As it was previously mentioned, to store information about given location, we use sets. For example a rack `rsn-27` peers can be found in the set `[HASH].[DC].rsn-27.N`. For communication and for triggering events on the peers, we use redis publish-subscribe channels. The functionality is simple – first, peers subscribe to given channel, e.g. for rack that's `[HASH].[DC].[rack].all.peers`. If a peers want to notify others about the new situation (like joining, or becoming a seeder), he just publishes a message to this channel. The redis will then notify all the subscribers (send them the published message). Redis provides libraries for various languages, we used ones for Python (dTrack is implemented in Python). We simply have a thread listening on the given channel – the thread is blocked on the read message call and unblocks once there is anything published.

To keep track of the transfer, we have sets `[HASH].PENDING` and `[HASH].FINISHED`. Once the peer starts the transfer, it adds itself into pending set (this is useful to debug if the peer failed before, or after starting the transfer). After the transfer is finished, peer moves itself to the finished set. Once the initial seeder sees that all the peers are in this set, he sends global finished transfer message and then decomposes the overlay. The peers also update on-line transfer statistics – current transfer speed and bytes already downloaded. This makes it easy to see in real-time, in one place, the progress of all the peers and thus the whole transfer. Stats are updated every 10 seconds. There is also time-to-live set on the speeds key, after which redis will automatically remove it. This is useful for seeing if the peer is alive or died (= didn't refresh his stats in time).

We have decided to make dTrack a distributed solution because it wasn't feasible to run many functions we needed centrally. Moreover, distribution didn't bring us only performance scaling, but also robustness and resiliency. The local dTrack client is tightly bound with the rTorrent. We can say that dTrack is de-facto just using rTorrent's (work stealing) file transfer services, but is doing everything else on its own. If any single peer has some performance problems, or even crashes during the transfer, it doesn't matter much to other peers. It doesn't slow or block the transfer, since there are always enough

other peers to provide data. The dTrack client can also manage peer's failures and restarts. When we restart a peer during the transfer, after the new start, rTorrent will check all the downloaded data for consistency and download only what is still missing. Also dTrack will re-read where the peer previously was in the overlay and again resume its role. This way we can say that the transfer wasn't actually restarted for this peer, it was just paused and resumed. The solution of restarting peers in case of trouble is also easily usable with various monitoring systems, like Nagios. If a peer is detected dead/too slow, just restart it. In most of the times the issue is somewhere outside our framework and would disappear after the restart. This also works for the restart of the initial seeder: during the time when redis is not available, there would be no changes made in the overlay, but all the connections, speed limits and transfers will still be in effect as they were previously set up. That means bandwidth wouldn't be dynamically uptated. However given efficiency of our solution, transfers don't last too long. Also we have made some choices to keep the overlay stable, therefore we already have quite good overlay after the initial 2-5 seconds of transfer. Yet initial seed remains SPOF: if he didn't upload all the data to the peers before his failure, there's no way they can finish the transfer. Restart does however usually fix a lot of issues.

Another important decision is a no-lock approach. Maintaining locks for synchronization among nodes in distributed environment is very difficult. We would have to deal with peer deaths while holding locks, also with the issues of the lock managers. Yet we see performance as the biggest issue: even with 1gbit network, RTT is about 1 ms. That limit's us to only about 1000 lock operations per second. Yet in our solution we see peaks of about 20000+ operations/s in the initial phases of building the overlay. Instead of using locks, we use atomic operations provided by redis (also via redis transactions, which can actually group more (non-atomic) commands). Therefore most of the time we wouldn't need the locks anyway. The fact that we avoid use of locks where possible results only in some occasional conflict, which doesn't break our system, just makes it lose a bit of efficiency. A good example of this is two representants deciding at the same time to download the same piece into their group. That piece would really be transferred twice, thus hurting efficiency a bit, but otherwise all is working fine.

Our implementation uses a specific representant selection strategy. For each rack group we select two Rs, for DC groups at least 5 Rs. We select two Rs both for redundancy and performance reasons. If we only had one R and that would fail, it would take a bit of time to recognize this situation and to replace it. When we are using two Rs, if one fails, the other one still keeps transferring the data, albeit the speed would not be fastest possible (one R of two is allowed to use only half of the available link speed), yet data would still be flowing. Another reason for having (at least) two Rs is that we avoid situation where the speed of the higher level link R would need to utilize is the same or higher than the speed of the lower level links R is assigned share of. Because of bandwidth allocation policy (higher level links have priority) that would mean that the R wouldn't have any spare capacity left for communicating with the lower level groups. Therefore our R could easily be present in the root group, but this membership could exhaust all its local link capacity and it wouldn't be able then to do any data uploads to the local rack group. In general, we call the ratio of the peer's assigned download rate and the peer's possible upload rate **duplication ratio**. When we have two Rs, each of them would be assigned half of the rack <-> DC link speed bandwidth. Let's say that this would be 0.5gbit. Because Rs themselves

have 1gbit links from their host to the rack, that leaves them with 0.5gbit spare capacity for communication with local rack peers. Two Rs then have combined retrieval speed from outside of rack of 1gbit (full link utilization), but also still have 1 more gbit of combined spare rack bandwidth. Those Rs can then easily both serve the roles they have in higher groups as well as saturating all the local demand (as local capacity is the same as the download capacity of the link, therefore it's impossible to get data to rack faster and we are running at highest possible speed). Each of the Rs thus then has 2:1 duplication ratio. The same situation occurs when selecting Rs for inter-DC communication. These links have 10gbit capacity, yet around 6.5 gbits are used almost permanently and we also keep 2gbits spare capacity. That gives us 5 Rs, each with 0.5 gbit capability (from rack link), to utilize 1.5gbits of bandwidth. As we can see, also here the duplication ratio is higher than 1 (2.5:1.5). This really is not 2:1 as in previous case, but still gives us 1gbit spare capacity for uploads in the DC location group. Because 1gbit is the speed of the link from rack to DC, it doesn't make sense to have more spare capacity, as we are physically bottlenecked to 1gbit anyway. On the other side, it is beneficial to have less Rs, because of complexity and scaling reasons (more about that in 4.8).

Page 9 in [17] also mentions the concept of **gateway peer** – in order to get the data to ISP, a single peer is selected to do this transfer and act as some form of a "gateway" for the data flow. In this paper they argue that performance of this peer is critical and influences how fast will the other peers inside ISP get the data. They recommend four times the speed of a regular internal peer. In our solution, the combined DC download bandwidth is only 2.5 times of the local peer's max capacity, but that is still enough to fully saturate it. Therefore even compared with the situation where our peer is connected to seeder directly and can use the full link capacity only for itself, we don't see any possible slowdowns for any peers, since in our setup, we always have at least the seeder's full link speed spare capacity (each group can upload 1gbit more than it can download).

The heuristics for selecting the best Rs are in our case based on first come first serve rule. Thus the first two peers from the rack to join the transfer will become Rs. This has two reasons – first is to make data flow as soon as possible, thus we need to construct fully connected overlay ASAP. The other reason is that if we have equal conditions for all the peers and they are all told to join the transfer in the same moment, we argue that the first two peers to react to this represent the two fastest and most capable peers in the rack, meaning they would be good Rs anyway. Moreover, before you start the transfer and actually do transfer some data, you actually don't know anything about the peers' capabilities and transfers speeds. Once the transfer runs already, you can then employ more elaborate R replacement and selection strategies. However in our case the selection of first peers yielded good results and full link utilization, thus we didn't see the need for more advanced R selection algorithms. Yet in other scenarios these might be necessary. It is also important to realize, that this can have quite some influence on performance, especially if the chosen Rs are not able to fully utilize the assigned link shares.

Peer exchange and distributed hash tables are disabled in our solution. These two methods of peer discovery and automatic inclusion into transfers would interfere with the overlay and throttle groups maintained by dTrack. There's no sense and logic about these in the rTorrent, thus rTorrent shouldn't be adding any peers on its own. We also use private torrents, which explicitly say that peer exchange and DHT should not be used for peer discovery for this transfer. This adds a layer of security: if any of the internal torrent files

should get out to the public network, there would not be any tracker or other way to find any peers.

Our solution also supports multiple concurrent transfers. Each transfer has its own initiator and it's own overlay. However dTrack takes the speed limits and throttle groups taking all the transfers into consideration. User might even specify the transfer priority – this is communicated to rTorrent, which would then prefer one of the transfers, all still within the safe speed limits set by dTrack.

## 4.6 **dTrack and control commands**

The dTrack client is the main controller of all peer's activity. It contains all the logic related to transfer management. The rTorrent client itself only does the actual data exchange, but it doesn't develop any activity by itself. Initially when we were testing functionality of work stealing, we had just a set of interconnected shell scripts which fed rTorrent. This simple solution worked well for simple scenarios, but later when we wanted to have better integration and use advanced logic, we have implemented the dTrack in Python. The application itself is multithreaded and event driven. Most of the activities dTrack provides (bandwidth management, "life" inside overlay – becoming R, S etc., connecting to peers and accepting connections) are quite independent and they should be able to happen concurrently. Therefore all the classes inside dTrack are actually threads. The group of the threads related to overlay and bandwidth management are event driven – the events are bandwidth changes, peers joining overlay, becoming seeders etc. Most of the events are triggered by a message on one of the redis channels the peer is subscribed to. Yet this is not a rigid condition, many times events are triggered by other threads (e.g. when receiving external command from user). In general, it's quite easy to do modifications to this design, since all the classes being treads means they are quite independent. This multi-threaded design also enabled us to easily manage multiple concurrent transfers – there are no limits on the amount of concurrency. There is very little locking inside dTrack. Communication with rTorrent, pushed via its SCGI socket is one of the parts using locking – this was because rTorrent 0.8.6 didn't support concurrent SCGI commands. Yet this functionality was added in 0.8.7 (after we had implemented dTrack). Therefore another lock can be removed. Nevertheless, this lock doesn't really limit performance or responsiveness, since it's not that many commands we need to send to the rTorrent.

Let's now present an overview of the interfaces dTrack uses (see also figure 5) and the commands available. One more remark – command reply codes resemble FTP commands syntax – first is always code, e.g. 200 for success, 400 for fail, 500 for unknown command, followed by space. These codes are parsed by the command sender. After the space each reply contains human readable comment, informing about the cause of reply or state of action.

***TCP socket***

dTrack is listening on an ordinary TCP socket on the port and interface specified at startup. Default port is 10002. Communication via this socket is used by other dTrack clients. Available commands:

**CONNECT** <peerIP> <rack> <dc> <HASH>

A command to connect to the given peer. His location is specified, so that we can add him to the appropriate throttle group before we connect. Also HASH is specified – so that we know which transfer to join (this has to be communicated to rTorrent explicitly, it won't connect to peer and then start discovering which transfers it is actually doing). This command is used by other dTrack clients of peers that want to connect to us. Reply is `200 OK` if successful, or `400 FAIL` if the peer couldn't have been added to throttle group. This way the other peer knows if it can proceed with the actual connecting.

**TRANSFER** <HASH> <redis host> <destination_path> <torrent_path> <md5 path> <speed_limit> <priority>

This command is used to tell the dTrack to join the given transfer. The destination path is where we want to download the contents of the transfer to. Torrent and md5 paths are local paths to .torrent and .md5 files, uploaded to us already by the transfer initiator. The speed limit is maximum possible speed to use for this transfer (useful to manage multiple concurrent transfers). Priority is transfer priority for rTorrent. Because bandwidth groups don't know about different transfers (they shouldn't since these transfers use the same links), all the bandwidth of the given group would be used by all the transfers (bandwidth is shared for all the transfers). Here the priority tells rTorrent how much bandwidth to allocate to which transfer. Reply is always `200 OK`, as dTrack will keep on trying to join/re-join transfer until it is told to stop.

**PID**

Kind-of ping – the reply is simply the PID of dTrack client.

**STOPALL**

Command to stop all the transfers. This command triggers internal events for each of the currently running transfers to stop them. The reply `is 200 Stopping <number of transfers> transfers`.

***UNIX socket***

dTrack is also listening on a local unix socket (path specified at startup). This socket is mainly used for getting information about finished transfers and starting init peer.

**HASH** <HASH> <STATE>

This command is posted by a script, which is called by rTorrent after finishing the download. This script then does md5 check of all the files and reports either success or failure to dTrack via this socket. This command means dTrack will change the peers role to the seeder in overlay. There is no reply to this command.

**INIT** <HASH> <redis host> <source path> <torrent_path> <md5 path> <speed_limit> <remote destination path> <priority>

This command tells the dTrack that he is the initial seeder for this transfer. It specifies where the files to transfer are located, also where we can find .torrent and .md5 files. These files will be uploaded to all the peers and then they will be contacted directly

by using `TRANSFER` command on their TCP socket. The peers to transfer files to are specified by the user. They are read from redis set `<HASH>.PENDING`. Reply is `200 OK`.

Exactly same commands as for TCP socket (same syntax, functionality, responses):

**`TRANSFER`** `<HASH> <redis host> <destination_path> <torrent_path> <md5 path> <speed_limit> <priority>`
**`PID`**
**`STOPALL`**

### 4.8 **Scaling**

Because our overlay topology tries to (abstractly) copy the real world network topology, our solution scales quite well – because the real networks have to be built in a scalable way. Otherwise they would be slow by design and no overlay would help. Given that today most of the networks are compromised of routers and switches, which all use star topology, the "network scaling" is quite good: with every hop further, we can reach exponentially more peers. In the real network one hop is represented by a single network element. In our solution it is represented by a group of few representants. It is important that the number of representants is lower than the number of peers in the group they represent. Otherwise too many peers would make it to upper level groups and these would then be extremely huge, running all-to-all communication (e.g. letting everyone know which piece we `HAVE` or have `START`ed) wouldn't be feasible. The solution wouldn't scale.

Let's now look at this situation from other point of view: peers in the given group will send `HAVE` and `START` messages for every piece they download to all their peers. This means $2n^2$ messages for all peers together. Yet because switches and routers have star topology, every new peer (a "leaf of a star") adds connectivity of 1. Therefore on one side we need to transfer $n^2$ messages (or use $n^2$ bandwidth), but on the other side, we have n times more connectivity. More connectivity makes sending one message cheaper/easier – n times easier (or, in the same time we can transfer n times more messages). Therefore the overall communication burden grows linearly with the number of peers: number of messages from point of view of a peer has linear complexity. This however still means – the more peers, the more burden. But because we have roughly n-times less representants for the higher level group (as there is n times less connectivity for the external "uplink"; roughly because in reality we have one more R for redundancy), our approach can really scale well – the number of peers in a higher group depends on how many ports the switch/router has and on the number of Rs needed to saturate each of the ports.

# 5 Evaluation, practical experience and further possibilities

In this chapter we evaluate our solution with various settings, to see how they affect the performance. Because of the research and comparisons already done in [5], the only other solution we compare against is bare, unorganized bit-torrent (albeit using our low-latency modification). We use it to demonstrate the efficiency of our overlay. In sub-section 5.2, we show how various enhancements (work division, work stealing) influence the real world results. We also demonstrate some real usage results and applications of our solution (**p2p server deployment**). In the end we discuss applicability of our solution in different environments and suggest possible improvements.

## 5.1 Overlay performance and efficiency

First we will demonstrate why we needed to create any kind of overlay, or advanced peer organization, at all and couldn't rely just on the ordinary Bit Torrent defaults. The problem with normal BT is that we don't control which peers our node connects to. Therefore we cannot determine for sure where would the traffic flow – if it would be inside rack, or outside using the shared link. Therefore to stay safe and non-intrusive, we would have to limit each peer to the speed, which would be his share on the available rack link capacity, if all the peers inside rack are going to use it. For example, if there are 15 peers inside rack, available capacity is 600mbit, each peer would be limited to 40mbit speed. If we now consider also the link between the datacenters, the limit would be further lowered: say we have 850 peers inside one DC, available link capacity of 3,5 gbit would mean around 4mbits per second limit. This can be seen as an example to our statement from the introduction – if you simply take existing solution and just slow it down, so that it won't introduce any latencies, it becomes so slow that it's almost unusable.

The possible improvement here would be that we start to be location aware, thus our overall speed limit won't be minimum of all limits, but each link would have its own. Peers would be added to throttle groups the same way we are using in our approach. This would make this solution already much faster, yet if we think about it, this is equivalent to all the group peers becoming Rs. As we have shown in 4.8, that gives bad scaling. Another problem with not controlling who connects to whom, is that BT clients limit the amount of peers they connect to. Therefore even if we assign shares of the available rack link capacity to all the peers, it is not sure that all of them would actually use their bandiwdth to the full extent. Some of the peers could connect only inside rack, or only inside DC. Because we have identified these links as bottlenecks, not using all the available capacity means narrowing them further down. Nevertheless we have decided to use this enhanced default BT as a reference for comparison with our solution's speed and efficiency. Figure 8 shows the testing setup. We have used lower, fixed speeds, so that we have results less dependant on the real link utilization (where the available link capacity would also depend on e.g. time of a day). File size is 173MB and the presented results are averages from 5 tests. In our test we have 5 peers in B-5-rsn rack, 4 peers and init seeder in B-12-rsn, 7 peers in Esy17c and one in Esy7b:

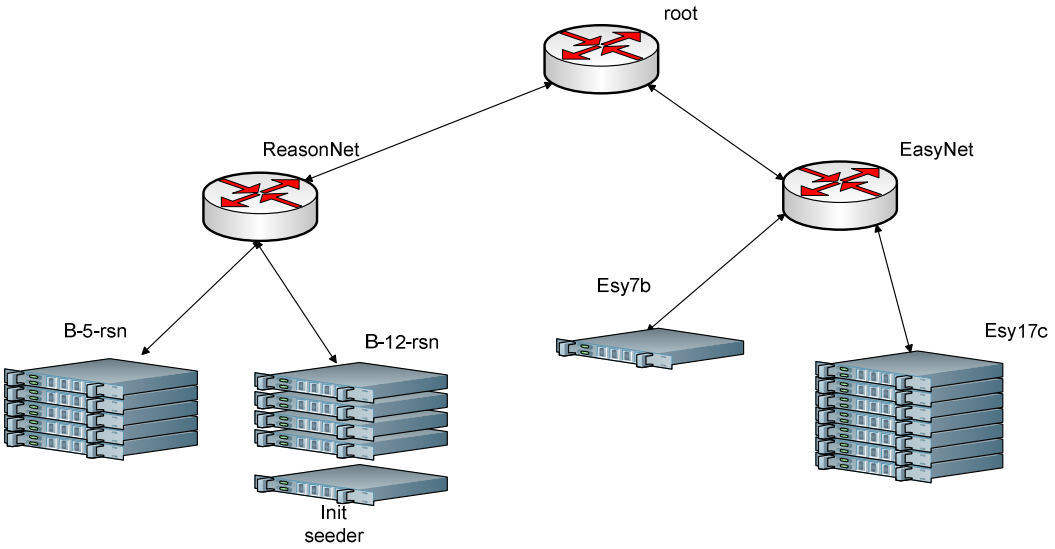



Figure 8 – test scenario physical topology, link speed is limited to 10mbit

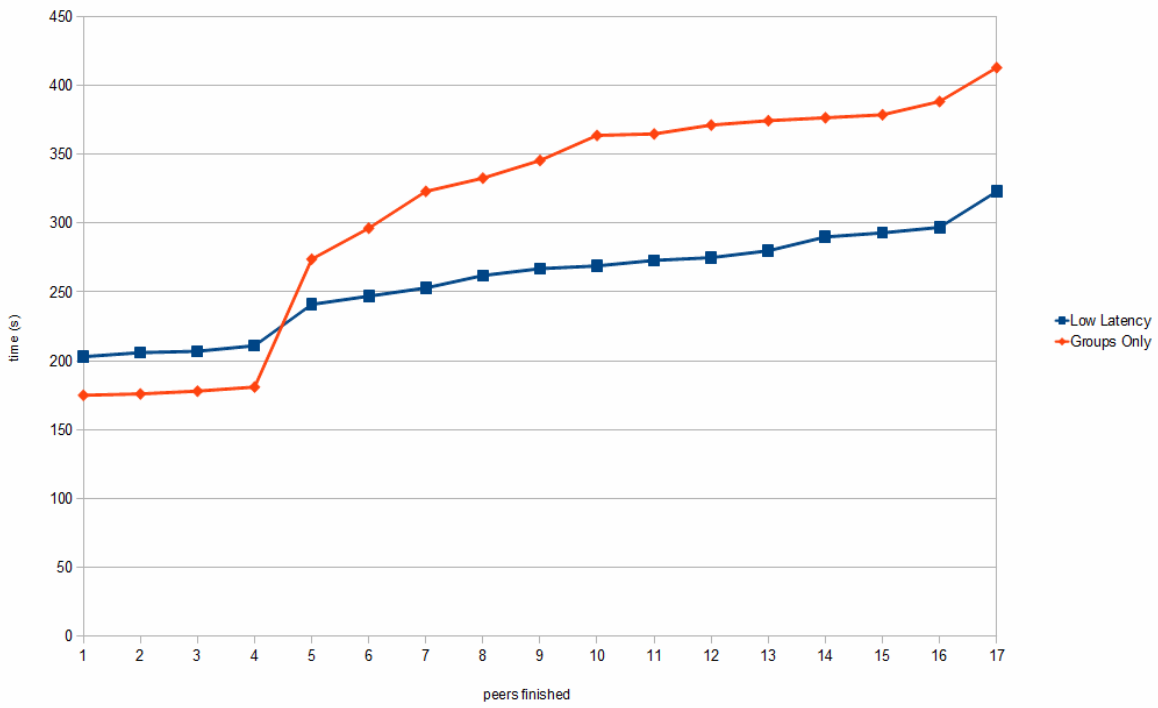



Figure 9 – finish times comparison

As we can see on the download finish times in figure 9, with the simple groups-only solution, it depends a lot where the peer is located: the peers located in the same rack as the init seeder finished quickly, but others had to wait a bit longer to get all the data. With our solution, finish times are much less spread. The peers in the init seeders rack don't have that big advantage anymore – in our overlay the init peer always goes to the root group, so that all the locations can get equal share of his upload bandwidth. The relative slowness of the enhanced default groups-only BT can be attributed to both duplicate piece retrievals but also not full utilization of all the assigned bandwidth shares during the whole transfer. One more interesting parameter to mention is the init peer's seeding ratio (how much data was uploaded compared to the file size) – for the unorganized groups only scenario it was 4.78, for the overlay + low latency client the seeding ratio was 1.47. This means that there is much more data flowing between peers and the initial seeder doesn't have to be contributing a lot of upload during the whole transfer.

## 5.2 **Effects of work stealing and work division**

We'll have now closer look on how each of the features we have added to the rTorrent affects the performance of the transfers. For comparison purposes, our test setup is the same as in 5.1, thus described on figure 8.

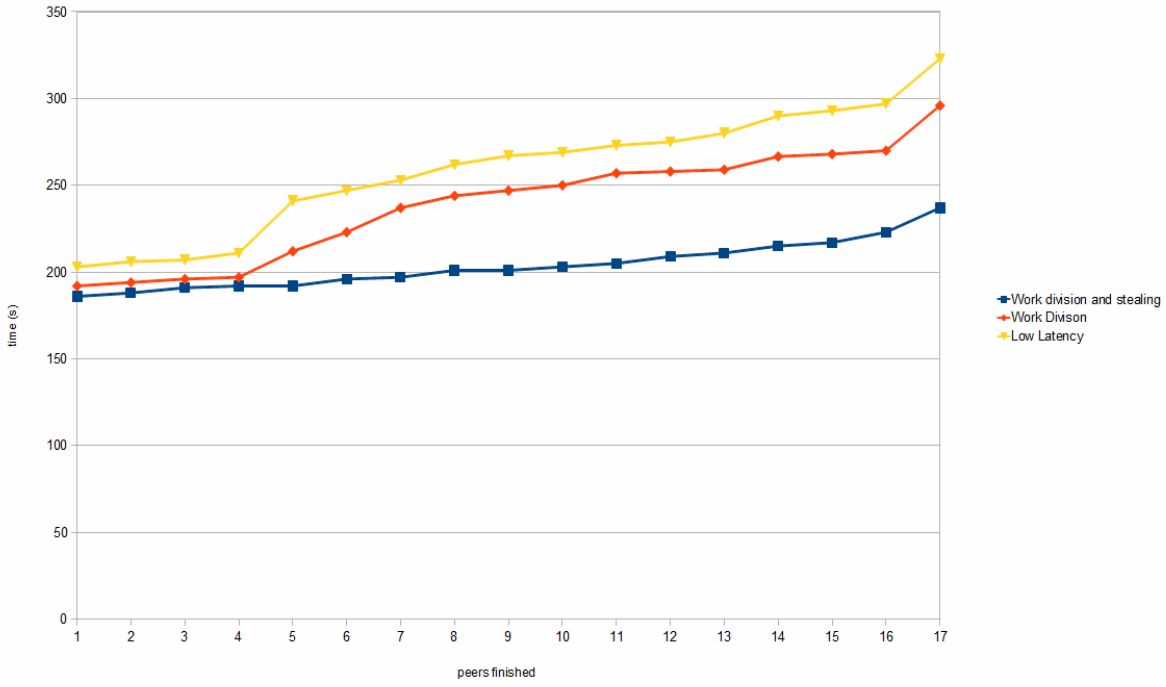



Figure 10 – comparison of enhancements influence

As we can see on the finish time distribution graph, the work division already has quite positive effect. The 4 peers that can connect to the init seeder directly have always the fastest finishing time, but due to work division, peers from outside rack are more effective on getting the data from the init seeder. This is visible as a smaller "jump" in the graph compared to low latency only solution.

The work stealing adds the biggest improvement. The work division works fine, but is static – according to the peer's ID, which doesn't have to change for the whole transfer. Work stealing is based on dynamic reaction to the current situation, therefore it can adapt better to the slightest changes and fluctuations in the data flow.

The init seeder ratios for these three methods are 1.47 for low latency only client, 1.31 for work dividing client and 1.12 for work stealing client. As we can observe, the more efficient solution, the less the init seeder has to upload. Especially with work stealing, peers are able to get the data mostly from themselves. We should also explain why is the ratio actually higher than 1: if we let the init seeder just upload the file once, by e.g. enabling init seeding in rTorrent, the distribution would work, but there would be two drawbacks: 1. The init peer wouldn't contribute any further bandwidth, even if it could. This makes transfer slower. But most importantly, 2. there is possibility that a peer that has some rare piece dies. Because init seeder is not seeding anymore and because no one else would have that piece, the whole transfer would be stuck. Therefore init peer being present from the beginning till the end is contribution to robustness.

### 5.3 **Practical experience and adaptability**

In practice we have used the developed file transfer framework mainly for file transfers. For demonstration purposes, how this looks in real practical situation with live traffic on the site, see figure X. On this graph we can see the cumulative bandwidth used by all the servers in the "bricks" deployment. These are servers storing user media and are very busy with the disks, therefore our solution limits itself to a lower speed. The two following spikes present the transfers with low-latency rTorrent, followed by rTorrent with work stealing and work division enabled. The spike couples represent transfers to 100, 200, 600 and 1200 hosts. What we can note is also background traffic of about 1GB/s which drops off in the end of the test. As we can observe, the second transfers are faster – the peers are able to achieve faster rates due to more effective link utilization. This real live site test demonstrates that our solution scales very well – even with 1200 hosts, the average transfer speed per host is almost the same as with 100 hosts only. This can be also seen on the time the transfer took, which grows only slightly. The actual reason for this growth is, that the more hosts you transfer data to, the more likely you are to encounter very slow hosts, which take much longer time to download the data.

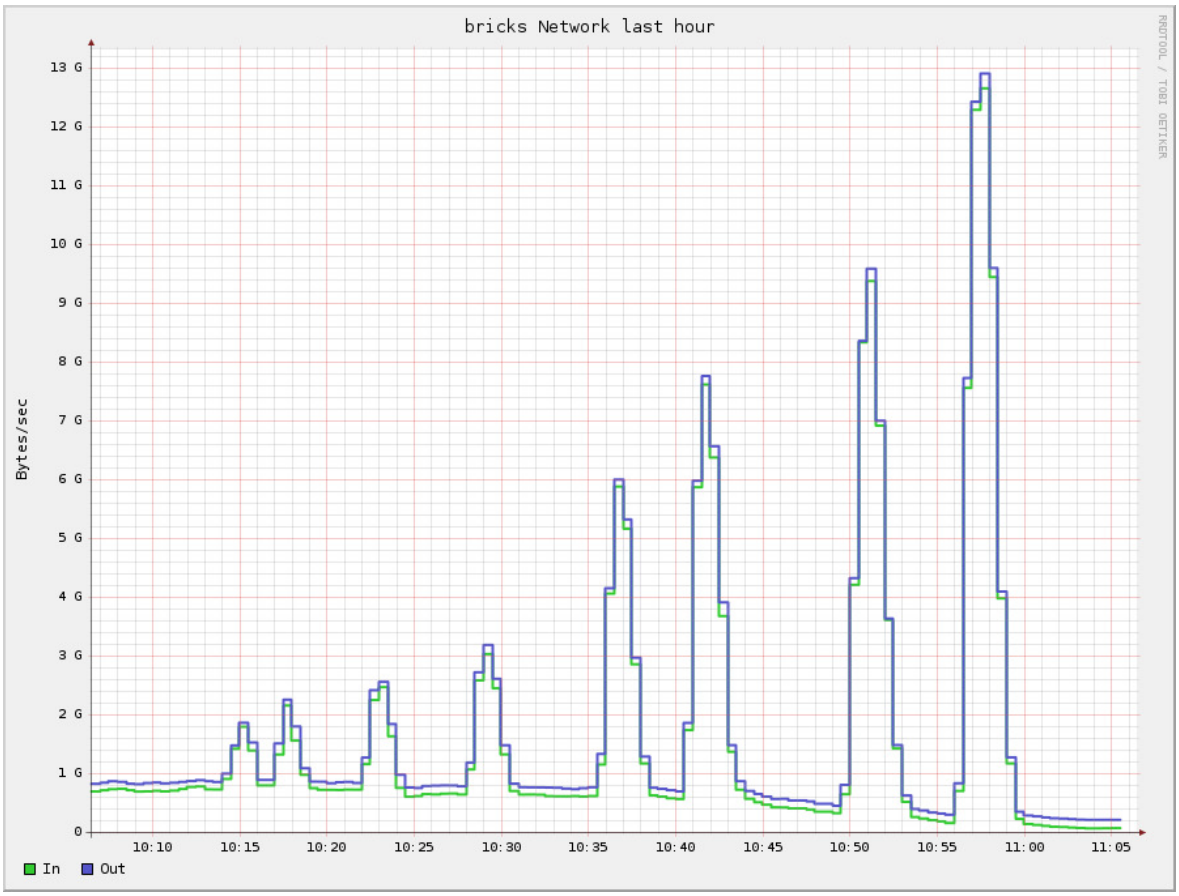



Figure X – live site file transfers, 100, 200, 600 and 1200 hosts

The second major use of our framework is **rapid concurrent deployment of servers**. We have established a so-called p2p boot server. This is an ordinary boot server, except the scripts are modified to use our peer-to-peer framework for transferring the disk image (cca 880MB). This makes concurrent deployment of almost any number of servers very effective and fast. The portion that needs to be downloaded directly from the boot server, before we can start rTorrent and dTrack client is very small: After the initial kernel is transferred via TFTP, remote disk is mounted via NFS. This is the disk from which the system does a very minimalistic boot. Because disk is NFS mounted, only the blocks which are actually read are transferred. The local partitions are created and our framework is used to transfer the disk image. After that is done, system either restarts, or only chroots to the local disk. The amount of data which is transferred from the boot server is very small – up to 40MB. This is the only non-scaling part of the process. If required, this can be reduced by further optimization of the boot image. The main part of the transfers (the 880MB disk image) is however transferred via our framework, thus has no problems with scaling. We mention this use case just to illustrate how can be our framework integrated into seemingly unrelated functionality and that no complicated solutions are necessary to have a very fast and scaling server deployment solution.

Our framework can be also used well in virtualized environments. If a server hosts multiple VMs, these can see it just as belonging to the same lowest level location in the topology, equivalent of a rack group from our case. Moreover, VM migration is easily doable: upon realizing that VM has moved (or being explicitly told), the peer just needs to leave the old location, disconnect from its old peers and join the new location, connecting to the new local peers. This of course also works fine with live VM migration.

Usability in the cloud environments is good as well. Here the user doesn't know his exact location, yet shouldn't maybe care so much, since here the links are shared anyway, so not utilizing them fully only means someone else will have more bandwidth. The situation gets more interesting when your application is deployed across multiple data centers (in EC2 nomenclature – availability zones). This is usually necessary for high availability and fail resistance. Yet communication between these DCs is slower, but mainly costs. Here our solution (albeit with more simplistic overlay – only one layer) helps to ensure both high transfer rates and also efficient bandwidth usage, resulting in lower expenses on bandwidth.

## 5.4 **Development experience and possible enhancements**

During the development of our solution we have gained quite some experience and encountered some issues. The main approach, emphasized by Hyves, of enhancing the framework step-by-step has proven to result in a working solution at most of the times. On the other side, we didn't implement all the advanced features we wanted (e.g. dynamic bandwidth management). Still, even without this additional functionality, our solution has good results. The implementation itself was many times tricky. Debugging a distributed application is not a trivial process. If you don't have clocks on all the hosts synchronized up to a millisecond (which you usually don't), searching for causality can be tricky – transfer of a block, or sending a command usually happens within milliseconds. Even the most trivial issue of segfault can be difficult to debug: When you have multi core machine, where the OS is using signal queues on all the CPUs, and you have a lot of signals in these queues (because you're doing a lot of network IO or because of other processes running on the same machine are busy), it can take some time till your segfault signal is processed. In the mean time, your application might process few more messages, write some more log lines. When it then finally exists upon segfault, the actual message/issue which caused it, is not the last one written to log.

As for the future enhancements, advanced R selection and management algorithms look like an area which should be investigated. The initial selection can stay, but after some time when transfers are actually running, the detailed information about how much bandwidth is each peer actually using – telling us how fast/capable it is, can be used to potentially adjust who is R or how big share of bandwidth he gets allocated. The main reason behind this need of bandwidth management is to ensure that the available speed is always utilized as much as possible, otherwise the transfers are slowed down.

Another open area is distributing the information storage: Currently all the overlay information for the given transfer is stored on single host's redis. Yet this can be easily distributed – the first R in each group could store only that group related information (peers, speeds, common channel etc.). The main redis would then only store pointers to the

storage hosts of the given group. Currently having all on one host is not an issue – performance wise, but also traffic wise. Yet if we would like to implement some advanced algorithms that require a lot of communication, or do frequent changes in the overlay, then the distribution would be necessary. It also brings one more advantage – the group related information (and the associated traffic) remains physically local in that group.

Each peer in our solution currently depends on being explicitly told about its location. In practice, this information is not always available. For discovering local peers (to put them into our lowest level local group) we could possibly use services of the local peer discovery, or some other tool, like TopoMon [5]. Another possible network-level heuristics seems to be to try to find out either MAC of the closest switch, or the IP of the router more hops away. This can then be used as a group identifier: all the peers that find the same MAC/IP are connected to the same switch/router. Talking about network level enhancements, utilization of UDP for `START` command looks like good idea. `START` commands themselves are very small (they just send the chunk number), UDP could remove overhead of TCP connection and improve delivery latencies. The last remark is about networks which use shared media. This can be for example the old BNC bus coaxial cable Ethernet, or currently WIFI. Here the network level multicast is very desirable. The issue is that every peer needs to get every piece, yet all the peers share the same medium. Thus without network level multicast, the same piece would get transferred over the same shared link multiple times, which is something we wanted to avoid in the first place. Unfortunatelly this time there really is no way to avoid it, except the network level multicast.

# 6 **Conclusion and future work**

In this thesis we have presented a peer-to-peer file distribution framework. We have evaluated our solution and demonstrated that it scales well. If we look back at the introduction and check our results against the initial requirements, we can conclude that we have managed to implement the most important features, which give our solution its performance and efficiency. Every multicast is about organizing the transfer. In the chapter 2 we have shown some of the possible approaches. The most useful ones were based on optimizing the transfer paths and the second group on used ad-hoc real time organization. In our solution we implement the best functionality from the peer-to-peer based methods, but we have also added a network aware overlay, which is de-facto defining the data distribution paths. The best distribution tree optimizing solution (Balanced Multicast from [5]) uses multiple paths concurrently. If we look from this point of view on our solution, we can observe that our overlay also defines multiple concurrent distribution paths: with each hop, a piece reaches another peer group where it has as many possibilities to continue as there are members. Because we also have at least two connections for each link, the possibilities how to get from initial seeder to the final peer are almost exponentially many, all concurrently possible. This gives robustness: there always is a delivery path and resilience – there always is a good path.

The major new addition of our approach is the organization of the overlay. In optimizing methods from chapter 2, the overlay doesn't contain network members, only nodes. Therefore it is difficult to consider the underlying network topology. This is important – the data actually get transferred over this real network, regardless of how our "virtual" topology looks like. If we want to do any real-world optimization, we have to work with the overlay that projects the real (physical) situation. Problem here is that the network members, like routers or switches, are actually not peers. In our solution we abstract these members by so-called representants. Representants are few peers selected from the underling hierarchy that "represent" the given location. They represent it, because together they have capacity to utilize all the available bandwidth of that location. Therefore for the given location we don't need to take all the present peers into consideration, only the representants. On the other side, we need to realize that these representants are in reality physically located in the leaves of the network graph. Thus even if they represent some higher in the topology located router, all the data they transfer, will be transferred physically directly to them, utilizing all the links on the way. Therefore we have implemented advanced bandwidth management algorithms which take this into consideration. This way we can ensure that we will never overload any of the links. This was critical for our solution, as the non-intrusiveness was priority #1.

The solution we present in this thesis has good real-world applicability. It supports multiple concurrent transfers and is quite well adaptable/usable in different environments, such as clouds. It can handle underlying topology changes in the real-time, thus is e.g. VM-migration friendly. It can be also easily integrated into bigger solutions, which only use it as a file transfer framework. One example of such is deployment of the new servers. Both rapid software/data and server deployment seem to be common problems nowadays. This can be presented by interest into solutions similar to ours by companies having huge numbers of servers (e.g. IBM). Twitter has presented Murder [28] when we were in the

later stage of this thesis. It is also a peer-to-peer deployment solution, except it is using chain topology, which simply wouldn't work in our environment (if one peer is slow/breaks, all the others in the chain are affected).

There have been a lot of papers about Bit Torrent. Many of them try to explain and model the changes in peer behavior and transfer dynamics introduced by modification of some of the default algorithms and settings. There are also many enhancements for BT, a lot of them focus on picking the best peers. Solutions which have some form of network topology awareness usually have better performance and use less traffic. Except [5], there's not much research done on piece selection strategies. We believe that the two new features our solution brings – bandwidth management and mainly network topology aware overlay deserve a bit of further investigation for possible improvements and better applicability in the real world (e.g. automatic location discovery). The `START` command looks like a good addition to the local peer discovering solutions, where it could further improve local bandwidth utilization and overall efficiency.

# References


[1] '4 Seconds' as the New Threshold of Acceptability for Retail Web Page Response Times, 2006 Akamai and JupiterResearch, www.akamai.com/4seconds

[2] Secure copy, Timo Rinne <tri@iki.fi>, Tatu Ylonen <ylo@cs.hut.fi>, BSD September 25, 1999 BSD

[3] NetCat (nc), *Hobbit* <hobbit@avian.org>, IPv6 support by Eric Jackson <ericj@monkey.org>, BSD June 25, 2001 BSD

[4] Rsync, http://rsync.samba.org/, GNU GPL license, maintainer Wayne Davison

[5] High-throughput Multicast Communication for Grid Applications, Mathijs den Burger, 2009

[6] J. Jannotti, D. K. Gifford, K. L. Johnson,M. F. Kaashoek, and J. JamesW. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation (OSDI'00)*, pages 14–14, San Diego, CA, USA, October 23-25, 2000.

[7] R. Cohen and G. Kaempfer. A Unicast-based Approach for Streaming Multicast. In *20th Annual Joint conference of the IEEE Computer and Communications Societies (IEEE INFOCOM 2001)*, pages 440–448, Anchorage, AK, USA, April 22-26, 2001.

[8] M. S. Kim, S. S. Lam, and D. Y. Lee. Optimal Distribution Tree for Internet StreamingMedia. In *23rd International conference on Distributed Computing Systems (ICDCS '03)*, Providence, RI, USA, May 19-22, 2003.

[9] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang. Mag-PIe: MPI's Collective Communication Operations for Clustered Wide Area Systems. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 131–140, May 4-6, 1999.

[10] N. T. Karonis, B. R. de Supinski, I. Foster, W. Gropp, E. Lusk, and J. Bresnahan. Exploiting Hierarchy in Parallel Computer Networks to Optimize Collective Operation Performance. In *14th International Parallel and Distributed Processing Symposium (IPDPS'00)*, pages 377–384, Cancun, Mexico, May 1-5, 2000.

[11] V. Jarník: *O jistém problému minimálním* [About a certain minimal problem], Práce Moravské Přírodovědecké Společnosti, 6, 1930, pp. 57–63

[12] R. Izmailov, S. Ganguly, and N. Tu. Fast Parallel File Replication in Data Grid. In *Future of Grid Data Environments workshop (GGF-10)*, Berlin, Germany, March 9, 2004.

[13] A. Cayley. A Theorem on Trees. *Quarterly Journal of Pure and Applied Mathematics*, 13:376–378, 1889.

[14] Bram Cohen, Bit Torrent, what is bt: http://www.bittorrent.org/introduction.html, protocol specification: http://www.bittorrent.org/beps/bep_0003.html

[15] M. Fras, S. Klampfer, Ž. Čučej, Impact of P2P traffic to the IP communication network performances, Faculty of Electrical engineering and computer science, University of Maribor, Slovenia

[16] The IBIS project - an efficient Java-based platform for distributed computing, http://www.cs.vu.nl/ibis/

[17] Do incentives build robustness in BitTorrent?, Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, Arun Venkataramani

[18] R. Bindal, P. Cao, W. Chan, J. Medval, G. Suwala, T. Bates, and A. Zhang. Improving Traffic Locality in BitTorrent via Biased Neighbor Selection. In *The 26th International conference on Distributed Computing Systems (ICDCS 2006)*, Lisboa, Portugal, July 4-7, 2006.

[19] P4P: Explicit Communications for Cooperative Control Between P2P and Network Providers**,** Arvind Krishnamurthy, University of Washington; Haiyong Xie, Avi Silberschatz, Y. Richard Yang, Yale University

[20] D. R. Choffnes and F. E. Bustamante. Taming the Torrent. In ACM SIGCOMM'08, Seattle, WA, USA, August 2008.

[21] Marco Slot, Guillaume Pierre, Vivek Rai, VU University Amsterdam; Paolo Costa, Microsoft Research Cambridge, Zero-Day Reconciliation of BitTorrent Users With Their ISPs

[22] Stijn Nijhuis, Faculty of Sciences,Vrije Universiteit: Network-aware BitTorrent

[23] Shansi Ren, Microsoft corporation; Songqing Chen, George Mason University; Lei Guo, Yahoo Inc.; Enhua Tan, Tian Luo, Xiaodong Zhang, The Ohio State University, TopBT: A Topology-Aware and Infrastructure-Independent BitTorrent Client

[24] BitTorrent has new plan to shape up P2P behavior,

http://arstechnica.com/old/content/2008/12/bittorrent-has-new-plan-to-shape-up-p2p-behavior.ars

[25] redis, an advanced key-value store, a data structure server, redis.io

[26] libtorrent by Rasterbar software, http://www.rasterbar.com/products/libtorrent/projects.html

[27] the libTorrent and rTorrent Project, http://libtorrent.rakshasa.no/

[28] Murder: Fast datacenter code deploys using BitTorrent, http://engineering.twitter.com/2010/07/murder-fast-datacenter-code-deploys.html

# Appendices

## A **Source code online**

All the source code, changes and enhancements we did in scope of this thesis can be found online at the address https://github.com/mtekel :

- *libTorrent-work-stealing*
  libTorrent 0.12.6 with our enhancements and modifications

- *rTorent-work-stealing*
  rTorrent 0.8.6 – low latency client with work stealing, work division and added xml-rpc commands

- *dTrack*
  A distributed tracker we have created for this thesis

- *redis-cli 2.0.4*
  Bare command line output addition/modification to the redis command line client, version 2.0.4. This client can be used for various on-the-fly modifications of the overlay (for testing), but mainly for viewing monitoring and statistics data.

- *This thesis*
  Text of this thesis is available in the PDF document format.

## B **List of figures**