Date of acceptance Grade

Instructor

Large-scale Experiments on Cluster

Liang Wang

Helsinki October 27, 2010

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Liang Wang

Työn nimi — Arbetets titel — Title

Large-scale Experiments on Cluster

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	October 27, 2010	75 pages + 0 appendices

Tiivistelmä — Referat — Abstract

Evaluation of large-scale network systems and applications is usually done in one of three ways: simulations, real deployment on Internet, or on an emulated network testbed such as a cluster. Simulations can study very large systems but often abstract out many practical details, whereas real world tests are often quite small, on the order of a few hundred nodes at most, but have very realistic conditions. Clusters and other dedicated testbeds offer a middle ground between the two: large systems with real application code. They also typically allow configuring the testbed to enable repeatable experiments. In this paper we explore how to run large BitTorrent experiments in a cluster setup. We have chosen BitTorrent because the source code is available and it has been a popular target for research.

In this thesis, we first give a detailed anatomy on BiTorrent system, such as its basic components, logical architecture, key data structures, internal mechanisms and implementations. We illustrate how this system works by splitting the whole distribution process into small scenarios. Then we performed a series of experiments on our cluster with different combination of parameters in order to gain a better understanding of the system performance. We made our initial try in discussing "How to design a rational experiment" formally. This issue did not receive as much attention as it should in the previous research work.

Our contribution is two-fold. First, we show how to tweak and configure the BitTorrent client to allow for a maximum number of clients to be run on a single machine, without running into any physical limits of the machine. Second, our results show that the behavior of BitTorrent can be very sensitive to the configuration and we re-visit some existing BitTorrent research and consider the implications of our findings on previously published results. As we show in this paper, BitTorrent can change its behavior in subtle ways which are sometimes ignored in published works.

ACM Computing Classification System (CCS):
A.1 [Introductory and Survey],
I.7.m [Document and text processing]

Avainsanat — Nyckelord — Keywords
Peer-to-peer, overlay network, high performance cluster, large-scale experiment, BitTorrent

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Internet applications and services have greatly changed our life style. Most of these popular applications can be viewed as large-scale distributed systems. They have tens of thousands of simultaneous on-line users, and are deployed on huge amount of machines, which are usually geographically distributed. Peer-to-peer(P2P) overlay network is a typical example.

The emergence of P2P network is mainly because of the impasse that traditional Client/Server architecture has to confront due to the fast grow of internet. Especially when content distribution is concerned, the centralized content distribution paradigm imposes great burden on the server. It is very common that a server is quickly overloaded when the arriving requests are beyond its processing capacity. The traditional solution for centralized architecture is to upgrade physical resources to increase the system capacity, which is very expensive and impossible for many individual content distributors.

P2P systems address this issue well. The system's workload is amortized to all participants. The system capacity increases as the number of peers increase, so P2P system is very scalable. From the emergence of Napster till the newest DHT, we have experienced four generations of P2P systems. Their structures vary from centralized to purely distributed architectures; from unstructured to structured overlays [SW05]. P2P systems have become a part of everyday life on the internet.

The great success of P2P paradigm not only give rise to a bunch of different P2P systems in the industry, but also arise the academy's interest. As one of the most successful P2P systems, BitTorrent receives much attention these years. Different analytical models were built and various experiments were performed to measure BitTorrent's performance or validate the models. However, the issue is most experiments are performed under the inexplicit assumption that they are rationally designed. Different experiment settings are used in the papers by different researchers. For example, the upload rate varies from several kilobytes per second to megabytes per second; and the distribution file size from dozens of megabytes to several gigabytes. What's more, someone uses one peer per node configuration, and some deploy multiple peers on one node.

Arbitrarily choosing experiment settings not only makes it difficult to compare the experiment (data) in different papers, but also may endanger the accuracy and the rationality of experiments. In fact, many researchers have already been aware of

this issue, thus carefully and conservatively choose low transfer rates and large file in their experiments, in order to guarantee the experiments be performed within the system capacity and achieve better accuracy.

However, the rationality of an experiment design still hasn't received as much attention as it should. Even though everyone knows BitTorrent's performance and behaviors are heavily influenced by the experiment settings(parameters), few serious discussions are made on these issues, and there is no clear boundary about the "safe region" for an experiment design.

In this thesis, we solved the following practical issues and answered some related questions:

1. We first demonstrate how to tweak BitTorrent in order to run multiple peers on one physical node. We show how system performance is impacted by several BitTorrent's key parameters, and how to tune them to gain the best system performance. The work in this part is trying to push the experiments to large-scale even with limited physical resources.

2. By reading through the code carefully, we give a detailed anatomy on BitTorrent system, such as the basic components, logical architecture, key data structures, internal mechanisms and implementations. The documents in section 3 can be used as references in future research.

3. We show how to do the capacity planning in one-node case with naive method, and also claim naive method will not work if more nodes are used for deploying peers. We proposed a more elaborated but simple analytical method to estimate the system capacity, and performed various experiments with different settings to validate our method.

4. By running the experiments around the system capacity limit, we show how the two core mechanisms(*peer selection strategy* and *piece selection strategy*) affect BitTorrent's behaviors. We not only observed BitTorrent's clustering property from its *peer selection strategy*, but also observed another kind of clustering from its *piece selection strategy*, which has not been discussed in the previous papers yet.

   We claim BitTorrent's behavior is the results from the combined-effects from both *peer selection strategy* and *piece selection strategy*. Limiting the upload bandwidth weakens influence from *piece selection strategy* on BitTorrent's behaviors, and makes it difficult to be observed in the experiments.

The thesis is organized in the following sections: First, we start our discussion from the necessity of large-scale experiments, and the reason why we choose cluster as our experiment platform. In section 2, we revisit some previous work done by other researchers and enumerate some relevant papers. Then we introduce the background knowledge about BitTorrent in great details in section 3.

In section 4, we explain the terms, the general methodology and specific methods used in this thesis. In section 5, we introduce our experiment environment, discuss some practical issues we have to face, and their corresponding solutions, and how various parameters impact BitTorrent's performance on cluster. Then we show how to do capacity planning for the experiments on cluster in section 6. We performed various experiments in section 7, and show two different clustering properties. What's more, we also give detailed behavior analysis in section 7 to explain why these two clustering properties happen. Finally, we conclude our paper in section 8.

1.1 Why we need large-scale experiment?

The most important reason is that an application may exhibit quite different characteristics in a large-scale experiment. Some application's behaviors may become even unpredictable when experiment scale is large enough. So, every P2P system should undertake thorough and intensive tests before we put them into real use.

Secondly, large-scale experiments can also generate abundant statistical information. By studying these data, we can analyze the system's behaviors, evaluate its performance, test the quality of a service. Then we are able to locate the bottlenecks of performance and further improve the system.

1.2 Why on clusters?

As far as large-scale experiments are concerned, among the options are simulation, emulation, clusters and real internet. Each of them has its own features.

1. Simulation: Simulations are a common way to move from the analytical models towards a more realistic setting. However, they are limited by the accuracy of the simulator, but generally allow to evaluate systems of thousands of nodes, possibly even up to a few million. Simulators typically have to abstract out many details about the actual network between nodes, but are able to capture many details of application aspects.

2. Internet: Real internet offers of course the most realistic setting, by requiring real, running code and using a real network. However, real world tests are often very limited in scale, with system sizes of a few tens of nodes being common and maximum sizes of a few hundred being more or less the upper practical limit (e.g., on testbeds like PlanetLab).

3. Cluster: Running the experiment on a cluster attempts to strike the middle ground between these two. It requires writing the real programs, so that all aspects of the application are included. Also, a cluster environment allows for a fine-grained setting of the network parameters, so as to model a real network between the nodes. Furthermore, the experiments are typically repeatable, allowing for more control in setting the experiments.

Cluster provides us a closed and exclusive environment for the experiments, so the irrelevant interferences can be minimized to the lowest level. There are some arguments that the experiments on the cluster do not take the following things into account, such as heterogeneity in the platforms, bandwidth, RTT, packet loss rate and so on. However, just because of this reasonable simplification in the experiment environment, it is much easier for us to figure out the true causes for a problem. Furthermore, such heterogeneous factors can be added to the experiments manually to make it closer to the real-world environment.

Another advantage that we can benefit from a cluster is the data collection. For example, in the study of BitTorrrent, it is impossible to collect all the data from every peer in the real-world swarm. Without the complete information, the thorough study on the system is difficult to perform. One solution is resorting to the tracker's log file, as the method used in [IUKB+04]. But this information is too coarse-grained to do the detailed study on peer level. Another popular solution is using instrumented client. Many researchers inject instrumented BitTorrent client into the real-world swarms, then collect data from itself and its buddies. However the data collected is only the partial information of the whole swarm. The research in [RR07] shows this approach cannot provide a representative view of BitTorrent's behaviors. Since the configurations of the instrumented client has already determined what kinds of data it will collect.

1.3 Popular testbeds

Considering the difficulties in testing and evaluating distributed systems and internet applications, various testbeds are built to provide researchers with an ideal experiment environment. In this section, we will introduce three widely-used testbeds in research area briefly.

1.3.1 PlanetLab

PlanetLab[Pla10] is an overlay testbed initiated by Prof. Larry L. Peterson in Princeton University in 2002[1]. It aims at building a global research network for computer networking and distributed systems. Currently(Sept. 2010), PlanetLab consists of 1130 nodes at 511 sites. The sites spread from North America, South America, Europe to Asia, and most of them are located in the U.S.A and Europe.

Any academic organizations or research institutes can join in PlanetLab by contributing some nodes to it, but the hardware of contributed nodes must comply with permitted configurations in order to minimize heterogeneity headaches. An individual cannot take part in under personal names. Each site should have a *PI*(Principal Investigator) responsible for the credentials for each account. And each account will be given some simple quotas on storage and CPU usage. People can build their own "private PlanetLab" with the standalone package – *MyPLC*, which is offered at [Pla10].

With PlanetLab, researchers have access to large set of geographically distributed nodes. The traffic will go through the realistic network substrate and experience congestion, packet loss and various realistic network conditions. PlantLab is not only an overlay testbed, but also a deployment platform supporting seamless migration of an application from early prototype[PACR03]. The direct consequence of PlanetLab's dual use paradigm is the nodes have to be shared by different experimental services. As L. Peterson et al. indicated in [PACR03], the dual use paradigm leads to an obvious tension between the needs of "test & measure" researchers for reproducible results, and those interested in the system as a deployment platform. That is the key reason why we did not choose PlanetLab as our experiment platform.

In PlanetLab, multiple *VMs*(Virtual Machines) run on a node. A *service* is set of distributed and cooperating programs running within multiple *VMs*. A user has access to his corresponding *VMs*, but without root privilege. The resources are

[1] also with other researchers from Princeton, UC Berkeley, MIT etc

shared in terms of *slices*, which is a horizontal cut of global PlanetLab resources allocated to a given *service*. Each *service* corresponds to a *slice*. In a nutshell, a *slice* is a collection of VMs. From node perspective, the resources allocation is realized in terms of *ticket*, which is issued by a node and specifies the resource amounts allocated to a *service*.

The detailed specifications about PlanetLab can be found in various PDNs(PlanetLab Design Notes) on [Pla10].

1.3.2 Emulab

Emulab[Emu10] is a widely-used network testbed for the researchers in the fields of networking and distributed systems. It is developed at the University of Utah, aims at making the networking experiments easier to design and perform. There are a bunch of Emulab in operation today, each of them has its own objectives and limitations. So an experimenter should carefully choose those matching his goals as experiment platform. Considering the hardware price keeps dropping and Emulab software keeps improving, it is feasible to build one's own private Emulab as W. Laverell et al. suggested in [LFG08].

Emulab provides a powerful GUI tool to create network topology for experiments. Then Emulab server will create the corresponding VLANs and emulate the link properties by adding delays, packet loss and limiting the bandwidth artificially according to the experiment configurations. In order to control various system parameters, the user is usually given root privilege in the system. Based on previous experience, the root access may also cause some problems if a user does not have enough administration skills, he may mess up an experiment completely. The lucky thing is every user's behaviors are isolated and can not interfere other's experiments.

Another useful feature in Emulab is *virtual node*, which is just lightweight virtual machines running on top of a host system. *Virtual node* is based on either FreeBSD's jail mechanism, or OpenVZ container-based virtualization on Linux. Since the physical node can be multiplexed by running multiple *virtual nodes* on it. The direct benefit is we can easily enlarge experiment scale without adding new physical resources. But the capacity planning should be taken into account when designing the experiments.

1.3.3 Grid5000

Grid5000[gri10] is a high performance computer cluster, which is designed to support experiment-driven research in large-scale parallel and distributed systems. The infrastructure is provided by INRIA, and geographically distributed in 9 sites in France. The sites are interconnected with the network provided by RENATER[2]. The nodes within a sites are interconnected with high performance network. The detailed specification about the node and the network can be found on [gri10].

As we have mentioned in 1.3.1, PlanetLab is not an ideal platform for "test & measure" researchers, since the competition for the physical resources may cause unpredictable turbulence on system performance and further lead to inaccurate experiment results. What's more, measurement experiments are usually performed with different combinations of various parameters, which are usually not under the control of an experimenter in PlanetLab. Compared with PlanetLab, Grid5000 provides a dedicated and exclusive environment which enables us to control various key system parameters.

Thus the experiments on Grid5000 are reproducible, which makes Grid5000 an ideal testbed for measuring and evaluating large-scale distributed systems. However, not like PlanetLab and Emulab, which provide some kinds of all-in-one solutions for a complete experiment environment; in Grid5000, an experimenter has to choose his own suitable tools to help him manage the whole experiments. Luckily, most of such tools can be found on the official website[gri10].

1.4 Further discussions

As far as testing and evaluating large-scale distributed system is concerned, besides the options mentioned above, Choffnes et al. proposed another option in [CB10] - Edge-measurement[3]. Even though there are still arguments on these options, we can foresee an end for such arguments in the near future. Since the boundaries among simulation-, cluster-, testbed- and edge-measurement are becoming clearer and clearer.

Simulation $\longrightarrow$ Cluster $\longrightarrow$ Testbed $\longrightarrow$ Edge Measurement

As we can see from above, from left to right, the coverage becomes larger and larger,

[2]French National Telecommunication Network for Technology, Education and Research.

[3]Section 2 has more discussions on Edge-measurement.

the environment becomes more and more realistic, and the data becomes more and more representative accordingly. Which is the best option for distributed system evaluation is always researcher's concern.

The key point is there is no such "BEST" option. As we have known, a sound and full-featured system can only be achieved under many iterations. Our opinion is different option plays a different role in the whole process of developing, deploying and evaluating a distributed system.

How to make choice depends on "how are we going to use the measurement data?", in other words, "what purpose are the measurements for?" Simulation is suitable for modelling the system in design phase before development, or before real-world deployment. Cluster is suitable for figuring out the bottleneck and measuring certain mechanisms in improving phase before real-world deployment. Edge-measurement provides us representative view of our internet, thus can be used to tune the system/model parameters.

We also think simulation should be used to provide the theoretical performance for a system. The study of peer-level behavior should be done on cluster. The study of user-level behavior should be done in edge-measurement. Finally, the whole system should be shaped into real-world environment (by tuning parameters or policies) based on edge-measurement.

Our current bewilderment revolves around testbeds - "Is testbed still a useful option?". Take PlanetLab as an example, L. Peterson mentioned in [PACR03] that testbed is not suitable for "test & measure" evaluation because of the dual-use paradigm. Physical resources are shared, thus the experiments are not reproducible, the data measured is not accurate. So people supposed testbeds can provide us a representative view of internet, such that we can modify our system/model based on the real-world parameters. However, [CB10] claims that testbeds can neither provide the representative view. All the implications from the recent work lead us to a very practical question - what else can testbeds be used for?

PlanetLab just builds us an environment more realistic than cluster, but not as good as edge-measurement. If edge-measurement finally becomes feasible, does it imply the PlanetLab will become useless in evaluating distributed system? Or will it evolve to a pure deployment platform, or an infrastructure for public services in the end.

All these guesses will be testified in the future.

2 Related Work

BitTorrent has been a popular target for research over the past several years, almost every aspect of BitTorrent is well discussed and lots of work are done. Most of the research work can be divided into two categories: model-based and experiment-based. However, there is no strict boundary, and some work are a mixture of both. Section 2.1 gives a summary on analytical model-based research by introducing two widely-used analytical models of BitTorrent. Section 2.2 discusses the related work in experiment-based research, which our work mainly revolves around.

2.1 Research based on analytical models

Basically, there are two kinds of analytical models for analyzing P2P systems. One is Fluid-based model, and the other is Chunk-based model. The most essential difference is how they treat the data, whether the data is infinitely divisible or not.

In fluid-model, since data can be divided into arbitrarily small pieces, a peer can distribute the data to others as soon as it receives the first bit of the data. There is no delay between these two operations. However, in chunk-model, a file consists of many chunks with certain length. A peer can distribute a chunk to other peers if and only if it receives the complete chunk. The delay between the two operations are taken into account in the model. An informal but more vivid analogy is, in fluid model, the process of content distribution is just like water flowing through the pipes; while in chunk-model, it is like carrying bricks to different peers.

Each of the models has its own pros and cons. Generally speaking, chunk-model is more realistic, and can be approximated by fluid-model very well in some occasions. However, fluid-model is more widely used in research area, since the formulas derived from the fluid-model are much simpler than those from chunk-model. So fluid-model is a very good way to describe the overall characteristics of a system in most occasions, and chunk-model can be used as reference model to provide upper and lower bound of the system.

2.1.1 Chunk-Based Model

Biersack et al. studied three topologies in chunk-based model in [BRF04], which are Linear topology, $Tree^k$ topology and $PTree^k$ topology in table 1. (k is the outdegree of the node)

Table 1: Three topologies in chunk-based model

Topology	Indegree	Outdegree
$Linear$	1	1
$Tree^k$	1	k
$PTree^k$	k	k

Besides deriving the closed-form formula to calculate the download completion time, they also made several important conclusions. Let N be the number of peers and C be the number of chunks. In chunk-based model, peer to chunk ratio N/C plays an important role. If $N/C > 10^{-1}$, the $PTree^k$ outperforms $Linear$ significantly. If the transmission time for one chunk is negligible compared to the transmission time of the whole file, the benefits of $PTree^k$ diminishes.

If we don't consider the overheads in communication introduced by each chunk during the transmission, these conclusions imply the file should be divided into many small chunks to keep the N/C value at a low level. And the system performance improves exponentially as the C increases.

In real internet, since the nodes keep joining and leaving the system, the topology cannot remain stable. When the *peer selection* and *piece selection* strategies are adopted in the system to live with the changing topology, the $PTree^k$ evolves into mesh-based topology.

2.1.2 Fluid-Based Model

Qiu and Srikant proposed their fluid-based model in [QS04], and derived very neat formulas to describe the system performance. Their model exposes the characteristics of P2P system well, the useful conclusions are: 1) the system scales well in terms of the number of peers in the system; the average download time is independent from the peer's arrival process. 2) The data distribution in BitTorrent is very effective and efficient in terms of the probability of locating the fresh data. There are also some other works [KR06, MPES09] based on their fluid model.

The basic starting point of their model is data is infinitely divisible, a peer can forward the data to its buddies immediately whenever it receives the data. There is no delay between receiving and forwarding. Every time one bit is injected into the network, all the nodes can obtain it at once(if physical bandwidth permits). Apparently, this starting point is not realistic. However, it can be a very good

approximation for chunk-model if the the number of chunks is larger enough than the swarm size. Another criterion is the file is large enough and the piece is small enough, which means the time for transferring one chunck is negligible compared with the total transmission time, then the fluid-model can be applied. They also made some unrealistic assumptions to simplify the the real-world complexity, such as symmetric homogeneous bandwidth, Poisson arrival pattern(which we will discuss in section 3.2), etc.

Another implication is that linear topology is meaningless in fluid-based model. Since whenever the head node receives a bit, the tail node obtains it immediately. Whenever the distributor finish uploading the complete file, all the nodes finish downloading at the same time. So the queue can grow arbitrarily long without degrading the performance.

2.1.3 Effectiveness of Connection - η

In fluid-model, there is a very important parameter η, which represents the effectiveness of the connections. In other words, it represents the probability that a peer can get fresh data from its buddies. High η means a healthy and effective system. So it should be thought as an important indicator for system performance. η is first introduced by Veciana and Yang et al. in [VY03], [QS04] gives the explicit formula to calculate its value.

To some degree, η is a bridge between the traditional C/S architecture and P2P architecture. In C/S architecture, η is zero, since there is no data exchange among the peers(clients), and the seed(server) has to upload data to all its clients. As η increases, the system transforms from C/S architecture to P2P architecture. The more η increases, the less workload needed from the seed.

Theoretically, if a peer has global information of a swarm, η is able to reach 1. Namely, a peer must be able to connect all the others to construct a complete graph. However, it is only possible for very small swarms in real world. For a swarm consisting of tens of thousands of peers, there will be unacceptable overheads in maintaining these connections. Some research investigated DHT module in BitTorrent, which is adopted to support trackless work mode, and point out that DHT's maintenance is responsible for 80% of communication overheads. It shows the expensive cost in maintaining excessive connections in a large distributed system.

Furthermore, other factors may also affect η. For example, a seed should avoid

uploading duplicated pieces to leechers before a complete file is injected into a swarm. The duplicated pieces will increase the probability that two connected peers have the same data, thus further reduce the uplink utilization and increase the seed's burden. In [BHP06], Bharambe et al. proposed a modified piece selection strategy for seeds, with which a seed only upload distinct pieces before a complete file is injected. The number of buddies also affects η. The more buddies a peer has, the more likely it can find the data it wants.

η is dynamic and keeps changing during the lifespan of a swarm. As more and more leechers become seeds, the other leechers will have higher η. Even for a single peer, η will not remain static. For example, as more and more pieces a peer gets, it will be more difficult for it to locate fresh data. Actually, the "last block" problem is the result of small η at the end of downloading. That's why *End Game strategy* is introduced.

2.2 Research based on experiments

In some previous work, e.g., [MPES09, KR06, SHRY07, LLKZ07, LUKM05], researchers use a real BitTorrent client in experiments to validate their models and conclusions. However, less papers have concerned themselves with the accuracy of their experiments and possible bias in their methodologies.

Legout et al. [LUKM05, LUKM06] conducted a thorough measurement-based research on the two core mechanisms of BitTorrent, piece and peer selection. However, the influences from these two mechanisms are discussed separately. The authors showed that the rarest first algorithm guarantees a close to ideal entropy, while the choke algorithm guarantees the fairness in the system. None of the results presented in the papers investigate the combined effects of both mechanisms, which as we have shown, also occurs and can have significant effects on BitTorrent's behavior.

Antoniu et al. [ABJM04] discuss the difficulties in validating large-scale peer-to-peer systems. The authors also proposed a framework for performing large-scale experiments based on grid services. However, how the experiments are affected by the underlying details and the experiment settings are not touched.

Only a few papers, e.g., [RR07, ZIea10, RLD10] concern the accuracy of experiments and the bias of measurements. Work in [ZIea10] investigated the sampling bias in BitTorrent experiments. Even though the discussion merely focuses on the approach of using instrumented client to obtain data from real-world swarm, the

recommendations proposed in this paper are simple heuristics and guidelines. We have followed their recommendations and have designed our Logger module to follow them. Our Logger module takes a snapshot for the peer every second during its whole life span. This strategy yields very reliable experiment data.

On the other hand, Rasti and Rejaie [RR07] claim that the data obtained with this approach (injecting an instrumented client into real-world swarm) is not representative and has already been biased in the beginning. The main reason for their claim is that BitTorrent clients tend to cluster with other clients having similar upload bandwidths. This observation is definitely valid for measuring a real-world swarm on the Internet, but as our experiments are performed on a cluster where all peers are instrumented to provide logging information, such a bias does not exist in our experimental setup.

A lot of analytical work has also studied the clustering properties of BitTorrent. Based on the analysis of the choking algorithm, [LLKZ07] provides empirical evidence of BitTorrent's clustering and show that peers with similar bandwidths tend to get clustered.

Meulpolder et al. [MPES09] extend an earlier analytical model from [QS04] and propose a new model for analytical investigation of BitTorrent's clustering. Their model only takes into account peer selection in BitTorrent and ignores the effects of piece selection. They observe similar clustering behavior as we have observed. However, their model and measurements exhibit a small discrepancy which they conjecture is the result of probabilistic effects from too small experiments. Our results show that clustering in BitTorrent is actually an interplay of both peer and piece selection algorithms, and we believe that their observed discrepancies are a result of their model ignoring piece selection. Although the effects of piece selection on clustering are small and hard to observe, our work, in particular on the download-constrained experiments, has shown that it cannot be ignored. Both [MPES09] and our work find the same effect of upload connections going to foreign peers while the majority of data comes from native peers.

The work by Rao et al. [RLD10] is the closest work to ours. The authors discuss the rationality of performing BitTorrent experiments on a cluster. However, the discussions focus on the marginal influences on the average download rate from various RTT and packet loss rates and conclude that the effects from changing RTTs and packet loss rates are so small that they can be discounted in the evaluation. Our work focuses on how to design an experiment on a cluster properly, i.e., what

is the 'safe region' for a correct experiment and how BitTorrent behaves when the experiments are performed around the system capacity limit.

The experiment setup in [RLD10] is very similar to the case discussed in our paper. The authors used 3 nodes for deploying leechers (100 leechers on each node) and performed a homogeneous upload-constrained experiment. The maximum upload rate was set to 100 KB/s. They did not consider possible bottlenecks in their experiment setup. Using our capacity planning method from Section 6.3, we can see that their experiments require only on the order of 3 MB/s of bandwidth between nodes and on the loopback. Given that they were using modern computers on the Grid 5000 testbed, they should be well below the system capacity limit. Our work therefore validates their experiment setting as being correct.

Choffnes and Bustamante et al. claimed in [CB10] that the data measured through testbed's vantages can not give us a representative view at internet scale because of the limited coverage. The authors argued from the following aspects:

1) A large part of links are invisible in testbed-measurement, especially at lower tier;

2) Latencies measured in edge-measurement are higher than testbed-measurement;

3) TIV(*Triangle-Inequality Violation*) is much higher in edge-measurement than in testbed-measurement;

4) Capacities are overestimated in testbed-measurement, which is much lower in edge-measurement.

Thus, the authors of [CB10] claim it is not proper to infer the system performance by using the data from testbed-measurement. What's more, they proposed an independent edge-measurement-based services, which can be integrated into the existing distributed system or providing certain incentives to encourage people installing it on their machines.

The idea of edge-measurement looks beautiful. Nonetheless, it also has to confront the same problem as testbeds do – the coverage(or scale). The larger scale, the better. But how to "control" so many nodes, in other words, how to collect data from the edge node, is the most difficult issue to handle. Our biggest concern is user's privacy. What's more, according to the information on [edg09], most ONO users are from Europe and the United States. But there are also great number of BitTorrent users in other countries not being taken into account, such as China. So the coverage is still limited, even it is much larger than PlanetLab.

The second issue is edge-measurement cannot be performed "arbitrarily". Com-

pared with the evaluation on cluster, edge-measurement-based evaluation is sort of "passive". Since we cannot design and perform arbitrary experiments to measure certain mechanisms of a system as we like. The evaluation of overall system in edge-measurement is based on mining the collected data.

3 BitTorrent Basics

3.1 A brief introduction

In this section, we will give a brief introduction on BitTorrent and its core mechanisms based on [Coh03]. The content here is closely related with the experiments and analysis in our thesis. The detailed anatomy on BitTorrent's internals is given in section 3.6. Since there are many BitTorrent implementations, the same terms sometimes may have different meanings. In order to avoid confusion, we restrict the discussions on the official implementation(*Mainline ver4/5*), and use terms by widely-accepted conventions in P2P research area.

BitTorrent is a popular P2P content distribution software, which can be categorized into the third generation P2P systems. The most significant feature of BitTorrent is its scalability. The content can be distributed efficiently among large amount of peers. There is a broad range of discussions about its scalability.

To join a swarm, a peer first need to obtain the corresponding meta file, which is usually referred as a torrent file. Then the peer will contact the tracker by extracting its url address from the torrent file. The tracker will make a peer-list by randomly selecting 40 peers in the swarm and return it to the requesting peer. With the peer-list, the peer can connect to those already in the swarm and join the distribution process. By default, a peer will keep connecting others until it has 40 buddies. After that, it will stop initiating new connections, however it can still accept connections from others. When a peer has 80 buddies, it stops accepting new ones, any more incoming connections will be dropped immediately. If the number of buddies dropped below a certain threshold, it will re-request a new peer-list from the tracker. So, during the life span of a peer, it usually maintains 40 to 80 buddies.

In BitTorrent, a distribution file is cut into *pieces*. The usual size of a piece can range from 256KB to 1MB[4], but it must be power of 2. Larger piece size can reduce

[4]In ver5, piece size is a function of file size, detailed discussion is in section 3.3.5

the torrent size. When exchanging data, a piece will be further divided into smaller units, which are referred as *slices* or *chunks*. In such a way, the uploads can be pipelined to improve the performance. So *slice* is the basic transmission unit.

As one of the core mechanisms, BitTorrent's *piece selection strategy* is widely known as *rarest-first*. More precisely, it should be called *local rarest-first* since the decision is made based on local information. By requesting those rare pieces, a peer can attract more buddies to download from it. As a result of tit-for-tat, it will be more likely to be served by others.

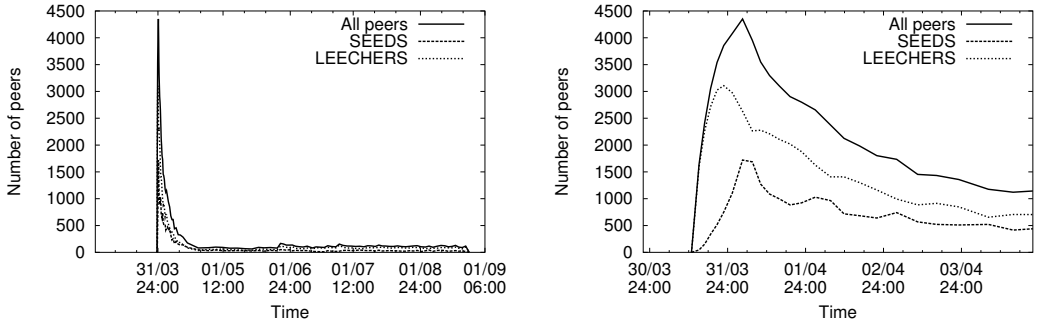
Another core mechanism is *peer selection strategy*. Leechers and seeds have different peer selection strategies. A leecher will upload to those who can provide it better download rate, while a seed will upload to those who can download from it fast. The leecher's strategy is rate-based tit-for-tat, the purpose is to guarantee the fairness in the system. And seed's strategy tries to make sure the new replicas can be generated fast. Every 30 seconds, a peer selects the buddies to upload to based on these strategies, the others will be choked.

3.2 Peer arrival process

In order to simulate a realistic environment, we try to make every aspect of our system close enough to the real-world swarm. So we studied the peer arrival pattern carefully.

In general queueing system, the clients' arrival process is usually modelled as Poisson process for analysis. If we consider the 'BIG' BitTorrent system consisting of thousands of swarms(torrents), Poisson process might be applied. But for a single swarm(torrent), the Poisson process is not a very suitable approximation. Because it assumes a constant arrival rate, which cannot reflect the initial flash-crowd, death of a swarm and other typical characteristics of BitTorrent system.[IUKB+04]

The Poisson process was first introduced to study BitTorrent system by Qiu et al. in [QS04], in which the authors assumed the system will reach its equilibrium and remain stable. However, in real world, the system seldom reaches stable state; only in the latter phase of a swarm's lifespan, when newcomers become fewer and fewer, the arrival rate will stay at a stable but low level. Usually, at this moment, a swarm is close to its death. Without any long-stay seeds, a swarm will die out quickly. And this is the common situation in real world. So the constant arrival rate assumed in Poisson process doesn't hold in real BitTorrent system.



(a) Complete trace

(b) Zoom on the first five days

Figure 1: Number of active peers over time [IUKB+04]

Actually, the figure 1, which is plotted based on statistics from real-world swarm, exposes actual peer's arrival process very well. In [GCX+07], Lei Guo, Songqing Chen and et al. proposed a formula to simulate this process, which is $\lambda(t) = \lambda_0 e^{-\frac{t}{\tau}}$. In their model, $\lambda(t)$ is peers' instant arrival rate at time t, λ_0 is initial arrival rate. Lei Guo et al. introduced an attenuation parameter of peer arrival rate τ. So the instant arrival rate will decrease exponentially as time goes by. To some extent, Poisson and other specific distributions are just faster and cheaper (but mathematically tractable) alternatives when lacking of real-world statistics.

In our experiment, we did not use any formulas to generate arrival process on the fly. On the contrary, we used pre-defined arrival process based on the data in [IUKB+04]. This strategy not only makes the system close enough to the real world, but also makes the experiment reproducible.

3.3 Basic Components

In BitTorrrent system, every peer is supposed to follow the standard protocol to guarantee the efficiency and fairness during the content distribution. To distribute a file, firstly, the distributor needs to make a torrent file for the file. Then he must make this torrent available to the people who are interested in the content. This can be done in many ways (e.g email, website, IM), usually the torrents are uploaded to a website.

In a typical BitTorrent system, there are four basic components – web server, tracker, seed and leecher. The image below shows the basic components and their relation-

ship. **Ryan: miss a figure here!**

3.3.1 Web Server

Web server is a machine where people can search and download the torrents that they are interested in. But it is not indispensable. Since BitTorrent is the third generation of P2P system. In its design principle, it does not rely on any central server responsible for indexing the files. The central indexing server used to be the strategy adopted in Napster, and was also the key factor finally brought down Napster. However, in BitTorrent system, in order to download a file, people only need to obtain the torrent file, no matter with what means. For example, the torrent can be obtained from search engine, from forums, from IM software etc. A web server just provides people some convenience in locating interested file.

3.3.2 Tracker

Tracker is an obligatory component. It provides the service to other peers such that they can get involved into the distribution process. The tracker's address(URL or IP) and service port number are embedded into a torrent file. So the leechers can contact the tracker by extracting these information from the torrent. All the peers register themselves to the tracker and get a peer-list from it, then they can communicate with each other. Multiple trackers may be embedded into a single torrent file to increase the robustness of the system and prevent single point failure.

What's more, tracker is also used to collect statistical information for the torrents it hosts. Those statistical data are very valuable, since it is not easy to collect such data in large-scale distributed systems in the real world. The tracker's special role in the swarm makes it possible to perform this task. Many research work are based on these data, such as the peers' behavior, swarm's evolution and lifespan, peers' arrival pattern and so on.

3.3.3 Seed

A seed is the peer who holds the complete file in a swarm. A seed uploads the file to other leechers but download nothing. It acts like a server and is responsible for distributing the content. The number of seeds is a very important indicator for the availability. Usually, there are only very limited number of seeds(maybe the only

Table 2: Native Data Types Used In Bencode

Data Type	Representation	Example
integer	i<decimal number>e	i3e; i17e; i-51e
string	<length>:<content>	8:Helsinki
list	l<content>e	li100e8:studentsee
dictionary	d<content>e	d6:animal5:fruit5:tiger6:orangee

initial one) in the beginning of distribution. As more and more leechers complete their downloads, they will become the seeds and serve the others.

3.3.4 Leecher

A leecher is a peer who does not hold the complete file. It downloads from other peers and uploads at the same time. People call such leecher a free-rider, if a leecher only downloads but uploads nothing, or uploads very limited data compared with its downloads. Generally, free-riders have impacts on the topology of overlay network, and are considered harmful to the system since it bring down the overall performance. However, Kangasharju, J. pointed out in [Kan09] that freeriding should not be always considered as harmful to system performance.

3.3.5 Torrent File

Torrent file is the glue that connects every component, make it possible to form a swarm. It contains the metadata for a distribution task. For example, the file list, the size of files and information about the tracker. These information is formatted with *Bencode*, which is dedicated encoding schema used in BitTorrent.

Bencode is a straightforward schema. It only supports limited native data types, including integer, string, list and dictionary in the Table 2. However, like other data languages such as JASON, Bencode is flexible and capable of storing complex yet loosely structured data. For the complex data objects, they can be first serialized, then be Bencoded and stored in the torrent file. (e.g. The pickle module in Python provides very good supports for object serialization)

The original specification of Bencode does not deal with other code sets except ASCII, so different implementations use their own way. The content in list and dictionary can be any Bencodeable data types. But the keys in a dictionary must

be organized in lexicographical order.

Since BitTorrent is expected to run on multiple platforms, the first consideration is platform-independence but not efficiency. That's why Bencode does not adopt pure binary encoding. Even though Bencode uses ASCII characters to represent basic data types, it is not thought as human readable, because the encoded data usually contains binary code, such as complex data object.

Another important information stored in the torrent file is the digest for each piece in a file. In official implementation, Ver4 uses 256KB as default piece size. However, Ver5 decides the piece size according to the content size. Ver5 guarantees the content will be cut into no more than 2^{12} pieces. So, for very large files, the piece size can reach 2MB or even more. As a result, the torrent file generated by Ver5 is usually several dozens of kilobytes. For example, for a 5000MB file, Ver5 generates a 50KB torrent file, while Ver4 generates a 391KB torrent file. Ver5 cuts it into 2500 2MB-pieces, Ver4 cuts it into 20000 256KB-pieces.

The change in Mainline Ver5 reflects the truth that more and more large files are being distributed on the internet. And the bandwidth keeps improving. In order to easily distribute torrents, Ver5 guarantees the torrent size won't become too large even for large files.

Then each piece will be hashed (SHA-1) and the digest will be stored in the torrent file. This processing mechanism also applies to the torrent including multiple files(usually called a *batch torrent*). Because BitTorrent treats multiple files as a single block of data, there is no difference between a single file and multiple files. The only thing needs to be pointed out is that piece boundary may overlap the file boundary.

3.4 Tracker Protocol

Within BitTorrent system, as far as communications are concerned, the protocol can be subdivided into peer protocol and tracker protocol. Tracker protocol defines how the peers communicate with the tracker; while peer protocol defines how the peers communicate with each other in the system.

The tracker protocol is built upon HTTP/HTTPS. Usually, we refer the communication from a peer to a tracker as *requests*; and the communication in reverse direction as *responses*. All the requests must be translated into HTTP GET method with URL encoding, then submitted to a tracker. Multiple request parameters can be embed-

Table 3: Request Parameters (*Peer*$\longrightarrow$*Tracker*)

Type Name	Usage
info_hash	SHA-1 value used to identify a torrent
peer_id	SHA-1 value used by a peer as its id, to register in a tracker
ip	Used to tell the tracker the ip address of a peer <optional>
port	Port number that a peer is listening on
uploaded	The amount of data a peer has uploaded
downloaded	The amount of data a peer has downloaded
left	The amount of data a peer still need to download
compact	Indicate in what form a peer-list will be sent back. '0' for dictionary model and '1' for binary model.
numwant	Number of peers that a peer requests from the tracker.
event	If value is specified, it must be one of 'started', 'stopped' or 'completed'; if not, used as regular(KEEP_ALIVE) request.

Table 4: Responses (*Tracker*$\longrightarrow$*Peer*)

Type Name	Usage
interval	Interval a peer must wait before sending next regular request
complete	The number of peers with complete file (seeds)
incomplete	The number of peers with incomplete file (leechers)
peers	The peer-list sent back to a peer, can be a binary string or a dictionary, depending on 'compact' request type.

ded in one request by separating them with "&". The response from a tracker is normal 'text/plain' HTTP document containing a Bencoded dictionary. And each piece of information corresponds to a key in this dictionary.

The Table 3 and Table 4 summarize some important request parameters and responses respectively. When a peer joins in a swarm, the first request sent to a tracker must contain "event" parameter and the value must be set to "started". And when it completes the download, it must inform the tracker with "event" parameter and set the value to "complete". What's more, a peer will regularly send request containing only "event" parameter to the tracker without any specified values. This kind of non-value events serves as KEEP_ALIVE messages, which is a widely-used mechanism in a distributed system.

There is another request named "scrape", which does not appear in the table above.

Since it is not supported by every implementation of tracker. If "scrape" is supported, tracker will report simple statistical information about all the torrents it hosts. By specifying a torrent id explicitly, the corresponding information of that torrent will be returned.

3.4.1 Peer-list – Ticket for the entrance

Not like other P2P systems(e.g emule), BitTorrent constructs one overlay for one torrent, which is usually referred as swarm. From a peer's point of view, a tracker is the middleman who can introduce it to a swarm by returning a peer-list.

Tracker will return certain amount of active peers in this swarm to a requesting peer. The requesting peer can specify the number of peers it wants in a peer-list with "numwant" request parameter (in Table 3). The actual amount of peers returned is calculated with the formula $min(numwant, max\_give)$. [5] If no value is specified, 50 peers will be returned by default.

In a typical swarm, it is common that new peers keep joining and old peers keep leaving. An active peer may leave the swarm for good after some time. To guarantee the download efficiency, A peer must maintain the number of its buddies above certain level. If the number drops below this level, the peer will request a new peer-list from the tracker to compensate the loss of buddies.

In official implementations, $\_check$ function will be called regularly (every 60 seconds) to check its internal states and variables. If it finds the current buddies are less than the threshold defined in the $config$, it will request for more peers if it hasn't requested any peers within 5 minutes. Usually, BitTorrent keeps 20 to 80 buddies.[6] After having 40 buddies, a peer stops initiating new connections to others, but it still can accept incoming connection. After having 80 buddies, it stops accepting incoming connections either. Any more incoming connections will be closed immediately.

[5] max_give is a parameter defined in the tracker, indicating the maximum number of peers that a tracker can return

[6] Actually, three parameters - min_peers, $max\_initiate$ and max_allow_in decide the range of buddy number together.

3.5 Peer Protocol

In BitTorrent system, Peer Protocol is an application layer protocol. It is used to guarantee every peer can understand each other in the communication. Peer Protocol operates upon TCP protocol. To make our introduction simple and clear, we assume two peers, which are peer A and peer B in our scenario. In this section, we only introduce the messages used in Peer Protocol, the details about how the behaviors are defined upon these messages will be discussed in the section 3.6.

3.5.1 Message formats

There are two kinds of messages used in Peer Protocol. The first is handshake message, which is used to establish the communication between two peers. The other one is normal message, which is used to carry control messages and exchange data. Their structures are shown in figure 2.

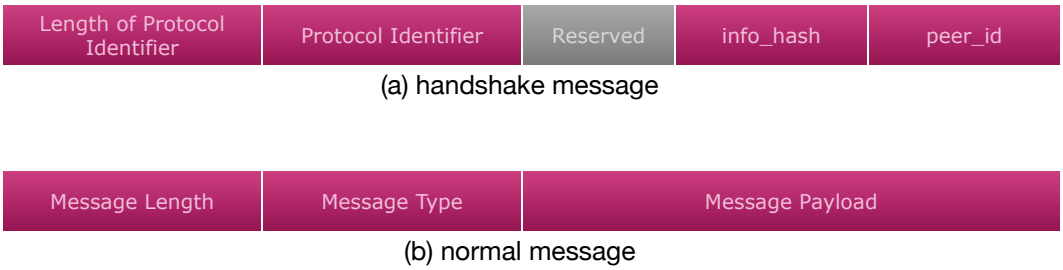


Figure 2: Two kinds of messages used in Peer Protocol

3.5.2 Handshake message

Very similar to TCP's 3-way handshake, handshake mechanism is also used in BitTorrent. The establishment of TCP connection between peer A and B doesn't mean the communication between them is established successfully. They should also finish another 3-way handshake in application layer. We refer the handshake message from A to B as *initiator's handshake*; and B to A as *recipient's handshake*. The difference is that peer_id is omitted in initiator's handshake message, since A will send it separately after it receives the recipient's handshake.

From the figure 2 (a), we can see how handshake message is organized. The first section indicates the length of the second section. The second section indicates the

protocol's name, and the usual value is "BitTorrent protocol". The third section carries a 20-byte torrent id, which can be used to group peers into different swarms. The fourth section carries 20-byte long peer_id, which is used to identify a peer uniquely. Different implementations use different way to generate this peer_id.

3.5.3 Normal message

From figure 2 (b), we can see the normal message is divided into three sections. The first section, which is four-byte big-endian value, indicates the length of the latter two sections(type section plus payload section). The message type section is very important, since it not only decides the first and third section, but also reflects how the peers communicate with each other. The table 5 lists some important message types. (Suppose A sends message to B)

Table 5: Message Types Used In Normal Messages (A⟶B)

Type ID	Type Name	Usage
none	KEEP_ALIVE	A informs B that A is alive
0	CHOKE	A informs B that B is choked by A
1	UNCHOKE	A informs B that B is unchoked by A
2	INTERESTED	A informs B that A is interested in B
3	NOT-INTERESTED	A informs B that A is not interested in B
4	HAVE	A informs B that A has a new piece of data
5	BITFIELD	A sends its bitfield information to B
6	REQUEST	A requests a slice from B
7	PIECE	A sends a slice of a piece to B
8	CANCEL	A informs B that A wants to cancel a slice request A sends before
9	PORT	A informs B which DHT port A is listening

Since these normal messages are closely related with peers' behaviors defined in Peer Protocol and internal implementation. We will explain them in details in the following section 3.6.

3.6 Internal implementation and mechanisms

Compared with Tracker Protocol, Peer Protocol is more complicated and important. Since Peer Protocol directly decides the peers' behaviors in a swarm, which will fur-

ther affect the overall performance of the system. Illustrating how a distributed system works is always considered as a very challenging job. It is easy to get overwhelmed by technical details at peer level and lose the overall understanding of the whole system. However, as a typical complex system, BitTorrent system's behaviors are indeed composed of many individual's behaviors. Without the exact knowledge of peer-level mechanisms, we can not study the system at all.

To solve the dilemma above, we use a different way to illustrate how BitTorrent system works. Since peer-level communications compose the core of the system, we explain the Peer Protocol by explaining how two peers cooperate in a file distribution process. What's more, to make our illustration more orderly, we split the whole communication process into small scenarios. The relevant message types and mechanisms will be explained in those scenarios.

3.6.1 Logical Architecture

Figure 3 shows the logical architecture for BitTorrent Mainline ver4. It is called "logical architecture" since the interactions and relations among these objects are more complicated in the actual implementation. In order to make the application's logic clear and easy to understand, some simplifications are made.

For example, *SingleDownload* object is actually created by *Downloader* object for each successful connection. However, it is more logically reasonable to connect *SingleDownload* object with *Connection* object. Since each *Connection* object has a corresponding *(SingleDownload, upload)* tuple, which are responsible for download and upload job of this connection respectively.

Furthermore, the *Config* object is created in *MainApp*. But almost every object in BitTorrent uses it, and keeps a reference to it.

Another thing worth mentioning is the *BitField* object. The one in *StorageWrapper* object is local peer's bitfield, while the one in *Upload* object is its buddy's bitfield on the other end of this connection.

Multitorrent: this object manages multiple torrents' download jobs, responsible for creating _SingleTorrent object for each torrent; maintain a RawServer object internally, which all the BitTorrent's traffic will go through.

Config: this object stores all the parameters of a BitTorrent client. Many of them have great influence on system performance, so they should be tuned carefully. The parameters reflect the essences of the BitTorrent Protocol.

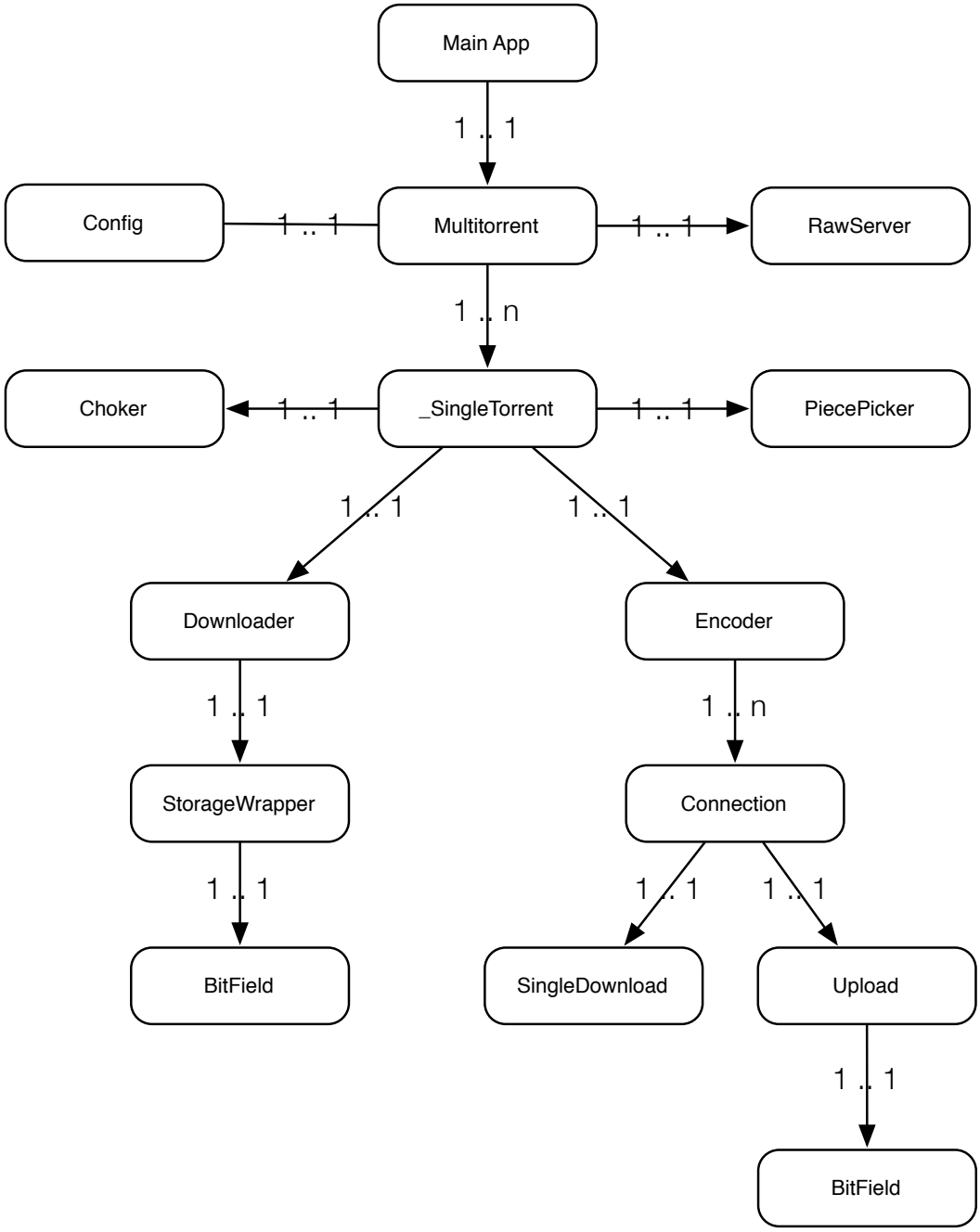


Figure 3: Logical architecture of official BitTorrent implementation – Mainline Ver4

RawServer: this object is responsible for sending and receiving data for all the torrents, there is only one RawServer in the application. So all the internet traffic will go through this object.

_SingleTorrent: this object manages single torrent's download job, every torrent has a corresponding _SingleTorrent, every _SingleTorrent has one Encoder, one Downloader, one PiecePicker, and one Choker object.

PiecePicker: this object realizes piece selection strategy.

Choker: this object performs peer selection strategy periodically.

Encoder: this object manages all the incoming and out connections of a _SingleTorrent; every Encoder objects has multiple connections.

Connection: every connection object represents a logical connection between two peers, and every connection has one SingleDownload and one Upload object.

Downloader: this object does some necessary work for _SingleTorrent, such as creating SingleDownload object for each connection.

StorageWrapper: this object stores a peer's own BitFileld object; while the SingleDownload in Connection stores its buddy's bitfield.

SingleDownload: this object represents the download job in one connection.

Upload: this object represents the upload job in one connection.

3.6.2 Connection Management

BitTorrent utilizes two dictionaries to maintain all connections, *connections* dictionary and *complete_ connections* dictionary. The difference is *complete_ connections* dictionary only includes the connections with successful 3-way handshake, which means every peer in it is a buddy. *connection* dictionary is the superset of *complete_ dictionary*, it also includes the connections that have not finished handshake procedure yet.

3.6.3 HANDSHAKE - Let's start talking

Suppose peer A is the initiator of the communication. First, A obtains a peer-list from the tracker, from which A can choose a peer it wishes to connect, suppose it is peer B. After extracting B's IP address from the peer list, A will try to establish TCP connection to B. The very first message sent to B must be the handshake

message to establish the application layer communication.

Figure 4 shows the logical processes in both A and B, and figure 5 shows the corresponding message flow during handshake. As the initiator, A is supposed to send the initiator's handshake message immediately after TCP connection is established. As soon as B receives the info_hash part, B will check whether there is a match among the torrents it hosts. If there is not, B will drop the connection immediately. If there exists a torrent matching the info_hash B has just received, B will response A by sending the recipient's handshake message, which contains the same info_hash and B's peer_id. After receiving B's recipient handshake, A will send back its peer_id. If B successfully receives A's peer_id, then the application layer communication is established.

In official implementation, the call of function *connection_completed(self, c)* means the successful establishment of the communication, then this function will create corresponding *Upload* and *SingleDownload* objects for this connection.

3.6.4 BITFIELD & HAVE - What can I share?

The most charming feature of BitTorrent system is that peers can exchange data among themselves, not like in C/S architecture, all the downloaders have to compete for the limited server's bandwidth. This feature rise an obvious issue - how does a peer locate a piece it wants? In other words, how can a peer know what data is offered by its buddies?

To inform the others about what pieces a peer has, two message types can be used - BITFIELD message and HAVE message. In short, the BITFIELD message is used in the beginning of a connection between two peers, and will be send only once during the lifespan of this connection. However, HAVE message will be send multiple times.

After the application layer handshake, the immediate message exchanged between peer A and B is the BITFIELD message. The only exception is for a newcomer who has no pieces at all, then there is no need for this newcomer sending BITFIELD message. The BITFIELD message carries a peer's bitfield indicating what pieces a peer possesses currently. Because bitmap size of a file correlates with the file size, the payload of this message is variable. We need to indicate explicitly the length of the payload. The receiver B will check the bitfield from A with the metainfo in the corresponding torrent file. If there is any error in the bitfield from A, B will drop

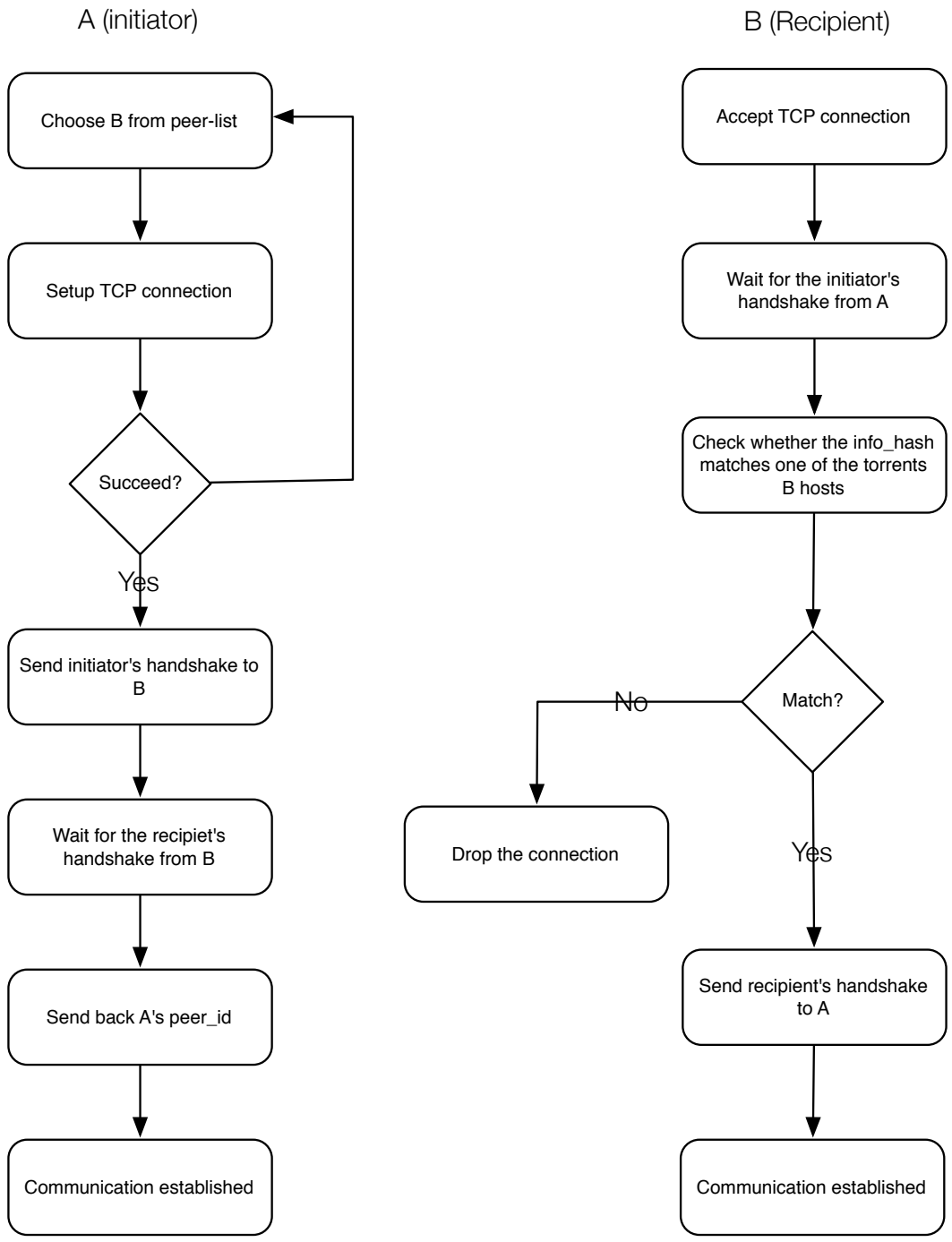


Figure 4: Flowchart for handshake process

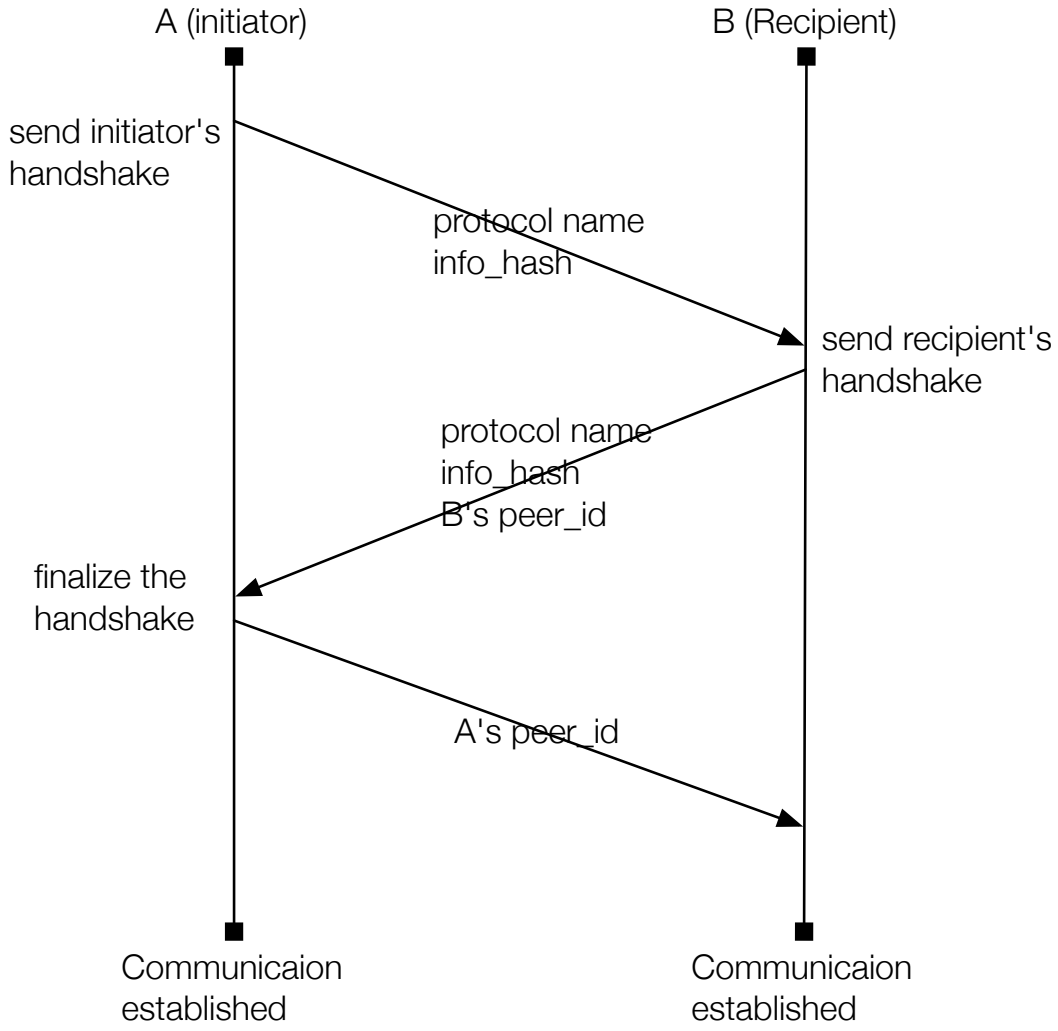


Figure 5: Message flow for handshake process

the connection, and vice versa.

Since the download process of a pees is always running, peer A may download a new piece after exchanging its bitfiled with B. At this time, A will use HAVE message to inform B about this event. More precisely, all A's buddies will receive a HAVE message from A, telling them that A has a new piece and they can request this piece from A if they need it. HAVE message is much shorter than BITFIELD message, the payload section only carries the piece index.

However, this mechanism may change a little in different real world implementations. One example is so-called lazy bitfield. With lazy bitfield, the bitfield A sends to B does not coincide with the pieces A actually possesses, there might be some bits are cleared by A on purpose. A will use HAVE message to complete the missing piece later. The purpose of this modification is said to be useful in against ISP's filtering. However, the true effects are not formally investigated.

Another example is, in some implementations, A will not send HAVE messages to all its buddies, since some of them have already possessed that piece and will never request for it. A can obtain the necessary information from its buddies' bitfields and their succeeding HAVE messages. By suppressing HAVE messages, the overheads in the communication can be reduced. But it also introduces another problem, *rarest first strategy* will not work properly without enough information about the piece distribution. Since it cannot calculate the exact number of HAVE messages for each piece any more.

3.6.5 State information for a connection

Peer A and B become buddies after they successfully established the communication. The connection between them is not stateless, both A and B have to maintain some state information. By storing buddies' state information, the management tasks become easier in BitTorrent.

We define two functions to represent two kinds of relationship between A and B. For *interest* relation from A to B, we write:

$$interest(A, B) = \begin{cases} 0 & \text{If A is not interested in B} \\ 1 & \text{If A is interested in B} \end{cases} \tag{3.1}$$

For *choke* relation from A to B, we write:

$$choke(A,B) = \begin{cases} 0 & \text{If A is not choking B} \\ 1 & \text{If A is choking B} \end{cases} \tag{3.2}$$

Because *interest* and *choke* relations are not symmetric[7], A should not only store the relations from itself to B, but also store the corresponding relation from its buddy to itself. So each peer at the both ends of a connection has to maintain the same pair of 2-tuple state information, which are $(interest(A,B), choke(A,B))$ and $(interest(B,A), choke(B,A))$.

By default, the initial values for both $choke(A,B)$ and $choke(B,A)$ are "1"; and for $interest(A,B)$ and $interest(B,A)$, the initial value are "0". It means two peers are not interested in each other, and will block each other in the beginning. The CHOKE and UNCHOKE messages from A to B will change the value of $choke(A,B)$. Similarly, the INTEREST and NOT-INTERESTED messages will change the value of $interest(A,B)$. And no matter how they change the value, this pair of 2-tuples at both ends should always remain the same during a connection's lifespan.

Besides the 2-tuples mentioned above, a peer also stores its buddy's bitfield, download and upload rate and some other information. We have mentioned the two objects, which are *Upload* and *SingleDownload* are created for each connection. These two objects store the state information for the link. Figure 6 shows how these state information are organized in real implementation.

3.6.6 INTERESTED & REQUEST - I want some data from you

By default, a peer can have maximum 80 buddies. From each buddy, it receives a corresponding bitfield. Then the peer will store the buddy's bitfield into *have[]* list in *SingleDownload* object, and check whether its buddy can provide any fresh data. If the buddy can, the peer will inform the buddy its interest by sending INTERESTED message. If the buddy is not choked, whenever a peer receives an INTERESTED or NOT-INTERESTED message, a new round of *peer selection* will be triggered.

After sending INTERESTED message, the peer will wait until it is unchoked by its buddy. As soon as the peer receives the UNCHOKE message, it will re-check the

[7] i.e $choke(B,A) = 0$ doesn't imply $choke(A,B) = 0$, vice versa. And the same for *interest* relation.

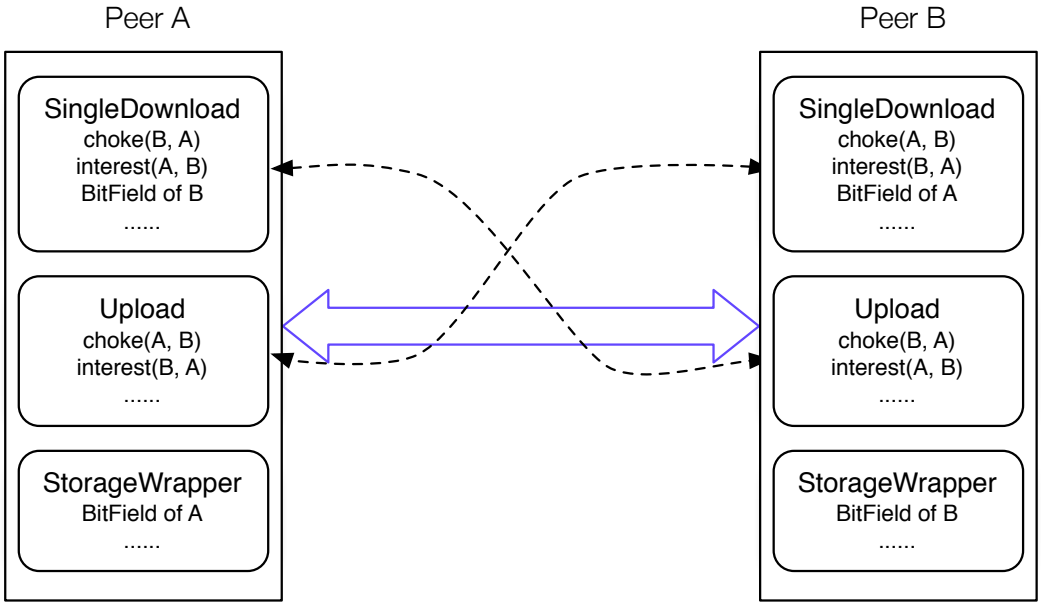


Figure 6: State information of a connection in actual implementation

state information $interested(A, B)$. If it is still interested in its buddy, it will send REQUEST messages for the slices it wants and start downloading. The re-check for $interested(A, B)$ is necessary. Because the slices A wanted from B might have already been downloaded from other buddies during the period A was choked by B. In this case, B possibly cannot provide any more fresh data to A.

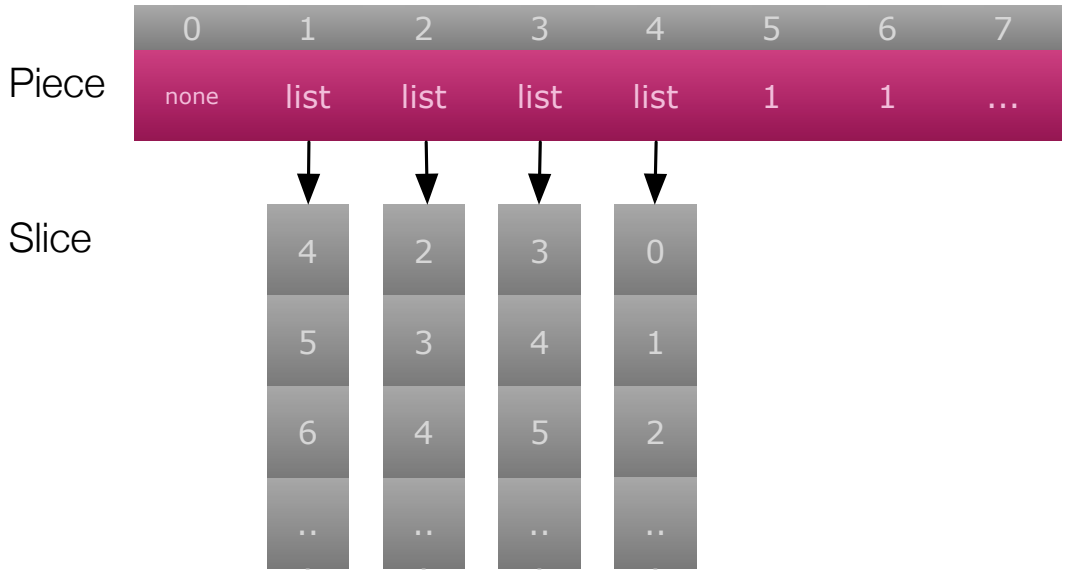


Figure 7: inactive_request list in BitTorrent

BitTorrent generates a REQUEST for each slice. Internally, it maintains a list as the one in Figure 7. Each element in *inactive_request* corresponds to a piece, if the value is 1, it means the requests for the slices in this piece are not generated yet. If it contains a sub-list, then each element in this sub-list corresponds to a REQUEST message for one slice. Every time a REQUEST message is sent out, the corresponding element will be removed from the sub-list. Till all the REQUEST messages for one piece are sent, the sub-list will become empty, then the corresponding element in *inactive_request* will be set to None.

We must point out, *none* value in the element of *inactive_request* only means all the slices in this piece have been requested, it does not say anything about whether the slices are received or not. Furthermore, since data exchange is based on slice, it implies the slices in one piece may be downloaded from different peers.

BitTorrent limits the maximum number of REQUEST that can be sent to a buddy.[8] So a peer will keep sending REQUEST to its buddy until it reaches this threshold or its buddy cannot provide any more fresh pieces. In the latter case, the peer will send a NOT-INTERESTED message to its buddy.

To avoid sending duplicated REQUEST for the same slice, BitTorrent checks the other buddies, and send NOT-INTERESTED messages to those who can only provide the slices that have already been requested. However, in EndGame mode, things would be different, and we will discuss it in section 3.6.11.

People may ask why we don't reduce the piece size to the slice size directly, such that there is no need to use this two-level mechanism. The reason is small piece size will lead to large torrent file, since more digest information has to be stored in it. However, large slice size is not good for the efficient transmission on the internet, since the whole slice needs to be retransmitted if it is corrupted. Usually, small slice pieces are more reliable and can be pipelined to improve the download performance.

3.6.7 Peer Selection

As we have introduced in section 3.4.1, a peer usually maintains 40 to 80 buddies during the download process. However, it can only upload data to some of them, the others will be temporarily choked. The decision about "uploading to which buddy" is called *Peer Selection Strategy*.

A leecher and a seed have different peer selection strategies, which can be summa-

[8]The parameter limits this number is *_backlog*.

rized in Table 6

Table 6: Peer Selection Strategy

Role	Strategy
Seed	uploads to the peers with fastest download rate
Leecher	uploads to the peers with fastest upload rate

By uploading to the peers with fastest download rate, a seed can speed up the process of "replicating". However, a leecher only chooses those peers who provide it with better download rates, then upload data to them. The reason is to keep the system as fair as possible, the tit-for-tat strategy is adopted. Besides, a leecher also reserves one upload slot for *optimistic unchoking*. We will discuss it separately in section 3.6.9.

In official implementation, *_rechoke(self)* function is invoked every 10 seconds to perform peer selection. And the actual concurrent uploads can be specified explicitly with the parameter *max_uploads*. If *max_uploads* is set to negative, then the concurrent uploads is calculated on the upload bandwidth a peer is willing to share, which is set by *max_upload_rate*. The higher the upload bandwidth, the more concurrent uploads a peer can have.

Let *uploads* denote the number of concurrent uploads, *rate* denote the max upload rate, the equation (3.3) shows the calculation.

$$uploads = \begin{cases} 2 & \text{if } 0 < rate < 9, \\ 3 & \text{if } 9 \leqslant rate < 15, \\ 4 & \text{if } 15 \leqslant rate < 42, \\ \sqrt{rate \times 0.6} & \text{if } rate \geqslant 42, \\ 7 & \text{if } rate \leqslant 0. \end{cases} \tag{3.3}$$

3.6.8 Piece Selection

When peer A is unblocked by B, A can then request data from B. The A's decision on which piece to request is called *Piece Selection Strategy*. Two main strategies are used in different phases of downloading, which are *Random* and *Rarest-First*.

The decision tree in Figure 8 illustrates how BitTorrent chooses its piece selection strategy.

In the beginning of download, when A has no data at all, it will use *Random* strategy.

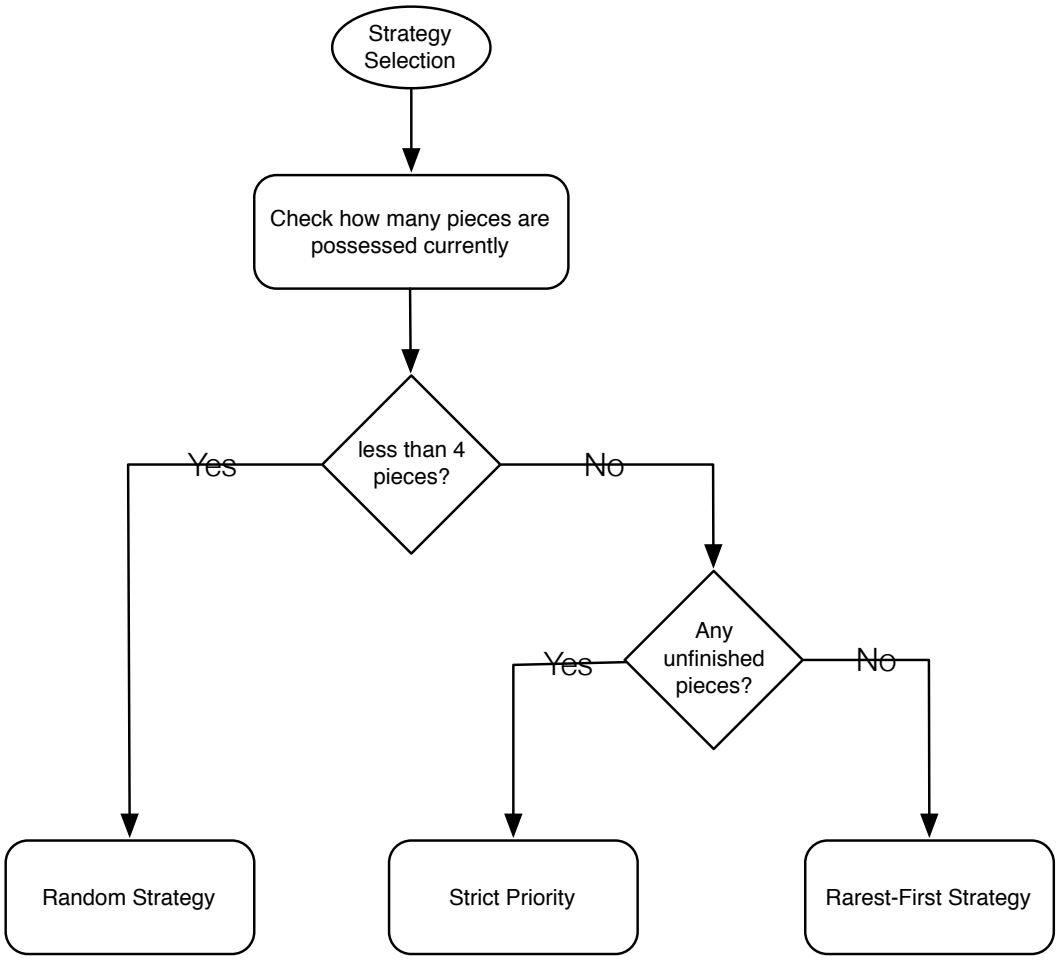


Figure 8: Decision tree for piece selection strategy

A will choose a piece randomly and then request the piece from one of its buddies. After *A* successfully downloads several pieces, it will switch to *Rarest-First* strategy. In official implementation, the threshold value for switching from *Random* to *Rarest-First* is defined by *rarest_first_cutoff*, whose default value is 4. So a peer will not use *Random* piece selection for long time.

Bharambe et al. compared the system performance by using different piece selection strategies in [BHP06].[9] Their work indicates *Rarest-First* can outperform *Random* greatly when the seeds' aggregate upload bandwidth is low. However, if the seeds' aggregate bandwidth is high, then *Random* can perform as good as *Rarest-First*.

From figure 8, we can see a subtle issue. Actually, peer *A* will check whether there are any unfinished pieces first. If there are, *A* will request the missing slices for those unfinished pieces, then *Rarest-First* will take effects. This extra step is usually referred as *strict priority*. Because slice is the basic unit for data exchange among peers, it is possible that *A*'s buddy will choke *A* before it can download a complete piece. However, *A* cannot forward any data in a piece unless it has a complete one. The *strict priority* guarantees that *A* can get a complete piece quickly and will not be suffered from many unfinished ones.

From system perspective, *Rarest-First* strategy tries to maximize the diversity of the content; from peer level, it tries to make a peer as attractive as possible in a tit-for-tat system. If A possesses some rare data, it will attract more buddies requesting for it from A. In such a way, A will get more chances to download fresh data from them.

How does a peer decide a piece is rare? It is impossible for a peer to have the global piece distribution in the swarm, so the decision for the rarest piece must be made on the local information.[10] The solution is HAVE message. We have known that a peer will use HAVE message to inform its buddies whenever it receives a new piece completely. In such a way, a peer is able to keep track of the number of HAVE messages for each piece. The piece with the least HAVE messages from the buddies is the *rarest*.

As we can see in Figure 9, three important lists are used to implement *rarest-first* strategy.

[9]The benchmark used here is 'uplink utilization', which is the ratio of aggregate bandwidth of all peers to aggregate capacity of the system.

[10]That's why some people also refer it as *Local Rarest-First*.

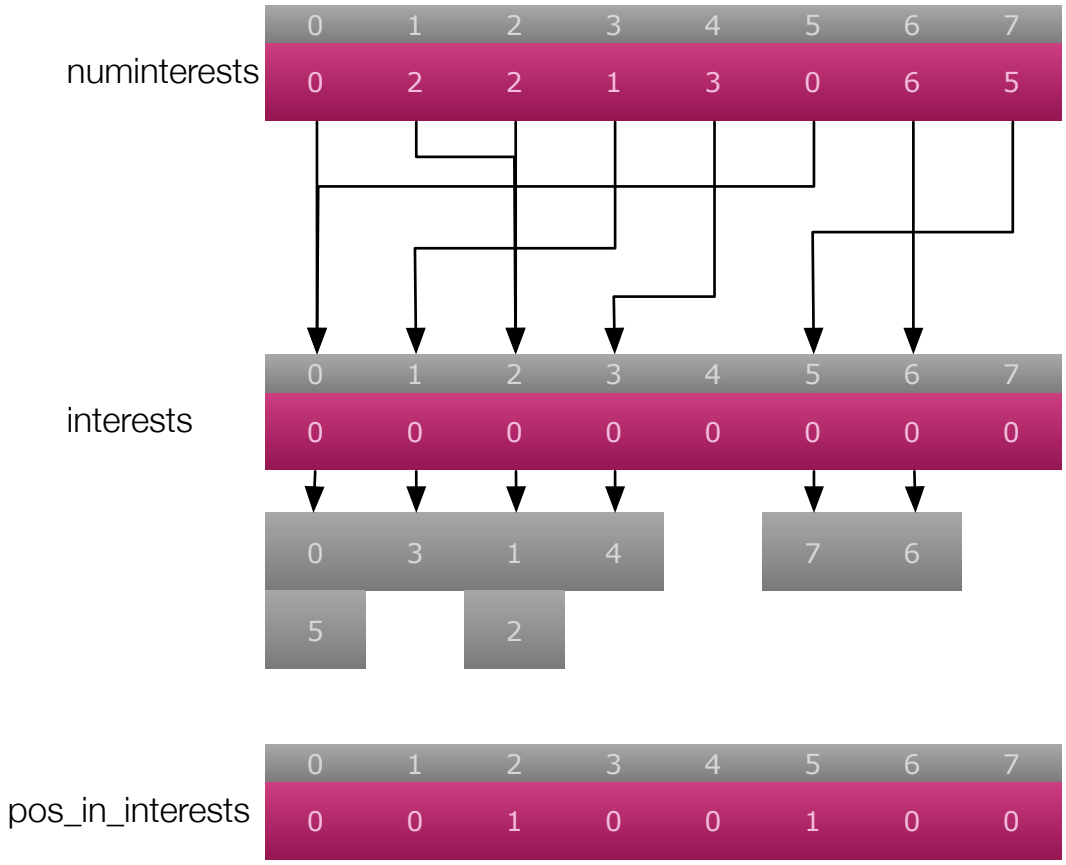


Figure 9: Three important lists used to implement Rarest-First

- *numinterests* list: index of each element corresponds to the piece index, value of each element corresponds to the number of HAVE messages received for this piece. In Figure 9, we can see piece 0 and piece 5 get no HAVE messages, piece 1 and piece 2 get two HAVE messages, and piece 3 get one.

- *interests* list: index of each element corresponds to the number of HAVE messages, value of each element is another list consisting of the pieces with the same HAVE message number. For example, *interests[0]* stores the pieces which have not any HAVE messages, namely, piece 0 and piece 5; *interests[3]* stores the pieces which have three HAVE messages, namely, piece 4.

- *pos_in_interests* list: index of each element corresponds to the piece index, value of each element corresponds to the piece's position in the sub-list stored in *interests* list. For example, *pos_in_interests[5]* is 1, which means piece 5 is in the position 1 in the sub-list stored in *interests[0]*.

If we consider *interests* list as a two-dimensional matrix, then for each piece i, tuple $(numinterests[i], pos\_in\_interests[i])$ indicates the exact position of piece i in the matrix. With the help of this matrix, the *Rarest-First* strategy can be easily achieved.

3.6.9 Optimistic Unchoking

As we have mentioned in section 3.6.7, BitTorrent performs a peer selection every 10 seconds. Besides those unchoked by the normal peer selection strategy, a peer will randomly unchoke one buddy in every third round of unchoke, which means every 30 seconds. This mechanism is referred as *optimistic unchoking* in [Coh03].

The optimistic unchoking serves two purposes. The first is to help a peer explore the network, the second is to help a newcomer bootstrap quickly.

As we have seen, a peer cannot connect to all the peers in a swarm, and the tracker only returns certain amount of peers every time. So a peer only has partial information of the swarm. In order to find the peers who can provide better upload bandwidth, a peer has to periodically perform optimistic unchoking. Optimistic unchoking is also very important for a newcomer, since it has nothing for exchange when it first join in a swarm. If tit-for-tat strategy is performed strictly, the system is impossible to bootstrap at all.

From another angle, optimistic unchoking means a peer is willing to upload to a buddy altruistically even it cannot download anything from that buddy for 30 seconds. However, as a side effect, this mechanism gives a chance to free-riders.

3.6.10 Tit-for-Tat

Tit-for-Tat(TfT) is the application of Game Theory in BitTorrent. TfT in BitTorrent is based on the upload rate but not on the data volume. This results in the the positive correlation between a peer's download rate and its upload rate. Izal et al. point out in [IUKB+04], rate-based TfT also means a peer may upload more data than it actually downloads. Especially for those with higher upload bandwidth, if they are connected with many low-bandwidth buddies, the asymmetry in data exchange is almost inevitable in terms of amount of date.

The essential reason for the asymmetric data exchange is peers' heterogeneous upload bandwidths. What's more, with rate-based TfT, people found that peers will

cluster based on their upload bandwidth in the long run. With optimistic unchoking, a fast peer can find more and more peers with the similar upload bandwidth. Since it can get better download rates from them, the peer also tends to upload data to them in return. For a slow peer, it is very difficult to join in fast peer's game. The only thing that slow peers can count on is optimistic unchoking.

Bharambe et al. thought this asymmetric data exchange is unfair and proposed a "Pairwise Block-Level Tit-for-Tat" strategy in [BHP06]. The proposed strategy is based on the amount of data exchanged between two peers and more strict. A peer only uploads one or two blocks more than the amount of data it downloads from one buddy. As a consequence, the asymmetry in rate-based TfT is eliminated. However, this strategy may cause a peer cannot fully utilize its upload bandwidth, since the upload rate of a fast peer may be throttled by a slow peer. From system perspective, Block-Level TfT degrades the system's overall performance by decreasing the replication speed.

In [BHP06], Bharambe et al. also proposed a new tracker protocol which provides a fast peer with a peer-list containing more peers with similar upload bandwidth. In such a way, the effects of asymmetric data exchange is weakened by helping a faster peer find more fast ones. In other words, the new tracker protocol encourages and speeds up the clustering process.

It is known that BitTorrent makes up a peer-list by choosing peers randomly. This algorithm leads to a random graph when constructing swarm in the initial phase[11], which has high connectivity and is resilient to attacks. However, the proposed tracker in [BHP06] will change the initial topology of a swarm, the cliques will appear earlier and more.

3.6.11 End Game Strategy

People observed that the download speed of BitTorrent usually dropped significantly at the end of downloading, especially when there are only a few pieces left to complete the download. The reason is it becomes more and more difficult for a peer to locate fresh data when it has almost all the pieces, since it does not know the global piece distribution. This issue is often referred as "last piece problem".

For example, if peer B is not a seed, it is highly possible that B cannot offer A any fresh data when A almost finishes its download task. Even if B is a seed,

[11] We use "initial phase" here, because rate-based TfT makes the peers cluster in the later phase.

$choke(B, A)$ might be 0 at that moment, which also makes A not able to download anything from B. In order to solve this issue, people introduced *End Game Strategy* into BitTorrent.

The End Game strategy is supposed to start working automatically in the final phase of downloading. The exact time when End Game will be triggered depends on the threshold value set in the applications. In official implementation, BitTorrent enters into End Game mode when there are no inactive requests left, which means all the slices have been requested. There are many discussions about the suitable time to enter into End Game mode, and some people think this threshold should be based on the complete percentage or on pieces. In our opinion, threshold based on percentage is not a good idea. Since when distributing a large file, even 1% of the file is still quite a lot of data. Entering into End Game mode too early also makes BitTorrent become very inefficient, unless this percentage is a function of file size and can be adjusted automatically.

In normal mode, if a slice has been requested, BitTorrent will not request it again unless the request is lost or the slice is corrupted. However, when BitTorrent enters into End Game mode, it will request all the missing slices from all its buddies, even those have been requested. When it successfully receives a slice from a buddy, it will send CANCEL messages to the other buddies to invalidate the previous request for this slice. This will prevent BitTorrent system from becoming inefficient, since sending a CANCEL message only cause very little overhead, much less than re-downloading the slice from the and then abandon it.

4 Methodology

4.1 Terminology

Besides those commonly used terms, in this thesis, we also use the following terms to simplify the discussion.

We refer two connected peers as *buddies*. If a peer's buddy is on the same node with this peer, we refer it as a *native buddy*; otherwise a *foreign buddy*. *Aggregated bandwidth* represents the total traffic generated by a group of peers in every second. It can be further divided into *aggregated download bandwidth* and *aggregated upload bandwidth*. In this thesis, we only concern the average value, not the instantaneous one, so the *aggregated download bandwidth* is calculated as the product of

average download rate and the number of peers. Likewise for the *aggregated upload bandwidth*.

All the experiments we performed can be divided into two categories, in one of which, we set a limit on the leecher's max upload rate, the download rate is unconstrained, we call this kind of experiments *upload-constrained experiments*; in another kind, we set a limit on the leecher's max download rate, but the upload rate is unconstrained, and call them *download-constrained experiments*. In all of our experiments, two distinct nodes are used for deploying the tracker and the seed respectively. There is only one seed in every experiment and its max upload rate is always constrained.

Since every peer will register itself to the tracker before joining into a swarm. Our experiment scripts query the tracker periodically to monitor the number of peers in the swarm. That's the way how we calculate the start-up peers. So if a peer cannot register itself successfully to the tracker, even the BitTorrent instance is running, we don't consider it as a successfully start-up peer.

4.2 General principle

Our general principle will partly be based on the theories and approaches in the research area of complex systems. In a complex sytem, the individuals usually have very simple mechanisms, but the whole system can exhibit very complicated behaviors. Undoubtedly, P2P system is a complex system. Despite its complexity, [GK99] shows it is still feasible to study such systems and proposed three modes for investigation, which we are going to apply in our research – experimental, computational, and theoretical.

Adaptability represents how the systems respond to external conditions. It is the key characteristic of various complex systems [AO04] and also our principal concern. Basically, BitTorrent's adaptability is determined by its peer selection strategy and piece selection strategy. The clustering property is a representation of adaptability.

4.3 Specific methods

Our basic method is using aggregated bandwidth to probe the system capacity. We believe the experiments should be performed within the system capacity. If the workload generated by an experiment is beyond this capacity limit, the data will be biased.

There are several things need to be clarified here. One is the *system capacity* is determined by the minimum capacity of CPU, memory, network or any other factors that may restrict the experiment scale, in other words, restrict the number of peers can be run on one node. The bottlenecks from the CPU, memory and storage are easy to detect. But the bottleneck from the network is difficult to handle, especially when running multiple peers on one node.

Average download rate is an important indicator for system performance in the study of P2P system. It also plays an important role in our experiments. Intuitively, the average download rate should start dropping after the experiments reach the system capacity limit. However, our research shows the average download rate and the corresponding aggregated bandwidth cannot reflect the system capacity correctly. The average download rate still remains at a stable level even though the network has already been saturated. The reason originates from BitTorrent's innate characteristic.

In many previous research papers, the researchers usually perform upload-constrained experiments, and set the max upload rate to a relatively low value. The main reason is most BitTorrent users connect to the internet with ADSL whose upload bandwidth is much smaller than the download bandwidth. Another reason is the official BitTorrent client (Mainline Ver.4) is a popular target in research area, because it is open source, simple and includes all the core functionalities. The biggest problem is there is no download rate limiter in Mainline Ver.4. It is ok with homogeneous experiments, since the upload rate is usually the bottleneck of the whole system. But for heterogeneous experiments, especially when we take peers' arrival pattern into account, download rate must be constrained.

The most significant difference in our experiments is that we set max upload rate to a relatively high value, 5000KB/s. This decision is based on two considerations. The first is the bandwidth keeps improving during these years, more people connect to the internet using Fibre/LAN with symmetric upload and download bandwidth. According to the reports on [oec], in Korea, 16.4 of every 100 inhabitants use Fibre/LAN connections.

The second consideration is it is easier to probe the system capacity by using high upload rate. As we have mentioned, there are various bottlenecks restricting running multiple peers on one physical node. Using high upload rate guarantees the network will become the first bottleneck, thus simplified our problem. Another benefit is we can easily observe how BitTorrent reacts to the network changes.

What's more, we also performed download-constrained experiments. Because in such experiments, BitTorrent exhibits very interesting changes in its behaviors. And we cannot find better example than the data from download-constrained experiments to show how the piece selection and peer selection affect the clustering property.

5 Preparing Experiment Platform

5.1 Experiment environment

Our experiments are performed on a cluster consisting of 30 nodes. Each node is equipped with a 8-core 2.8GHz CPU, 32GB memory and connected to a Gigbit Ethernet. The underlying operating system is Ubuntu smp with linux 2.6 kernel. The TCP congestion control used in the network between the nodes is CUBIC TCP. The parameters `net.ipv4.tcp_wmem` (controls the sending buffer – size of `cwnd`) and `net.ipv4.tcp_rmem` (controls the receive buffer – size of `rwnd`) are set to "4096, 16384, 4194304" and "4096, 87380, 4194304" respectively (minimum, default, and maximum).

For the BitTorrrent client used in our experiment, we considered several ready-made clients. However, they are not full-fledged, and cannot satisfy our requirements. For example, the max download rate cannot be set, data logged is not comprehensive enough. What's more, all of them are only for the small-scale experiments, are not qualified for the use in large-scale and heterogeneous experiments. So, we modified client by ourself, the target client is official version - BitTorrent Mainline Ver.4

Our instrumented BitTorrent supports various useful features such as bypassing the disk I/O, controlling the buffer size, setting different log level. It is also able to mimic the peers behind a firewall or the free-rider's behaviours by setting different switches such as `firewalled`, `free_rider` and so on.

5.2 Enlarge experiment scale

The original design of BitTorrent only allows only one instance running on one node. Considering the configuration of the nodes in our cluster, one peer per node scheme is really a waste of our resources! What's more, even we have hundreds of machines, they are still not enough if we want to enlarge the scale to thousands and even ten thousands of peers. So running multiple peers on one physical node is an attractive

solution. With the capability of running multiple instances on one node, it is possible to deploy large-scale experiments with limited nodes. If there are abundant available physical nodes, as we are going to show in the following sections, the experiment scale can be easily enlarged without even considering the effects from the loopback interface, and the capacity planning is easier to handle.

We also considered virtual machines like VServer and KVM. One advantage of virtual machine is they can provide strong isolation at very low level. Some low-level parameters such as IP address, physical upload and download bandwidth can be configured for each peer respectively. This is a very attractive feature when deploying heterogeneous experiments. However, virtual machines are relatively resources-consuming. The system performance degrades fast if we try to run hundreds of instances on a physical machine. Virtual machine can be a good choice if and only if very few peers are going to be deployed on one machine such that the experimenter can guarantee the system performance will not degrade.

In our work, different instances are isolated on the application layer. More peers can be supported compared with the virtual machine scheme. In order to support multiple instances, we modified the code and added some helper functions such as creating working and configuration directories on the fly to avoid conflicts among different instances. The resident memory for each instance is 10MB to 14MB, so the memory won't be our bottleneck.

We must point out one side-effect introduced by running multiple peers on one node, even though it is not a problem in our research. In BitTorrent, a peer uses a 20-byte long peer-id to identify itself uniquely. A 'greedy' peer may try to make multiple connections to another peer by using different peer-ids. In such a way, it can obtain better download performance. To prevent such things, BitTorrent allows only one connection from one ip by default.

In order to run multiple instances on one machine, an ip address is shared among different instances, this feature has to be disabled. Since all the clients running on the cluster are under our control, this feature is not useful to us. But if it is an important factor in some other's experiments, it is better to know how it affects BitTorrent's behaviours.

5.3 Data collection

High-quality experiment data plays an important role in the analyzing peer-level behaviors. Zhang et al. in [ZIea10] gave very valuable guidelines to avoid the sampling bias in BitTorrent experiments. We followed their recommendations in designing our data collection scheme.

We implemented a Logger module and embedded it into BitTorrent Client. The Logger module is used to collect important information during the lifespan of a peer in the system. It will record the important events happening within the client, such as the timestamps for starting the client, joining the swarm, finishing downloads, leaving the system and so on. Besides that, the Logger module also takes a snapshot for the peer every second. The snapshot includes the information such as, the current upload and download rate, share ratio, transferred data size, and the connections maintained by the client at the moment.

Since the Logger module records almost all the important information, it gives us a good chance to study the BitTorrent behaviors in details. Especially the ability of connection tracking, which proves to be helpful in study of peer selection strategies.

5.4 Bypass hard-disk I/O

Our first experiment was performed in the simplest setting, one seed and one leecher. Since we didn't limit the upload and download rate, the transfer rate should reach the network bandwidth, which is around 125MB/s. However, the stable transfer rate in our experiment is only 70MB/s, far below the value it is supposed to be.

By monitoring the system resources carefully, we found the bottleneck is I/O operations to the hard disk. The speed of writing data to the hard disk cannot keep up with the speed at which BitTorrent receives data. Lots of CPU resources are wasted on I/O wait. To solve this issue, we must prevent BitTorrent from writing data to the disk. Another good reason for doing so is because of the limitation on storage capacity. In our experiment, we use large-size(2-GB) file as distribution content. The reason is we can amortize the system's "warm-up" time to the long distribution time, the data will be more accurate. However, the hard disk cannot provide us enough space if all the peers really write data to the disk.

We considered two methods to bypass write operations:

1. Method 1: simply discarding all the data received can eliminate all the write

operations. However, a peer is both a client and a server in a P2P system. It is uploading to others while it is downloading at the same time. If it discards all the received data, then what can it do if someone else requests those data later?

2. Method 2: manipulating the file in the memory. The advantage is it can improve both read and write operations greatly. However, we don't have that large memory if hundreds of peers keep their own copy of the file of several GB in memory.

Our solution is a combination of these two methods. We intercept the read and write operations within a peer. We let BitTorrent do nothing in write operation requests, just drops the received data. At the same time, all the read operations from different instances are redirected to the same file. Since there is only one copy of the distribution file, the storage space is saved and issues above are solved.

Considering all the nodes are equipped with large memory, we let the operating system cached the complete file in an experiment. We also adopt memory-mapped file mechanism in the instrumented client, and make the cached file shared by all the peers on the same node. In order to eliminate the *major page-faults* caused by the first access to the file, we pre-load the file into the memory before every experiment. With these methods, overheads caused by page-faults and system-calls can be reduced. The CPU resources spent on I/O wait is almost zero even hundreds of peers run on the same node.

Then we repeated the experiment with the simplest setting above, the transfer rate increased from 70MB/s to more than 115MB/s. The improvement can be seen in the table 7.

Table 7: Average download rate with and without I/O to hard disk

I/O bypass	Transmission rate	CPU on I/O wait
NO	70MB/s	85%
YES	115MB/s	almost 0%

5.5 Tune BitTorrent's parameters

BitTorrent has several dozens of parameters can be tuned, some of which have great influence on the performance. Many developers spend quite amount lot of time

on tuning and testing those parameters to gain better performance. And these parameters are set to different values in different implementations. Even in the official implementation, same parameters are changed in different versions. These changes on the parameters reflect the changes in the network environment, at least from the developer's perspective.

Basically, the BitTorrent is designed for low speed internet. Some parameters which give BitTorrent good performance on the internet may not be suitable in a high performance cluster. What's more, the impacts from these parameters can be amplified when we run multiple peers on one node. By carefully tuning them, we can deploy more peers on a node. We investigated the impacts from various parameters, and below, we will explain 3 important parameters we discovered.

5.5.1 Sending buffer

The first parameter can be tuned to improve the performance is upload_unit_size. It controls the sending buffer in the application layer. When BitTorrent sends data, it writes 1380 bytes into TCP layer every time by default. As a result, it generates huge amount of I/O operations in our experiment setting(high transfer rate, multiple peers on one node, etc.). However, when BitTorrent receives data, it will try to fetch up to 100 KB from TCP buffer every time.

By increasing this upload_unit_size, more data can be passed to TCP layer in a single write operation. So the number of I/O operations can be reduced given the same amount of data. In our experiments, we increased this number to 64 KB. Then we repeated the upload-constrained experiment and observed obvious improvements.

5.5.2 Slice size

As we have introduced in section 3, slice is the basic transmission unit. If data in a slice is corrupted, the whole slice needs to be re-transmitted. So in a unreliable network, large slice size may make the transmission inefficient.

In Mainline ver4, there are two parameters controlling the slice size. The first one is *download_slice_size*, which decides how many bytes this peer will request from others (as a slice). The second one is *max_slice_length*, which decides the maximum length of a slice can be send to other peers, the connection will be closed if a larger request is received. *max_slice_length* should be no less than *download_slice_size*. In ver4, *download_slice_size* is 16 KB, and *max_slice_length* is also 16 KB by

default.

In Mainline ver5, Bram Cohen changed *download_ slice_ size* to *download_ chunk_ size*, and *max_ slice_ length* to *max_ chunk_ length*. Names are changed, but their functionalities remain the same. The default value of *download_ chunk_ size* is still 16 KB, but the default value of *max_ slice_ length* is increased to 32 KB. One reason is network condition keeps improving, 16 KB is really small, compared with the huge files being distributed on the internet. Another reason is some other BitTorrent implementations have already adopted larger sizes, in order to exchange data with other implementations, official version has to enlarge the *max_ chunk_ length* a little bit. Or the requests from others with larger *download_ chunk_ size* will be dropped.

In our experiment, We experimented with various slice sizes and observed significant improvements in performance when the slice size was increased from 16 KB to 32 KB. And further increasing to 64 KB resulted in a clear improvement over 32 KB. However, beyond 64 KB, further increase on slice size cannot bring any more significant improvements on the performance.

5.5.3 Concurrent uploads

The number of concurrent uploads[12] plays an important role in BitTorrent's clustering property. The larger this value, the more difficult for the peers to get clustered. In the extreme situation, when a peer uploads to all its buddies, then it is impossible to observe any clustering at all. As we have introduced in section 3.6.7, the concurrent uploads can be calculated according to the equation (3.3). We can see, when the max upload rate is set to unlimited, 7 concurrent uploads will be used, which is quite a conservative number. This can be viewed as an implication that BitTorrent is designed for low-speed network.

Concurrent uploads also has strong influence on the system capacity. The larger the concurrent uploads, the smaller the system capacity. Because too many concurrent uploads also cause excessive I/O operations.

[12] also referred as upload slots

5.5.4 Peer Set Cardinality

"What is the minimum peers we should use in an experiment?" – is an issue worth discussion. In many previous papers about BitTorrent, the researchers ignored the issue about the minimum peers they should use in an experiment, while they try their best to enlarge the experiment scale. However, our research shows the minimum peer set cardinality can also affect the experiment data.

We found that as the swarm size grows from 0 upwards, the average download rate keeps decreasing until there are 40 peers in the swarm. Then the average download rate will remain roughly constant until we hit the system capacity limit. The reason for this behavior is that the peer-list that a peer gets from the tracker contains 40 peers. Hence, when the swarm has less than 40 peers in total, every peer knows every other peer and the connection graph between them is a full mesh. This means that every peer has to maintain more buddies and thus the overhead of maintaining the connections increases as more peers join in (but less than 40). In large swarms, peers only maintain connections to about 40 peers, so the overhead remains stable after that point, until we reach the system capacity.

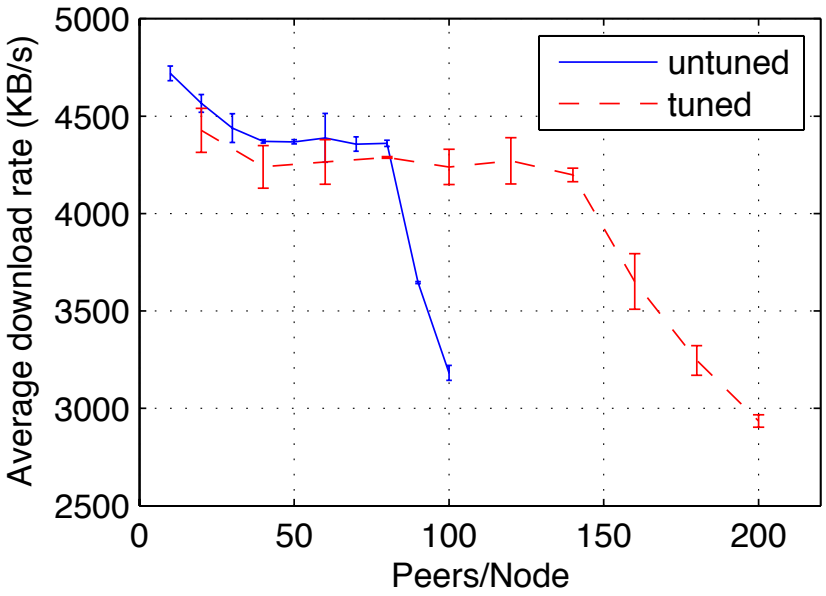
The important lesson we learned is the swarm size in any experiment should be larger than the peer-list.

5.6 Improved result

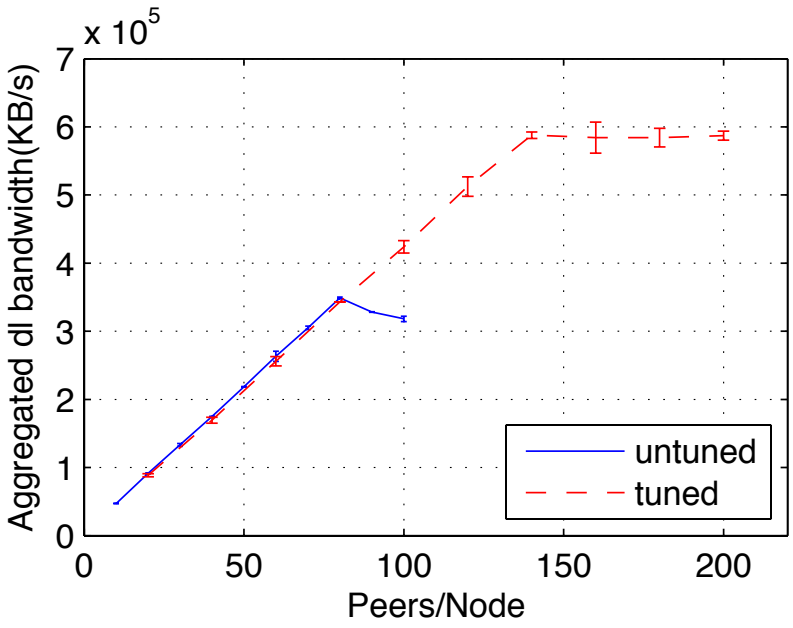
In order to check the effects of the parameters described in section 5.5 on the system capacity, we carefully tuned these parameters and performed download-constrained and upload-constrained experiments respectively. Then we compared the data with those in the untuned case. The results are shown as below.

The figure 10 shows how many peers we can deploy on a single node with or without tuning BitTorrent parameters in a download-constrained experiments. In these experiments, we had only one seed on a node and set its max upload rate to 5 MB/s. All the leechers were deployed on a different node and all their max download rates were constrained to 5 MB/s, the concurrent uploads is set to 7. The blue solid line in figure 10(a) shows the average download rate as a function of peers per node in untuned case. We can see the average download rate enters into the stable stage at 40 peers/node [13] , and remains stable till it reaches 80 peers/node. After

[13]recall that the peer list has 40 peers, as discussed in section 5.5.4



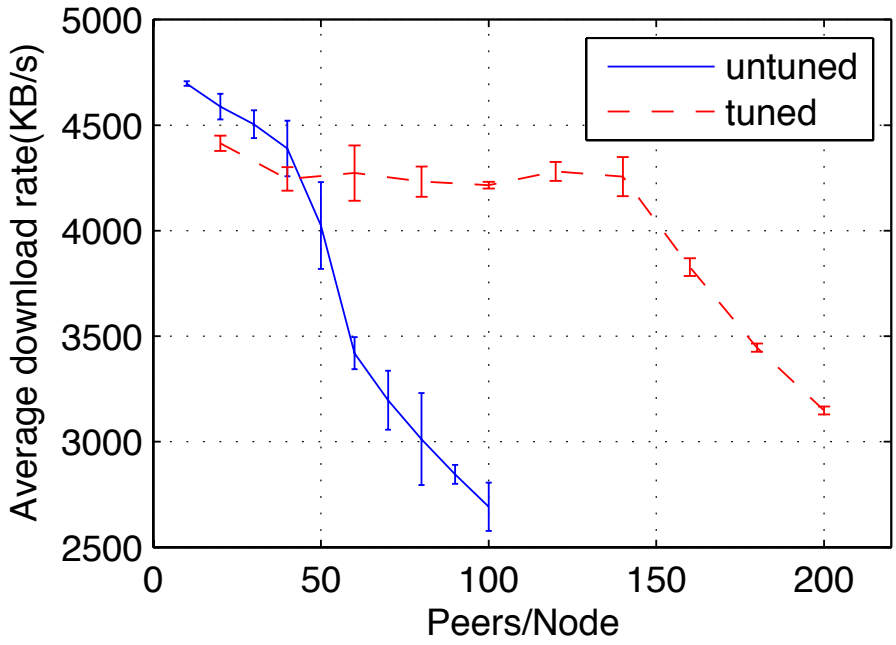
(a) Average download rate



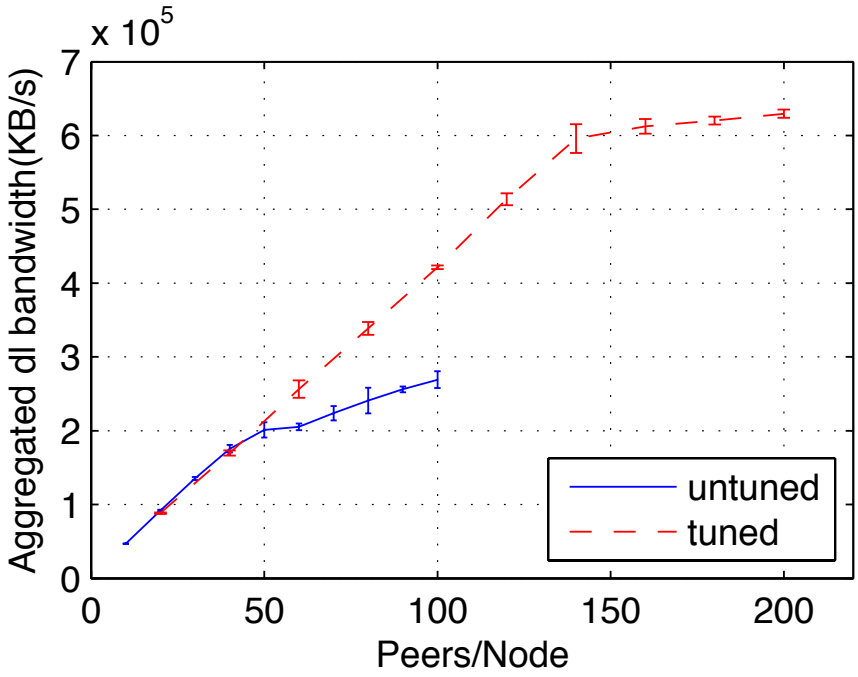
(b) Aggregated download bandwidth

Figure 10: Effects of tuning BitTorrent parameters on average download rate and aggregated bandwidth as a function of peers per node in download-constrained experiments. The bars show 99% confidence intervals.

80 peers/node, the average download rate drops sharply. This change can also be observed clearly on the corresponding aggregated bandwidth in figure 10(b) . Before reaching the system capacity at 80 peers/node, the aggregated bandwidth keeps in-



(a) Average download rate



(b) Aggregated download bandwidth

Figure 11: Effects of tuning BitTorrent parameters on average download rate and aggregated bandwidth as a function of peers per node in upload-constrained experiments.

creasing linearly to 350 MB/s. Without tuning the parameters, we can only deploy maximum 80 peers on a node with the experiment configurations described above.

The red dashed line in figure 10(a) shows the average download rate as a function of peers per node in tuned case. From figure 10(a), we see the average download rate can remain stable in the interval from 40 peers/node to 140 peers/node, which is almost doubled compared with untuned case. And the corresponding aggregated bandwidth can reach almost 600 MB/s. With tuned parameters, we can deploy 140 peers on a node maximum.

The figure 11 shows the results from upload-constrained experiments, which are similar to those in download-constrained experiments. In upload-constrained experiments, the seed's settings remained the same as in the previous experiment; the leecher's max upload rates were constrained to 5 MB/s and max download rates were unlimited.

In figure 11(a), the average download rate in untuned case(blue line) never entered into the stable phase. It kept decreasing after 40 peers/node. Even the aggregated bandwidth increased to 260 MB/s at 100 peers/node, the increase is not linear any more. The significant drop in average download rates also indicated the system was already overloaded. So we don't consider the experiments are performed "safely".

In tuned case, the system showed almost the same capacity limit as that in download-constrained experiments. The average download rate remained stable from 40 peers/node to 140 peers/node, and the corresponding aggregated bandwidth was able to reach 600 MB/s.

There are two things worth noticing. The first is in tuned case, the curves have almost the same shape no matter in upload-constrained or download-constrained experiments(compare the red lines in figure 10 and 11). However, in untuned case, the shapes of the curves are quite different in these two kinds of experiments. The reason is in untuned case, the *concurrent uploads* is set to 7 explicitly in both kinds of experiments. In untuned case, BitTorrent will calculate *concurrent uploads* by itself according to the equation (3.3) in section 3.6.7. So in upload-constrained experiments for untuned case, BitTorrent will set *concurrent uploads* to 54, which is much higher than 7. This resulted in large amount of I/O operations, can cause system's instability.

The second is that figures 10(a) and 11(a) show that the average download rate for the tuned case is slightly lower than untuned case before reaching the system

capacity limit. This is because the tuned case uses a larger slice size, hence a piece will be divided into a smaller number of slices. Request pipelining which allows efficient parallel downloads is not as efficient as before in this case, hence the average download rate suffers slightly. But the benefit we have is less I/O overhead.

So, the important lesson we learned here is, to maximize the resources utilization, the BitTorrent's parameters should be carefully tuned.

5.7 Other restrictions from OS and BitTorrent

If an experimenter decides to run multiple peers on one node, most probably, he will be overwhelmed by tons of underlying details and parameters. Knowing these issues enables us to control the experiment completely, even though it is very difficult. In this section, we will talk about some other restrictions either from the operating system or from the BitTorrent itself.

The first restriction is from BitTorrent itself. By default, BitTorrent tries to listen on port 6881 for incoming connections. If port 6881 is occupied, it will try the others in the range 6881–6999 sequentially. This means we can only start 119 peers maximum. After that, BitTorrent will report error of unavailable ports. So we set this range to 6881–9999 to guarantee there are enough ports to listen on.

We also observed an unexplained limit on the number of BitTorrent clients we were able to start (quasi) simultaneously. After starting 700 clients, the speed of starting new processes slowed down and after 800 clients it practically stopped. We were not able to get more than 835 clients started in this manner. We investigated several possibilities, but were not able to find a cause for this behavior. It was not an OS limit on starting processes, file handles, available local ports, nor the tracker. The behavior is repeatable, but so far we have not been able to find the cause.

In practical terms, this means that we have a hard limit on the number of peers that can start 'simultaneously'. In our experiment setting, we were able to start 500 clients on a single node within 15 seconds.

Besides the above restrictions, there are also some others from the kernel and TCP, such as the maximum processes a user can start, maximum sockets, queue length for loopback interface, *tcp_max_syn_backlog* and so on. All these parameters have influences on the experiments and system performance. We did experiments with tuning the kernel and TCP and did observe small potential performance gains, but none were significant enough to merit the added trouble of tweaking them.

6 Capacity Planning

After intensive study on how to tune the parameters to achieve the best system performance, we started our investigation in capacity planning. Capacity planning is very crucial for large-scale experiments on the cluster. Our goal is to determine general rules of thumb which a system designer can use to evaluate the performance of the system and to guide the experiment design.

We start from a very simple experiment setting in section 6.1. We will show how to decide the system capacity with a naive method. In the section 6.2, we show the safe region found with naive method is not safe any more in two nodes setup. In section 6.3, we will show a more elaborate method to estimate the system capacity and claim the method used in section 6.1 is immature. The actual experiment scale is much smaller than the naive method predicts. The average download rate is not sufficient in estimating the system capacity.

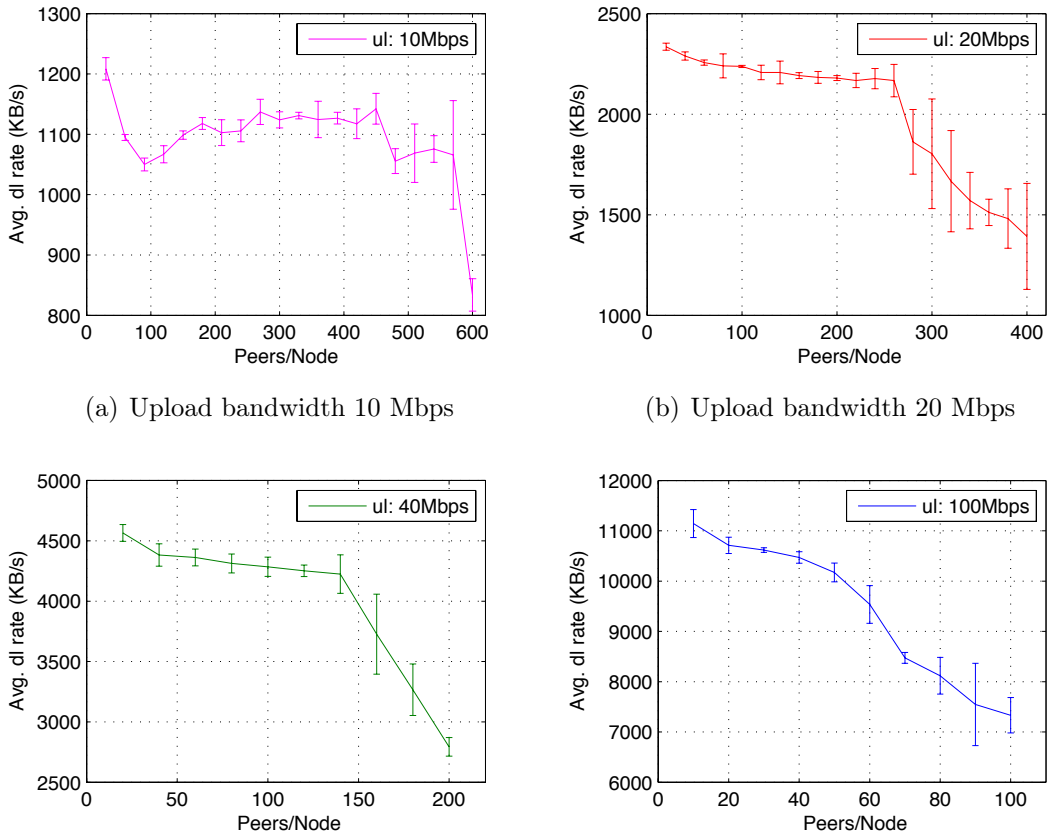
6.1 Naive capacity planning for single node

In the naive method, we only take average download rate into account. If there is no significant drop in average download rate, then the experiment is considered to be reasonable.

First we experimented with placing all the leechers on a single node. We increased the number of leechers until the average download rate was no longer stable, and the corresponding aggregated bandwidth cannot increase linearly. All leechers were upload-constrained and we used different values for upload bandwidths: 10, 20, 40, and 100 Mbps.

We investigated whether we can use the simple formula $y = \frac{a}{x}$ to roughly estimate how many peers we can put on a single node.(y is the maximum peers we can put on a single node. x is the max upload or download rate we set. a is a constant constant related to the aggregated bandwidth). If the transmission rate and max number of peers on a node have this simple relation, then we need not redo the capacity probing every time we change the experiment settings.

Figure 12 plots the average download rate against the number of peers. Even though the max upload rates are set to different values, the shapes of those curves are similar. The average download rate decreases gradually before reaching 40 peers/node, then it enters into a relatively stable stage. After reaching the system capacity, the



(a) Upload bandwidth 10 Mbps

(b) Upload bandwidth 20 Mbps

(c) Upload bandwidth 40 Mbps

(d) Upload bandwidth 100 Mbps

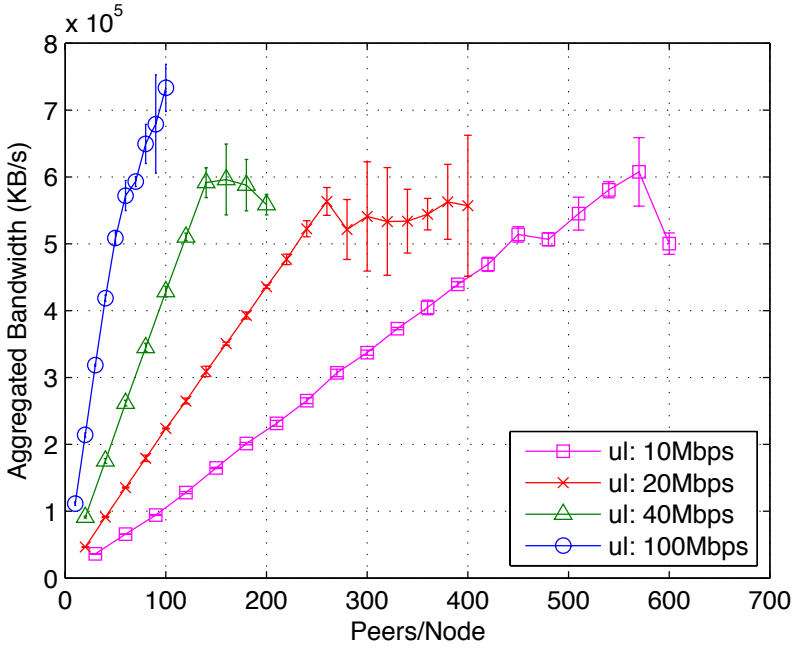
Figure 12: Average download rate as function of peers per node for different upload bandwidths for case of 1 node being used.

average download rate drops sharply.

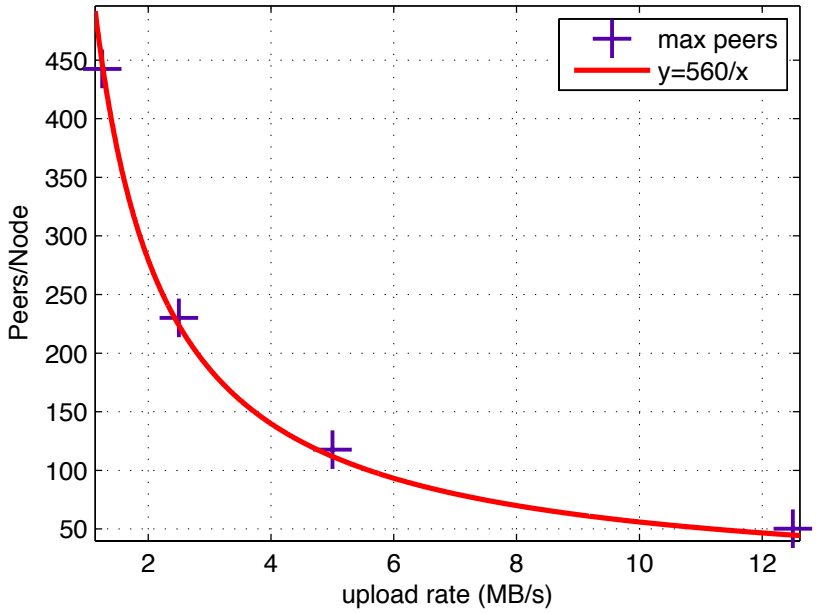
Figure 13(a) plots the corresponding aggregated download bandwidth based on the same experiment, with the curves from the different cases combined. As the figure shows, the aggregated download bandwidth can increase at least to 500 MB/s, and the corresponding average download rates remain stable. Thus we can define 500 MB/s of aggregated download bandwidth as the system capacity, and any value below that is considered safe. Since the curves in the safe region are basically straight lines, it is easy to fit a curve and find the corresponding x when $y = 500$ for each line. Then we can obtain the relation between the number of peers per node, x, and maximum upload rate per peer, y, as follows

$$y = \frac{560}{x}. \tag{6.1}$$

Figure 13(b) shows the curve for equation (6.1) and our data points. This curve can be used to set the values for upload bandwidth and number of peers in an experiment



(a) The corresponding aggregated download bandwidth for the cases shown in Figure 12.



(b) Number of peers/node vs. per-peer upload rate.

Figure 13: Aggregated download bandwidth with different upload rate and the corresponding capacity plan curve.

when all leechers are placed on a single node. In download-constrained experiments, we get similar results as in the upload-constrained experiments.

6.2 Naive capacity planning for more nodes

In this round of experiments, we move a step forward from the simplest setup in *single node capacity planning*. We try to answer the question: if we use more nodes in the experiment, will the naive method still work?

We deployed the leechers on two nodes(`cln008` and `cln018`) equally, starting from 20 peers/node, and increased 20 peers on each node in every succeeding experiment until it reached 200 peers per node. The max upload rate of each leecher is constrained to 5 MB/s, the download rate is unconstrained. Results for average download rate and aggregated bandwidth are shown in Figure 14. At the first glance, the results are quite similar to the ones obtained for the single node case (Figure 12(c) and 40 Mbps line in Figure 13(a)). In a download-constrained experiment, we obtained similar curves.

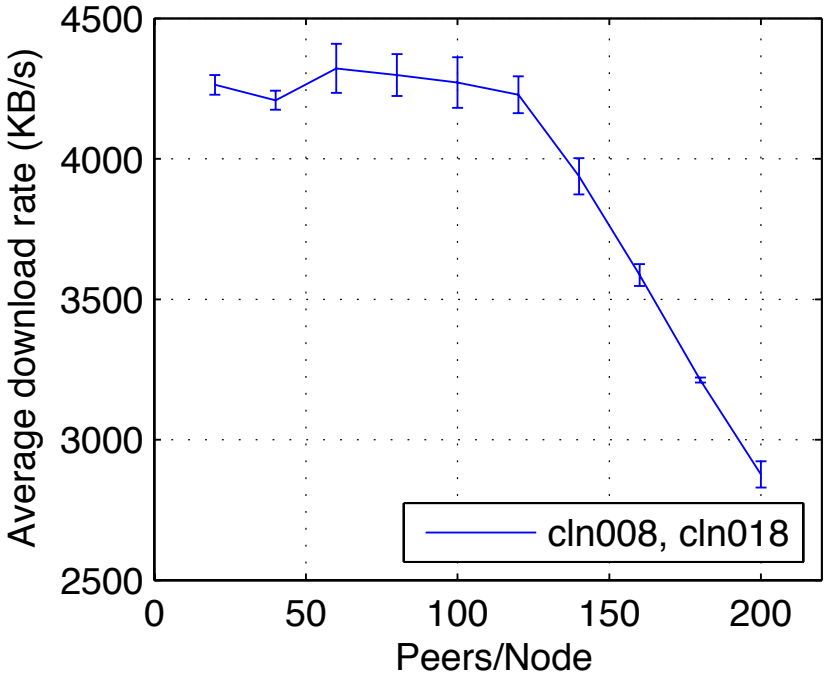
As the figure 14 shows, when we deployed the leechers on 2 nodes, the whole system still exhibit a capacity of more than 500MB/s. And before 120 peers/node, the average download rate remained stable. The results led us to the conclusion that we can deploy 120 leechers on each of the two nodes. However, by inspecting the actual network traffic and connections made by the peers, we noticed that already at 60 peers per node, the network between the nodes had been saturated (see details below). Figure 14 cannot exhibit the change in BitTorrent behaviors. That's the reason why we call this method naive and average download rate cannot be used as the only benchmark when we design the experiment.

However, from another angle, we consider the lack of observed change in the average download rate in changing network conditions as excellent evidence of BitTorrent's ability to adapt to varying conditions.

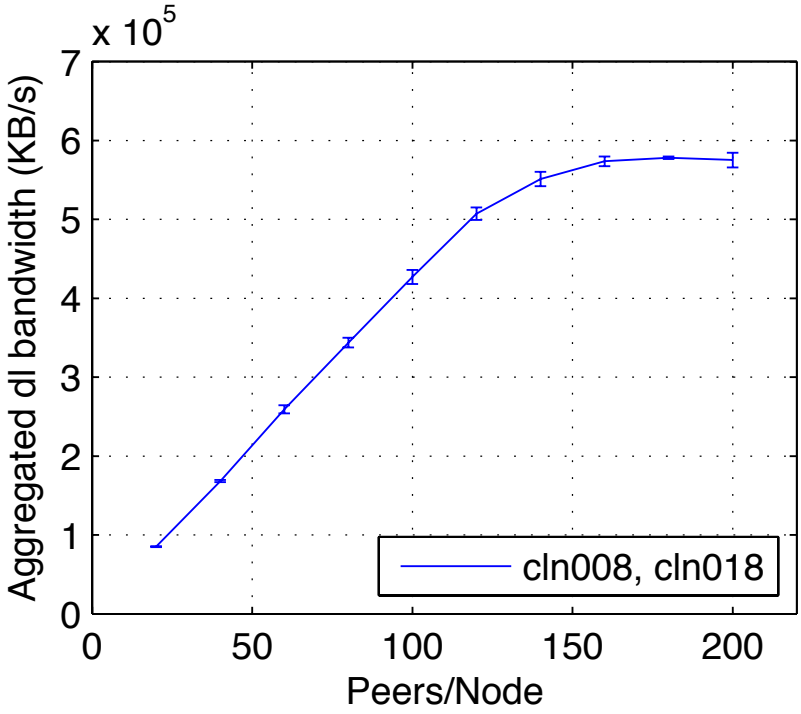
In section 6.3, we will introduce a better method to estimate the capacity. The behavior change will be discussed in details in section 7.1 and 7.2.

6.3 Capacity planning formulas

In this section, We will present a simple analytical means of determining whether a planned experiment falls within the system capacity limits or not. Table 8 lists the notation used in the following. Let $i, j, k \in \{1, 2, 3...n\}$. Since we only use one seed on a separate node in every experiment, compared with the traffic among the leechers, the traffic from the seed is negligible and we have excluded it for reasons



(a) Average download rate



(b) Aggregated download bandwidth

Figure 14: Leechers deployed equally on two nodes; upload-constrained experiment

n	number of nodes in an experiment
m_i	number of peers on node i
U_i	aggregated upload bandwidth generated by the peers on node i
D_i	aggregated download bandwidth generated by the peers on node i
L_i	physical capacity of loopback device on node i
C_i^{ul}	physical upload capacity of network card on node i
C_i^{dl}	physical download capacity of network card on node i
P_{ij}	probability that a peer on node i will connect to peers on node j

Table 8: variables used in the discussion

of simplicity. The discussions here can still be applied to multiple seeds setup after minor modification.

When we deploy multiple peers on one node, a peer will not only try to connect and upload to the native peers, but also to the foreign peers. P_{ij} is the probability that a peer on node i will connect to peers on node j, and assume all the peers on node i have the same probability. Then we have

$$P_{ij} = \begin{cases} \frac{m_i - 1}{\sum_{k=1}^{n} m_k - 1} & \text{if } i = j, \\ \\ \frac{m_j}{\sum_{k=1}^{n} m_k - 1} & \text{if } i \neq j. \end{cases} \tag{6.2}$$

When $i = j$, P_{ii} actually denotes the probability that a peer will connect to the native peers.

U_i and D_i denote the aggregated upload and download bandwidth on node i respectively. Obviously, U_i equals the sum of all peers' upload bandwidth on node i and D_i equals the sum of all peers' download bandwidth on node i. Then the traffic from node i to node j is[14]

$$T_{ij} = P_{ij} \times min(U_i, D_j) \tag{6.3}$$

[14]We have made the assumption that all peers on a node have the same limits on upload and download bandwidths.

We can construct a matrix to show the traffic flows between the nodes:

$$T = \begin{bmatrix} T_{11} & T_{12} & T_{13} & \dots & T_{1n} \\ T_{21} & T_{22} & T_{23} & \dots & T_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ T_{n1} & T_{n2} & T_{n3} & \dots & T_{nn} \end{bmatrix} \tag{6.4}$$

In matrix T, row i represents the distribution of the traffic flowing out of node i, and column i represents the distribution of the traffic flowing into node i. The elements on the diagonal represent the traffic going through the loopback interface of a node. This traffic in T must be constrained by the physical capacity of a node. Then for each i, j we have:

$$\sum_{i=1, i \neq j}^{i=n} T_{ij} \leqslant C_j^{dl} \tag{6.5}$$

$$\sum_{j=1, i \neq j}^{j=n} T_{ij} \leqslant C_i^{ul} \tag{6.6}$$

$$T_{ii} \leqslant \frac{L_i}{2} \tag{6.7}$$

Now, consider an extreme situation, when all the traffic goes through loopback interface or the network card, then we have the following constraints:

$$U_i \leqslant C_i^{ul} \tag{6.8}$$

$$D_i \leqslant C_i^{dl} \tag{6.9}$$

$$min(U_i, D_i) \leqslant \frac{L_i}{2} \tag{6.10}$$

To some extent, (6.5), (6.6) and (6.7) define the upper bound of the experiment, while the (6.8), (6.9) and (6.10) define the lower bound. The upper and lower bound will converge at two points. The first is when only one node is used for deploying leechers. Then there is only one element T_{11} in the matrix. The (6.7) and (6.10) will be the same, since all the traffic will go through the loopback interface.

The second is when an infinite number of nodes is used. Considering that we can only deploy a limited number of peers on a node, the probability that a peer will

connect to native peers decreases to zero. As a result, all the traffic will go through network card. Then (6.5) and (6.6) will be the same as (6.8) and (6.9). T_{ii} will be zero since no traffic will go through the loopback interface.

Furthermore, we need to clarify several things to make our method sound. Our analysis above is under the assumption that all the peers are started up 'simultaneously'. The upper bound applies to the beginning of an experiment. Since at that time, peer selection strategy has not taken effects yet, the traffic will be distributed only based on the probability we have calculated above. As time goes by, the traffic may shift among the nodes. How the traffic shift depends on the specific experiment configurations. However, the shift will always be constrained in the range defined by the equations (6.5), (6.6), (6.7), (6.8), (6.9) and (6.10).

From the discussion above, we can conclude when running multiple peers on a single node, it would be better to include as many nodes as possible into the experiments. Then the experiment scale will mainly be decided by the lower bound formulas, and the capacity planning becomes easy to handle.

7 Clustering and Analysis

As we claimed in section 6.2, if two nodes or more are used for deploying peers, naive method cannot be used for capacity planning any more. In this section, we will substantiate our above claim that BitTorrent's behavior has changed and that the average download is not an accurate indicator of a correct experiment in two ways. First, we will experimentally investigate how the connections between the peers are formed in the above experiment. Second, we will use the analytical methods derived in section in 6.3 to demonstrate that the above experiment with two nodes violates these intuitive conditions.

7.1 Clustering in upload-constrained experiments

We ran the experiment with two nodes as above, i.e., start with 20 peers per node, increasing it by 20 peers per node until we reach 200 peers per node. Upload rates were constrained to 5 MB/s and download rates were unlimited. In every experiment we kept track of all the connections maintained by all the peers and identified which connections are *native* (to peers on same node) and which are *foreign* (to peers on the other node). Every experiment was repeated 3 times and we present the averages

and the standard deviations in the figures.

Figure 15(a) shows the fraction of native buddies in the peer list given by the tracker. As we can see, the value hovers around 50% which is to be expected since two node are used for deploying peers equally, and the tracker picks the peers for the peer list uniformly at random. Investigating the fraction of native buddies (and consequently foreign buddies) allows us to determine how BitTorrent is choosing where to download from.

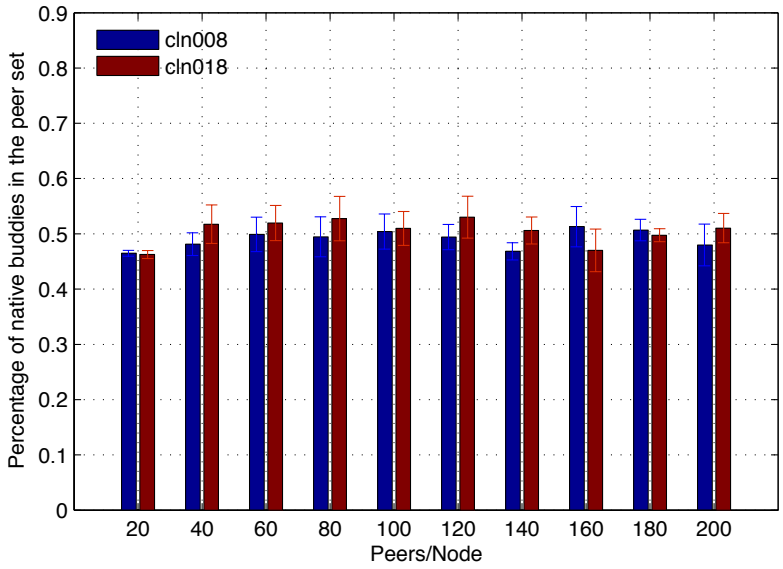
Figure 15(b) shows the fraction of upload connections to native buddies in an upload-constrained experiment. We used two nodes, `cln008` and `cln018` and show the values for both of them, as a function of the number of peers per node. As we can see, from 60 peers per node onwards, the peers tend to *favor native buddies* and the fraction of connections to native buddies keeps on increasing throughout the experiment. At 200 peers/node, more than 80% upload connections were made to native buddies.

The explanation is quite simple. Because the peers obtained in the peer list are evenly distributed, so are the connections in the smaller tests. Because both the native and foreign peers are able to serve data equally fast, a peer has no reason to prefer one over the other. (Recall that BitTorrent selects the peers to upload to or download from based on the bandwidth it obtains to/from that peer.) At around 60 peers per node, the amount of data going between the nodes is enough to saturate the 1 Gbps network link, whereas the local loopback device still has a lot of unused capacity. Hence, what we are seeing in Figure 15(b) is simply the normal BitTorrent's peer selection algorithm at work. In other words, the peers have clustered themselves locally but this effect is not visible in the average download rates or aggregate bandwidth shown in Figure 14.

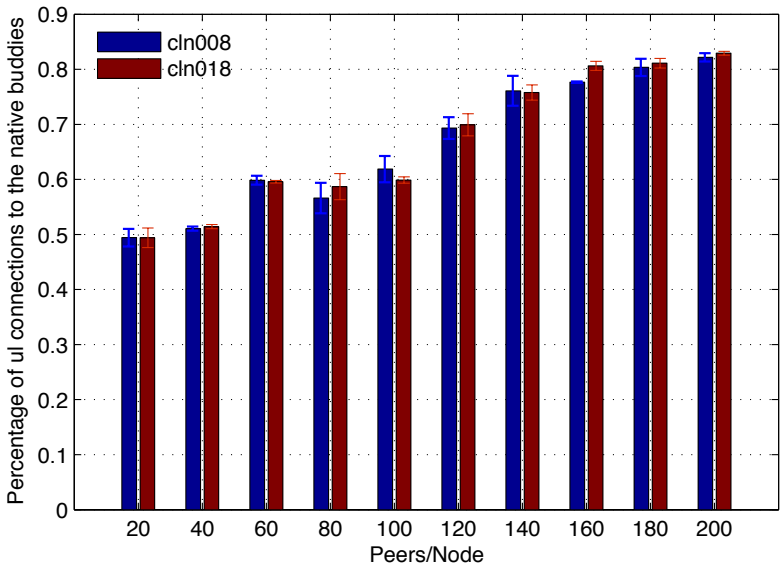
7.2 Clustering in download-constrained experiments

We repeated the above experiment, but this time constrained the download rate of every leecher to 5 MB/s and left the upload rates unlimited. Seed's maximum upload rate was 5 MB/s as in the other experiments.

Figure 16(a) shows the results from this experiment. As with the upload-constrained case, the network is saturated at around 60 peers per node, but the effects are drastically different from the upload-constrained case. The peers start favoring *foreign buddies* as opposed to native buddies for a longer spell and return to favoring



(a) Fraction of native buddies in the peer list

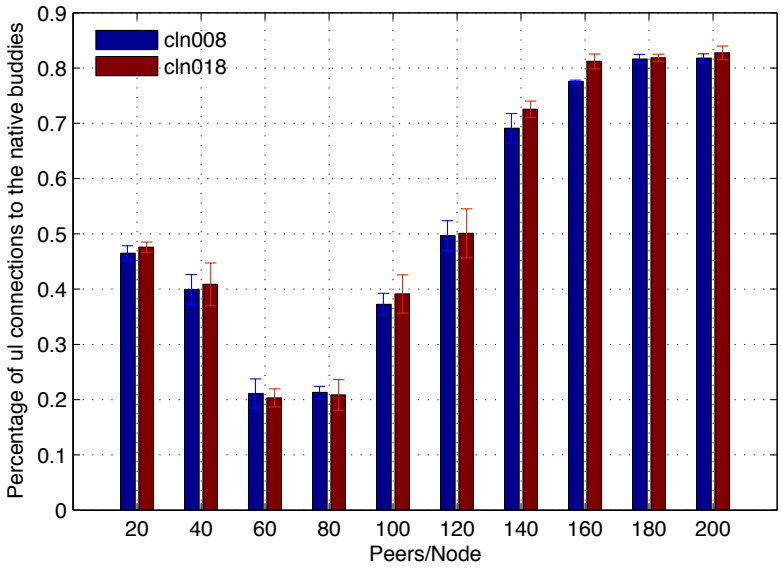


(b) Fraction of upload connections to native buddies in an upload-constrained experiment with 2 nodes

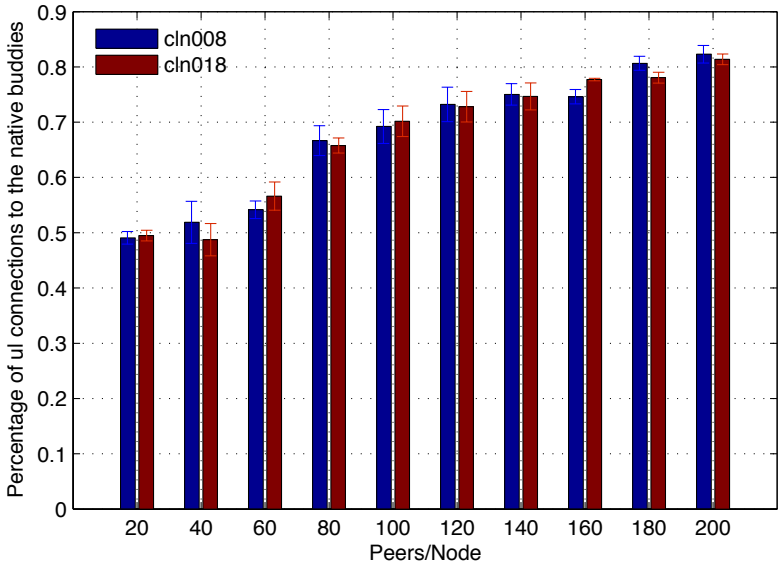
Figure 15: Track connections in upload-constrained experiments, 2 nodes

native buddies only in very large experiments.

Interestingly, our analysis of the situation showed that although most of the upload connections in the range of 60–80 peers per node were to foreign buddies, the peers received most of the data from native buddies. For example, with 20 peers per node,



(a) Fraction of upload connections to native buddies in a download-constrained experiment with 2 nodes.



(b) Fraction of upload connections to native buddies in a download-constrained experiment with 2 nodes and random piece selection.

Figure 16: Track connections in download-constrained experiments, 2 nodes

54.4% of the traffic came from native buddies, at 60 peers per node this was 60.1% and at 100 peers per node 77.2%. (The detailed traffic distribution on `cln018` is given in table 9.) Turns out that this behavior is a result of BitTorrent's *piece selection strategy*. Piece selection strategy in BitTorrent is based on a mechanism called

peers/node	20	40	60	80	100	120	140	160	180	200
Native (%)	54.4	56.9	60.1	68.4	77.2	79.0	79.3	79.4	81.6	82.8
Foreign (%)	45.6	43.1	39.9	31.6	22.8	21.0	20.7	20.6	18.4	17.2

Table 9: Percentage of traffic through the loopback interface(Native) and percentage of traffic through eth0 interface(Foreign) as a function of peers per node on `cln008`, corresponding to the figure 16(a)

peers/node	20	40	60	80	100	120	140	160	180	200
Native (%)	49.8	49.6	57.1	66.5	71.6	73.7	74.3	75.0	74.8	75.0
Foreign (%)	50.2	50.4	42.9	33.5	28.4	26.3	25.7	25.0	25.2	25.0

Table 10: Percentage of traffic through the loopback interface(Native) and percentage of traffic through eth0 interface(Foreign) as a function of peers per node on `cln008`, corresponding to the figure 16(b)

rarest-first. The purpose is to make a peer attractive to the others by requesting the rarest pieces first in the swarm, and quickly turn a peer into a productive member of the swarm.

Peers make the decision on which piece they consider to be the rarest based on locally available information from other peers. (This is why in some cases BitTorrent's piece selection algorithm is called local rarest first.) Peers obtain information about the pieces other peers possess through BitTorrent's HAVE-control messages. A peer sends a HAVE-message to its buddies when it has completed the download of a piece, in order to let its buddies know that they can download the piece from the peer. Peers keep track of the HAVE-messages and use them to calculate which pieces are the rarest among their buddies.

BitTorrent's control messages (of which HAVE is one) have to share the network with the actual data transfers. When the network (or loopback device) becomes congested, both the data and control messages are slowed down. At the 60 peers per node point, the network between the nodes starts becoming congested, but the loopback is still far below its capacity. Hence, peers receive a lot of HAVE-messages from the native buddies but the HAVE-messages from foreign buddies slow down. As a result of this, the peer (correctly) considers the pieces from the foreign buddies to be rarer than native pieces (which spread very fast within the node to many peers) and wants to request the rarer pieces from the foreign buddies first. As the network is only approaching the saturation point and is not yet completely congested, the

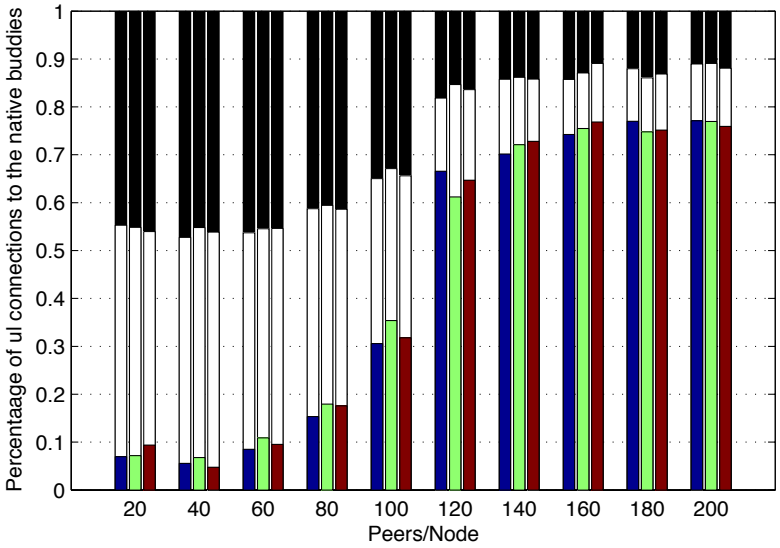
peer is able to provide uploads to foreign buddies so that they are willing to upload pieces to it (recall the use of tit-for-tat).

From the results, we can conclude that BitTorrent's piece selection algorithm is very sensitive to changes in network conditions in the download-constrained cases. In fact, piece selection strategy overrules peer selection strategy in the early part of the experiment. As the network gets more and more congested, the peers are no longer able to provide good enough upload rates to foreign buddies, so in accordance to the tit-for-tat policy, they are choked. Hence they have to resort to the native buddies for actually getting the data. Since there are no limits on upload rate, the actual injection of new information is limited by the seed's upload rate (which was limited), but the native buddies are enough to feed new data within the node. Eventually, we see the same kind of clustering between peers on a single node as we saw in the upload-constrained case.

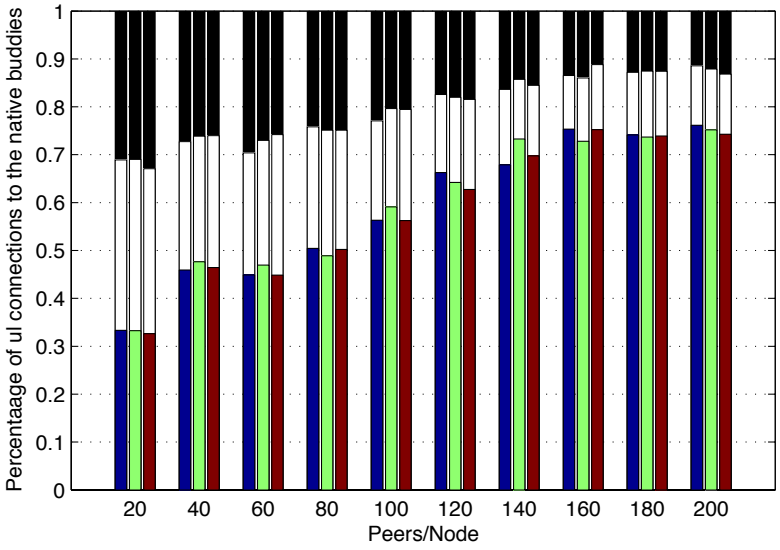
To verify our claim that the behavior above is due to the piece selection algorithm, we repeated the experiment with a random piece selection algorithm. Because peers exhibit no preference for pieces, peer selection algorithm should be the deciding factor. Results are shown in Figure 16(b). The results are similar to the upload-constrained case in Figure 15(b) where peer selection is known to be the deciding factor.

Table 10 shows the traffic distribution on `cln018` corresponding to figure 16(b). Compared with the traffic distribution in table 9, we can find another interesting thing. Even though the figure 16(a) shows the peers with *rarest first* piece selection algorithm are more clustered(higher ration of connection the native buddies) than those with *random* algorithm in figure 16(b), the ratio of traffic within the node (Native row) in table 9 is lower that in table 10.

As further evidence, we ran the download-constrained experiment with leechers placed equally on three nodes and the fraction of connections to other nodes is shown in Figure 17(a). The three parallel bars represent the three nodes. The lowest section of each bar shows native connections and the two upper sections show connections to the two other nodes. We see the same preference for foreign buddies in the beginning, with connections between the other two nodes being rather uniformly split, as is to be expected. After the network gets congested, we see the same kind of clustering as in the case with two nodes.



(a) Fraction of upload connections to native buddies (lowest section of bars) and foreign buddies (white and black sections) with 3 nodes in a **download-constrained** experiment.



(b) Fraction of upload connections to native buddies (lowest section of bars) and foreign buddies (white and black sections) with 3 nodes in a **upload-constrained** experiment.

Figure 17: Track connections in upload-constrained and download-constrained experiments, 3 nodes

7.3 Example: Case of 2 Nodes

We revisit the case of using two nodes in an experiment shown in Figure 14. In the experiment, we obtained an average download rate of 4.25 MB/s and loopback

capacity $L_i = 500\ MB/s$. The network between the nodes is a Gigabit Ethernet, so $C_i^{ul} = 125MB/s$ and $C_i^{dl} = 125MB/s$. $(i \in \{1, 2\})$

When there are 40 peers on each of the two nodes, we get the traffic distribution matrix T^{40} as below:

$$T^{40} = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} = \begin{bmatrix} 83.9 & 86.1 \\ 86.1 & 83.9 \end{bmatrix} \tag{7.1}$$

We can see from the equation (7.1), for node 1, $T_{12} \leqslant C_1^{ul}$, $T_{21} \leqslant C_1^{dl}$ and $T_{11} \leqslant \frac{L_1}{2}$. The same applies to node 2. We can see all the equations hold, the experiments are designed within the system capacity.

When there are 60 peers on each of the two nodes, we obtain the traffic distribution matrix T^{60} as below:

$$T^{60} = \begin{bmatrix} T_{11} & T_{12} \\ T_{21} & T_{22} \end{bmatrix} = \begin{bmatrix} 126.4 & 128.6 \\ 128.6 & 126.4 \end{bmatrix} \tag{7.2}$$

We can see from the equation (7.2), for node 1, $T_{12} > C_1^{ul}$ and $T_{21} > C_1^{dl}$. Both equations (6.5) and (6.6) are violated. Since equation (6.10) still holds, then in a upload-constrained experiment, a peer will not treat native buddies and foreign buddies equally. They start showing preference in uploading to native buddies, and the clustering happens. The same analysis can be applied to node 2. This analysis yields the same result as the investigation on the actual behavior of BitTorrent above.

7.4 Example: Case of 3 Nodes

In this section, we give another example to show the capacity planning for the experiments shown in Figure 17(b). As in section **??**, we use average download rate of 4.25 MB/s and loopback capacity $L_i = 500$ MB/s in the calculation. And $C_i^{ul} = 125MB/s$, $C_i^{dl} = 125MB/s$. $(i \in \{1, 2\})$

Since we deployed peers on 3 nodes equally, if a peer show no preference between native buddies and foreign buddies, the upload connections to the native buddies should be around 33.3%.

When there are 20 peers on each of the two nodes, we get the traffic distribution matrix T^{20} as below:

$$T^{20} = \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{bmatrix} = \begin{bmatrix} 27.4 & 28.8 & 28.8 \\ 28.8 & 27.4 & 28.8 \\ 28.8 & 28.8 & 27.4 \end{bmatrix} \tag{7.3}$$

We can see from the equation (7.3), for node 1, $T_{12} + T_{13} \leqslant C_1^{ul}$, $T_{21} + T_{31} \leqslant C_1^{dl}$ and $T_{11} \leqslant \frac{L_1}{2}$. The same applies to node 2 and node 3. All the capacity planning formulas hold, the experiments are designed within the system capacity. The results can be verified in figure 17(a), at 20 peers/node, the upload connections to native buddies are about 33% on each node.

When there are 60 peers on each of the two nodes, we obtain the traffic distribution matrix T^{60} as below:

$$T^{40} = \begin{bmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{bmatrix} = \begin{bmatrix} 84.1 & 85.5 & 85.5 \\ 85.5 & 84.1 & 85.5 \\ 85.5 & 85.5 & 84.1 \end{bmatrix} \tag{7.4}$$

We can see from the equation (7.4), for node 1, $T_{12}+T_{13} > C_1^{ul}$ and $T_{21}+T_{31} > C_1^{dl}$. Both equations (6.5) and (6.6) are violated. The peers start showing preference in uploading to native buddies, and the clustering happens. The same analysis can be applied to node 2 and node 3.

We must point out a subtle issue here. When there are 40 peers on each of the three nodes, the calculation shows experiments should be within the system capacity, but very near to it. What's more, the actual network bandwidth is usually smaller than our assumption (125 MB/s), which the theoretical bandwidth for a Gigbit Ethernet. So the actual workload from the experiments has already exceeded the system capacity limit, which can be verified in the figure 17(a).

The important lesson we learned is an experimenter should be very cautious when the experiments are performed near the system capacity limit. Even the calculations show the experiments are reasonable, the actual workload may have exceeded the capacity limit because of some over-optimistic assumptions.

7.5 Experiment conclusion

Piece selection heavily relies on the control messages(HAVE message), while the peer selection relies on data messages. That's why piece selection is more sensitive to the network changes and states. When the network is saturated, piece selection

senses the change in the number of HAVE messages, thus changes the BitTorrent's behaviours greatly. At the same time, even though the number of data messages also decreases, the decision of peer selection strategy is made upon the total amount of data received within a certain period. A little change in the number of data message won't change the amount of data greatly. So the peer selection will not react as drastically as piece selection, and will take effects slowly and slowly.

8 Conclusion

Experimental evaluation of large scale systems is an important topic in networking research. Currently no ideal environment exists for such evaluations, with simulations, real Internet, and cluster-based testbeds being the commonly used solutions. In this thesis, we gave our understanding on these solutions and the suitable situations to apply them. We believe that cluster-based testbeds offer the best of both worlds, realistic applications with a real (albeit not necessarily realistic) network in between.

In this thesis we have shown how to design BitTorrent experiments on a cluster. Our focus has been on identifying how the physical limits of the host machine affect the tests and how many clients can be deployed on a node. We have shown that the number of peers per node depends on many factors, but up to 500 peers per node is realistic for certain values of allocated per-client bandwidth. We have shown that the simple metric of average download rate is not sufficient for determining when an experiment is 'safe', but that a more complex analysis is needed. We provide a simple set of formulas, intended to be used as rules of thumb for determining if an experiment runs into the physical limits of the machine.

Our work has also extended previous work on BitTorrent, by showing that the previously observed clustering behavior is actually a result of both the peer and piece selection algorithms, and not simply the peer selection algorithm as previously believed. Although the effect of the piece selection algorithm is small, it cannot be ignored in all cases. By performing the pairwise experiments with different piece selection strategies and comparing the experiment data, we are able to show how piece selection strategy influence clustering property even it is marginal influence.

In our future work, we plan to verify our results using a 10 Gbps network between the nodes. This is likely to change some of the details of our results, since in that case the loopback will saturate before the network; hence the clustering behavior

will be different. Furthermore, we will also start our study in other P2P systems to accumulate enough experience in evaluating various distributed systems. Our research focus in future will be developing a systematic methodology for testing and evaluating large-scale distributed systems.

References

ABJM04 Antoniu, G., Bougé, L., Jan, M. and Monnet, S., Going Large-scale in P2P Experiments Using the JXTA Distributed Framework. Research Report RR-5151, INRIA, 2004. URL `http://hal.inria.fr/inria-00071432/en/`.

AH00 Adar, E. and Huberman, B. A., Free riding on gnutella. *First Monday*, 5,10(2000).

AO04 Amaral, L. and Ottino, J., Complex networks: Augmenting the framework for the study of complex systems. *The European Physical Journal B-Condensed Matter*, 38,2(2004), pages 147–162.

BHP06 Bharambe, A. R., Herley, C. and Padmanabhan, V. N., Analyzing and improving a bittorrent networks performance mechanisms. *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, april 2006, pages 1 –12.

BRF04 Biersack, E. W., Rodriguez, P. and Felber, P., Performance analysis of peer-to-peer networks for file distribution. *In Proc. Fifth International Workshop on Quality of Future Internet Services (QofISâ04)*, 2004.

CB10 Choffnes, D. R. and Bustamante, F. E., Pitfalls for testbed evaluations of internet systems. *SIGCOMM Comput. Commun. Rev.*, 40,2(2010), pages 43–50.

Coh03 Cohen, B., Incentives build robustness in bittorrent, 2003.

edg09 EdgeScope Project – Sharing the view from a distributed Internet telescope, 2009. URL `http://tinyurl.com/2c7ykgn`.

Emu10 Emulab Website, 2010. URL `http://www.emulab.net/`.

Fan05 Fang, Y., Modeling and performance analysis for wireless mobile networks: a new analytical approach. *IEEE/ACM Trans. Netw.*, 13,5(2005), pages 989–1002.

GCX+07 Guo, L., Chen, S., Xiao, Z., Tan, E., Ding, X. and Zhang, X., A performance study of bittorrent-like peer-to-peer systems. *Selected Areas in Communications, IEEE Journal on*, 25,1(2007), pages 155 –169.

GK99 Goldenfeld, N. and Kadanoff, L. P., Simple Lessons from Complexity. *Science*, 284,5411(1999), pages 87–89. URL `http://www.sciencemag.org/cgi/content/abstract/284/5411/87`.

gri10 Grid5000 Website, 2010. URL `https://www.grid5000.fr/`.

Har68 Hardin, G., The tragedy of the commons. *Science*, xx, pages 1243–47.

IUKB+04 Izal, M., Uroy-Keller, G., Biersack, E., Felber, P. A., Hamra, A. A. and Garces-Erice, L., Dissecting bittorrent: Five months in torrent's lifetime. 2004, pages 1–11.

Kan09 Kangasharju, J., Freeriding not (always) considered harmful. *Information Networking, 2009. ICOIN 2009. International Conference on*, 21-24 2009, pages 1 –5.

KR06 Kumar, R. and Ross, K., Peer-assisted file distribution: The minimum distribution time. nov. 2006, pages 1 –11.

LFG08 Laverell, W. D., Fei, Z. and Griffioen, J. N., Isn't it time you had an emulab? *ACM SIGCSE Bulletin*, volume 40, 2008, page 246.

LLKZ07 Legout, A., Liogkas, N., Kohler, E. and Zhang, L., Clustering and sharing incentives in bittorrent systems. *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, New York, NY, USA, 2007, ACM, pages 301–312.

LM99 Liberzon, D. and Morse, A., Basic problems in stability and design of switched systems. *Control Systems Magazine, IEEE*, 19,5(1999), pages 59 –70.

LUKM05 Legout, A., Urvoy Keller, G. and Michiardi, P., Understanding BitTorrent: An Experimental Perspective. Technical Report, 2005. URL `http://hal.inria.fr/inria-00000156/en/`.

LUKM06 Legout, A., Urvoy-Keller, G. and Michiardi, P., Rarest first and choke algorithms are enough. *IMC '06: Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, New York, NY, USA, 2006, ACM, pages 203–216.

MH09 Minyi, K. and Huazhong, J., Cluster-based job management system research oriented to spatial data processing. may. 2009, pages 1 –4.

MPES09 Meulpolder, M., Pouwelse, J., Epema, D. and Sips, H., Modeling and analysis of bandwidth-inhomogeneous swarms in bittorrent. sep. 2009, pages 232 –241.

oec Oecd – organisation for economic co-operation and development. URL `http://www.oecd.org`.

PACR03 Peterson, L., Anderson, T., Culler, D. and Roscoe, T., A blueprint for introducing disruptive technology into the internet. *SIGCOMM Comput. Commun. Rev.*, 33,1(2003), pages 59–64.

Pla10 PlanetLab Website, 2010. URL `http://www.planet-lab.org/`.

QS04 Qiu, D. and Srikant, R., Modeling and performance analysis of bittorrent-like peer-to-peer networks. *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, 2004, ACM, pages 367–378.

RLD10 Rao, A., Legout, A. and Dabbous, W., Can Realistic BitTorrent Experiments Be Performed on Clusters? *Peer-to-Peer Computing (P2P), 2010 IEEE Tenth International Conference on*. IEEE, 2010, pages 1–10.

RR07 Rasti, A. and Rejaie, R., Understanding peer-level performance in bittorrent: A measurement study. aug. 2007, pages 109 –114.

SHRY07 Sirivianos, M., Han, J., Rex, P. and Yang, C. X., Free-riding in bittorrent networks with the large view exploit. *In IPTPS â07*, 2007.

SW05 Steinmetz, R. and Wehrle, K., *Peer-to-Peer systems and applications*. Springer, Berlin, cop. 2005.

VY03 Veciana, G. D. and Yang, X., Fairness, incentives and performance in peer-to-peer networks. *In the Forty-first Annual Allerton Conference on Communication, Control and Computing*, 2003.

ZIea10 Zhang, B., Iosup, A. and et al, Sampling bias in bittorrent measurements. Euro-Par 2010, Ischia, Italy, 2010.

A Terminology

Mainline Ver4/Ver5: Official BitTorrent implementation. All of our discussions and terms used are based on *Mainline*. *Ver4* stands for *Mainline Version 4*, and *Ver5* stands for *Mainline Version 5*.

metainfo: Meta information about the distributed content, such as creation time, announce url, file-name list and so on. metainfo is Bencoded and stored in Metainfo file, which is widely known as ".torrent" file. We use "metainfo file" and "torrent file" interchangeably in this paper.

batch torrent: If a distributor includes multiple files into distribution content, the corresponding torrent file is called *batch torrent*. A *batch torrent* contains the corresponding file-list and all the meta-information for each file.

sharing content: Since multiple files can be distributed with a batch-torrent, and in BitTorrent system, there is no difference in distributing single or multiple files in one swarm. We don't care how many files are involved in one torrent, so we use the term *sharing content* for both cases in this paper.

piece: BitTorrent divides the distributed content into pieces. The size of a piece should be power of 2 and varies from 512KB to 1MB. Every piece has corresponding hash_info in the metafile. In chunk-based model, a piece is also called a chunk. We use them interchangeably in this paper.

slice: To make the distribution efficient, a piece is further divided into slices. The data exchange among peers are based on the slices. The usual value is 16KB or 32KB. Small slice size will causes high overhead, while large size will introduce long latency. Usually, value below 128KB are recommended.

bitfield: bitfield is the data structure used in BitTorrent, indicating what pieces a peer have currently. bitfield is just a simple bitmap, in which every bit stands for a piece of the distributed content. "0" means missing and "1" means having, and the spare bits in the last byte are set to "0".

buddy: In this paper, we call two connected peers buddies. Buddy relation is symmetric, but not reflexive or transitive.

native buddies: peers on the same physical node are referred as native peers; if they are buddies, then they are referred as native buddies.

foreign buddies: peers on the different physical nodes are referred as foreign peers; if they are buddies, then they are referred as foreign buddies.