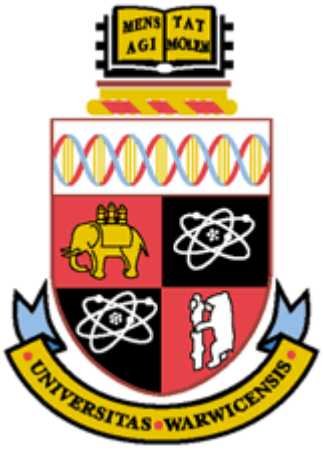


Parallelisation for Data-Intensive Applications over Peer-to-Peer Networks

by

Xinuo Chen

Thesis

Submitted to the University of Warwick

In partial fulfilment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

September 2009

THE UNIVERSITY OF WARWICK

Contents

List of Tables

List of Figures

Acknowledgements

I would like to express my most sincere thanks to my supervisor, Professor Stephen Jarvis, for offering me a great environment, unlimited support and pertinent suggestions. Without his continual guidance, this work would not have been possible.

I would also like to thank Dr. Guang Tan. He is one of my best friends and one of the kindest persons I ever met in my life. He gives me many valuable advices on my research and much help even in my personal life. Without the vivid discussions we had on the research, the process of finishing this work would have been much less fun. Another important PhD mate who I cannot forget is Dr. Wen Jun James Xue. He encouraged me a lot in my studies.

This PhD study is the best time in my life. Thanks to the members and associates of the High Performance Systems Group at University of Warwick both past and present, including Dr. Ligang He, Dr. Daniel Spooner, Lei Zhao, Paul Isitt, Brian Foley, and Gihan Mudalige.

Thanks to my parents. To them, I am always a little boy. They give me their unlimited love to support me in all the years. No matter how busy they were, they always prepared the best food for me to take the cooking burden off my shoulder so that I have more dedicated time on this work.

Last but definitely not least, thanks to my beautiful and lovely girlfriend Shuangyan Liu. She accompanies me in the last year of this work. The finalising process of this work was hard, but she stands behind me and encourages me not to stop progressing. She brings tremendous fun to my life and I will not forget any slice of memory of being with her.

Declarations

This thesis is presented in accordance with the University of Warwick regulations for the degree of Doctor of Philosophy. It has been written by myself and has not been submitted in any previous applications for any degrees. This work described in the thesis has been undertaken by myself except where otherwise stated.

During this period of research, I have contributed to the following publications:

X Chen, SA Jarvis, Shuangyan Liu, "ppBLAST: A Computational Service over Peer-to-Peer network for BLAST", International Conference on Advances in P2P Systems (AP2PS 2009), Sliema, Malta, October 11-16, 2009. [4]

X Chen, SA Jarvis, "Analysing BitTorrent's Seeding Strategies", 7th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC-09), Vancouver, Canada, August 29-31, 2009. **BEST Paper Award.** [5]

X Chen, SA Jarvis, "Design and Implementation of Efficient Range Query over DHT Services", International Conference on Signal Processing and Communication Systems (ICSPCS 2007), Australia, Gold Coast, 17-19 December 2007. [6]

X Chen, SA Jarvis, "Distributed Arbitrary Segment Tree: Efficient Range Query Over Public DHT Services", 12th IEEE International Workshop on Computer Aided Modelling and Design of Communication Links and Networks (CAMAD 07), held as part of the 18th IEEE International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC 2007), Athens, Greece, September, 2007. [7]

X Chen, SA Jarvis, "Analysing Seeding Strategies and Fairness in BitTorrent-based Networks", 22nd Annual UK Performance Engineering Workshop (UK-PEW06), Bournemouth, UK, July 6-7, 2006. [8]

X Chen, SA Jarvis, G Tan, L He, DP Spooner, GR Nudd, “An implementation of BLAST over peer-to-peer and its performance validation through simulation”, 8th International Conference on Computer Modelling and Simulation, Oxford, UK, 6-8 April 2005. [9]

G Tan, SA Jarvis, X Chen, DP Spooner, GR Nudd, “Performance Analysis and Improvement of Overlay Construction for Peer-to-Peer Live Media Streaming”, *Simulation: Transactions of the Society for Modeling and Simulation*, **82**:93-106, Feb 2006. [10]

G Tan, SA Jarvis, X Chen, DP Spooner, GR Nudd, “Performance Analysis and Improvement of Overlay Construction for Peer-to-Peer Live Media Streaming”, 13th IEEE Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), Sept. 2005. [11]

G Tan, SA Jarvis, L He, X Chen, DP Spooner, GR Nudd, “Modelling Web transfer Performance over Asymmetric Networks”, 1st International Workshop on Performance Modelling in Wired, Wireless, Mobile Networking and Computing (PMWMNC-2005), 11th IEEE International Conference on Parallel and Distributed Systems (ICPADS'05), Fukuoka Institute of Technology, Japan, 20-22 July 2005. [12]

L He, SA Jarvis, DP Spooner, X Chen, GR Nudd, “Dynamic Scheduling of Parallel Jobs with QoS Demands in Multiclusters and Grids”, 5th IEEE/ACM International Workshop on Grid Computing (Grid2004), Pittsburgh, USA, Nov 8, 2004. [13]

L He, SA Jarvis, DP Spooner, X Chen, GR Nudd, “Hybrid Performance-based Workload Management for Multiclusters and Grids”, *IEE Proc.-Softw.*, 151(5):224-231, October 2004. [14]

L He, SA Jarvis, DP Spooner, X Chen, GR Nudd, “Dynamic, Hybrid Performance-oriented Scheduling of Moldable Jobs with QoS Demands in Multiclusters

and Grids", 3rd International Conference on Grid and Cooperative Computing (GCC 2004), Wuhan, China, October 2004. [15]

L He, SA Jarvis, D Bacigalupo, DP Spooner, X Chen, GR Nudd, "Queueing Network-based Optimisation Techniques for Workload Allocation in Clusters of Computers", IEEE International Conference on Services Computing (SCC 2004), Shanghai, China, September 15-18, 2004. IEEE Computer Society Press. [16]

L He, SA Jarvis, DP Spooner, X Chen, GR Nudd, "Hybrid performance-oriented optimisation mechanism for scheduling QoS-requesting parallel jobs in multi-clusters and grids", 20th Annual UK Performance Engineering Workshop (UK-PEW' 2004), University of Bradford, July 7-8 2004. [17]

Glossary of Terms

BLAST

A Bioinformatics tool set which is used to compare two DNA sequences.

Peer-to-Peer networks

The application level of overlays where computers over the Internet connect to each other to share resources such as files, storage, computing power, and so on.

BitTorrent

The most popular and effective Peer-to-Peer file sharing protocol which splits large files into pieces and distributes them among peers.

Seed

The peer that have a complete set of file pieces in BitTorrent networks. They upload to others without the need of downloading.

Leecher

The peer that downloads file pieces in the BitTorrent networks.

Selfish leecher

The leecher who wants to download but not to share.

Freerider

A type of selfish leecher who downloads many file pieces without sharing any with others.

Exploiter

A type of selfish leecher who shares file pieces while downloading, but leaves the network immediately after finishing.

Seeding strategy

The strategy which guides seeds to upload file pieces to leechers.

Tit-For-Tat

One of the key mechanisms in BitTorrent protocol. It forces leechers to upload.

BitTorrent Choking Algorithm

The file chunks are exchanged between peers. Every peer employs the Choking Algorithm to decide whom it will upload to.

Choking Slot

When a peer uses the choking algorithm to decide whom to upload, the choking slot is actually the number of other peers this peer will upload to.

SHA-1

Secure Hash Algorithm - 1. An algorithm is used for encryption.

Tracker

A server that stores the locations of all peers (in the BitTorrent overlay) and provides every peer with the locations of others.

Homogeneous and Heterogeneous network environments

In homogeneous network environment, every peer is assumed to have the same network bandwidth. For heterogeneous network environment, the network bandwidths of peers are different.

DHT

Distributed Hashtable. A Peer-to-Peer overlay that provides lookup and storage functions.

Traditional Segment Tree

A traditional segment tree is a binary tree and the value of every parent is split and assigned to its two children.

Bootstrap Peer

A peer whose job is to give the locations of other peer to the peers who wish to join the peer-to-peer overlay.

Abstract

In Data Intensive Computing, properties of the data that are the input for an application decide running performance in most cases. Those properties include the size of the data, the relationships inside data, and so forth. There is a class of data intensive applications (BLAST, SETI@home, Folding@Home and so on so forth) whose performances solely depend on the amount of input data. Another important characteristic of those applications is that the input data can be split into units and these units are not related to each other during the runs of the applications. This characteristic helps this class of data intensive applications to be parallelised in the way where the input data is split into units and application runs on different computer nodes for certain portion of the units. SETI@home and Folding@Home have been successfully parallelised over peer-to-peer networks. However, they suffer from the problems of single point of failure and poor scalability. In order to solve these problems, we choose BLAST as our example data intensive applications and parallelise BLAST over a fully distributed peer-to-peer network.

BLAST is a popular bioinformatics toolset which can be used to compare two DNA sequences. The major usage of BLAST is searching a query of sequences inside a database for their similarities so as to identify whether they are new. When comparing single pair of sequences, BLAST is efficient. However, due to growing size of the databases, executing BLAST jobs locally produces prohibitively poor performance. Thus, methods for parallelising BLAST are sought.

Traditional BLAST parallelisation approaches are all based on clusters. Clusters employ a number of computing nodes and high bandwidth interlinks between nodes. Cluster-based BLAST exhibits higher performance; nevertheless, clusters suffer from limited resources and scalability problems. Clusters are ex-

pensive, prohibitively so when the growth of the sequence database are taken into account. It involves high cost and complication when increasing the number of nodes to adapt to the growth of BLAST databases. Hence a Peer-to-Peer-based BLAST service is required.

This thesis demonstrates our parallelisation of BLAST over Peer-to-Peer networks (termed *ppBLAST*), which utilises the free storage and computing resources in the Peer-to-Peer networks to complete BLAST jobs in parallel. In order to achieve the goal, we build three layers in ppBLAST each of which is responsible for particular functions. The bottom layer is a DHT infrastructure with the support of range queries. It provides efficient range-based lookup service and storage for BLAST tasks. The middle layer is the BitTorrent-based database distribution. The upper layer is the core of ppBLAST which schedules and dispatches task to peers. For each layer, we conduct comprehensive research and the achievements are presented in this thesis.

For the DHT layer, we design and implement our DAST-DHT. We analyse balancing, maximum number of children and the accuracy of the range query. We also compare the DAST with other range query methodology and state that if the number of children is adjusted to more two, the performance of DAST overcomes others. For the BitTorrent-like database distribution layer, we investigate the relationship between the seeding strategies and the selfish leechers (freeriders and exploiters). We conclude that OSS works better than TSS in a normal situation.

Chapter 1

Introduction

1.1 Overview of BLAST

The Basic Local Alignment Search Tool (*BLAST*) is a tool set which is used throughout Bioinformatics to compare two nucleotide or protein sequences. Normally, all sequences that have already been identified are categorised and packed into databases, while newly discovered sequences can be encapsulated into queries. BLAST compares each sequence in a query with each sequence in a database. For each sequence in the query, the statistical similarity that is derived from all comparisons is calculated in order to decide whether it is novel or not. If novel sequences are indicated by the results, they will be added into the existing databases and, therefore, the size of the existing databases will grow.

The current most popular version of BLAST is *NCBI-BLAST*, which is implemented by the National Centre for Biotechnology Information (*NCBI*) [18]. NCBI-BLAST consists of five programs: *blastn*, *blastp*, *blastx*, *tblastn* and *tblastx*

Table 1-1 BLAST programs and their functions

BLAST program	Function
blastn	Compare nucleotide with nucleotide
blastp	Compare protein with protein
blastx	Compare translated nucleotide with protein
tblastn	Compare translated protein with nucleotide
tblastx	Compare translated nucleotide with translated protein

(their functions are summarised in Table 1-1). Each program can be executed through the command line. The parameters for the commands of the five programs are similar; the two major parameters express the query and an associated database, both of which contain a number of sequences. More details about NCBI-BLAST and BLAST are presented in Chapter 2. In this chapter, for simplicity, we only use the term *BLAST* and the terminology for a command line execution of a BLAST comparison is expressed as a "*blast query database*".

BLAST provides high efficiency algorithms to compare two sequences; however, for the daily tasks of discovering new nucleotide or protein sequences, the usage of BLAST is restricted by the exponential growth in the databases [19]. As described above, every officially identified novel sequence will be added to the existing databases, e.g., GenBank [20]. In order to identify one newly discovered sequence, each of the existing sequences in the databases has to be compared with it using BLAST. Hence, along with the growth of the databases, more sequences will join the comparison queue for one query sequence, and the search time of a single BLAST job also increases exponentially. To illustrate the current performance of a typical BLAST job, we present an example. The current size of the nucleotide sequence database (*nt* database from GenBank in FASTA format) is 7.29GB. For a query of sequences, whose size is 1MB, if we compare it with the *nt* database using BLAST on a PC with 2GB memory and an Intel Core2 Duo 2.4GHz CPU, it will take approximately 8 hours and 34 minutes. This is the current status, and clearly the performance will deteriorate yet further as the databases associated with BLAST continue to grow.

In order to reduce this performance impact, a number of advances (research and / or industry oriented) have emerged to parallelise BLAST[1]. While some of these propose special hardware to directly accelerate BLAST comparisons [21, 22], most focus on software level parallelisation, which splits a job into a number of tasks and distributes them among nodes in a cluster [23-25].

[1] More details of these BLAST parallelisation approaches are presented in Chapter 5.

BLAST has the following characteristics that make it possible to split a BLAST job into tasks at a software level. First, one unit of comparison in BLAST is between two single sequences. Although BLAST accepts a query or database that contains more than one sequence, the actual operation that BLAST conducts is to compare each pair of sequences (one sequence from the query and the other from the database) until all comparison units are finished. For example, if a query has three sequences and a database has five sequences, a blast query on the database will execute 15 comparisons one by one. Moreover, all comparisons are independent to each other, which means that one comparison operation will not affect another. Thus, if one splits the database into several parts, each of which contains a number of sequences, a query and one portion of the database can be then combined to create a task. When those portions of the database are distributed among nodes in a cluster, a BLAST job can then be completed in parallel by all nodes.

The feasibility of splitting BLAST jobs and utilising clusters to improve BLAST performance has been proven by [9, 19, 23, 25-29]. However, the cluster-based parallelising approaches also face problems of scalability and resource limitations. While the sizes of the existing databases grow exponentially [20], more nodes have to be continuously added into a cluster to match the growth in speed required in order to satisfy the demand for BLAST performance. This has already been shown to involve tremendous cost and an increased complexity in the maintenance and administration of these supporting clusters. Another burden on the scalability of cluster-based BLAST is the number of BLAST job requests from the users. If the number of job requests increases, more nodes are needed in a cluster to satisfy the requests of the users. For example, Darling et al. [24] claims that their cluster-based approach of parallelising BLAST can obtain near linear speedup for one BLAST job. Nevertheless, if an increasing number of BLAST jobs are submitted to the cluster simultaneously, which is commonly the case, the performance will unlikely reach the optimal degree.

In order to tackle the scalability problems in parallelising BLAST and also adapt to the growth of the databases and the potentially large number of BLAST

job requests, we investigate another type of available and usable resources – Peer-to-Peer networks. The Peer-to-Peer technology has become popular since the success of Napster [30] in 1999. Traditional approaches to sharing music files were based on the server / client Internet model. If a user wants to share his own music files, he has to upload the files onto the server. The server was responsible for hosting the music files and all clients downloaded from the server. This approach suffers from the scalability problem: if a large number of clients try to download music files simultaneously, the server has to either put limits on the number of connections or stop the service directly. In either case, the result is that the clients cannot obtain satisfactory service. Napster solved the problem by maintaining an index server which did not host any files. If a client has a file to share with others, it connects to the server and informs it which file it wants to share and what is the IP:Port it provides. The server will then index this information. When another client wants to download the file, the server will tells it who has the file and the client will try to connect to the file owner's computer and download from it directly. In other words, downloading and uploading files will occur between clients, not between clients and servers. This solves the scalability problem for the server.

After the birth of Napster, many Peer-to-Peer applications emerged covering areas such as file-sharing [2, 31], storage [32-34], online video streaming [35], information routing [36], processing power sharing [37-39] and so forth. Their achievements have shown the availability and scalability of Peer-to-Peer resources. The projects on SETI@Home claim that over millions of peers have contributed their resources since the projects started [40]. Their successes demonstrate the richness of the Peer-to-Peer resources and peers are willing to share not only their storage space but also the CPU processing power. Therefore, inspired by the model of SETI@Home, we have targeted Peer-to-Peer networks as the new computing resource provider for BLAST and have carried out extensive research on utilising scalable Peer-to-Peer networks in parallelising BLAST.

Parallelising BLAST over Peer-to-Peer networks has the following features that need to be taken into consideration:

- *Network Bandwidth*

  In a cluster environment, the network links between nodes normally have high bandwidths. In a cluster, there is a master node that assigns tasks to worker nodes and hosts the database fragments. If a worker node needs a fraction of a database at a certain point in time to finish a task, it can download it from the master node directly (and immediately) without considering the performance impact that will be brought about from the transfer delay [24]. This is the so-called “on demand”.

  In a Peer-to-Peer network overlay, however, this is hard to achieve. Most peers are normal home PCs which are connected to the Internet through home broadband. Their download and upload bandwidths are much more limited compared to cluster interlinks. If we adapt the “on demand” design to the database distributions in a Peer-to-Peer network, the time spent on the comparatively low transfers will decrease the overall performance significantly.

  Therefore a more efficient file distribution approach needs to be investigated and employed for Peer-to-Peer BLAST in order to save the time cost in distributing databases among peers.

- *Self-organising (has advantage)*

  Cluster-based BLAST parallelisation is in a centrally controlled paradigm. The master node hosts database fragments and is responsible for organising all worker nodes to cooperate with each other to finish a BLAST job. This makes the master node the central point of failure. Without a properly functioning master node, the BLAST service cannot operate.

  For Peer-to-Peer BLAST, however, we design it in a loose and self-organising paradigm in order to avoid the dependency of single point.

In addition, the self-organising mechanism will also remove the central control from Peer-to-Peer BLAST and further increase the scalability of the system.

- *Storage*

  Since there will be no master element in Peer-to-Peer BLAST, one remaining question is where the queries or results should be stored after users submit their BLAST jobs. As users will not wait for all worker peers to obtain the queries or return results, queries or results need to be temporarily stored in the overlay for worker nodes or users to retrieve them later.

- *Self-scheduling*

  When a job is submitted to the Peer-to-Peer BLAST overlay, it is automatically split into a number of tasks and this number depends on how many fragments the database has. As previously described, all tasks are independent. However, each task will produce only one result file and all result files have to be finally collected for a complete BLAST job. Therefore, a job may be delayed due to a small number of incomplete tasks. This represents a task scheduling problem. Another scheduling issue is that peers have different network bandwidths and thus at a certain point in time, peers have a different distribution of database fragments. If one peer does not have database fragment A, it cannot finish task A, which means that not every peer can finish every task. One peer may have a large number of fragments while another may only have few. The working abilities of these two peers are then different. Note that there is no master-like element in the overlay to centrally schedule the tasks, how to self-schedule the tasks to achieve BLAST jobs efficiently must also be investigated.

1.2 Thesis Contributions

Through solving the above problems, we design a Peer-to-Peer BLAST service, *ppBLAST*, where volunteer peers (which also can be BLAST users) construct an overlay, receive BLAST job requests from users, and return the results. In this thesis, we demonstrate the achievements of this research on BLAST parallelisation over a Peer-to-Peer network. The three specific contributions of this thesis are as follows.

The first contribution is the analysis of BitTorrent seeding strategies. In order to boost the database distributions over the Peer-to-Peer network, we employ the efficient Peer-to-Peer BitTorrent file sharing protocol. Although the efficiency of BitTorrent is one of its key advantages, the seeding strategy of the protocol has been largely overlooked. In order to obtain the optimum performance from BitTorrent to aid the database distributions for ppBLAST, we carry out comprehensive analysis dedicated to BitTorrent seeding strategies. This contribution is presented in Chapter 3.

The second contribution is in the design and implementation of efficient range queries over Distributed Hash Table (*DHT*) services. The DHT is based on a Peer-to-Peer overlay to provide a lookup service similar to a hash table. {*key*, *value*} pairs can be inserted into DHT and the *value* be retrieved through the *key*. DHT can be used as a light-weight information storage and retrieval system. In addition, DHTs are purely distributed and self-organised. These characteristics of *DHT* fit to the requirements of ppBLAST and Peers in ppBLAST can be self-organised through the DHT mechanism. However, normal DHTs can only support single key retrieval, which means that only one key can be queried at a time. If one wants to retrieve the values for a series of keys, the query operations have to be carried out for each key one by one. This significantly increases the overheads of query operations and decreases ppBLAST performance because there are always many pairs of {task, result} to be retrieved for a BLAST job. Therefore, we improve DHT by adding a range query layer so that the values for a range of keys

can be retrieved through one query operation. This contribution is presented in Chapter 4.

The final contribution is the ppBLAST service and its performance measurements. A BitTorrent-like file distribution with correct seeding strategy is employed in ppBLAST to boost the database distribution process. A DHT infrastructure with support for range queries is also added as a fundamental layer of ppBLAST to provide a self-organising ability, peer seeking and locating functionalities, as well as the storage of queries and results. We then design ppBLAST framework and self-scheduling algorithms using the two research contributions above.

The remainder of the thesis is organised as follows: Chapter 2 documents the background research and related work; Chapter 3 presents the analysis of BitTorrent seeding strategies and the changes required to support ppBLAST; Chapter 4 describes the range query support for DHTs and how this work has been extended for the ppBLAST design; Chapter 5 illustrates the design and implementation of ppBLAST and demonstrates its relative performance; Chapter 6 concludes the thesis and discusses future works.

Chapter 2

Background

In this chapter, we present the necessary background research for this thesis. Since this thesis covers three research areas – BLAST (data intensive computation), Peer-to-Peer file distribution and DHT - we split the background chapter into these three areas. It should be noted that each of the following chapters contains its own related work, and therefore we do not repeat this in this chapter.

In Section 2.1, we detail BLAST and its characteristics related to this thesis. In Section 2.2, we introduce Peer-to-Peer overlays, Peer-to-Peer file sharing, DHTs and Peer-to-Peer-based computation. The documentation that is more topic-related is addressed at the beginning of the associated chapters.

2.1 BLAST

BLAST, the **B**asic **L**ocal **A**lignment **S**earch **T**ool [41, 42] is mainly employed by bioinformatics area to compare two amino-acid sequences from different proteins or the nucleotides of DNA. The results are then used to find the similarity between the two sequences. For example, if a gene from an ape is obtained, one can compare it with each identified human genome to see whether humans carry a similar gene; alternatively it can be compared with all identified ape genomes to find out if the gene is new.

The original BLAST contains algorithms to efficiently carry out the comparison. The National Centre for Biotechnology Information (*NCBI*) [18] has developed five programs (listed in Table 1-1) and combined them into a toolset (*NCBI-BLAST*). NCBI also maintains a large library of databases to store all iden-

tified nucleotide and protein sequences. Every identified sequence is added to the library. Bioinformatics scientists, who are BLAST users, will compare newly obtained sequences against the ones in certain databases in the library through BLAST. Although these five programs of NCBI-BLAST are used to compare different types of sequences, their basic usages are similar. They accept a query and a database as the input, and produce the result to show the similarities between the query and database. Both query and database can have one or more sequences. If either of them has more than one sequence, each sequence will be compared against the target. For example, if a query has three sequences, each of them will be extracted and compared with the target database. If the target database has five sequences, then each possible pair of sequences (one from the query and the other from the database) will be compared. All five programs are executed through a command line in the form of "*blast –p program_name –i query_name –d database_name –o out_result*" with a number of other parameters. For simplicity, an execution of the BLAST program is termed a "*blast query database*".

Although some characteristics of BLAST have been introduced in Chapter 1, more detail is required in the content of this work. BLAST is both data- and computation-intensive. The time overhead that is spent on a BLAST job has two parts. One is the disk operation time which is spent on reading the sequences from the hard disk to memory and writing text-based results from memory into the disk. The other overhead is the computational comparisons for sequences. Simply comparing two single sequences using BLAST is very effective; however, once the number and the sizes of sequences (from either query or database) in a job are large, the execution time will become tremendous, as shown by the example in Chapter 1. However, the independent characteristics of BLAST provide a means for parallelisation.

Despite the number of sequences in the query or database, BLAST always extracts one sequence from a query and one from the database, and compares the pair. After one pair is finished, BLAST will carry on the comparison for another different pair until all possible combinations of pairs are finished. Most importantly, the comparisons of all pairs are independent. In other words, we can split

the sequence pairs, compare them separately and finally combine all results. Therefore, in order to parallelise a BLAST job, we can split the database into fragments and distribute them among nodes (in cluster-based parallelisation, the term "node" refers to cluster node; in our Peer-to-Peer approach, it refers to peer over the Internet). Each node will therefore have a number of fragments and can compare the query against its own database portions. In this way, a BLAST job can be executed in parallel and the overall performance and efficiency are improved. Another advantage of this parallelisation approach is that the core of BLAST does not need to be modified and the parallelisation is focused on BLAST jobs but not on the core of BLAST programs.

2.2 Peer-to-Peer Network

As introduced in Chapter 1, the popularity of Peer-to-Peer networks in both research and industry arises largely from the success of Napster [2, 30]. The term *Peer-to-Peer overlay* indicates that a number of nodes are interconnected through a particular distributed architecture, which is laid on top of the physical network links. In the overlay, peers share resources with each other and during the sharing, peers are both servers and clients; this is different from the traditional network sharing model where nodes are normally clients and obtain services only from servers.

2.2.1 Peer-to-Peer File Sharing

The most important application area for Peer-to-Peer networks is file sharing. Traditional file sharing approaches heavily rely on servers. Popular files are collected or produced by the hosting servers manually, or uploaded to the servers by clients. If a client wants to obtain a file, he has to connect to the hosting server and download it. The upload / download links exist only between a server and a client. The downloading rate is restricted by the server's output bandwidth or the client's download bandwidth. It is also affected by the number of users connected in the server. The output bandwidth of a server is always limited and it has to be

shared by all downloaders. In the worst case, the server may stop working if the number of simultaneous downloaders reaches a threshold because the capacity of the server is exceeded. Peer-to-Peer file sharing applications solve the problem by allowing peers to connect to each other without considerable central server activity. Peer can browse shared files of other peers or upload to / download from others. The bandwidths of the links between peers are then utilised more efficiently. In addition, the capacity of the whole system increases when more peers join the network, because the newly joined peers can supply more files / file chunks and contribute their bandwidths.

In June 1999, Napster emerged and started the popularity of Peer-to-Peer file sharing overlays. Gnutella [1], eDonkey2000 [43] and Freenet [44] were released in 2000. Gnutella [1], employs a search driven file sharing mechanism. Peers do not browse for files; instead, they query their neighbours for the files that they want to retrieve. If their neighbours do not have the files, queries will be sent to the next level of neighbours to further explore the existence of the requested files. The whole overlay is *unstructured*, i.e., there are no particular algorithms to decide who peers should be connected to and peers randomly make connections to others through bootstrap nodes. eDonkey2000 [43] also follows this *unstructured* design. However, it has a central indexing server to record which files peers share in the overlay and where they are located (IP:Port information of peers). This design is similar to Napster, which is called a *Hybrid Unstructured* architecture. Freenet [44] has put much effort on the anonymity of peers. Peers in Freenet do not browse or query for files and they simply choose whether to receive files from others. Sharing a file normally starts from one peer, and that peer will send files to neighbours. If the neighbours choose to receive them, they will start the next round of sending after finishing their retrieval. Every peer knows only its neighbours who send the file, but not the information on which peer initialises the sharing.

By 2002, the BitTorrent Peer-to-Peer protocol [31] dominated the file sharing arena. This protocol splits every large file into small chunks. When a peer initialises the sharing of a file, it builds a *torrent* file and publishes it. Peers who

wish to download this file will open the torrent file, retrieving the meta-information in it and starting the BitTorrent client. The client application will follow the records that reside in the torrent file and connect to the *tracker* server. The tracker server is the only centralised element in the BitTorrent overlay; however, its function is limited to only recording the IP:Port of peers who are currently download the file in the overlay. The key factor that leads to the BitTorrent success is the Tit-For-Tat mechanism (*TFT*). The TFT algorithm forces peers to upload to others whose uploading rates are the highest. The reason for this design is that if one is willing to upload, it then can download faster from others. Peers use TFT to decide which peer to upload chunks to and which to download from. BitTorrent is suitable for the sharing of large files and has been demonstrated to exhibit very high efficiency [3, 7, 8, 31, 45-56]. This is one of the reasons that we select BitTorrent as our database distribution layer.

2.2.2 Peer-to-Peer Computation

The most representative applications in Peer-to-Peer computation area are SETI@Home [37, 38, 40] and Folding@Home [39]. Although the initiatives of these two projects are different, the internal Peer-to-Peer architectures are similar. Thus, we describe only SETI@Home in this section.

The term *SETI* is an acronym for the **S**earch for **E**xtra-**T**errestrial **I**ntelligence. SETI@Home is the project that makes use of home PCs to analyse radio transmissions from outer space to search for possible evidence of alien life. Since the analysis work intends to be continuous and long term, this project builds on a Peer-to-Peer overlay and normal home PCs can participate in to contribute their processing power. The architecture of SETI@Home is straightforward, in that it employs a centralised server to coordinate the whole Peer-to-Peer computation process. If a home PC wants to contribute to the project, it can connect to the indexing server and become a peer through SETI@Home client application. The server then sends the peer computation task to compute. Once the peer finishes the task, it returns the result back to the server and a new round of task assigning starts. Peers can join or leave at any time without any restriction. The server just

iteratively locates any available peers and assigns them with tasks. There are no communications between peers and the whole running process of SETI@Home is driven purely by the server.

SETI@Home is enormously successful. Since SETI@Home was launched on 17th May, 1999, it has had over 5.2 million participants worldwide. On 26th September, 2001, it has performed a total of 10^{21} floating point operations, which is the largest computation in history. On the 14th September, 2009, SETI@Home had 278, 832 active peers concurrently in the overlay and 2.4 million peers having been in the system. The computing capacity of the SETI@Home Peer-to-Peer overlay is over 684 TeraFLOPS. This provides the inspiration for the initiative of utilising Peer-to-Peer networks to parallelise BLAST. ppBLAST does however adopt a different Peer-to-Peer architecture for BLAST-style computation jobs.

2.3 DHT

A *Distributed Hash Table* (DHT) is a Peer-to-Peer system that provides a lookup service similar to a traditional hash table. A *key* and a *value* are combined as a {*key*, *value*} pair and stored in participating peers through a DHT construction algorithm. Peers can submit a *key* to the overlay and retrieve its *value*. This functionality seems simple but provides many other services to Peer-to-Peer applications. For example, using the DHT it is possible to store the location information (IP:Port) of peers, so centralised indexing servers are no longer necessary [57]; DHTs can also act as online Peer-to-Peer storage systems, where disk storage of peers can be easily utilised.

Unlike ordinary Peer-to-Peer file sharing applications, DHT systems are normally *structured*. All peers join the overlay by following a design of construction so that various DHT properties, such as the number of replicas for {key, value} pairs, the list of peer's neighbours etc., can be maintained. Although various DHT systems exist [32, 36, 58-60], their construction algorithms are similar. In DHTs, when a peer joins the overlay through a bootstrap node (a node whose

IP address is known to every peer and is used as the entrance to the overlay), it will be assigned an *id* which is generated by hashing its IP:Port information. The peer will then query the bootstrap node for a list of peers whose ids are closest to its own. Once the list is obtained, it will connect to those peers and add them into its neighbour list.

When a {key, value} pair is to be stored into the DHT, the key will be hashed in the same hashing space as the peer id. The remaining question for the storage of the {key, value} pair is which peers should physically store it. The peer that accepts the request of storage will not physically store them. For example, if peer A asks peer B to store {key, value} pair, peer B does not necessarily need to store the pair on its own hard drive. Instead, it will query its neighbours for the peers whose ids are closer to the pair's key hash than itself. Neighbours will continue to query the next level of neighbours for closer ids. If a peer cannot find any other neighbours whose ids are closer, it becomes the final peer that will then eventually store the pair physically in its hard drive. The DHT value retrieval process is similar to the storing, thus we do not repeat it here.

DHTs have the following characteristics:

- *Decentralisation*. There are no coordinating or indexing servers in DHT overlays and DHTs are purely decentralised [58]. All peers collectively form a DHT system without any central coordination. This removes the possible single point of failure (single place where can causes the failure of the whole system) from the system and thus obtains better robustness.
- *Scalable.* When more peers join the overlay, the performance of the DHTs will not decrease. Firstly, the capacity of the storage of the DHT will increase as peers will contribute more local disk space. Secondly, because every peer is equal and the interconnections between peers are constrained by the DHT algorithms, the participation of peers will not overload other peers. In addition, as the DHT algorithms emphasise the distributed storage, {key, value} pairs will not be stored on a single or a limited number of peers. On the contrary, DHTs intend to equalise the storage for the pairs

over all peers, i.e., every peer stores similar numbers of pairs [36]. Therefore, with the participation of more peers, the load on existing peers will be more relaxed as the new peers will share portions of pairs and balance them with the existing peers.

- *Fault tolerant.* DHTs have a replica mechanism to try to avoid the loss of {key, value} pairs due to any failure of peers [36]. Every {key, value} pair will have a number of replicas in the overlay. If a certain amount of peers fail to function, pairs will not be lost. Also, a timeout parameter exists in DHTs which is used to check whether a neighbour of a peer is still functioning or not. If the timeout expires and there is no signal being returned back from a neighbour, the peer will try to connect a new possible neighbour candidate and a minor connection modification will occur.

Chapter 3

Analysing BitTorrent's Seeding Strategies

The analysis in this chapter is key to identifying how database distributions over Peer-to-Peer networks can be conducted; this in turn is crucial for employing BitTorrent in the implementation of ppBLAST.

Various terminology is introduced (e.g., freerider, exploiter, selfish leecher, etc.). A summary of this terminology can be found in the thesis glossary.

3.1 Introduction

BitTorrent is a typical peer-to-peer (P2P) file distribution application that has gained tremendous popularity in recent years. A considerable amount of research exists regarding BitTorrent's choking algorithm (see Glossary), which has proved to be effective in preventing freeriders. However, the effect of the seeding strategy on the resistance to freeriders in BitTorrent has been largely overlooked. In addition to this, a category of selfish leechers (termed exploiters), who leave the overlay immediately after completion, has never been taken into account in previous research. In this chapter two popular seeding strategies, the Original Seeding Strategy (OSS) and the Time-based Seeding Strategy (TSS), are chosen and we study, via mathematical models and simulation, their effects on freeriders and exploiters in BitTorrent networks. The mathematical model is verified and we discover that both freeriders and exploiters impact on system performance, despite the seeding strategy that is employed. A selfish-leecher's threshold is identified; once this threshold is exceeded, we find that TSS outperforms OSS – that is, TSS reduces the negative impact of selfish lechers more effectively than OSS. Based

on these results we discuss the choice of seeding strategy and speculate as to how more effective BitTorrent-based file distribution applications can be built.

3.2 Seeding Strategies and Performance

In a traditional client/server file distribution paradigm, a server takes responsibility for transmitting data to all clients. This service model is limited in scalability, especially when the files are large. As a successful Peer-to-Peer file-sharing system, BitTorrent [31] solves this problem by dividing a large file into many small sized blocks and encouraging clients to exchange blocks during their downloading processes. This mechanism reduces load on the server and improves the system service capacity.

Like many other peer-to-peer systems, BitTorrent faces the challenge of *freeriders* [1, 49], which are peers who never upload blocks to others. A peer can act as a freerider by setting the upload rate to a very low value or even zero. Fortunately, BitTorrent can effectively penalise those freeriders using its Tit-For-Tat (TFT) policy, which determines how peers with incomplete files (called *leecher*s) exchange blocks. With the TFT policy, all leechers exchange blocks only with those who upload to them at a higher rate, thus freeriders cannot obtain blocks because they never upload. While most previous research [31, 45, 49, 54] focuses on the behaviour of leechers, the role of the *seed*s (peers with complete files), in the process of preventing freeriders has been largely overlooked.

Since seeds own all the data blocks, they are dedicated to uploading to others. Seeds use a seeding strategy to decide which leechers to serve. Currently, a widely used seeding strategy, the *Original Seeding Strategy* (OSS), ensures that seeds upload to leechers which have the highest download rates, in the hope that new seeds can be produced quickly, which can then serve others. However, when freeriders with relatively high download bandwidths exist in the overlay, due to the mechanism of OSS, there is a possibility that those freeriders dominate the resources of seeds and delay the downloading processes of other unselfish leech-

ers. Thus the OSS strategy may benefit freeriders rather than necessarily fostering contribution. In order to solve this problem caused by freeriders, a new seeding strategy, called the *Time-based Seeding Strategy* (TSS), was proposed in [50]. By employing this strategy, seeds serve each leecher in turn and for the same amount of time so that no single leecher (including freeriders) can dominate the resources of seeds. However, the negative side of TSS is that the speed of producing fresh seeds in the overlay is slowed down and eventually the overall performance may be impacted. Due to the lack of a comprehensive analysis of TSS in [50], it is not clear whether TSS is better than OSS in preventing freeriders and as such this remains an open research question.

Furthermore, an issue largely ignored is that the newly generated seeds can choose not to act as expected – they may stay for only a short time [48, 53] or simply quit the system once they have obtained the whole file. In this chapter, we term these types of peers *exploiters*, i.e., they serve others while downloading, but quit the overlay immediately after its completion. So far little attention has been paid to the influence of exploiters; therefore it is unclear how OSS and TSS are resilient to their behaviour.

In order to answer these questions and direct the selection of seeding strategy for BitTorrent clients, we conduct a comprehensive analysis of OSS and TSS. First we establish a mathematical model to assess OSS's effectiveness in reducing the impact of selfish leechers (freeriders and exploiters) in a homogeneous environment where all peers have identical downlink and uplink bandwidths. We then introduce BitTorrent simulation experiments and provide experimental results that verify our model and compare TSS with OSS. The investigation is then extended to a heterogeneous environment. We show that under either OSS or TSS, freeriders and exploiters degrade system performance. If the number of selfish leechers increases, the performance of OSS-led BitTorrent drops faster but is still better than a TSS-led version. However, there is a threshold for the scale of the selfish leechers. Once the threshold is met, TSS performs better than OSS and provides better resistance to freeriders and exploiters.

The remainder of this chapter is organised as follows. Section 3.3 presents an overview of BitTorrent and documents related work; Section 3.4 analyses the different seeding strategies using a mathematical model; Section 3.5 describe our simulation methodology; Section 3.6 and 3.7 discuss the simulation results; finally Section 3.8 summaries the chapter.

3.3 Understanding BitTorrent

BitTorrent employs a series of sophisticated mechanisms to encourage peers to upload data to each other, and thus achieves scalable and highly efficient content distributions. In this section, we first give an overview of BitTorrent. Several key factors behind the success of the BitTorrent protocol, such as the *choking algorithm* and the *local-rarest-first mechanism*, are already described in previous research [3, 31, 50, 54, 61]. We thus present only the two popular *seeding strategies*, OSS and TSS, in detail. Throughout this chapter, we use terminology first introduced in [50]; much of this terminology is summarised in the Glossary of Terms at the front of the thesis.

3.3.1 BitTorrent Mechanism

Prior to the content distribution, the content provider splits the file into a number of pieces and obtains the SHA-1 hashes for all pieces. Together with the IP address and port number of the tracker, the provider encapsulates the file piece information into a torrent. Normally the provider itself will then connect to the tracker and thus become the initial seed. After peers retrieve the torrent, they obtain the IP address and port number of the tracker and then join the BitTorrent overlay through the tracker. From this time point, peers become leechers or if some peers already have the complete set of file pieces when they join, they become initial seeds (we do not consider this case in our research). Every peer updates its own peer set by obtaining a peer list, which contains a random set of peers (their IP:Port), from the tracker at a certain time interval. Seeds upload data to leechers using the *seeding strategy*, while leechers interact with each other, i.e.,

decide who to download data from or who to upload data to, by executing the *choking algorithm*. When a peer starts to download data from others, leechers decide which file piece to retrieve by following the guidance of the *local-rarest-first algorithm* (LRF). Once a leecher finishes downloading, it either rejoins the overlay to be a seed or leaves immediately so becoming an exploiter.

3.3.2 Seeding Strategy

The initial seed and the regular seeds that come from the leechers all have a complete set of file pieces and thus do not need any data from others. Note that the choking algorithm uses the uploading rates of leechers to decide whom to upload to, thus it is not applicable for seeds because seeds cannot calculate other leechers' uploading rates. In order to effectively contribute to the distribution overlay optimally, seeds employ seeding strategies. There are currently two deployed seeding strategies: the *original seeding strategy* (OSS) and the *time-based*

Algorithm 3-1: OSS, invoked by seeds every 10 seconds

remove unchoked leecher from the *interested_leecher_list*
for *every leecher* in the *interested_leecher_list* **do**
  calculate the rate (*download_rate*) at which the leecher downloads from *this_local_seed*
end for
sort *interested_leecher_list* in descending order based on the leecher's *download_rate*
for $i \leftarrow 1$ **to** 3 **do**
  unchoke *interested leecher list*(i)

seeding strategy (TSS). These are described in the following text and further analysed in Section 3.4.

Algorithm 3-2: **TSS, invoked by seeds every 10 seconds**

global variable: t

sort the *interested_leecher_list* based on the leecher's last unchoke time, with the most recently unchoked leecher last

if $t = 0$ **then**

  for $i \leftarrow 1$ **to** 3 **do**

    unchoke *interested_leecher_list*(i)

  end for

  int $r \leftarrow$ random integer between 4 and n (n is the number of interested leechers)

  unchoke *interested_leecher_list*(r)

  $t \leftarrow t + 1$

else if $t = 2$

  for $i \leftarrow 1$ **to** 4 **do**

    unchoke *interested_leecher_list*(i)

  end for

  $t \leftarrow 0$

end if

OSS (see algorithm 3-1) has been employed since BitTorrent was invented [31]. In the BitTorrent distribution process, there are seeds that always stay in the overlay for a limited time period [54, 61]. Thus, OSS aims to force seeds to contribute to the overlay as much as possible before they leave. In following OSS, seeds upload data to leechers whose downloading rates are highest. In other words, seeds aim to upload data as quickly as possible. There are three aspirations behind this process: 1. Seeds can deliver a maximum number of pieces to leechers; 2. Leechers that download from seeds can become new seeds quickly; 3. New seeds can continue to serve the remaining leechers.

TSS (see algorithm 3-2) was introduced in the official BitTorrent client 4.0.0 [50]. In following TSS, seeds upload data to leechers uniformly. In other words, seeds serve each of their neighbour leechers in turn based on the time stamp of the last service, regardless of the leechers' download rates. A seed can perform *s* parallel uploads. After a seed has been uploading data to a leecher for sixty seconds (typically), it chokes the leecher and selects another leecher to serve. In this manner, all leechers that are connected to a seed will be served for a similar time period. The purpose of this strategy is: 1. To prevent any single leecher from monopolising seeds; 2. To reduce the amount of duplicate data a seed needs to upload before it contributes a full set of file pieces to the overlay.

3.3.3 Additional Related Work on the Analysis of BitTorrent

There has been a good deal of research on analysing the BitTorrent mechanism. The methodologies that this work employs can be categorised into three groups: creating mathematical models, carrying out simulations and analysing BitTorrent traffic.

Qiu *et al.* [54] construct a fluid model for BitTorrent in order to address several issues: peer evolution, scalability, file sharing efficiency, local availability and incentives to prevent freeriders. Their work indicates that the mean download completion time of leechers does not relate to the peer arrival rate. The utilisation of the uploading/downloading bandwidth of each peer is fairly high. However, the success of the distribution of a file is related to the number of freeriders in the overlay. If the number of freeriders is increasing, the numbers of seeds will exponentially decrease and the distribution terminates.

Massoulie *et al.* [62] introduce a probability model of *coupon replication systems*. The major conclusions, which are directly related to BitTorrent, are 1. the peer arriving or departing rate does not affect the system performance significantly; 2. the efficiency of the distribution does not critically depend on the local-rarest-first algorithm.

Tian *et al.* [56] create a simple mathematical model to study the performance of BitTorrent file sharing, with particular interest on the completeness of the downloading process. They find that the current choking algorithm improves the distribution efficiency but cannot help to maintain the file availability and may not prevent the system from termination (the rest of the leechers cannot finish downloading) because of the lack of certain pieces in the overlay.

Fan *et al.* [46] investigate how BitTorrent achieves incentives and prevents freeriders. Felber *et al.* [63] conduct a simulation to investigate how BitTorrent-like protocols handle flash-crowds. By the means of simulation, they find several tradeoffs inside each algorithm that BitTorrent uses and that the system performance depends on many factors.

Bharambe *et al.* [3] use an event-driven simulator to comprehensively evaluate the performance impact from the core algorithms (TFT choking algorithm, local-rare-first algorithm and so forth) that BitTorrent employs. They find that the distribution rates are not optimal although they were reported to be high [48]. The choking algorithm is very effective and unfairness, which is defined in terms of serving rates, has been prevented to a satisfactory degree. However, if the fairness is defined in terms of the data served by nodes, the choking algorithm does not perform well. The local-rarest-first algorithm outperforms alternative piece selection strategies but does not effectively solve the last piece problem when leechers wait for the downloading of the last piece. The last finding from their work is that the initial seed is very important and it should distribute a full set of pieces into the overlay as quickly as possible.

Iza *et al.* [48] conduct a comprehensive analysis on the data that is derived from the tracker log of the most popular Redhat 9 torrent. The measurements involve the downloading and uploading characteristics of thousands of peers. They observe that the mean downloading rate of leechers is 50KB/s, which indicates the good connectivity that most leechers have. They also find that peers, including both leechers and seeds, are constantly sending data to other leechers, which implies that the choking algorithm is able to provide incentives to peers for upload-

ing. However, one fact that is also discovered is that major data contributions are from a relatively small number of peers.

Pouwelse *el al.* [53] present a measurement study on BitTorrent. The major contribution of their work is the analysis of the flashcrowd effect, i.e., the phenomenon of the sudden popularity of a new file distribution. They indicate that the peer arrival process is not following the Poisson distribution and this gives guidance to the future BitTorrent-related simulation research work.

Guo *et al.* [47] focus on the fact that peers always download though multiple torrents simultaneously. With this fact, the research in which single torrent distribution is assumed is not accurate any more because the downloading / uploading bandwidths of peers are shared among multiple BitTorrent distribution swarms. Guo *et al.* also find that the arrival / departure rate is exponential and that service availability becomes poor quickly when the distribution is close to the end. Finally, they propose a system design for multi-torrent cooperation.

Our work on BitTorrent differs from the previous work discussed above in the following respects. First, we identify a special kind of selfish leecher: termed exploiters. We show that the behaviour of the exploiters impacts on the system performance although the harm done is not as severe as that caused by freeriders. Second, we analyse two seeding strategies (OSS and TSS) in detail and focus on their resistance to the selfish leechers, which none of the previous research work has conducted. Finally, our experiments show that choosing a seeding strategy to adapt to the scale of the selfish leechers in the overlay is very important. We thus propose in our future work that a mechanism is in need to detect the scale of the selfish leechers so that the seeding strategy can be loaded or changed dynamically.

3.4 Modelling Seeding Strategies

As described in Section 3.3, there are currently two kinds of seeding strategies, the *original seeding strategy* (OSS) and the *time-based seeding strategy* (TSS), which have been deployed through different BitTorrent client applica-

tions. In this section, we present a mathematical analysis to investigate the impact that freeriders or exploiters have on the mean download completion time of unselfish leechers if OSS is employed.

3.4.1 Metrics, Assumptions and Scenarios

Leechers join a BitTorrent network in order to obtain a complete shared file from the overlay. They have one major concern - the time that they need to spend to complete the download. Since uploading data is the driving force of a BitTorrent system, unselfish leechers are important to the overlay and the quality of service that they get from the distribution should be kept at a certain level; otherwise, the fairness in the system is violated. Thus, we choose the *mean download completion time* of unselfish leechers, denoted by t_u, in the system as the metric of our mathematical analysis. Using this metric, we can investigate how freeriders and exploiters affect the fairness and the benefits that unselfish leechers obtain if BitTorrent employs OSS.

Another concern that leechers have is whether a complete set of blocks of the file can be finally obtained. [54] suggests that the probability of a leecher finding a desired block among its neighbours is very close to one, therefore, for simplicity of analysis, the first assumption that we make is that at any point in time, the network owns the complete set of blocks of a file.

The environment in which a file is distributed is assumed to be homogeneous. This is the second assumption that we make. Initially, there are N leechers and one initial seed. All peers have identical uplink bandwidths of u, but varying downlink bandwidths of d_i. The size of the file being distributed is A.

Two other assumptions are made:

- A peer's downloading bandwidth is not the bottleneck of data transfer. The experimental results in [54] suggest that the mean utilization of d_i is practically quite low (20-40%). Therefore, it is reasonable to assume that the

downlink bandwidth of each leecher will not restrict the downloading process.

- Peers have the same uplink bandwidths u, and the mean utilisation of peers' uplink bandwidths is 100% [3] in this homogeneous environment. We will consider the heterogeneous setting where the uplink bandwidths are not equal in our simulation study.

Three scenarios are considered in the model. All leechers in the first scenario are unselfish leechers. In the second scenario, a number F of leechers are freeriders and the remainder of the leechers are unselfish. In the final scenario, only unselfish leechers and a number E of exploiters are present in the network. The different t_u for the three scenarios are calculated to show whether OSS can guarantee fairness in the system.

3.4.2 The Model

In all three scenarios, unselfish leechers act as seeds for a mean time T_s after finishing downloading, regardless of whether they are being served by seeds. T_s is assumed to be a constant value, which is long enough to let new seeds contribute sufficient blocks to the network [3, 61]. Let T_{s0} denote the staying time of the initial seed. Freeriders and exploiters have higher downlink bandwidths than unselfish leechers. The mean download time of exploiters is denoted by T_E.

In the first scenario, all the unselfish leechers keep uploading to others before they become seeds. The amount of data that they upload is uNt_u (t_u is the period of time that unselfish leecher uploads for). After that they become seeds; they stay for a mean period of time T_s, hence, the amount of data they send out to the system is uNT_s. The initial seed contributes to the network with the amount of data uT_{s0}, where T_{s0} is long enough to ensure that at least one copy of the entire file blocks are distributed into the network.

In the distribution process of the whole file (i.e., within the period of the torrent lifetime), each of the N leechers eventually obtains a complete copy of the

file. Therefore, an amount NA of data has been uploaded by all peers. Thus, we have $NA = uNt_u + uNT_s + uT_{s0}$ and t_u is given by

$$t_u = \frac{NA - uNT_s - uT_{s0}}{Nu} = \frac{A}{u} - T_s - \frac{T_{s0}}{N} \tag{1}$$

When the F freeriders with the highest download rates join the system, the seeds pass on all of their contributions to the freeriders (following the OSS policy) before they, or the freeriders, leave the network. Because the freeriders do not upload blocks at all, all data are distributed through only the unselfish leechers and the seeds; that is, $NA = (N - F)ut'_u + (N - F)uT_s + uT_{s0}$.

In the second case, t'_u is given by

$$\begin{aligned} t'_u &= \frac{NA - (N - F)uT_s - uT_{s0}}{(N - F)u} \\ &= \frac{N}{N - F} \cdot \frac{A}{u} - T_s - \frac{T_{s0}}{N - F} \end{aligned} \tag{2}$$

To show the performance degradation caused by the freeriders, the increase rate (IRD) of mean download completion time of unselfish leechers is

$$IRD = \frac{t'_u - t_u}{t_u} = \frac{F}{N - F} \cdot \frac{\frac{A}{u} - \frac{T_{s0}}{N}}{\frac{A}{u} - \frac{T_{s0}}{N} - T_s} > \frac{F}{N - F} \tag{3}$$

In [1], it is discovered that nearly 70% of Gnutella users share no files. If we assume F to be 70%, IRD will be more than 230%, which implies that unselfish leechers will have to spend 230% more time on downloading because of the existence of freeriders. At an early stage, freeriders dominate the resources of initial seeds, and unselfish leechers seldom exchange blocks with each other, because new blocks are rarely delivered to them. This blank transfer period causes a significant increase to the download completion time of unselfish leechers. Note that when $T_{s0} = NA/u$, $IRD = 0$. This implies that when the initial seed stays in

the network for a sufficiently long period of time, the mean download completion time of unselfish leechers will not be increased.

If an alternative strategy, TSS (see algorithm 2), is applied, the freeriders will not be the only consumers of the initial seed. The seed tries to equalise its contributions to every leecher in the system. The unselfish leechers have equal chance to obtain new blocks from the initial seeds and the blank transfer period no longer exists. The mean download completion time of the unselfish leechers is thus reduced. This result will be further demonstrated in the simulation experiments.

In the final scenario, the E exploiters keep exchanging blocks with the leechers while they receive data from the seeds. However, the exploiters will leave the network as soon as they finish downloading; that is, they do not stay as seeds for a period of time T_s . Therefore, $NA = (N-E)ut_u'' + Eut_E + (N-E)uT_s + uT_{s0}$ and t_u'' is given by

$$\begin{aligned} t_u'' &= \frac{NA - Eut_E - (N-E)uT_s - uT_{s0}}{(N-E)u} \\ &= \frac{N}{N-E} \cdot \frac{A}{u} - \frac{E}{N-E} \cdot t_L - T_s - \frac{T_{s0}}{N-E} \end{aligned} \tag{4}$$

where t_E is the mean download time of the E exploiters.

To show the performance degradation, IRD' is calculated

$$\begin{aligned} IRD' &= \frac{t_u'' - t_u}{t_u} = \frac{E}{N-E} \cdot \frac{\frac{A}{u} - \frac{T_{s0}}{N} - t_E}{\frac{A}{u} - \frac{T_{s0}}{N} - T_s} \\ &= \frac{E}{N-E} \cdot \left(1 - \frac{t_E - T_s}{\frac{A}{u} - \frac{T_{s0}}{N} - T_s}\right) \end{aligned} \tag{5}$$

Note that $t_E < t_u$ because exploiters tend to be served by seeds with a higher priority, thus we have

$$IRD' = \frac{E}{N-E} \cdot \left(1 - \frac{t_E}{\frac{A}{u} - \frac{T_{s0}}{N} - T_s} + \frac{T_s}{\frac{A}{u} - \frac{T_{s0}}{N} - T_s} \right) > \frac{E}{N-E} \cdot \frac{T_s}{\frac{A}{u} - \frac{T_{s0}}{N} - T_s} \quad (6)$$

The negative impact that the exploiters bring to the system is less than that of the freeriders. When t_E increases, IRD' drops. The reason for this is that if the exploiters have to stay as leechers in the network for a longer period of time, they serve more blocks to the unselfish leechers and the download rates of the unselfish leechers therefore increase. OSS does not try to force exploiters to stay as leechers for a longer period of time; on the contrary, it helps the exploiters finish downloading in a faster manner if exploiters have higher download rates.

3.5 Simulation Methodology

In this section we conduct simulations to: 1) Verify our mathematical model, which is presented in Section 3.4; 2) Analyse seeding strategies in both homogeneous and heterogeneous network environments; 3) Further explore the performance impact of seeding strategies using more comprehensive metrics.

3.5.1 Simulator Details

A discrete-event-based simulator written in Java is implemented to simulate peer activities, such as joining, leaving and block exchanging, as well as most of the BitTorrent mechanisms (the local-rare-first algorithm, the choking algorithm, OSS, TSS, etc). In the simulator, we treat every BitTorrent or network related operation as an event. Every event is associated with an event timestamp. The event timestamp does not correspond to the real time and it is a relative time indicator for events. In the simulation, we provide launch events, and event chains will be formed because one event may lead to another. For example, when a peer

becomes able to serve others, the first choking round occurs and a preset choking event for the peer is generated. When the event finishes, it will generate or schedule the next choking round which will occur 10 seconds later. Note that the *10 seconds* is the timestamp inside the simulation. All events, including both the preset and the newly generated events, are pushed into a priority queue, which is sorted by event time. The events in the event queue are polled one by one and executed.

Each peer in the simulator is associated with a downlink and an uplink bandwidth, which resemble asymmetric network access as widely observed today. Based on these bandwidth settings, the simulator calculates the block transfer delay in the following way. When peer *A* is going to send a block to peer *B*, the simulator retrieves *A*'s upload bandwidth and *B*'s download bandwidth. The bandwidth with the least value is selected and the time delay is calculated through dividing the block size by the selected bandwidth value. The simulator then schedules a block-received event with the time delay for peer *B*. When the event is executed, peer *B* receives the block and its file block set is updated.

Since the time delay computation for each block transmission is expensive, we make a number of simplifications that have negligible impact on the performance aspects we are considering. For example, we ignore the interaction of BitTorrent control packets, which are normally very small compared with the data block size. Following Akella et al. [64], network bottlenecks are assumed to mainly occur in the edges of networks.

We simulate BitTorrent mechanisms, such as its choking algorithm, local-rarest-first algorithm, OSS, TSS, as comprehensively as possible. However, BitTorrent performance can be influenced by many configuration parameters at the client side or the network condition of peers (such as a peer's network link is not stable). Since our study is to find the performance impact of seeding strategies, we do not try to find a set of optimal parameters for BitTorrent performance but use those parameters proposed in related research [3, 50].

3.5.2 Metrics

In term of fairness, we focus on the benefits that the unselfish leechers can obtain in the BitTorrent distribution process. Hence the metrics we select for the simulations are all for unselfish leechers.

Mean download completion time of unselfish leechers: In the mathematical model, we use the *mean download completion time* of unselfish leechers as the metric to compare seeding strategies where freeriders or exploiters exist. Since the purpose of BitTorrent is to distribute a file among a number of peers, the mean download completion time indicates the efficiency of the distribution. Using this metric we can investigate the overall system performance. In the simulation experiments we continue to validate our model and extend the BitTorrent overlay environment from a homogenous to a heterogeneous setting. The download completion time of every unselfish leecher is summed and a mean value calculated.

Download rate and bandwidth utilisation: In the experiments with homogeneous settings, we use download rate and bandwidth utilisation of unselfish leechers as metrics to indicate performance differences between OSS-led and TSS-led BitTorrent overlays where selfish leechers exist.

Cumulative distribution of unselfish leechers' download completion times: For the experiments with heterogeneous settings, we plot all cumulative distributions of the download times of all unselfish leechers. This metric will help us to understand how individual leechers perform.

3.5.3 Setup of Experiments

Since our study focuses on the seeding status of the distribution, i.e. the finishing period, we assume a steady-state network where all leechers are already present. Bharambe *et al*. [3] also use a set of peer bandwidths, which was derived from the Gnutella study presented in [2]. We borrow the bandwidth values from [3] and assign them to peers in our simulation experiments.

In term of peers' bandwidths, our experiments have two settings: homogenous and heterogeneous. In the homogenous setting, all leechers have the same network bandwidths that we choose from the bandwidth values shared by [2, 3]. In the heterogeneous setting, the link bandwidths of leechers follow the distribution which is presented in [3].

The experimental setup is summarised as follows:

- File size: A = 200MB; piece size: 256KB; block size: 16KB
- Number of initial seeds: 1
- All unselfish seeds stay in the overlay for 1000 seconds
- Peers' bandwidth:
    - Initial seed uplink: 50KB/s
    - Leechers' bandwidth. Homogeneous: downlink = 150KB/s, uplink = 38KB/s; Heterogeneous: see Table 3-1
    - For freeriders and exploiters, the downlink remains 150KB/s and the uplink remains 38KB/s
- Number of leechers: N = 1000 and all leechers join the network simultaneously at the beginning of the distribution
- Unchoking slots for each peer: s = 5

Table 3-1: Bandwidth distribution of leechers (derived from actual distribution of the Gnutella network [2, 3])

Downlink (KB/s)	Uplink (KB/s)	Fraction of leechers
78	12	0.3
150	38	0.6
300	100	0.1

3.5.4 Roadmap of Experiments

We begin in Section 3.6 by applying OSS and TSS, respectively, to BitTorrent and examining their mean download times of unselfish leechers in a homogeneous environment (all leechers have the same uplink and downlink bandwidths) while freeriders or exploiters exist. For each of the seeding strategies, there are three experiments performed with this setting. In the first all leechers are unselfish. In the second and last we vary the number of freeriders / exploiters from 100 to 700 and record the mean download times. Through the three sets of mean download times we calculate the *Increase Rates* of the mean download times to study the impact of OSS and TSS on unselfish leechers while freeriders or exploiters exist. We finally use our mathematical model to predict the increase rates and compare them with OSS and TSS to verify the accuracy of our model.

In Section 3.6, we continue the experiments with heterogeneous settings. In addition to the mean download time of unselfish leechers, we use the cumulative distribution of download times for the three classes of unselfish leechers whose network bandwidths are different. This metric helps us investigate the details of individual completion time. We use the set of 100, 300, and 700 for the numbers of freeriders or exploiters, as the trend of the change of the download time is the important characteristics that we are studying and we believe this set is enough to highlight this trend.

Note that for the results of all experiments, they are related to only unselfish leechers unless we clearly specify otherwise.

3.6 Results: the Homogenous Setting

In this section, we present the experimental results in a homogenous setting. For each of the experiments, the figures are plotted from the mean values of the results which are provided from ten simulation runs.

3.6.1 Impact of Freeriders

Fig. 3-1 shows the mean download times of OSS and TSS while freeriders exist. When there is no freerider in the network, the mean download time when OSS is applied is 1,938.4 seconds, while applying TSS leads to a download time of 2,114.6 seconds. The download process therefore costs 8.3% more time to finish if TSS is employed; OSS clearly outperforms TSS when all leechers are unselfish. The advantage of OSS is kept, although debasing, along with the increment of the number of freeriders. When the number of freeriders reaches 300, TSS starts to out-perform OSS. Its mean download time is 2,812.6 seconds representing a 7.8% improvement over OSS. When the number of freeriders is 700, which means 70% of the leechers are freeriders, the mean download time of TSS is 40.9 percent less than OSS and the downloading performance of OSS becomes very poor (its mean downloading time is 7,275.9 seconds). Clearly, TSS performs better in resisting freeriders than OSS, but OSS out-performs TSS when the number

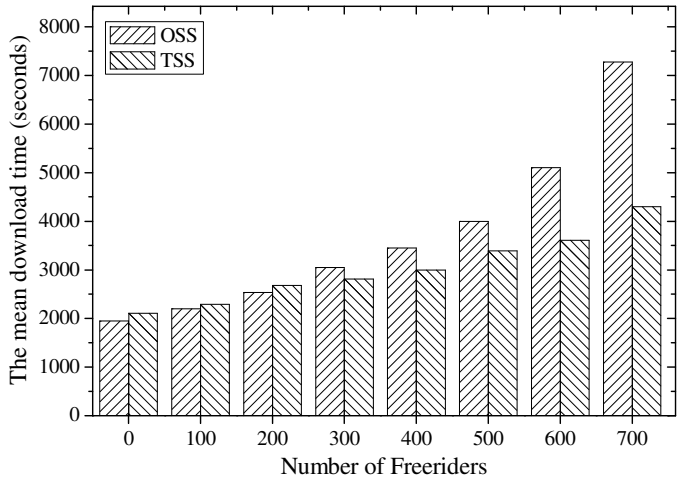


Figure 3-1. Results for homogenous setting with freerider: the mean download completion time comparison

of freeriders is below a certain value.

We plot the mean download rates of OSS and TSS in Fig. 3-2. We can see that the download rate of OSS is higher than that of TSS before the number of freeriders reaches 300, but drops quickly as the number of freeriders increases.

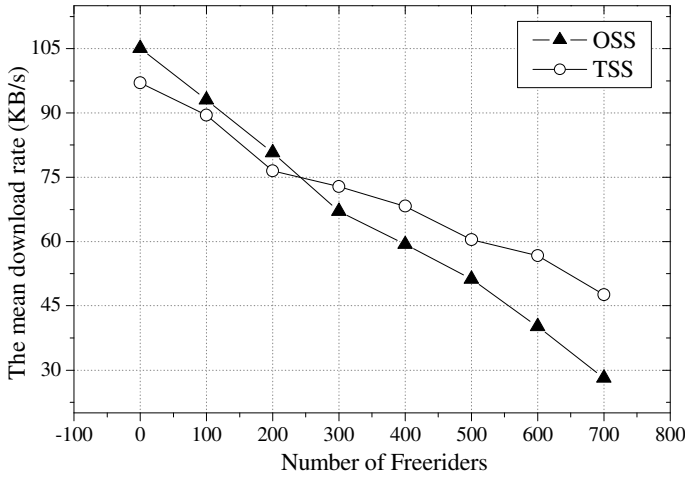


Figure 3-2. Results for homogenous setting with freerider: the download rate comparison

When 70% of the leechers are freeriders, the mean download rate of OSS drops to 28.1KB/s where the mean download bandwidth utilisation is only 18.7%. On the other hand, the mean download rate of TSS drops smoothly and slowly. Even in the worst case, the mean download rate of TSS is 47.6KB/s, although when there are no freeriders in the overlay, TSS performs 17.3% worse than OSS.

From Fig. 3-1 and Fig. 3-2, it can be concluded that before the number of freeriders reaches a certain value, OSS leads to a higher distribution performance than TSS. This is because OSS encourages leechers, who have higher download rates, to be seeds more quickly. Although there may be a number of freeriders in the overlay, a portion of unselfish leechers still have the chance to be served by seeds, and continue to serve afterwards and boost the whole distribution process. We know that freeriders will not serve others after they finish downloading and even while they are being served by seeds, they are not sharing file blocks with others. Thus, when the number of freeriders grows, the possibility for more freeriders dominating the resources of seeds will also increase, and the downloading performance of unselfish leechers are negatively and significantly impacted.

Instead of letting a particular set of leechers dominate the seed resources, TSS forces seeds to treat every leecher equally (despite their download rates) and

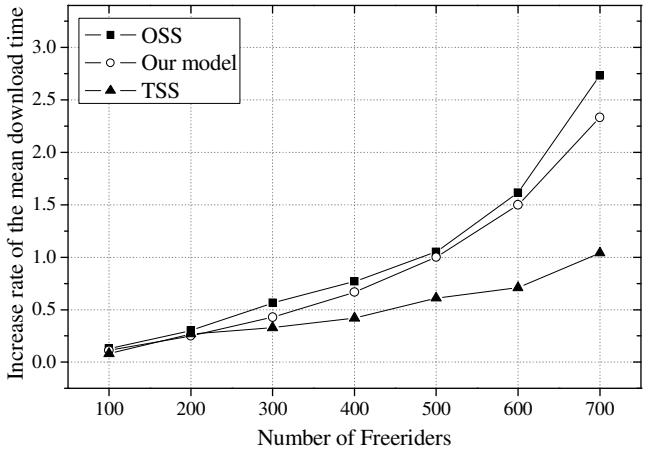


Figure 3-3. Results for homogenous setting with freerider: the *IRD* comparison

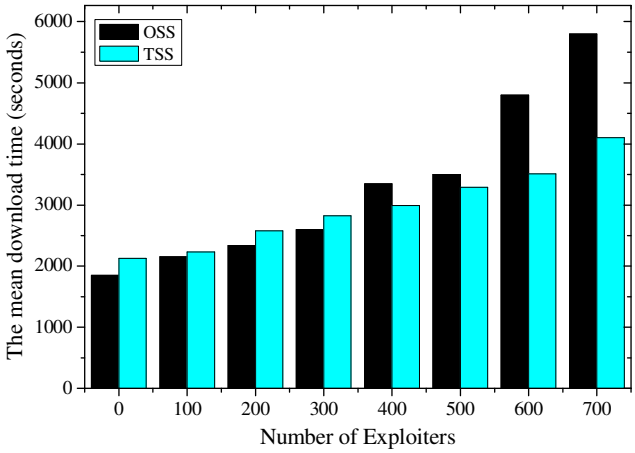


Figure 3-4. Results for homogenous setting with exploiters: the mean download time comparison

serve them one by one for a certain period of time. When the number of freeriders is below a certain value, the TSS-led distribution process has lower performance than OSS-led because it does not try to boost the whole process by making more seeds quickly (unlike OSS). While the number of freeriders is growing, the performance is negatively impacted but does not drop sharply like an OSS-led distribution, because freeriders will never dominate the seed resources and unselfish leechers still can be served by seeds even if the number of freeriders is large.

We compare the increasing rate of the mean download times *(IRD*s) of OSS-led BitTorrent with our model in Fig. 3-3. The reason for calculating the

IRD is to see how much the mean download time grows if the number of freeriders changes from zero to a certain value. Through the growth we can conclude which seeding strategy (OSS or TSS) performs better in resisting the freeriders. In addition, by calculating the IRD from the experimental results, we can justify how practical our mathematical model is. Note that our mathematical model is only for an OSS-led BitTorrent distribution process. We can see from Fig. 3-3 that the mathematical model matches the experimental results. The trends are approximately the same and our experience is that it is common for a mathematical model to underestimate values as it is, after all, an approximation of what is actually performed in practice. We can confirm that in real-world experiments the OSS-led BitTorrent performs worse than our model predicts, although the model still provides a reliable indication of IRDs, and that the mean download time grows at a slightly faster rate.

If we focus on TSS-led BitTorrent, we can see that the *IRDs* from this are very low; from the best case (0.08) to the worst case (1.04). When comparing *IRDs* from a TSS-led implementation with the OSS-led implementation, we find that the TSS-led version is on average 41.66% better than the OSS-led version, and the difference is consistent ranging from -62.01% to -10.75% (the negative sign indicates that the TSS-led version is lower than the OSS-led version). This shows that with TSS, the mean download time increases in a slow and steady manner, while more freeriders emerge.

3.6.2 Impact of Exploiters

Fig. 3-4 shows the mean download times of the OSS-led and the TSS-led versions when there are exploiters in the overlay. The OSS-led BitTorrent performs better than the TSS-led one before the number of exploiters reaches 400, after which the reverse is true. Exploiters in a TSS-led BitTorrent overlay cannot dominate the seeds; however, it is similar for unselfish leechers whether they receive file blocks from seeds or exploiters. Hence, there are no large differences despite employing OSS or TSS when the number of exploiters is below a certain value (400 in our experiments).

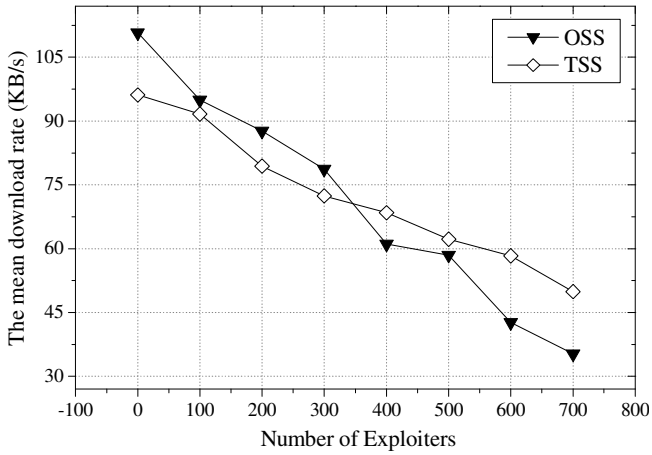


Figure 3-5. Results for homogenous setting with exploiters: the download rate comparison

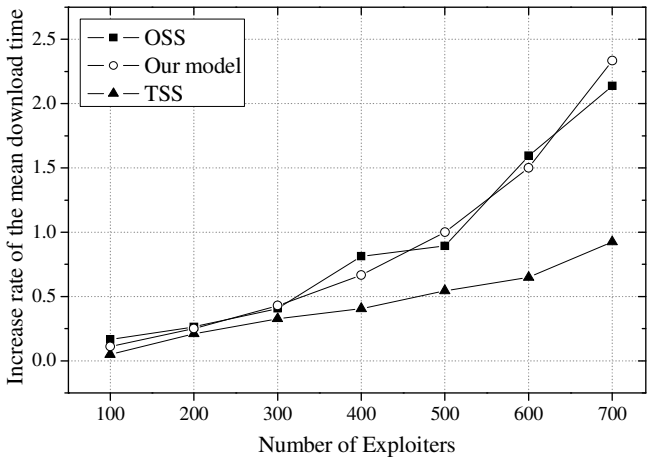


Figure 3-6. Results for homogenous setting with exploiters: the *IRD* comparison

Fig. 3-5 plots the mean download rate of the unselfish leechers. When there are no exploiters in the overlay, the OSS-led BitTorrent distribution shows a high download rate of 110.73KB/s and the download bandwidth utilisation is 0.74. Although it is predictable that the TSS-led BitTorrent performs worse than the OSS-led version, the mean download rate remains high at 96.12KB/s. When the number of exploiters is 100, the download rate of the OSS-led BitTorrent decreases and is close to that of TSS. OSS keeps performing better than TSS until the number of exploiters reaches 400. When the number of exploiters equals 500,

OSS is worse but close to TSS; however, the performance of OSS drops significantly afterwards, while TSS's degradation is slow. When the number of exploiters is 700, the download bandwidth utilisation of OSS is 0.24, lower than that of TSS (0.33).

From Fig. 3-6, we can see that the increase rate of the mean download time for OSS matches our model and that the average difference is 0.091. Before the number of exploiters reaches 400, the increase rates of the mean download time of OSS and TSS are close. This indicates that before the number of exploiters reaches a certain value, the negative impact on both seeding strategies are similar. After this, OSS suffers more acutely than TSS. In other words, TSS performs better than OSS in resisting exploiters.

3.7 Results: the Heterogeneous Setting

In this section, we study the impact of the seeding strategies on the performance of BitTorrent when the peer bandwidth is heterogeneous. As described in sub-Section 3.5.3, we categorise all unselfish peers into three classes whose network bandwidths are different. To emphasise the effects of freeriders and exploiters, we choose to make their bandwidths consistent: 150KB/s (down) and 38KB/s (up).

3.7.1 Impact of Freeriders

Fig. 3-7 plots the cumulative distribution of the download time when there are no freeriders in the overlay. It is clear that TSS performs much worse than OSS in this case. Using TSS, there are only 20% of the leechers able to finish the download within 3,000 seconds and 90% of them need nearly 6,000 seconds to complete. In the case of OSS, 52% of the leechers complete within 3,000 seconds and 93% of them finish within 5,000 seconds.

Recall that there are three classes of unselfish leechers in the overlay. In order to investigate the difference of the impact between OSS and TSS on each

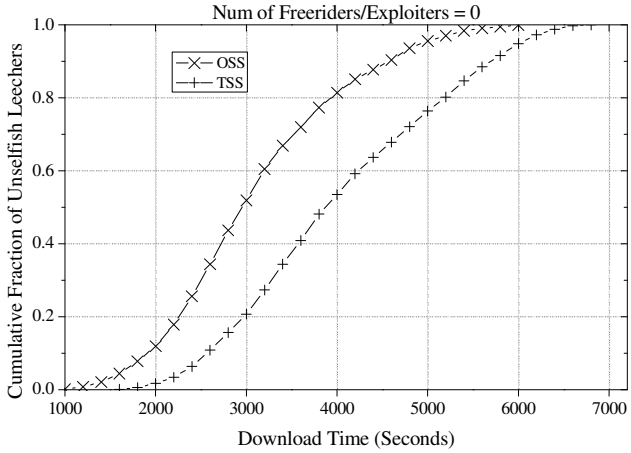


Figure 3-7. The cumulative distribution of the download times, when the number of freeriders or exploiters equal to zero, for all unselfish leechers

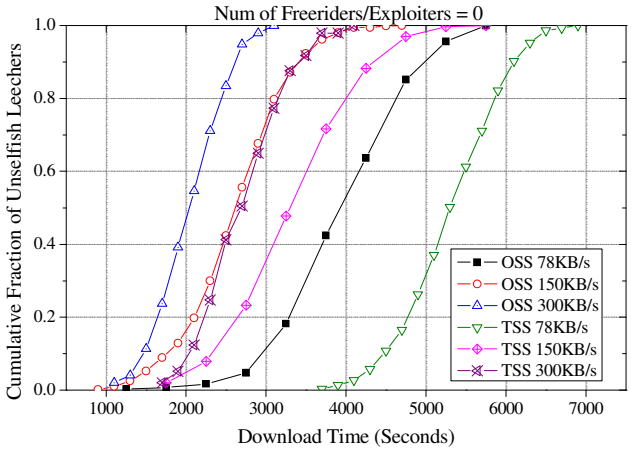


Figure 3-8. The cumulative distribution of the download times, when the number of freeriders or exploiters equal to zero, for each class of unselfish leechers

class, we plot the cumulative distribution of the download times for every class (Fig. 3-8). The download link bandwidths of the three classes of unselfish leechers are 78KB/s, 150KB/s and 300KB/s respectively. Thus, the optimal cases for the download times of the unselfish leechers are 2,625.6, 1,365.3, and 682.7 seconds. We can see from Fig. 3-8 that none of the leechers reach their optimal download time, despite the presence of OSS or TSS. For every class of unselfish leecher, OSS performs better than TSS. The 300KB/s class of leechers in TSS have similar download times to the 150KB/s class in OSS. The 78KB/s class of leechers, who

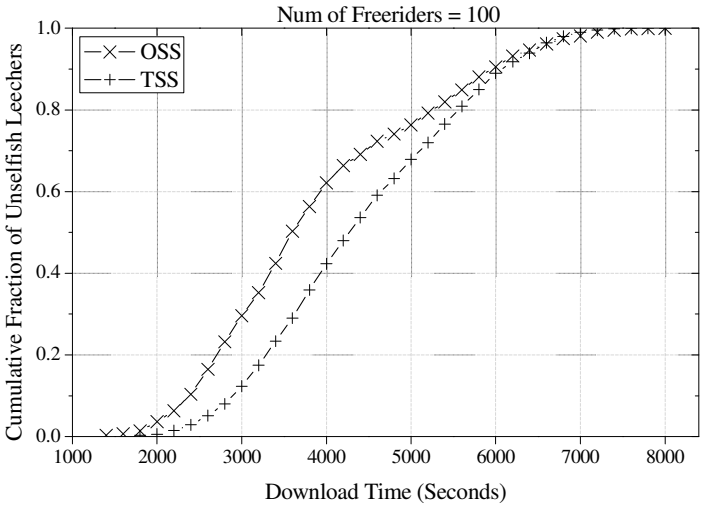


Figure 3-9. The cumulative distribution of the download times for all unselfish leechers when freeriders exist (100 freeriders)

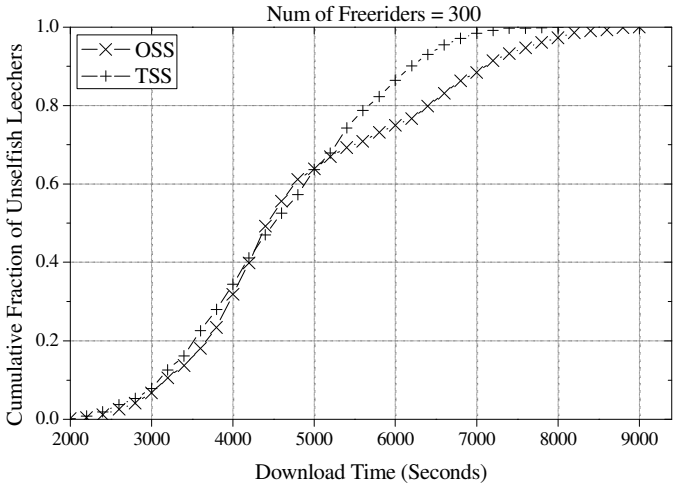


Figure 3-10. The cumulative distribution of the download times for all unselfish leechers when freeriders exist (300 freeriders)

have the lowest download bandwidths, spend the most time completing the download, particularly when TSS is employed.

When the number of freeriders is increased to 100, the two lines in Fig. 3-9 become closer and the segments of the lines where the download time is more than 6,500 seconds overlap. In Fig. 3-12, we can see that the class of TSS 300KB/s has surpassed the class of OSS 150KB/s, while they perform similarly if

the number of freeriders is 0. At the same time, the class of TSS 78KB/s and the class of OSS 150KB/s have very close download times.

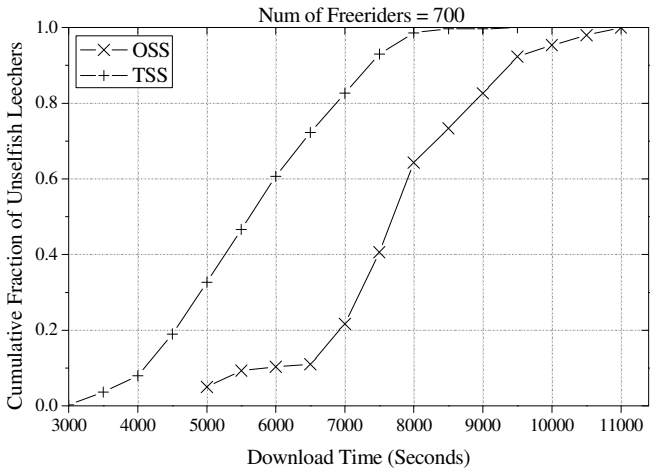


Figure 3-11. The cumulative distribution of the download times for all unselfish leechers when freeriders exist (700 freeriders)

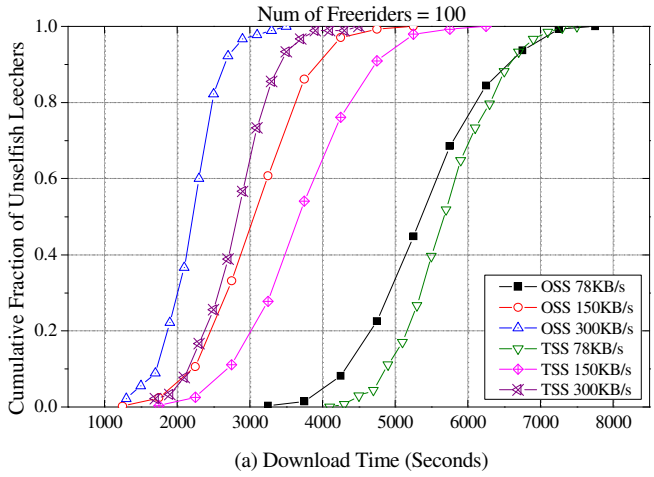


Figure 3-12. The cumulative distribution of the download times for each class of unselfish leechers when freeriders exist (100 freeriders)

This indicates that once the freeriders join the overlay, the performance of the OSS-led BitTorrent degrades towards the TSS-led version. The TSS-led Bit-Torrent also suffers performance degradation; however, the degree of this is less

than that of OSS. When the number of freeriders reaches 300, the lines in Fig. 3-10 are very close to overlapping, before the point where the download time equals 5,000 seconds. This means that there are similar numbers of unselfish leechers having finished their download within 5000 seconds. After 5000 seconds, more unselfish leechers with TSS finish before 7000 seconds than those with OSS. Fig. 3-13 shows that the low bandwidth (78KB/s) leechers with TSS complete sooner than those with OSS

For middle and high bandwidth leechers with TSS and OSS, they perform

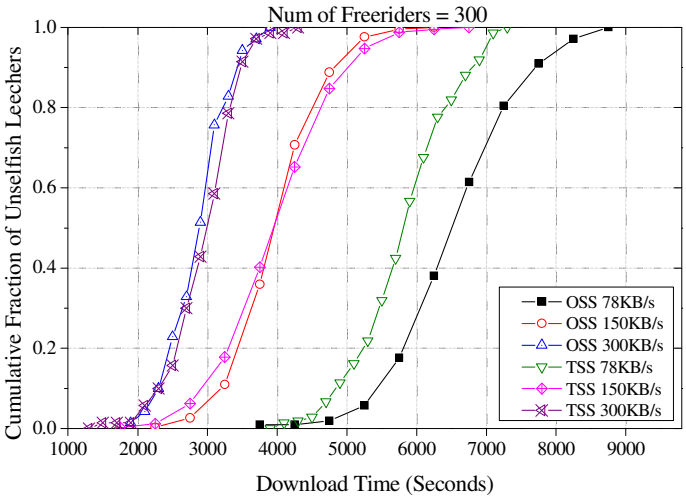


Figure 3-13. The cumulative distribution of the download times for each class of unselfish leechers when freeriders exist (300 freeriders)

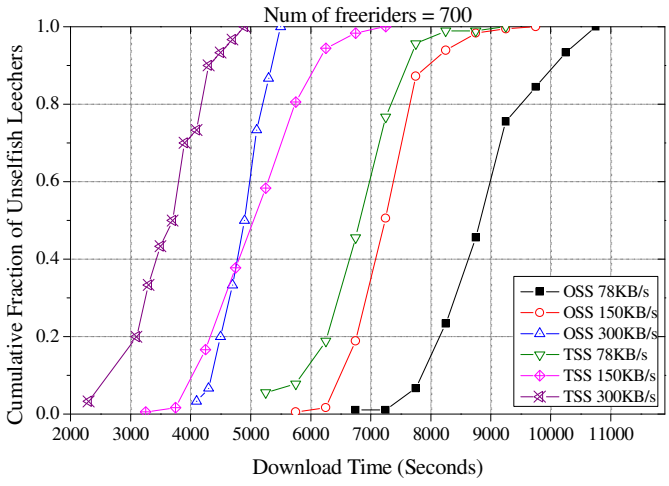


Figure 3-14. The cumulative distribution of the download times for each class of unselfish leechers when freeriders exist (700 freeriders)

the download similarly. This case is seen to be similar to the homogeneous setting. Where the number of freeriders is 300 and above, TSS starts to perform better than OSS. For the results of the experiments in which the number of freeriders is 700, we plot them on Fig. 3-11 and Fig. 3-14. From Fig. 3-11, when the number of freeriders is 700, the download time in TSS is less than in OSS. From Fig. 3-14, the download time in TSS is also less than in OSS, respectively for every kind of bandwidth. Especially in Fig. 3-14, the class of OSS 150KB/s performs even worse than the class of TSS 78KB/s. Comparing the class of TSS 150KB/s with OSS 300KB/s, we can see that, 40% of TSS 150KB/s spent less time than OSS 300KB/s leechers.

In addition to the conclusions that we draw from the above study, we also find two interesting effects of freeriders:

- Freeriders impact simultaneously on all leechers even if they have different bandwidths;
- OSS-led and TSS-led BitTorrent overlays both suffer from freeriders. However, The TSS-led version suffers less than the OSS-led one. This is clearer when more freeriders join the overlay.

3.7.2 Impact of Exploiters

In this section, we study the resistance of OSS and TSS to the exploiters. When the number of exploiters equals zero, the results of the download times are the same as those where there are no freeriders in the overlay (Fig. 3-7 and Fig. 3-8). Thus we do not repeat the results or discussion here. Instead, we continue the study from where the number of exploiters is 100.

When the number of exploiters is increased to 300 (Fig. 3-19), 8% of the class TSS 300KB/s leechers finish downloading before all high-level leechers in the OSS-led BitTorrent; however, the remainder of the OSS 300KB/s leechers perform better than TSS 300KB/s. The download times of the middle-level leechers are very close in both overlays, although those in the OSS-led overlay perform

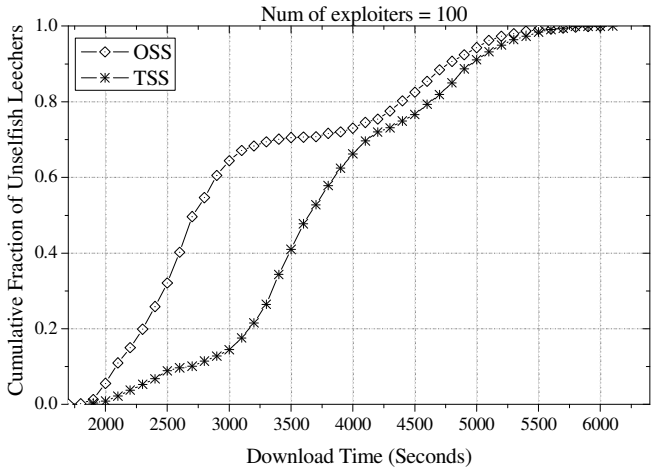


Figure 3-15. The cumulative distribution of the download times for all unselfish leechers when exploiters exist (100 exploiters)

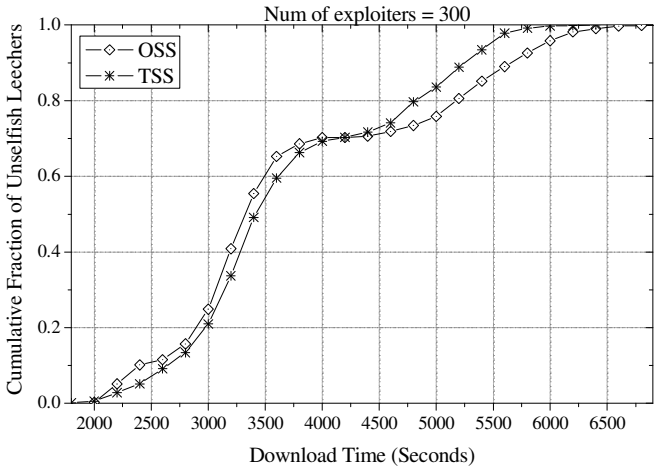


Figure 3-16. The cumulative distribution of the download times for all unselfish leechers when exploiters exist (300 exploiters)

slightly better than the TSS-led version. The low-level leechers in the TSS-led overlay complete sooner than the ones in the OSS-led BitTorrent. With the emergence of more exploiters, the middle-level leechers in the OSS-led overlay significantly impact on performance (compared with Fig. 3-7). This is because when exploiters finish downloading, they leave the network immediately.

Although high-level unselfish leechers will serve the overlay for an amount of time in an OSS-led overlay, their service targets are always other high-level leechers who have not yet finished. The services to the middle-level leechers

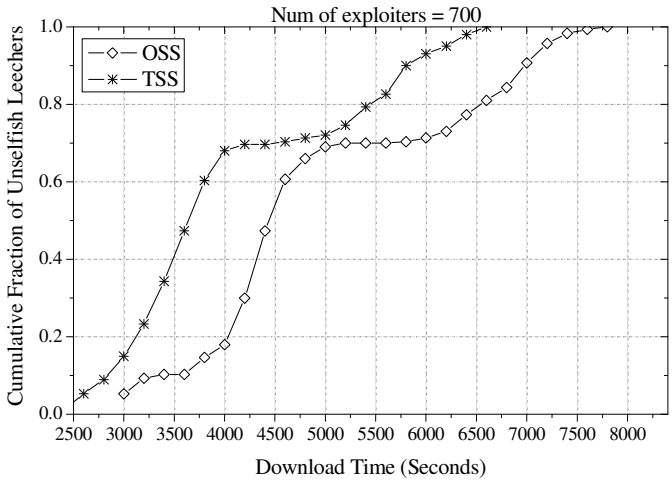


Figure 3-17. The cumulative distribution of the download times for all unselfish leechers when exploiters exist (700 exploiters)

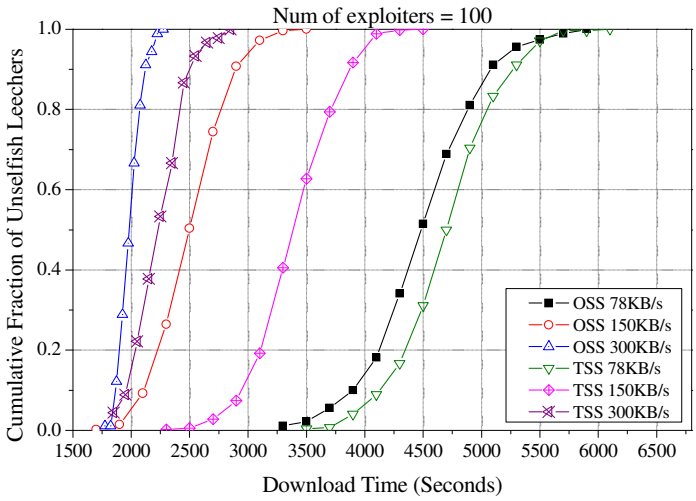


Figure 3-18. The cumulative distribution of the download times for each class of unselfish leechers when exploiters exist (100 exploiters)

are not therefore sufficient. In the TSS-led overlay, middle-level unselfish leechers always obtain their services from existing seeds, even if exploiters exist. Thus the performance degradation in the TSS-led overlay is not as severe. Once the number of exploiters is more than 300, TSS starts to transcend OSS. First, the middle-level unselfish leechers in the TSS-led overlay obtain better overall download speeds than the same class in the OSS-led version (shown in Fig. 3-17), 9% of them perform better than those in the OSS-led overlay when the number of exploiters is 700 (shown in Fig. 3-20).

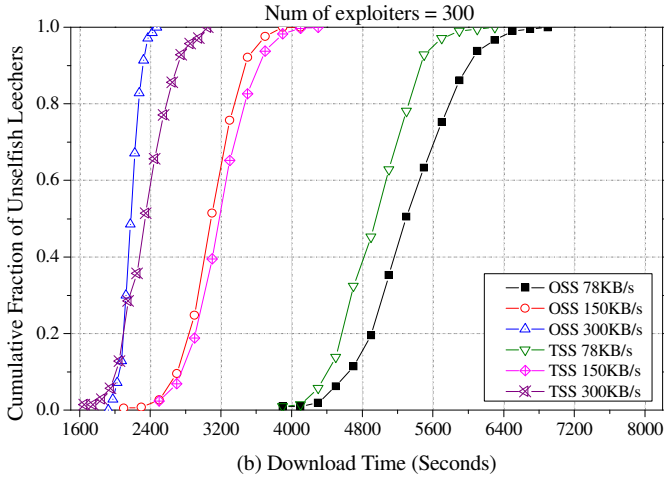
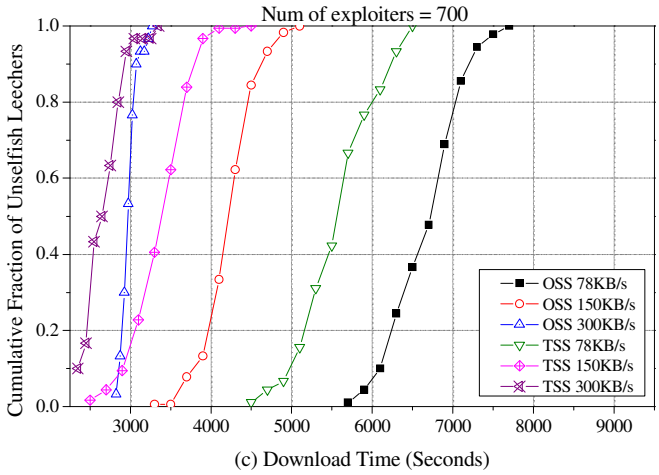


Figure 3-19. The cumulative distribution of the download times for each class of unselfish leechers when exploiters exist (300 exploiters)



The performance differences of low-level leechers between the OSS-led and TSS-led versions remain similar to the case where the number of exploiters is 300; however, their overall download times are increased. When the number of exploiters is 700, the download time window for 63% of the unselfish leechers in the TSS-led overlay is between 3,000 and 5,000 seconds, while this window is between 4,000 and 6,000 seconds for the OSS-led overlay.

3.8 Conclusions and Discussion

In this chapter we establish a mathematical model to analyse how the seeding strategies in BitTorrent affect system performance in the presence of selfish peers. We choose two popular seeding strategies, the Original Seeding Strategy (OSS) and the Time-based Seeding Strategy (TSS), for this study. We categorise selfish peers into two classes: freeriders and exploiters. A series of simulations are then conducted in both homogeneous and heterogeneous network settings.

First, we prove the practical uses of our mathematical model for OSS, which can be used to theoretically study the effects of the seeding strategies on a BitTorrent network. We then discover that when the number of selfish leechers is below a certain value, OSS performs better than TSS. Beyond this value, TSS outperforms OSS by equalising the contributions of seeds to every leecher. It is observed that this approach prevents freeriders from occupying the resources of seeds and successfully makes exploiters serve more blocks to other leechers.

We also discover that both freeriders and exploiters harm the system, despite the seeding strategy that is employed. TSS has better resistance (which means smaller performance degradation) to selfish leechers compared with OSS. Our experimental results are consistent with those shown in [50]: that is the download rates that leechers can obtain are directly proportional to leechers' bandwidths. Furthermore, we find that the selfish leechers can impact negatively on leechers of every level of bandwidth and neither of the seeding strategies can completely eliminate this impact. TSS can reduce the impact more effectively than OSS; this can be more clearly observed when the number of selfish leechers reaches a certain threshold.

In a BitTorrent network we believe that OSS should be employed to boost system performance (higher download performance) if the number of selfish leechers is relatively small; the exact threshold depends on the BitTorrent environment. Beyond this threshold, TSS should be deployed to equalise the contribu-

tions of the seeds into all leechers to prevent selfish leechers from dominating the seeding resources.

With this analysis work, suitable seeding strategies can be selected and optimum performance can be obtained after we employ BitTorrent for the database distribution in ppBLAST.

Chapter 4

Distributed Arbitrary Segment Tree

Distributed Hash Tables (DHTs) can provide functions such as lookup, storage, and self-organising Peer-to-Peer overlays. These functions are essential for ppBLAST. However, most DHTs support retrieval for only single key a time. For range queries, they always exhibit poor performance. ppBLAST involves many range query operations; therefore, we design Distributed Arbitrary Segment Tree structure and lay it on top existing DHTs to deliver high performance on range queries.

4.1 Introduction

This chapter describes the design and implementation of *DAST*, a Distributed Arbitrary Segment Tree structure that supports range queries for public Distributed Hash Table (DHT) services. DAST does not modify the underlying DHT infrastructure, instead it utilises the scalability and robustness of DHT while providing simplicity of implementation and deployment for applications. Compared with traditional segment trees, the arbitrary segment tree used by a DAST reduces the number of key-space segments that need to be maintained, which in turn results in fewer query operations and lower overheads. Moreover, considering that range queries often contain redundant entries that the clients do not need, we introduce the concept of *Accuracy of Results* **(*AoR*)** for range queries. We demonstrate that by adjusting *AoR*, the DHT operational overhead can be improved. DAST is implemented on a well-known public DHT service (OpenDHT) and validation

through experimentation and supporting simulation is performed. The results demonstrate the effectiveness of DAST over exiting methods.

4.2 DHT and DAST

There has been considerable research interest into Distributed Hash Tables in recent years. In addition to offering the advantages of scalability, load balancing and robustness, DHTs allow P2P applications to achieve efficient key insertion, lookup and retrieval over the underlying P2P network [58, 65-67]. Imperative to the success of DHTs is the hashing operation. Each DHT node has a unique node identifier represented with a predetermined number of bits, e.g., a Pastry node has a 128-bit id [66]. The node identifier is typically the hash of the node's public key or IP address, and the set of node identifiers is uniformly distributed. Before inserting a key into the P2P overlay, DHT also hashes the key over the node identifier space so as to locate the node whose node ID is the closest to the hash of the key. Once this mapping is complete, the hash of the key together with the value is stored at the target node.

The ID-based hashing effectively balances the load over all DHT nodes; however, this exact matching mechanism makes range query inefficient because clients can only search and retrieve one key at a time. If clients need to search for all available keys in a certain range, i.e., a *range query*, this is difficult to achieve via DHT lookup directly, since the DHT hashes the keys over the node identifier space before inserting, and the structural attributes of keys, such as the continuity of the key space, are erased by the DHT hashing functions. Consider for example that the keys to be inserted are the integers between 0 and 15. Each key is hashed before it is inserted into the DHT. If clients want to retrieve all keys in the range [3, 5], each key ("3", "4", "5") must be separately identified as even if one key is found, e.g., "4", it is not possible to conjecture the locations of its neighbours ("3", "5") through the hash value of "4" since the hashing is purely random and not structured. If the length of the range is very large, e.g. [2, 2^{10}], then clients have to carry out 2^{20}-1 retrieval operations to obtain all keys, which introduces considerable overheads to the DHT [68] and the efficiency of the query to itself.

To enable DHTs to support efficient range queries, we propose a Distributed Arbitrary Segment Tree (DAST), a data structure that is layered upon a traditional DHT. There exist a number of approaches to implementing a range query. In some designs keys are duplicated or the query results contain unnecessary keys in the interest of query efficiency. Nevertheless, the values associated with the keys are ignored. We believe that the size and type of the data associated with each key is crucial in understanding the efficiency of the query process. It is this data after all which is directly retrieved from the DHT and causes the storage load on the DHT. By considering the values associated with each key, DAST achieves a better balance between load and query performance. Moreover, we use the term *data item* of the form {*key*, *value*} when we describe DAST operations.

DAST constructs an arbitrary segment tree (*AST*), which is an enhanced form of a traditional segment tree [68, 69], to break down the entire key space into a number of segments (each segment is a node in the tree). For every insertion request of a data item {*key*, *value*}, DAST first locates all segments of the tree that contain the key, and then creates new data items in the form {*segmentId*, (*key*, *value*)}, i.e., DAST encapsulates the key and the value in the new data item, with *segmentId* being the new key. Finally, DAST inserts the new data items into the underlying DHT instead of the original data items. To process a range query, DAST looks for a minimum number of segments on the tree so that the union of the selected segments matches the range of the query. This way, by retrieving all *segmentId*s in the union, we obtain the result of the range query. Since every segment contains a number of keys, retrieving by *segmentId*s instead of the original keys can significantly reduce the number of DHT retrieval operations and consequently improve the efficiency of the range query.

A novel concept in DAST is the accuracy of the results for a range query. As mentioned, the efficiency of DAST is determined by the number of segments that constitute the query range. The use of the arbitrary segment tree guarantees that the DAST is able to find the union of segments exactly matching the range. However, if we relax the requirement of an exact match, that is, allow the union of segments to exceed the range of a query for a certain length, then fewer seg-

ments may be needed to cover the range, which in turn leads to fewer "get" operations to the DHT. This said, the query efficiency may not always be improved since the result of the query may contain unwanted data items due to the extra span of segments which may cause more traffic or longer latency. We thus define the accuracy of results (*AoR*) as the number of necessary keys divided by the total number of keys in the response. We analyse the balance between the efficiency of DAST and the value of *AoR* in this chapter. To the best of our knowledge, no existing research has introduced or analysed the *AoR*, which makes our contribution unique.

Significantly, our solution does not require modifications to the core of the DHT; instead, we layer the DAST over a DHT infrastructure and present it as a middleware component between clients and DHTs. As some DHT systems have already become public services [57], this layering approach brings simplicity of implementation and deployment to applications. Note that DAST is a tree based data structure, however, it does not require peers in the network to be organised to any form of overlay structure, i.e., the DAST tree does not need to be maintained as long as the range of the key space is determined. Section 4.4 describes this characteristic of DAST in more details.

The rest of the chapter is organised as follows. We describe related work and compare DAST with this work in Section 4.3. In Section 4.4 we present the details of the DAST algorithms and the concept of the *AoR*. We evaluate the performance of DAST in Section 4.5. Finally we conclude in Section 4.6.

4.3 Other Range Query Support

Range queries are used by many P2P applications, including P2P databases, distributed computing, and file sharing [70-72]. A variety of solutions have been proposed to address the range query problem for DHTs. These solutions can be classified into two broad categories: those that need to modify the core of the DHT, and those solutions that need not.

Mercury [73], SkipGraph [74], SkipNet [75], and PIER [76] are all representative examples from the first category. They either modify or redesign the core of the DHT to achieve a range query. Alternative designs include the Prefix Hash Tree (*PHT*) [60] and the Distributed Segment Tree (*DST*) [68], which represent examples of the second category, and subsequently do not need to know the internal mechanism of the DHT. We describe two examples, PHT and DST, and compare these with our own scheme DAST.

4.3.1 Prefix Hash Tree (PHT)

PHT employs a layer-based tree structure encapsulating the original tuples {key, value} in new data items with the label of the leaf nodes acting as the new key and inserting it into the underlying DHT. Each original key is expressed as a binary string of length *D*. All keys with the same prefix are stored on the same leaf nodes. The depth of the tree is decided by the load balancing mechanism in PHT, i.e., if the number of keys that are stored on a leaf node exceeds a threshold, the leaf node will split into two child leaf nodes.

Clients are not aware of the structure of the whole PHT. To determine which leaf node to insert, clients have to first look up all *D* possible prefix labels in parallel, e.g., if the binary string of a key is “00100”, a client has to perform parallel “get” operations to the DHT for the keys “0”, “00”, “001”, “0010” and “00100”; if one of the “get” operations returns a result, then the leaf node is located and the key is stored on it via a data item. The authors of PHT also suggests a binary search solution for locating the leaf node [60]. For the query of range (L, H), PHT first locates the PHT node corresponding to the longest common prefix of L and H and then performs a parallel traversal of its subtree to retrieve all the desired data items as the result of the query.

DAST differs from PHT in the following ways. First, the depth of the PHT grows with the increase in inserted keys, i.e., the structure of PHT keep changing over time and as a result additional “get” operations are required for each insertion operation. In contrast, the structure of DAST is stable as long as the entire

key space does not change. Clients locate the destination tree nodes for keys without any additional "get" operations, which results in lower latency for range queries. Moreover, as will be described in Section 4.2, the result of a range query in PHT may contain unnecessary data items, which may increase the latency. In comparison, DAST gives criteria for the accuracy of results. We find that with similar *AoR*, DAST requires fewer DHT operations and thus achieves lower latency for range query than PHT.

4.3.2 Distributed Segment Tree (DST)

The Distributed Segment Tree approach is the most similar to our work. Both DST and DAST use the concept of a segment tree [69], nevertheless, DST is a binary tree while each node on DAST may have more than two children. Each non-leaf node in DST has two children and the segment corresponding to the parent is split into two equal parts and assigned to the two children, respectively. Hence the entire key space is split into 2^i (i represents the level in the tree, counting from 0) parts on each tree level and the depth of the tree is $O(\log R)$ (R is the length of the entire space range). Therefore, keys need to be inserted to $O(\log R)$ DST nodes and there will thus be $O(\log R)$ duplications for each key (the number of duplications is not always $O(\log R)$ due to DST's load balancing mechanism which we describe in Section 4.4). The nodes in a DAST can have more than two children and through setting the maximum number (M) of children that each node can have, there will be arbitrary number of segments on each tree level (this is where the name "arbitrary segment tree" derives) and the depth of the tree is $O(\frac{\log R}{\log M})$. Consequently, each data items in a DAST will have $O(\frac{\log R}{\log M})$ duplications, which leads to lower DHT storage load and operational overheads. DAST also adopts a load balancing algorithm that achieves similar effects to the one in DST, but with a considerably simpler implementation. Finally, DAST incorporates the concept of the *AoR* to further improve the range query performance. DAST also provides clients with the flexibility to adjust the primary properties to suit their own range query requirements. Such an approach is not documented in DST or PHT.

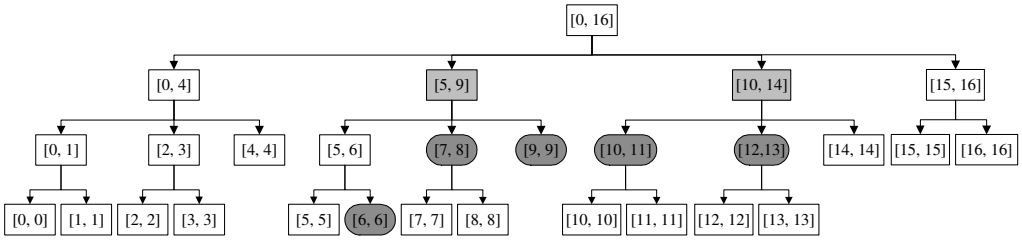


Figure 4-1. An example AST with the segment tree range [0, 16] and M = 4. We choose the segment tree range such hat each node can have an arbitrary number of children and the segments are uniformly split in each level while maintaining appropriate span length. An exemplar query for range [6, 13] is also illustrated here. The query union can be {[6, 6], [7, 8], [9, 9], [10, 11], [12, 13]} with AoR 100% or be {[5, 9], [10, 14]} with AoR 71.4%.

4.4 Design of DAST

In this section we present the design of DAST. We first introduce the Arbitrary Segment Tree data structure and then describe how to layer an AST over an existing DHT infrastructure to achieve range query functionality.

4.4.1 Arbitrary Segment Tree

The Arbitrary Segment Tree (*AST*) is based on the traditional segment tree (*TST*) data structure [69], where a range (henceforth we use the term *segment tree range* to distinguish from the range in a query) of non-negative integers[2] is iteratively split at each level into certain number of segments, and each segment is assigned to one tree node. However, the rule of splitting the segment tree range on

[2] The range that a segment tree represents can in fact include real numbers. In this paper, we only give examples of non-negative integers for practical purposes.

Algorithm 4-1: The pseudo code of the AST construction algorithm

```
// Parameters:
// ASTNode: the class of AST nodes.
// sf, st: bounds of the interval for the segment on the node.
// level: the tree level of the node
//ASTNode.children[]: an AST node's children.
//M: the maximum number of children; a global value.
//C: the number of children of a node

ASTNode(sf, st, level)
    C ← 0
    children ← new ASTNode[M]
    if sf ≠ st then
        from ← sf
        length ← (st - sf) / M
        to ← from + length
        while true do
            children[C] ← new ASTNode(from, to, level+1)
            C ← C + 1
            if to = st then
                break
            else
                from ← to + 1
                  if st > from + length then
                    to ← st
                  else
                    to ← from + length
```

Algorithm 4-2: The pseudo code of the dividing algorithm for the range of the query

```
// Parameters:
// rf, rt: bounds of the interval of query range
// cdt: the candidate segment for the union of range segments.
// newCdt: new candidate segment.
// cdtClt: the collection of candidates (cdt).
// cf, ct: bounds of the interval for the candidate segment.
// nf, nt: bounds of the interval for the current AST node.
//nri: number of redundant data items allowed in query results.
// results: the union of segments that match the range.

divideRange(rf, rt, AoR)
    cdtClt.add(interval(rf, rt))
    for each level on the tree do
        for each AST node on the level do
            if candidates is empty then
                return results
        else
         for each cdt in cdtClt do
            nri ← (cdt.to ← cdt.from) × (1-AoR)
                    newCdt ← interval(cdt.from-nri, cdt.to + nri)
            if newCdt covers the current node then
                    results.add(the segment of current node)
                if cf < nf then
                    cdtClt.add(interval(cf, nf-1))
                if ct>nt then
                    cdtClt.add(interval(nt+1, ct))
                cdtClt.remove(cdt)
                break
    return results
```

each level in AST is different from that found in TST (*Traditional Segment Tree*). TST is a binary tree where every internal node has two children. Therefore, starting from the tree root, the segment that every internal node represents is evenly split into two parts and allocated to the two children, respectively, until it has only one number within. In contrast, AST is a multiway tree in which each internal node can have an arbitrary number[3] of children. We denote M as the maximum number of children that one node can have, i.e., each AST node can have at most M children. Note that AST is a superset of TST, i.e., when the value of M is 2, an AST becomes a TST. At each tree level, AST splits the segment tree range uniformly to up to M segments while maximising the interval size of each segment. The properties of AST are as follows:

- Assuming the length of the segment tree range is R, the height of an AST is $O(\frac{\log R}{\log M})$.
- The root node has the entire segment tree range. Every other node represents a segment. The union of all segments on the same tree level is the segment tree range.
- Every non-leaf node has C_i children, where $1 < C_i \leq M$. The segment of each non-leaf node is split into C_i parts and distributed to the children. The value of C_i and the intervals of the segments for the children are decided by the tree construction algorithm (algorithm 4-1).
- Every leaf node has an atom segment, i.e., a segment that contains only one key. The union of all leaf nodes covers the segment tree range.
- Every node has a *segmentId*. DAST produces the *segmentId* by hashing its interval over the underlying DHT node ID space. Through the hash, the *segmentId* can be mapped to the node ID space of DHT and then used in the DAST operations.

Unlike PHT, an AST will not change its structure once it has been constructed, as long as the segment tree range does not change. This property ensures

[3] The number cannot be "1" because splitting a segment cannot be performed if a node has only one child.

the consistency of the positions for keys, i.e., the destination node that holds the (key, value) items. Fig. 4-1 depicts an example AST where the segment tree range is [0, 16] and the value of *M* is four. Note that the numbers of children of internal nodes vary between two and three through the tree and are purely determined by the segment tree range and the choice of the value of *M*.

4.4.2 DAST operations

The DAST data structure provides an interface between the client applications and the underlying DHT. Clients insert, delete or retrieve data items to or from DAST instead of DHT. We describe the DAST operations needed to achieve the range query functionality for clients.

- *Insert/Delete*: The insertion and deletion of a data item with a key in DAST is straightforward. When an insert request arrives, DAST looks for all nodes whose segments cover the key of the item (there must be one and only one such node on each tree level). For each of these nodes, DAST creates a new data item in which the key is the *segmentId* of the node and the value is the original data item. Finally, DAST inserts the new data items to DHT. The insert operation for one key in DAST needs $O(\frac{\log R}{\log M})$ DHT insertions and there will be $O(\frac{\log R}{\log M})$ copies of the key inside the DHT. When a data item is deleted, DAST finds all segments that cover the key and removes the data items accordingly.
- *Range query*: DAST first divides the range of the query into a union of segments that the AST contains, and then retrieves all *segmentIds* with associated data items from the DHT. The dividing algorithm is shown in algorithm 4-2. Since the AST ensures that leaf nodes have atom segments, the union of the segments is guaranteed to be found for the range. There may exist alternative ways to divide the range; however, our algorithm is dedicated to building a union containing a minimum number of segments, i.e., the intervals of the segments should be as wide as possible, so as to reduce the number of DHT retrievals.

- *Single key query*: DAST performs single key queries by simply retrieving the corresponding atom segment from the DHT.

4.4.3 The Value of *M*

The value *M* controls the maximum number of children an AST node can have. The key advantage that AST has over TST is that it provides more flexibility for clients to improve the performance of a range query. As previously described, the height of an AST is $O(\frac{\log R}{\log M})$ and hence a greater value of *M* leads to lower numbers of DHT insertions (improving performance of the DAST insertion) and less duplications of data items (reducing the DHT storage load). However, if *M* is too large, the segment of one node will be split into more parts and consequently the segments in the AST will be shorter. Therefore, when fulfilling a range query, the average number of segments in the union that covers the range will be greater. In other words, DAST has to perform more DHT retrievals to obtain the result. Due to this trade-off, clients have to carefully choose the value of *M* depending on their definition of the key space and their expectations for the lengths of the ranges that the queries may have. We investigate the impact of *M* on the performance of DAST in Section 4.5.

4.4.4 Accuracy of Result for a range query

We consider the Accuracy of Result (*AoR*) for a range query in DAST. This investigation is motivated by the fact that when using PHT we found that the responses of range queries may contain unnecessary data items, since one prefix tree node stands for a prefix of keys and consequently keys that do not belong to the same range may fall into one prefix node. This causes higher latency to the query responses and cannot be rectified because PHT does not modify the DHT layer and so cannot filter the query results before feeding them back to the clients. By default, DAST always returns the query results to clients with 100% accuracy, i.e., the responses of the query do not contain any unwanted data items. However, we found that if we relax the segment union for the query (to be larger than the

range of the query), i.e., the span of the union covers the range but has extra intervals on either end or both ends, the number of segments in the union may be reduced. Consequently, a number of unnecessary data items will exist in the results, however, the number of DHT retrievals needed for range queries will also drop. An exemplar range query [6, 13] is illustrated in Fig. 1. DAST builds a union {(6, 6), (7, 8), (9, 9), (10, 11), (12, 13)} for the query [6, 13] by default and has to perform five DHT retrievals for the result. If we relax the union construction to be {(5, 9), (10, 14)}, the result may contain only two extra items (5 and 14) but the number of retrievals drops down from five to two, which is 2.5 times lower than before.

Achieving a range query in DAST usually requires a number of DHT retrievals and these DHT retrievals are executed in parallel which significantly reduces the response latency. However, if clients submit range query requests to DAST simultaneously with high frequency, DAST has to in turn submit the retrieval operations for those range queries to the underlying DHT in parallel and the DHT may suffer high overheads in a short period of time (PHT also considers the overhead for a DHT when choosing a binary search or parallel search for a lookup, although there is no detailed analysis in the associated paper). To help the DAST clients reduce the overhead imposed on the DHT, we present the concept of the accuracy of result (*AoR*) for a range query. We will show that by adjusting the value of *AoR*, the number of DHT retrievals for range queries can be much reduced and the overhead on DHT can therefore be lowered. The *AoR* is defined as the number of necessary data items divided by the total number of data items in the result of a query. In the example above, the value of *AoR* is $\frac{5}{5+2} = 71.4\%$ after tolerating unnecessary items in the result. The implementation of *AoR* is demonstrated in Algorithm 4-2.

The *AoR* in a DAST range query is 100% by default since DAST builds a segment union that can precisely match the range of the query and the resulting response consists of only necessary data items. Clients can choose the desired *AoR* value to be less than 100% to suite their application environments. Note that

the desired *AoR* acts as a threshold in DAST, i.e., the actual *AoR* of range query may not precisely equal the desired one but it is guaranteed not to be lower. This is because we assume every key in the key space as having a data item in Algorithm 4-2, and calculate the *AoR* by the number of key slots not the number of actual items. In real range query cases, since some key slots may be empty, the actual *AoR* must be equal to or greater than the desired one. We demonstrate the relationship between *AoR* and the number of DHT retrievals in Section 4.5.

4.4.5 Load Balancing

Approaches based on segment trees have potential problems on load balancing. There are fewer nodes at the higher tree levels; however, these nodes are responsible for more data items, as each data item has to be inserted into every tree level. The extreme case occurs at the root node. Since the root node has the entire key space, it will have to maintain a copy of every data item. The actual DHT node thus experiences a heavy storage load.

DST [68] employs a load balancing mechanism, called *downward load stripping*. Each node maintains two counters for its children, the left one and the right one. If, when a key is inserted into a node, it can also be covered by one of its children, the corresponding counter is increased by one. When either counter reaches a threshold, the node stops receiving keys. What this mechanism actually does is to limit the high level nodes from having more data items than the threshold. However, it brings to the implementation the problem of how do clients locate the values of two counters for each DST node in such a distributed environment? The obvious solution is to put the counters into the underlying DHT as data items and let clients access them through specified keys. However, this solution will occupy extra DHT storage and the insertions or retrievals of the counters themselves take time. Consequently, concurrency or synchronisation problem may occur, e.g., one node may not stop receiving data items when it should, because the counters are not updated on time.

Load balancing is nontrivial in DHTs [77] and cannot be perfect since even if the keys are uniformly distributed onto the DHT nodes, some nodes will be responsible for a logarithmic factor more of the key space than others [58]. In other words, some nodes in the DHT will assume much higher storage and routing load than others. Due to inheritance, PHT, DST and DAST also suffer from the same problem. Even though data items are inserted at leaf nodes in PHT and it is easier to distribute leaf nodes uniformly into DHT nodes, some data items within a certain range may still gain high popularity and become responsible nodes and hence will have uncharacteristically heavy load; this is also true for the DST.

Therefore, we propose to reduce the effects of load in DAST but not to perfectly eliminate it. We ignore the nodes in the levels N - 1 and above in the AST and start to insert data items at level N. The value of N depends on how large the entire segment tree range is and how many nodes there are in the underlying DHT. We encourage applications or clients to carry out experiments to test their values of N before the deployment. Our evaluation in Section 4.5 provides suggestions as to how to choose a good value for N.

4.4.6 Tree Maintenance and Fault Tolerance

As described above, DAST is a data structure layer between the peer-to-peer overlay and the DHT infrastructure. When a peer carries out a range query operations, it gives the operation command to DAST and its associated algorithms and obtain the results from the DAST layer. Thus, the DAST data structure exists inside only the application functions which peers use to carry out the range query operations, and it does not influence the peer-to-peer overlay structure or the DHT infrastructure. Moreover, algorithm 4-1 and 4-2 show that the DAST structure will remain constant as long as the range of the whole key space does not change. Thus, the DAST does not require any maintenance work and this brings simplicity to the applications.

Since DAST is built upon a DHT service layer, it inherits all of the resilience and failure recovery properties of the underlying DHT. Although the DHT

has methods to guarantee a certain level of data availability and fault tolerance [57], the DAST can still lose data if all replicas in the DHT disappear (all nodes with those replicas leave the network). To avoid this catastrophic failure, the DAST employs a soft state refreshing mechanism. Each data item that is inserted to the DHT through the DAST layer has a time-to-live (TTL) associated with it. Peers have to regularly update the data items regularly by a TTL of seconds; otherwise, the data items are automatically deleted from the DHT. Hence, even if all replicas for a data item in DHT are lost, the item will be restored by the refreshing mechanism eventually.

4.4.7 Memory Requirement

In a DAST tree, a node will have at most M children. If a node's id is 160 bits (we take this value from [36]) which is 20 bytes, then a node's size in memory is $(M + 1) \times 20$ bytes and the memory size of the whole tree is $N \times (M + 1) \times 20$ where N is the number of nodes in the peer-to-peer network. Taking $N = 50,000$ [48] and $M = 6$ as an example, the whole tree's size is 7MB. Considering the hardware specifications of current PCs, this memory requirement is fully acceptable.

4.5 Evaluation

In this section, we evaluate the performance of DAST. First we investigate the internal structural properties of DAST. We then compare the range query operations of DAST, DST and PHT. Finally we compare their range query efficiencies in an OpenDHT deployment.

4.5.1 Implementation

We implement two versions of DAST, the first as a simulation and the second as a full-scale deployment. In both versions, the source code for the mechanisms of DAST are exactly the same. The only difference is that the simula-

tion version of DAST utilises a Java Hashtable object to simulate the underlying DHT, while the deployed version is layered on top of OpenDHT.

To shorten the time to conduct the experiments, we use the simulation version to investigate the structural properties of DAST and compare the range query operations of DAST, DST and PHT. For comparisons of the real range query efficiencies, we use our deployed version of DAST that accesses OpenDHT service on the Internet.

4.5.2 Setup

In the simulations, we assumed the segment tree range to be $[0, 2^{16}-1]$ and generated 2^{14} keys for insertions. The keys are uniformly distributed over the segment tree range space. The values associated with the keys are empty, i.e., the sizes of the values are zero. This is because the sizes of the data items do not affect the investigation of the internal mechanisms of DAST, DST or PHT - such a configuration also improves the simulation efficiency. We also randomly generate five sets of range queries, each of which has 1000 queries with span lengths (the length of the query range) of 512, 1024, 2048, 4096 and 8192, respectively.

In the deployment, the segment tree range remains the same but we generate only 2^{10} random keys. We chose a relatively small number of insertions because 2^{10} insertions are enough to demonstrate the insertion efficiency of all three approaches. Every key has 1KB of data associated with the value (the maximum size of a value in OpenDHT is 1KB). All the experiments were prototyped on a single PC to guarantee the correctness of the comparison of results. The range query setup is similar to that in the simulation except that each query set consists of 100 queries.

Each of the simulation experiments was conducted 100 times and the experiments on the OpenDHT deployment were repeated 30 times.

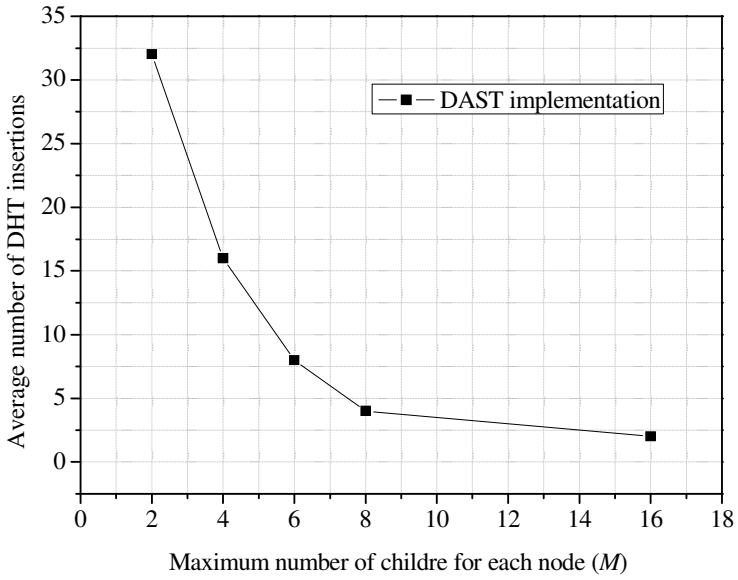


Figure 4-2. Plots of DHT operations for different values of *M* (Maximum number of children): the plot of the average number of DHT insertions for one DAST insert request;

4.5.3 Structural Properties of DAST

We study the number of children allowed in AST, the load balancing mechanism and the performance impacts from different values of *AoR*. Clients can choose their own settings to suit the demands or adapt to the different computing environments.

Maximum number of children (the value of *M*): As described in Section 4.4, the value of *M* controls both the number of DHT insertions and the number of DHT retrievals for range queries. Recall that the height of AST is $O(\frac{\log R}{\log M})$, if *M* is too large, AST may have only a very small number of levels (the extreme case is that the whole AST has only the root node when *M = R).* Thus to maintain the AST we choose the candidate *M* to be 2, 4, 8, 16 and 32. For each of the DAST examples with those *M* candidates, we insert the preloaded keys (for now we do

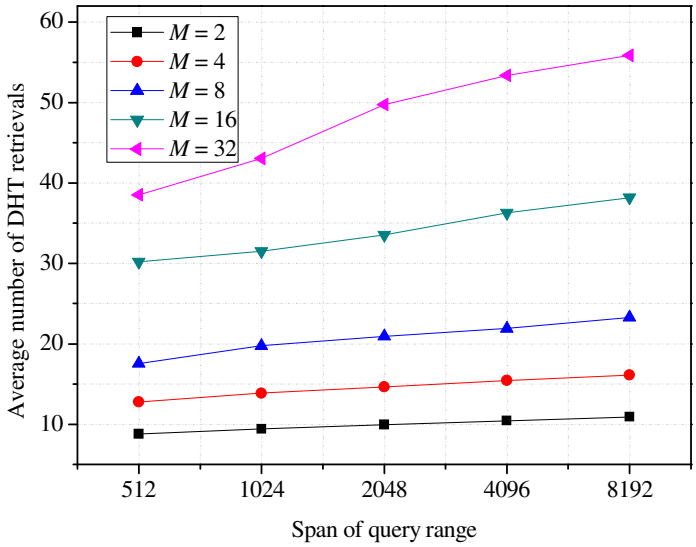


Figure 4-3. Plots of DHT operations for different values of *M* (Maximum number of children): the plot of the average number of DHT retrievals for one DAST range query request

not consider the load balancing problem and *AoR*) and plot the average number of DHT insertions involved. As depicted in Fig. 4-2 the number of DHT insertions drops sharply when *M* increases from 2 to 4 and this trend slows as *M* increases. When *M* reaches 16, the number of DHT insertions remains constant. To see how the value of *M* affects the range query, we send the five sets of predefined range queries to DAST and plot the results in Fig. 4-3. We can see that the higher the value of *M* leads to a larger number of DHT retrievals. The distance between the curves for $M = 8$ and $M = 16$ is large, indicating a sudden increase of DHT retrievals. Comparing Figure 4-2 and 4-3, we thus suggest that $M = 4$ is the optimal in our experiments.

Load balancing (the value of *N*): Our load balancing mechanism is simply that we start to insert data items from tree level *N* (if the root node is on level 1). The top *N* levels therefore contain no items. Using the result of $M = 4$ from the previous experiments, and testing *N* values from 1 to 6, the insertion and query

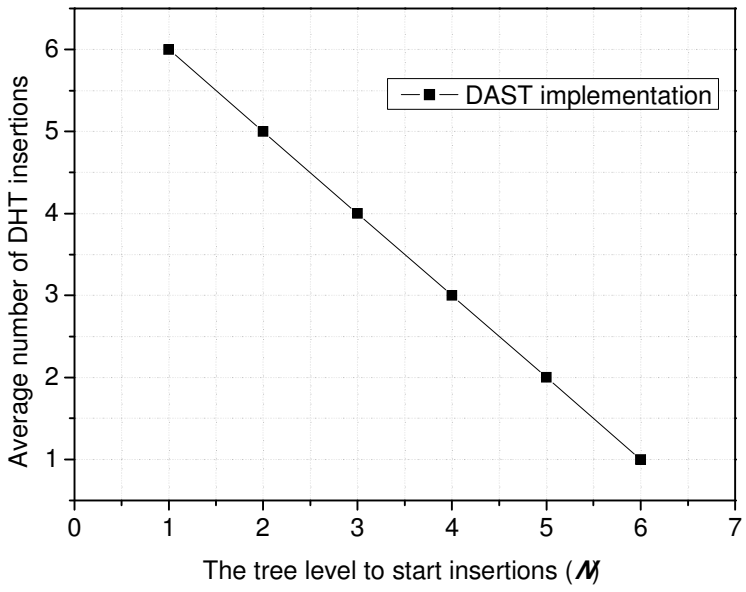


Figure 4-4. Plots of DHT operations for different values of *N* (the level number that DAST starts to insert data items): the plot of the average number of DHT insertions for one DAST insert request

range results are plotted in Fig. 4-4 and 4-5 respectively. The number of DHT insertions is reduced by one if *N* increases by one, which is apparent in Fig. 4-4. In Fig. 4-5, it is not easy to see the plots where $N = 1, 2, 3$ because they are overridden by $N = 4$, which implies the results for these three values of *N* are similar. When $N = 5$, the number of DHT retrievals starts to rise as the nodes on levels 1 to 4 do not have data items and cannot contribute to the range query. The increment is more pronounced when the value of *N* reaches 6. The results in Fig. 4-5 narrow our choice of *N* down to 4 or 5. We do not consider $N \leq 3$ because $N = 4$ gives better load balancing while providing a similar number of DHT retrievals. We present the detailed experimental results for $N = 4$ and $N = 5$ in Table 4-1 to illustrate choosing the optimal *N*. As we can see, the differences between the numbers of DHT retrievals of the two cases become larger when the span of the query increases. However, we should also note that the number of nodes on tree level 5 is 256, which is four times more than that on level 4. Considering that

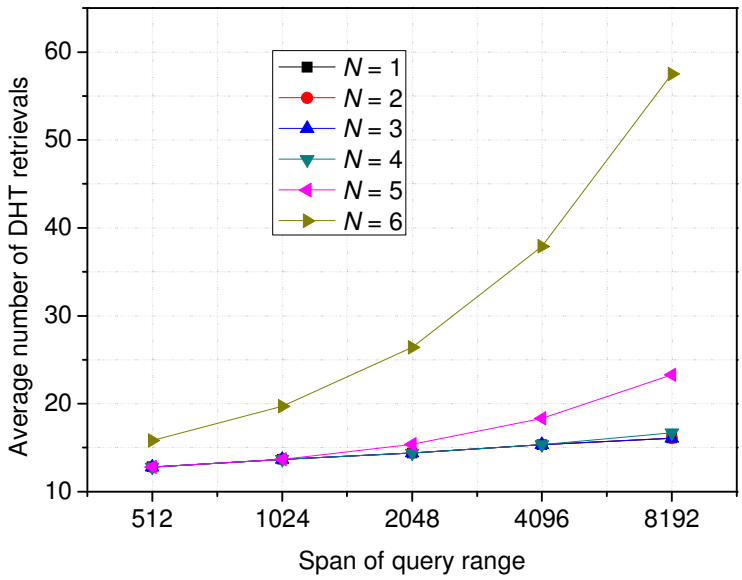


Figure 4-5. Plots of DHT operations for different values of *N* (the level number that DAST starts to insert data items): the plot of the average number of DHT retrievals for one DAST range query request

lowering *AoR* in DAST can further reduce the number of DHT retrievals, we thus chose 5 as the optimal value for *N*. This conclusion is validated in the next experiments considering *AoR*. Note, $N = 5$ is not universally optimal and clients should test for their own value of *N*.

The accuracy of the result for range query (*AoR*): To provide an analysis from the point of view of the *AoR*, we queried DAST for the same range sets seven times and each time we tested a different value of *AoR*. The value set of *AoR* are shown in Fig. 4-6. We do not present the results when $AoR < 70\%$ because these plots are masked by the plot for $AoR = 70\%$, which means the value of *AoR* stops affecting DAST when it is below 70%. As we can see in Fig. 4-6, the number of DHT retrievals needed for the range query drops along with the reduction of *AoR*. We confirm the precise pertentage of the drop (compared to that when $AoR = 100\%$) with the corresponding value of *AoR* in table 4-2. Through

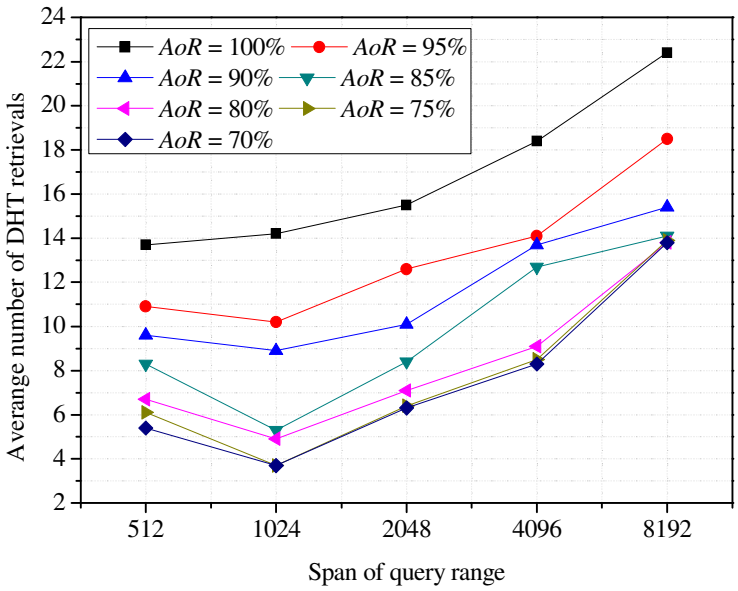


Figure 4-6. Plot of the average number of DHT retrievals for one DAST range query request with different values of *AoR* (the accuracy of result).

comparisons, we can see that if we reduce the value of *AoR* by even 5%, the number of DHT retrievals drops significantly (by 21.62%). If we allow 30% of the result to be unnecessary (*AoR* = 70%), the number of retrievals drops further to 57.43%.

Clients should be aware that lowering the value of *AoR* can also affect the response latency of the query depending on the sizes of the data items. If the size of the data item is small in the client application and the frequency of the range query request is high, having *AoR* of 70% can result in an approximate 50% lower overhead to the underlying DHT and may not negatively affect the response latency. Even if the size of the data items is large and the frequency of the request is high, allowing *AoR* to be 95% is worth considering since it still results in over 20% lower overhead to the DHT. A detailed analysis of the tradeoffs among the data size, overhead and *AoR* is required; this is precluded in this chapter as the implementation and evaluation of DAST is done entirely on a third party DHT layer.

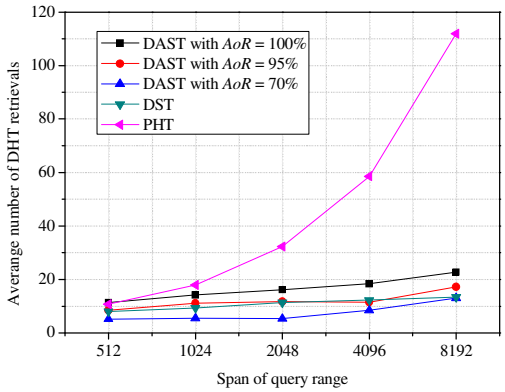


Figure 4-7. Comparison of DAST (with different *AoR*) against DST and PHT on average number of DHT retrievals for one range query.

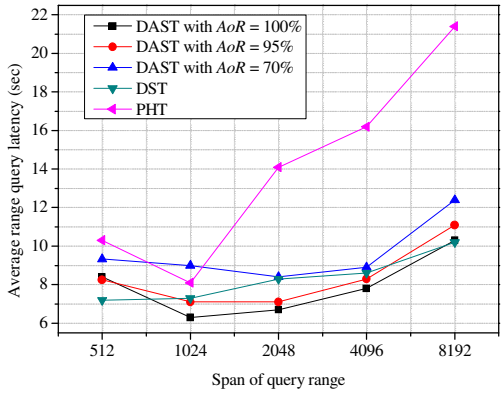


Figure 4-8. Comparison of DAST (with different *AoR*) against DST and PHT on query latency.

The results here provide suggestions rather than quantitative conclusions for reducing the potential DHT overhead through adjusting *AoR* in DAST.

4.5.4 Range query operations in DAST, DST and PHT

We compare the number of DHT operations (insertions and retrievals) that are needed for range queries in DAST, DST and PHT. The parameter settings M = 4 and N = 5 are selected for DAST and a block size of 60 is chosen for DST and PHT, which means that on each of the DST and PHT nodes they can have at most

60 data items stored (these settings replicate those found in related literature [68]). We insert the same set of data items to DAST, DST and PHT, and execute range queries using the same query sets in each of the three approaches. In DAST however, we also conduct range query experiments for three different values (100%, 95%, and 70%) of *AoR*; these results can be found in Figure 4-6.

For an insertion request of one data item, PHT always requires only one DHT insertion, however, it requires a number of DHT retrievals for the lookup of the leaf node. For PHT, we hence add the number of DHT retrievals for the lookup to the one DHT insertion and treat the sum as the number of DHT operations needed for one data item insert request. The simulation results indicate that the average numbers of DHT operations for one data item insert request are 5, 13, and 8, respectively for DAST, DST and PHT. DST requires on average 13 DHT insertions for one data item insert request and duplicates the data item 13 times in the DHT storage. DAST requires less than half the DHT insertions and one data item requires only 5 copies in DHT, which significantly reduces the storage load in DHT. PHT on the other hand needs only one DHT insertion and requires only one copy of a data item. However, it requires on average 7 DHT retrievals, which imposes a higher operational overhead than DAST. To conclude, DAST is demonstrably superior to DST for insert requests and trades extra storage for insertion performance when compared to PHT.

Fig. 4-7 depicts the simulation results for the range queries. For one range query, PHT performs many more DHT retrievals than DAST and DST, which represents potentially high DHT overheads. When DAST is configured with *AoR* set to 100% it requires more DHT operations than DST. This is because each DST

Table 4-1: The experimental results of load balancing evaluation for $N = 4$ and $N = 5$

N	# of nodes	Query span				
		512	1024	2048	4096	8192
4	64	12.8	13.7	14.4	15.35	16.7
5	256	12.8	13.7	15.4	18.4	23.3

node has fewer children and the splitting of segments is slower than in DAST; DST therefore has longer segment spans, leading to fewer query unions of segments and fewer DHT retrievals. Nevertheless, when the *AoR* of DAST is set to 95%, DAST achieves approximately the same number of DHT retrievals as DST. When *AoR* is configured to 70%, DAST surpasses DST.

Table 4-2: The experimental results for *AoR*

AoR	Average # of retrievals	Dropping percentiles for *AoR*	Dropping percentiles for # of retrievals
100%	16.84	N/A	N/A
95%	13.26	5%	21.62%
90%	11.54	10%	31.78%
85%	9.76	15%	43.19%
80%	8.32	20%	51.94%
75%	7.72	25%	55.98%
70%	7.5	30%	57.43%

PHT does not always achieve 100% *AoR* in the results of the range queries. We calculate the *AoR* and the average number of DHT retrievals for PHT and DAST responses, and present the results in Table 4-3. Through comparing the values of *AoR* in PHT and DAST together with the average number of DHT retrievals, we can see that DAST performs fewer retrievals while maintaining higher *AoR*.

4.5.5 Comparison of the latencies for insertions and range queries in DAST, DST and PHT

In this experiment, we deploy our DAST implementation on OpenDHT together with DST and PHT. We insert the preloaded data items into OpenDHT through DAST, DST and PHT, respectively. The latency of every insertion is recorded and the average of these values is presented in Table 4-4. The results clearly indicate one DAST insertion takes on average only 67% of the time that DST insertion requires. The advantage of DAST over PHT is more pronounced in that PHT insertions take twice as long as DAST insertions.

For the range query experiment, we deploy three versions of DAST, each of which is configured with *AoR* as 100%, 95% and 70%, respectively. With different values of *AoR*, we investigate the impact of *AoR* on the query latency. These results are presented in Fig. 4-8; the average latencies of the range queries can be found in Table 4-4. We can see that the average latency in DAST with *AoR* as 100% is very close to the one in DST. When *AoR* is reduced, the latency grows due to extra unwanted items in the results. PHT requires more time for range queries because it needs several sequential steps to lookup the leaf key, and the response contains unnecessary items. DAST does not have sequential operations and thus performs better.

4.6 Conclusions

In this chapter, we proposed a Distributed Arbitrary Segment Tree (DAST), a structure built on top of public DHT services to achieve enhanced range query functionality for clients. DAST incorporates the Arbitrary Segment Tree (AST), yet is designed so that the query union contains a smaller number of segments leading to fewer DHT operations and a lower overhead. In addition, the duplications of data items are significantly reduced in DAST as compared with DST. Moreover DAST introduces the concept of *AoR* (Accuracy of Result). By adjusting the value of *AoR*, we demonstrate that DAST can further reduce the number of DHT operations and therefore further reduce the overhead.

Table 4-3: The comparison of *AoR* between PHT and DAST

	PHT					DAST		
AoR	79%	86%	92%	96%	98%	70%	95%	100%
# of DHT retrievals	10.7	17.9	32.3	58.6	111.9	7.5	12.0	16.5

Table 4-4: The experimental results for the average latencies of insert and range query in DAST, DST and PHT

	DAST ($M = 4, N = 5$)			DST	PHT
	AoR =100%	*AoR*=95%	*AoR*=70%		
Insert (sec)	4.5			6.7	9.6
Query (sec)	7.9	8.37	9.606	8.32	15.88

An advantage of this scheme is that DAST does not modify the underlying DHT and instead acts as a middle layer between DHT and the applications that require range query functionality. The approach is also designed to provide DAST users with the flexibility to modify DAST to their application environments for best range query efficiency. Furthermore, the DAST structure is deterministic once the range of the whole key space is decided and therefore does not need any maintenance work which brings simplicity and less overhead to the client applications.

Validations are undertaken through both simulation and extensive real-world experimentation and the results demonstrate the effectiveness of DAST across a range of metrics.

DAST has delivered range queries support to the underlying DHT, which can be used for ppBLAST to efficiently lookup a range of keys and further improve the storage performance. Also, the DHT behind DAST organises peers for ppBLAST to build the self-constructing overlay. Thus, DAST is the important fundamental part of ppBLAST.

Chapter 5

ppBLAST: A BLAST Service over Peer-to-Peer networks

We now turn our attention to ppBLAST. In Chapter 3, we present the analysis of BitTorrent seeding strategies. It guides us to choose proper seeding strategy for BitTorrent which we can employ for ppBLAST database distribution process. Chapter 4 introduces our DAST over DHT. DAST-DHT provides lookup of range queries and Peer-to-Peer storage functions, therefore ppBLAST can utilise it to search ranges of peer candidates for BLAST tasks and store BLAST queries and results in the overlay for future retrieval. After these two essential layers are well formed, we begin integrating them to ppBLAST with a novel BLAST task scheduling algorithm and finalise the design of ppBLAST.

5.1. Introduction

BLAST is a popular toolset for protein and nucleotide sequence comparisons. Recent estimates show that these databases of protein and nucleotide sequences are set to grow exponentially and as a result local BLAST searches on single PCs cannot satisfy recent data processing requirements. Traditional approaches for the parallelisation of BLAST are focused on utilising clusters. However, scaling clusters to adapt potential growth of BLAST requests will significantly increase the cost and complexities of maintenance and administration. This chapter introduces ppBLAST, a BLAST service utilising the free computing resources in the peer-to-peer overlay on the Internet. ppBLAST uses several peer-to-peer technologies such as DAST, DHT and BitTorrent-like distribution and layered design. In this chapter, the feasibility of ppBLAST is validated through a small-scale deployment.

The performance of ppBLAST is evaluated; we show that the execution time of a BLAST job can be considerably shortened if the scale of the supporting peer-to-peer overlay is large enough.

The remainder of this chapter is organised as follows: Section 5.2 documents related work; Section 5.3 describes the design of ppBLAST; Section 5.4 presents a performance evaluation of ppBLAST; Section 5.5 concludes the chapter and discusses future work.

5.2 Additional Related Work

This section is divided into three parts: (1) we briefly describe existing approaches that attempt to parallelise BLAST and boost its performance; (2) we introduce the designs of SETI@home and Folding@home and compare them to ppBLAST; (3) several recent techniques, which support job scheduling on large data sets in distributed environments, are illustrated.

5.2.1 Parallelising BLAST

TimeLogic [21] has announced an FPGA-based hardware accelerator called DeCypher BLAST which cooperates with their TeraBLAST software. This method of parallelisation takes place during the sequence alignment itself, i.e., parallelising the comparison of a single query sequence with a single database sequence entry. BioScan [22] is also focused on parallelising BLAST at the hardware level.

Splitting BLAST queries, instead of databases, was proposed in [23, 27]. The query segments are distributed among nodes in a cluster or CPU on a symmetric multi-processor (SMP) system, each node then executes a BLAST comparison between the query segment and the entire database. By following this design, BLAST on each node still needs to load the entire database from a local storage system into core memory. Hence, if the database does not fit into the

memory, BLAST will suffer from the virtual memory bottleneck and the optimal performance cannot be obtained.

Segmenting BLAST databases are widely adopted [18, 24, 25, 78-80]. This approach requires clusters to cooperate on the parallelisation. Although all of these solutions claim high performance for BLAST search jobs, they require high cost clusters, complex maintenance and administration and suffer from scalability problems due to the growth of the databases and increasing BLAST search requests.

ppBLAST differs from all the above approaches. It parallelises BLAST at a software level and does not require any special hardware modifications or upgrades. It utilises peer-to-peer networks, where the computing resources are free, and addresses the scalability problems through the participation of peers.

5.2.2 SETI@home and Folding@home

The design philosophies of SETI@home and Folding@home are similar. There are a set of central servers connected to the Internet which act as coordinators for peers and job dispatchers. Peers who wish to contribute to the projects connect to the servers through the client applications. Once the connections have been established, the servers issue computation jobs to those peers and the results are returned once the jobs are finished.

The design of ppBLAST differs from both systems. ppBLAST does not have central servers to coordinate peers. The peers rely on range query supported DHT to be self constructed and organised. Users who wish to use the service submit BLAST jobs through the ppBLAST service broker and the program runs in a users' local machine instead of in a central server. Thus, ppBLAST does not have single point of failure and is ultimately scalable.

5.2.3 Recent distributed data intensive computing techniques

Several recent techniques exist in order to aid data intensive computing. MapReduce [81] was a programming model, invented by Google. It helps programmers who are not familiar with distributed computing to easily develop applications in environments that are based on cluster or multi-clusters over different locations. Apache Hadoop [82] is also a Java programming framework which supports data intensive distributed applications for clusters. Since both techniques are focused on clusters, we do not describe them in details. BOINC [83] is a open source software for volunteer and Grid computing. Science projects can be created through BOINC and volunteer PCs can contribute their storing and computing powers to those projects through BOINC client application. The basic design of BOINC is similar with the one of SETI@HOME and Folding@HOME. In fact, both projects are now using BOINC as their application support. We described the differences between both projects and ppBLAST in sub-Section 5.2.2, thus we do not repeat here.

5.3 Design of ppBLAST

We begin by recalling the characteristics of BLAST. Based on this the workings of ppBLAST and its components are described in detail.

5.3.1 Characteristics of BLAST

The BLAST toolset has five programs: blastn, blastp, blastx, tblastn, tblastx. Each of these corresponds to one specific comparison, e.g., blastn is used for the comparison of nucleotide sequences. All five programs have similar run-time features; a typical BLAST would involve the command line of *BLAST query database*[4]. The query and database are all text-based. After the run completes, the results are written to a text file. Because of the simplicity and success of the

[4] BLAST programs have a number of other parameters. As these are not relative to our design, we ignore them in this chapter for simplicity.

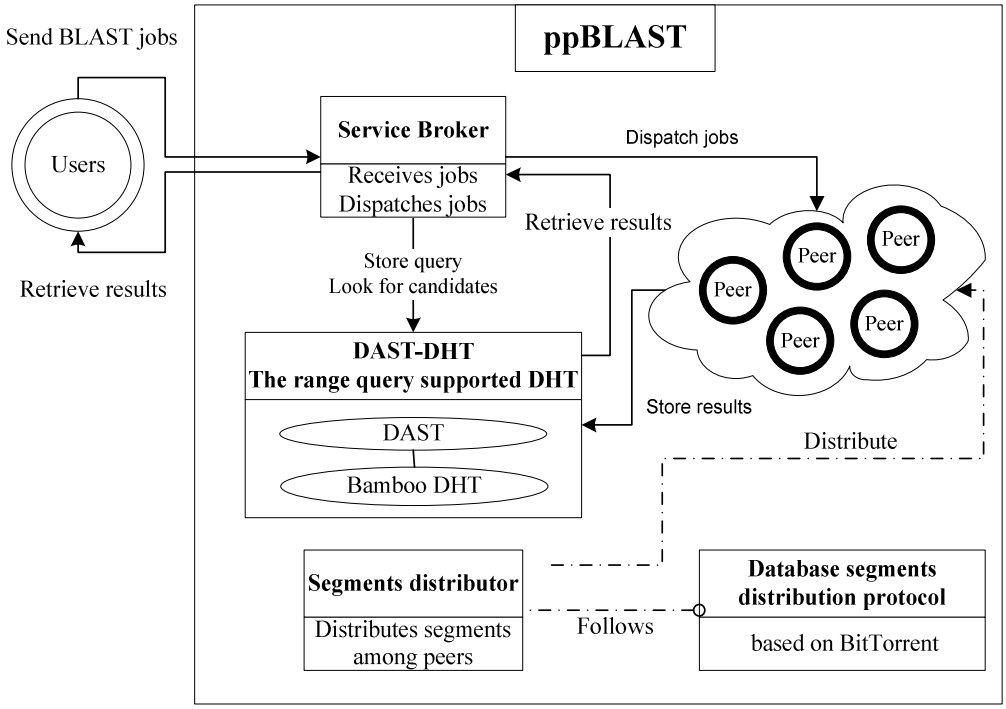


Figure 5-1. Design overview of ppBLAST

BLAST command style and its standalone capabilities, ppBLAST does not modify BLAST internally. Instead, after a peer accepts a BLAST task, it executes a command with the target query and database, and the standalone (original) BLAST will complete the actual search and produce the results. Furthermore, since the input and output of BLAST is all textual, it is feasible and convenient to segment the database and combine the results.

5.3.2 Overview of ppBLAST

Fig. 5-1 depicts the overview of the design of ppBLAST. The most significant component of ppBLAST is the underlying DHT infrastructure arranged as a *Distributed Arbitrary Segment Tree* (DAST). DAST is a result of our previous research [7] (described in Chapter 4), which lays over ordinary DHT and provides it with range query support. The peer-to-peer overlay is self-organised through the

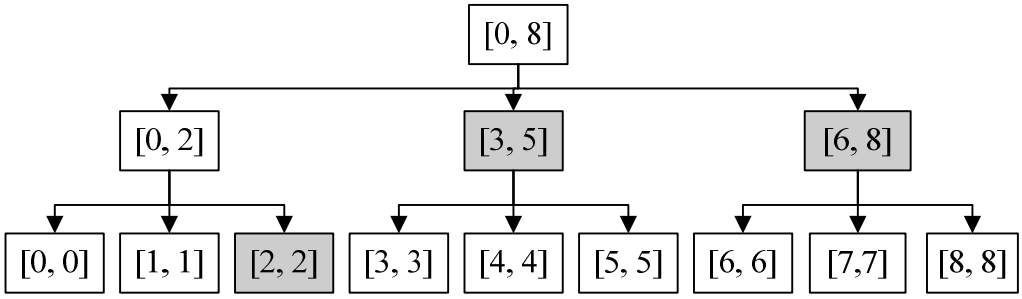


Figure 5-2. An example AST with the entire range [0, 8]

DAST-DHT and every peer represents a contributing node. Peers contribute to the function of the DAST-DHT and also use it at the same time. The major role of DAST-DHT is the storage facility. For the database segments it stores the following tuple {*dbId*, *dbSid_list*}. A *dbSid* is the id of a database segment. This tuple of information maintains a list of *dbSid* for a database, this maintenance is necessary because the databases are regularly updated as described in chapter 1 and new database segments may have to be added to the overlay. For a BLAST job, it stores {*jobId_query*, *query*}, {*jobId*, *taskId*}, {*taskId_done*, *true*}, {*taskId_dispatched*, *true*} and {*taskId*, *result*}. Every BLAST job has a unique Id associated with it and can be divided into a number of tasks. One task is defined as a comparison of a query and a segment of database. Thus a *taskId* is produced using a *jobId* and a *dbSId*. In order to store the locations of peers, it retains a {*dbSId, ip:port_list*}, i.e., the list of peers that have the according database segment, and {*ip:port, job_queue_len*}, i.e., the length of the queue of jobs of a corresponding peer.

A database is split by the *Segment Distributor* component prior to distribution. Normally this operation needs to run once at the very start. Future newly discovered sequences which are added to the database can be extracted and become separate segments for distribution. The segment distribution process follows a modified BitTorrent protocol (described in sub-Section 5.3.5). Thus, after the segmenting operation, the segment distributor can be considered as a seed and upload at least one copy of a complete set of segments into the overlay. Each of the

peers has a *segment list* and updates it after obtaining a new segment. Note that the segment distributor is the only central element in the service and may need to be hosted on at least one PC; this is however necessary as the sequence database needs to be published and its possible failure will not harm the ppBLAST service as its function is limited to the *seeding of the distribution* only.

We present a typical ppBLAST job workflow as follows. A user submits a BLAST request through the *Service Broker* of ppBLAST which resides on his local PC. The service broker generates a *jobId* for the request and connects to the overlay via a bootstrap peer. It puts the tuple {*jobId*, *query*} into the overlay via the DAST-DHT interface. Afterwards, it obtains the set of candidate peers from DAST-DHT. The sum of the segment lists of the candidate peers will cover the entire database. Depending on the queue sizes of the candidates, the broker chooses worker peers and dispatches the task requests to them. Workers will then get the {*jobId*, *query*} from the DAST-DHT and execute the BLAST task. The {*taskId*, *results*} are placed in the DAST-DHT for retrieval.

5.3.3 DAST-DHT

DAST-DHT provides the fundamental infrastructure to ppBLAST. It contains two layers: BambooDHT [57, 84] and DAST [6, 7]. BambooDHT is a DHT implementation based on Pastry [36]. It provides mechanisms that organise peers into an overlay and functions of DHT, i.e., a *put* operation for a single {*key*, *value*} pair and a *get* operation for a single query of a {*key*}. Every pair that is stored in BambooDHT has a TTL (Time-To-Live) associated with it. For every key, it may have a list of values and if more than one *put* operations is made on the same key, all values will be added to that list. BambooDHT does not support range queries. If a component in an application wants to retrieve values for a range of keys, such as 2 to 8, it has to execute a *get* operation 7 times, which is not efficient. With DAST enabling the range query function for BambooDHT, the number of necessary *get* operations is significantly reduced and higher efficiency can be obtained when a range of keys are queried.

Algorithm 5-1: The high level pseudo code of worker selection and task dispatching for the Service Broker

```
// Candidates: the list of peers. The sum of peers' segments cover the
                 according range of segments
// index: controls the range of the segments
// n: every n database segments constitute a task group
//N: the total number of segments for a database
// get(range_start, range_end): DAST-DHT range get interface
//worker1, worker2, work3: selected worker peers
//dispatch(worker, task_start, task_end, order): dispatch tasks
// task_start, task_end: represents the segment range for a task group
// order_1: blastn query from task_start to task_end ascending
// order_2: blastn query from task_end to task_start descending
// order_3: blastn query from the middle of task_start and task_end

SelectWorker()
   index ← n
    while true do
       candidates ← new List[]
       candidates[index/n] ← get(index-n+1, index)
       update task queue length of each peer in candidates[index/n]
        sort candidates[index] ascending by length of peer's queue
       worker1 ← candidates[1]
       worker2 ← candidates[2]
       worker3 ← candidates[3]
       dispatch(worker1, x-n+1, x, order_1);
       dispatch(worker2, x-n+1, x, order_2);
       dispatch(worker3, x-n+1, x, order_3);
       index ← index + n
       if index = N then break;
       if index > N then index ← N
    end while
```

Chapter 4 describe the design of DAST; However, we present the three simple interfaces of DAST-DHT here to clarify the essential DAST-DHT operations within ppBLAST:

- *put_single(key, value)*. This interface inherits from BambooDHT and premits the *put* operation that needs not be range queried;
- *put_range(key, value)*. If the key may be range queried this alternative interface should be used;
- *get(key_range_start, key_range_end)*. The two parameters indicate a range of keys. However, if *key_range_start* equals to *key_range_end*, then the *get(key)* function of BambooDHT is invoked.

5.3.4 Peers and the worker component

There are three roles for the peers in the overlay. First, they organise themselves to construct the DAST-DHT. They store the {*key*, *value*} pairs in their local disks and conform to the BambooDHT mechanisms to route the messages over the network. The second role that they enact is as the clients of DAST-DHT, i.e., they access the public DAST-DHT interfaces for the purpose of storing or retrieving information. Their last role is in participating in the segment distribution and acting as the workers for the BLAST jobs/tasks via the *worker* component. This three-layer design for peers eases the implementation of ppBLAST logically and since the functionalities of the layers are not overlapping, the design will not complicate the development.

Every peer maintains its database segment list and task queue. Every time a peer obtains a new segment, it invokes a *put_range*(*dbSId*, *ip:port*) over the DAST-DHT. This action will update the existing owners of the *dbSId*. Note that it uses the *put_range* because the *service broker* will need to query for a range of segments when looking for job candidates. In addition, peers are required to invoke *put_single* {*ip:port_live, true*} every *t* minutes to indicate their liveness. The TTL of pair {*ip:port_live, true*} is set to *t* such that if a peer does not update the pair, it is assumed to have left the overlay.

5.3.5 Service Broker

The *service broker* is a component that is accessed by only ppBLAST users. It can reside in a server that provides a web interface (a web page, for example) to the users as the frontend for BLAST requests or be an application in users' PCs in which case it will not be a central element in ppBLAST.

The major function of the service broker is offering a simple interface where users can select the BLAST program, upload a query and choose the target database. Once users complete these steps, the service broker will try to accomplish the remaining work; that is posting the query to the DAST-DHT such that worker peers can obtain it, identifying the worker peers and dispatching the tasks.

Suppose we split the database into *N* segments. A job can then be divided into *N* tasks each of which is a BLAST request on the query segment. The *N* tasks are then organised consecutively and uniformly into *N/n+1* groups and each group has *n* tasks. "Consecutively" means that the *dbSId*s of the segments of the *n* tasks in a group are contiguous. For every group of tasks, the service broker will query the DAST-DHT for a range of peers that own the according range of the segments of the tasks. Once the list of candidate peers for a group is obtained, the service broker will sort it into ascending order induced by the length of the peer's task queue. The top *k* peers (with the shortest queues) are then chosen to be the workers and the tasks in that group are dispatched to them with different orders. The information regarding which peers were assigned which tasks will be stored in the broker's local system.

There are three possible orders to the search. With the first, the worker peer executes a BLAST run on each segment in the range in ascending order, i.e., from *segment_1* to *segment_n.* The second is descending, where the peer runs BLAST from *segment_n* to *segment_1*. Finally a peer may start from *segment_(n+1)/2* and the series of target segments will be {segment_(n+1)/2+1, segment_(n+1)/2-1, segment_(n+1)/2+2, segment_(n+1)/2-2, …}. The reasons that we section tasks into groups and dispatch a group of tasks to three peers, each

of which finishes tasks in a specified order, are summarised as follows: 1) Compared to dispatching tasks one by one to peers, sectioning tasks can obtain higher efficiency and generate a lower overhead from the DAST-DHT *get* operation and network connections. 2) We must prepare for the situation where worker peers may leave the network before finishing all the tasks in that group. If any one or two peers leave, the remaining peer(s) will be required to finish the entire group. If all leave, which is the worst case, the service broker will restart the worker selection process again for the unfinished tasks. 3) Peers will *get {taskId_done}* before they decide whether or not to begin a task. By following the design of the three orders, the tasks that those three peers undertake will not overlap therefore no computing power is wasted. 4) We choose three worker peers whose task queues are the shortest to control overloading. After the workers are chosen for one group, peers' queues are updated locally within the service broker such that the information regarding their queues will not be queried again when deciding the workers for the next group. Algorithm 5-1 depicts the whole worker selection and task dispatching process.

For load balancing purposes, the service broker will hold a threshold on the task queues of peers. If the lengths of the task queues of all candidate peers exceeds the threshold, the service broker will pause the dispatching process, regularly querying peers' queues and restarting the dispatching when the length of any peer's queue is reduced below the threshold. Note that the service broker does not necessarily keep connections with worker peers after all tasks are dispatched. Once a worker peer finishes a task, it *put_single {taskId, result}* into the DAST-DHT with a certain TTL. Within the TTL, users can retrieve results via the service broker. Since for every task there is a result file in the DAST-DHT, retrieving the result files can begin as soon as the first results file is produced. Users do not have to wait until all the results files are in place. Retrieving results asynchronously can reduce the total time spent on a BLAST job. When the service broker *get*s the {*taskId*} and retrieves no results, it will query for the {*taskId_dispatched*}. If the task has been dispatched, the broker will locate the worker peer for the task and

then query for {{*ip:port_live, true*}. If the peer is still alive, the service broker will wait; otherwise, it will restart the dispatching process for that task.

5.3.6 Segment Distributor

The segment distributor is another component in ppBLAST. It has two major functions: splitting the database and distributing the parts into the overlay. All BLAST related nucleotide and protein databases are updated regularly by NCBI GenBank. Newly discovered sequences are added to the end of the existing databases; hence, they can be easily extracted and split into new segments for distribution while the old segments in the overlay do not need to be erased or redistributed. The size of one segment should be considered carefully. If it is too large, it is possible that a peer cannot receive a complete copy of the segment if the sending peer leaves the overlay. If it is too small, the number of tasks for one job may be extremely large and the overhead of dispatching a job to peers may be too high. We will give an evaluation in Section 5.4 to guide the choice of the size of the segment.

The segment distribution process follows a modified BitTorrent protocol. BitTorrent is one of the most successful peer-to-peer file distribution protocol [31] (for details of BitTorrent mechanisms and its performance, refer to [50]). We modify the BitTorrent protocol in the following respects to adapt it to ppBLAST.

In original BitTorrent, there is a server, termed as *tracker*, in the overlay. It collects statistical information of peers and tells peers the location of others. Without the *tracker*, BitTorrent stops working. We ignore the *tracker* in ppBLAST. Instead, when the segment distributor wants to distribute segments, it announces itself as a seed by *put_single {new_distribution, ip:port_of_distributor}*. The key {*new_distribution*} will be regularly queried by peers. Once a peer detects a new distribution, it will connect to the segment distributor using the value of the key to receive the *dbSIds* of the new segments. It will then choose the ones it wants to download and query the DAST-DHT for the peers who own them. If no peers have yet owned them, it will ask for them from

the segment distributor. When facing parallel requests for segments, the segment distributor acts as a seed in BitTorrent. From the conclusion of chapter 3, the Original Seeding Strategy (OSS) performs better than the Time-based Seeding Strategy (TSS) when the number of freeriders and exploiters is relatively small [8]. Thus, we choose OSS as the seeding strategy in ppBLAST.

In BitTorrent, when a peer selects a file chunk to download from others, the one with least owners in the overlay will always be chosen, while in ppBLAST segment distribution however, peers choose segments in an organised manner. As described in sub-Section 5.3.6, tasks are sectioned into groups where the segments for a group of tasks are actually consecutive. Peers also download groups of segments to match the design of the dispatching. When a peer starts to download a group of segments (group size = n), it chooses a random starting point from the *dbSId* set {1, n+1, 2*n+1 …}. The random starting point will avoid (with a higher probability) the case that most peers are downloading the same group of segments at the same time. Staggering the segments can give opportunities to peers to download from each other.

5.4 Performance Evaluation

We present the performance evaluation of ppBLAST. First, we investigate the feasibility of ppBLAST through its prototype over a limited number of peers. We then carry out a series of simulations to study the performance of ppBLAST

Table 5-1. The hardware specifications for the five PCs

Peer No.	CPU	Network
1	P4 3.0 GHz	University backbone 100Mbps
2	P4 3.0 GHz	University backbone 100Mbps
3	P Core^2 Duo 1.8GHz	Home broadband 2Mbps
4	P Core^2 Duo 2.4GHz	Home broadband 2Mbps
5	P Core^2 Duo 2.4GHz	Home broadband 2Mbps

in a large-scale peer-to-peer overlay.

5.4.1 Experimental Roadmap and Setup

The performance evaluation process is divided into two steps. The first is validating the feasibility of ppBLAST. We use a prototype version of ppBLAST and run it over five peers using Internet connections. The second step contains a group of simulations. We carefully set up the simulation parameters so as to reflect the real environment. The details of both steps are as follows.

1) Feasibility validation. *We implemented a prototype version of ppBLAST to test the feasibility of the design. The implementation is Java based.*
   a. DAST-DHT: Using the Bamboo-DHT sourcecode as a basis, we implemented our DAST layer on top of this without changing the core of Bamboo-DHT. Bamboo-DHT constrains the sizes of the {key, value} pairs by default. By using DAST-DHT as a storage facility, we are able to get around this constraint by setting the size of a single pair to 10MB. Note that for DAST, we assign 90% to the value of *AoR* and 3 to the level of the tree (*N*).
   b. There are two types of values in {key, value} pairs in ppBLAST: messages and files. The messages are normally small with sizes below 1KB. The files are the ones for results, query sequences and database segments. The sizes of queries that users submit to a real deployment of ppBLAST may exceed 10MB; however, in our experiments we constrain their sizes to be below 1MB to reduce the time of the experiments. In fact, we do not recommend users submit a query larger than 1MB as it may cause low routing efficiencies in the underlying DHT. We discuss this in Section 5.5.
   c. For the worker component that every peer runs, we implemented it as two layers. One is for the BitTorrent-based segment distribution and the other is for the BLAST tasks. These two layers do not communicate directly. If the task layer needs to know which segments the peer currently has, it simply checks the segment folder in the hard drive.

Also the task layer does not integrate with the BLAST programs. It simply executes BLAST command lines and reads the result files from disk after the BLAST programs finish. In all experiments, we choose blastn to represent BLAST and *nt* was selected to be the only target database.

d. Experimental Settings: we deployed a prototype of the ppBLAST service over five peers. The hardware specifications are listed in Table 5-1. We do not list the sizes of the memory in these peers because the segment size will be much less than their memory sizes and thus will not cause a bottleneck [24]. We chose these five peers in order to reflect a heterogeneous environment. Furthermore, in order to reduce the run time, we ran the segment distributor in peer1 to utilise its backbone network to speed up the segment distribution. The service broker application ran in peer2. For the BLAST program, we choose blastn and nt for the database.

2) Simulations: we extend the performance evaluation to a large-scale peer-to-peer overlay through simulation. This simulates the key algorithms and components in ppBLAST. However, for BLAST jobs, the simulator does not execute real BLAST searchings. The simulation of job execution is described as follows:

a. Factors that affect the search time of a BLAST job: query length, number of sequences in a query, sequence database size, number of sequences in database [24]. Even if all these factors in two BLAST jobs are identical, the comparison between the query and database will likely result in different search times.

b. The peers: The computers that peers are using are not always identical. Different CPUs or memory will result in different BLAST performance. Accurately simulating the computing abilities is not important here. Furthermore, we cannot consider all peers as dedicated workers. That is, peers may have other computing loads while serving ppBLAST, hence, it is unlikely to predict this senario.

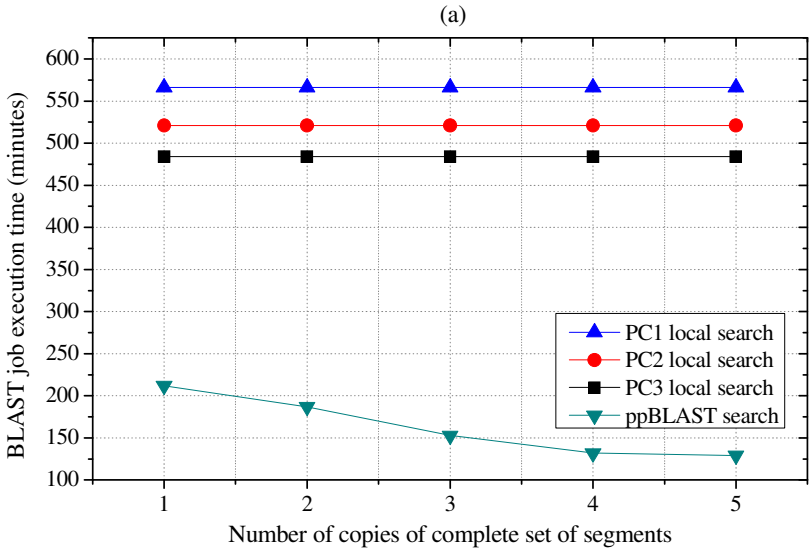


Figure 5-3. the BLAST searching time comparison of local search and ppBLAST search

c. Taking all these random factors into account, we simulate BLAST jobs as follows: First we categorise all peers into three groups of hardwares – CPUs: P4 3.0GHz, P4 Core^2 Duo 1.8GHz, P4 Core^2 Duo 2.4GHz. These hardware reflect three kinds of peers whose computing abilities approximately range from low to high. A peer is randomly assigned to one of these hardwares. Before developing the simulator, we extracted one query sequence of size 1MB from the nt database. We ran blastn searching the query against the nt database in three PCs (PC1, PC2 and PC3) whose hardwares match the specifications above. The time consumed by those searches are recorded. After this, the nt database was split five times (the segment sizes were 2MB, 4MB, 8MB and 16MB, respectively). We then ran the query against every segment in each set of the database and recorded the time for use inside the simulation. In the simulation, depending on the peer and the lengths of query and segment, the time of a blastn task in a peer was extracted from the benchmark above. Also we gave a random weight to each time to reflect the possible other computing loads in the peers.

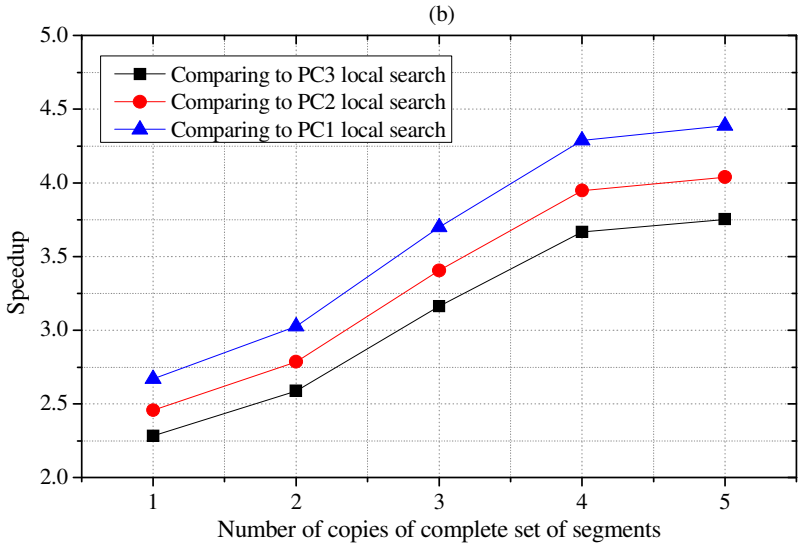


Figure 5-4. Speedup of ppBLAST comparing to the local searches on PC1, PC3, PC5

5.4.2 Results from the feasibility experiments

In this section, we present the experimental results to validate whether ppBLAST in a small scale peer-to-peer overlay can reduce the search time of BLAST. First we randomly extracted 10 sets of queries, each with 1MB size of sequences, from the *nt* database and searched (using *blastn*) against the *nt* database in PC1, PC2, and PC3 locally. For each PC, the average values of the time spent searching the 10 queries are calculated and recorded. We then split the *nt* database into segments, each with 3MB size. The segment distributor was set up in *peer2* and began the segment distribution. At each time when the overlay has a new complete copy of all segments (a maximum of five complete copies when each peer has one), we submit a random query (1MB) to ppBLAST via the service broker residing on *peer3*. A copy of the complete segment set in the overlay means that all segments that peers own can cover the segment set (but some peers may lack certain segments). For the time spent on a job in ppBLAST, we calculate

the time period between the point of submitting a job and the point at which all the results are retrieved from the DAST-DHT.

Fig. 5-3 plots the BLAST search time comparisons between ppBLAST and the local searches of PC1, PC2, and PC3. We can see that the execution time of BLAST jobs on ppBLAST are much less than those obtained from local runs on a single PC. The speedups are depicted in Fig. 5-4. As described in Section 5.1, the performance of a local BLAST search significantly decreases if the size of the database is bigger than the memory on the PC. The *nt* database used in the experiments has the size of 8.76GB and none of the three PCs have memories larger than this. Hence, the execution of the jobs run locally takes considerable time. *peer1* exhibits the worst case where it averages almost 10 hours to complete a BLAST search of a 1MB query against the *nt* database. ppBLAST improves the BLAST search jobs significantly because (1) the size of database segment is much smaller than the local memory; (2) five peers cooperate to finish one job. Note that there are several additional operations, such as peers retrieving queries and putting the results files into a DAST-DHT, which cost time in ppBLAST. However, the size of the query is small enough to be obtained quickly (larger queries can be split) and storing result files can be managed asynchronously to the BLAST searching, i.e., if a peer finishes a task, it can store the result file in the background while it is doing the next task. Compared to the original BLAST search time, the overhead for the retrieving and storing operations are insignificant.

Fig. 5-4 also shows the relationship between the ppBLAST execution time and the number of copies of the complete segment set in the overlay. It is obvious that the performance of ppBLAST is better if there are more copies of the complete segment set in the overlay. ppBLAST does not adopt a segment-on-demand mechanism [24] where a worker peer obtains a missing segment for a task from a central server. Instead, it assigns tasks based on what segments a peer current has. We design ppBLAST in this way because a central segment providing server may be the single point of failure, but without the server, segment-on-demand can force peers to download the lacking segments from others and the BLAST tasks

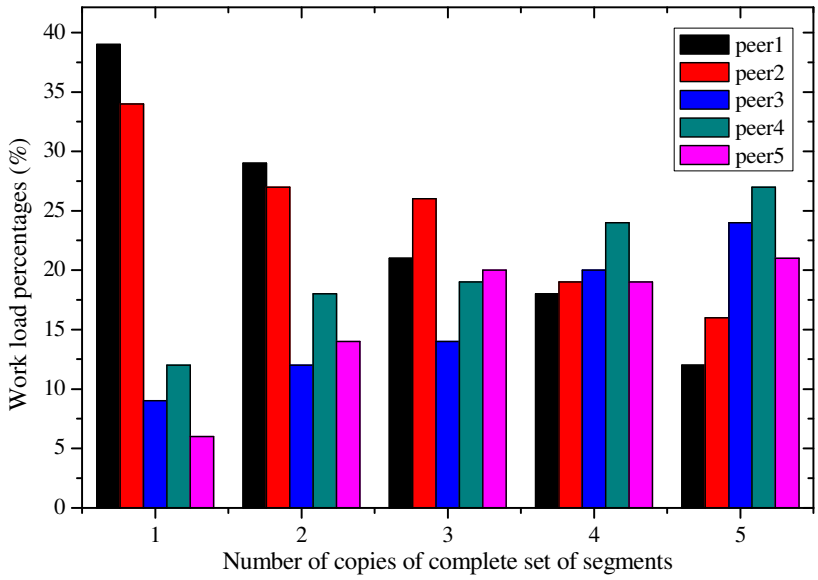


Figure 5-5. The work loads of each PC (the percentage of the total number of tasks in a BLAST job)

will have to wait, which itself is not time-efficient. Therefore, when there is only one copy of the complete set in the overlay, a job can be completed but a number of peers may not obtain tasks and be idle because they do not have the necessary segments. When more copies of segments are delivered among peers, the performance increases. This is shown in Fig. 5-3 and 5-4; when there are four copies of the complete segment set in the overlay, the job execution performance reaches near-optimal.

Fig. 5-5 presents the workload distributions of five peers. The workloads were calculated from the number of tasks that a peer took in a BLAST job. We can see that when there is only one copy of complete segment set in the overlay, *peer1* and *peer2* take the majority of the workload. As listed in Table I, *peer1* and *peer2* have high speed Internet access and they can download segments from others (including the segment distributor) quickly. Inside the first copy of the complete set in the overlay, they owned most of the segments. Hence, when the service broker dispatched the tasks, *peer1* and *peer2* were assigned more than 70%

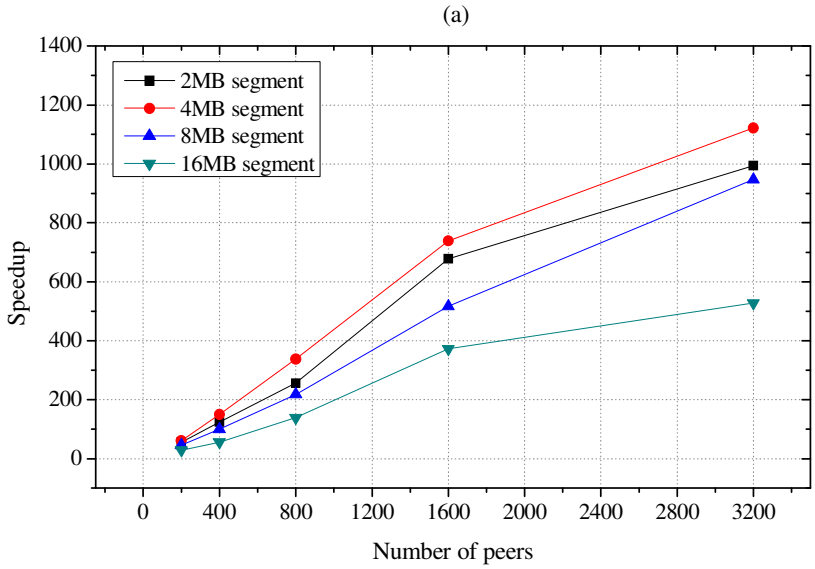


Figure 5-6. Speedup of ppBLAST when every peer has at least 60% of the complete segment set

of the tasks. The remaining peers only obtain 26% of the tasks due to lacking certain segments. The workload tends to balance while more copies of segments are distributed among peers. When every peer in the overlay owns a complete segment set, *peer3*, *peer4* and *peer5* obtain more tasks than *peer1* and *peer2* because of their higher computing abilities.

5.4.3 Results from the simulation experiments

In this section, we extend our performance evaluation of ppBLAST by adopting a large-scale peer-to-peer overlay via simulation. The numbers of peers that we have chosen are 200, 400, 800, 1600, 3200, and 6400. Since we analysed the relationship between copies of the complete segment set and the execution time in sub-Section 5.4.2, here we only present the results of the simulation runs in which a job is submitted when every peer has already obtained at least 60% or 100% of the segment set. The percentage is only relative to the number of segments, but not related to the contents of segments, i.e., two peers can have the same number of segments but not same ones. For the comparison target, we chose

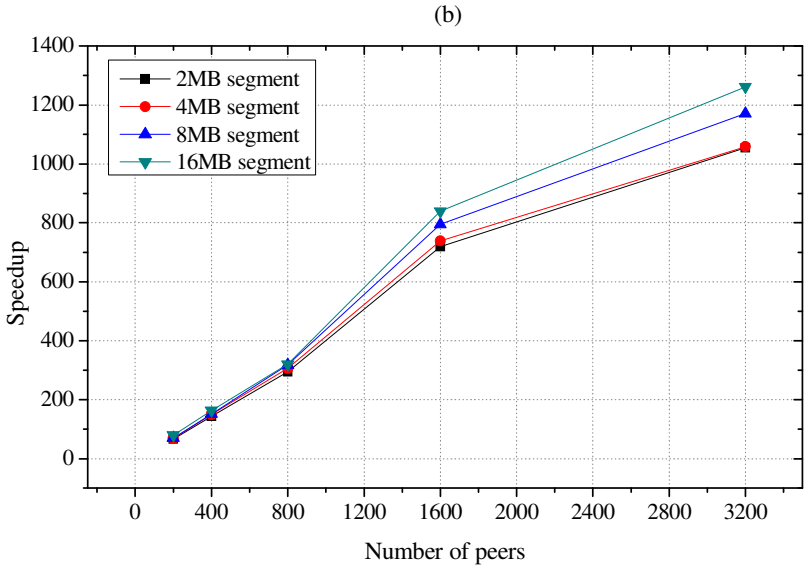


Figure 5-7. Speedup of ppBLAST when every peer has full segment set

the times spent on the BLAST job executed in *PC3,* which has the highest computing capability. The distribution of the bandwidths follows Table 5-2.

Fig. 5-6 and 5-7 shows the speedup of the execution times of ppBLAST. First of all we can see that the speedups in both figures indicate the high performance improvement in ppBLAST once the peer-to-peer overlay is at large scales. The execution time of a BLAST job is considerably reduced in ppBLAST compared to the local search time on one PC. In Fig. 5-6, four lines represent four types of segment into which the *nt* database is split. The speedups rise when splitting the database into 4MB segments instead of 2MB. However, if we segment the

Table 5-2. Bandwidth distribution of leechers (derived from the actual distribution of the Gnutella network [1])

Downlink (KB/s)	Uplink (KB/s)	Fraction
78	12	0.3
150	38	0.6
300	100	0.1

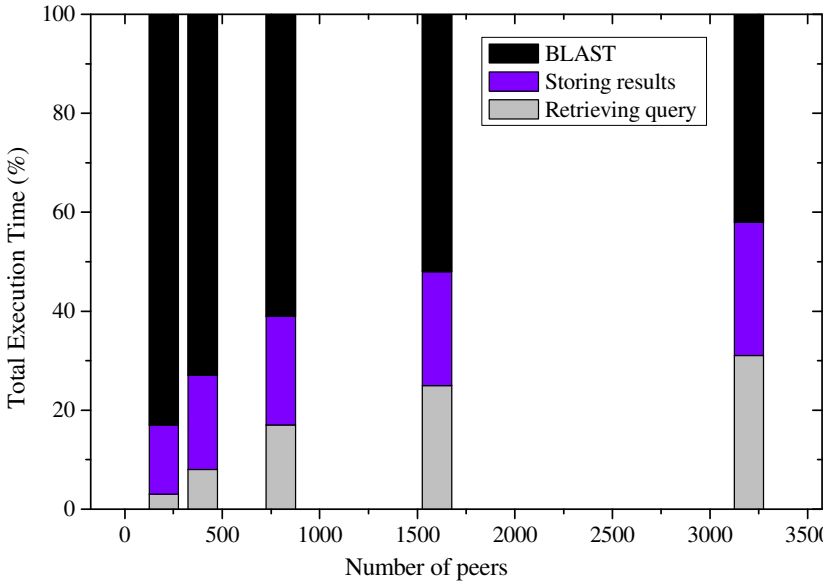


Figure 5-8. How is time spent in ppBLAST.

database into 8MB segments, the performance degrades (it becomes worse using 16MB segments). When a BLAST search begins, the program will read a query and a database from the hard drive and will write the results to the hard drive when finished. Constant hard drive operations (read and write) will result in high overheads on the local system. If we segment the database into 2MB segments, the system overhead is higher than that when using 4MB segments because there are more segments that need to be read and more result files need to be written. However, when we adopt 8MB segments or higher, the distribution of a single segment costs more time. A complete segment set is the unit for tasks. Hence, peers with high computing power but low bandwidth may not be able to take a task because of the low transmission rates of a single segment. This is clearly depicted in Fig. 5-6. In Fig. 5-7, the low transmission rate effect disappears since every peer has a complete set of segments.

Although the speedups in Fig. 5-6 and 5-7 indicate large performance improvement in ppBLAST, they are not optimal, i.e., speedup does not equal the number of peers. When the number of peers increases, the number of tasks that

each peer can take reduces. The total amount of time that a peer spends on all of its tasks is also shortened such that we cannot ignore the time spent on retrieving queries and storing result files (DAST-DHT operations, as described in sub-Section 5.3.3). We plot the average time compositions of peers for each job in Fig. 5-8. We can see that the percentage of the time spent on DAST-DHT operations becomes larger when the scale of the overlay grows. If the DAST-DHT operation time can be reduced, the performance of ppBLAST can be further improved. One potential solution to shorten the query retrieval time is to create an additional BitTorrent-like layer dedicated to distributing the query among worker peers.

5.5 Conclusions

This chapter introduces the design, implementation and performance evaluation of ppBLAST. ppBLAST focuses on utilising the computing resources over the Internet to provide a BLAST computing service. Its design involves three layers: DAST-DHT, BitTorrent-like distribution and BLAST job dispatching and execution. In order to simplify the implementation, all layers do not overlap but only use each other. A performance evaluation demonstrates the considerable improvement that ppBLAST can offer to BLAST searches.

The motivation of the attempt of utilising peer-to-peer network for BLAST is to look for another kind of possible computing resource to complement clusters. Clusters are expensive, in terms of the funding cost and the complexity of administration, and hard to scale with the growing sizes of the BLAST databases and jobs. Although one may argue that clusters are getting cheaper these days, we believe that these results demonstrate that the free and scalable computing resources from peer-to-peer networks will provide a cost-effective substitute for clusters.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we present our three contributions: Analysing BitTorrent seeding strategies, a Distributed Arbitrary Segment Tree to provide range query function for DHTs, and utilising Peer-to-Peer overlays to parallelise the bioinformatics application - BLAST.

Although the aim of our research is to build a Peer-to-Peer computation service for BLAST jobs, we had to first build a large file distribution facility to distribute the database among peers. BitTorrent has been shown to be the most effective Peer-to-Peer file distribution protocol, and its Tit-For-Tat algorithm is one of the key factors that have led to BitTorrent to success. However, another key factor – Seeding Strategy – has been largely overlooked. Without a proper seeding strategy, the performance of BitTorrent still can be negatively impacted by the existence of *freeriders* and *exploiters*. Thus, we conduct a comprehensive analysis on the seeding strategies of BitTorrent in order to make better selection of strategy and further improve BitTorrent's performance in preparation for the implementation of our Peer-to-Peer-based BLAST.

With the BitTorrent protocol enhanced to provide an effective seeding strategy to distribute databases among peers in ppBLAST, an additional facility was required to store queries, tasks and results through a simple and efficient interface. The DHT system fulfils this requirement. DHT supplies functions like a traditional hash table – put a {*key*, *value*} pair in and get the *value* out by a *key*. Peers can store queries and results through DHT layer easily and efficiently. In

addition, DHT also make peers self-organised and provides a convenient means for locating worker peers. However, current DHTs have the limitation of single key query, i.e., each time a peer can query for only one key and if it needs to query for a continuous series of keys, one query has to be made for each key. This negatively impacts the system performance. Therefore, we conducted research on building a range query function layer over existing DHTs. We utilise segment tree data structure, extending it to **D**istributed **A**rbitrary **S**egment **T**ree and layering it on top of DHTs. Every DHT *put* or *get* operation will first go through our DAST layer and DAST will rehash the *key* in a new hashing space in order to instrument the range query functionality.

With the above two fundamental services were determined, we were able to design and implement ppBLAST. We build three layers: the layer at the bottom is the DAST-DHT; it provides three functions: constructing the overlay for ppBLAST, providing lookup for peers' locations and storing queries, tasks, and results. The middle layer is the BitTorrent-like database distribution facility. It splits a large database into chunks and distributes them among peers. It also utilises the underlying DAST-DHT layer to locate peers and thus does not need an indexing server. The top layer is the BLAST job assigning and scheduling layer. It relies on the DAST-DHT layer to find the available target peers and assigns tasks through a scheduling algorithm.

6.1.1 Analysing BitTorrent Seeding Strategies

In this contribution, we study two seeding strategies – the *Original Seeding Strategy* (*OSS*) and *Time-based Seeding Strategy* (*TSS*). First we build a mathematical model to investigate the impact that freeriders or exploiters have on the mean download completion time of unselfish leechers if OSS is employed. For the model, we adopt homogeneous environment where peers have the same download / upload bandwidths. We then further extend our study via simulation. In the simulation experiments for the homogeneous environment, we first validate our mathematical model and then compare OSS with TSS through metrics of mean download completion time, mean download rate and mean bandwidth utili-

sation. For the heterogeneous environment, we use one additional metric which is the cumulative distribution of unselfish leechers' download completion time.

We summarise the results as follows. First, our mathematical model for studying the Original Seeding Strategy is validated and can be used to theoretically analyse the effects of the seeding strategies on BitTorrent networks. We find out that there is a threshold for the number of selfish leechers (freeriders / exploiters). If the threshold is exceeded, TSS performs better than OSS; otherwise, OSS outperforms TSS. We also discover that both freeriders and exploiters harm the system, despite the seeding strategy that is employed. However, TSS has better resistance to selfish leechers compared with OSS. Furthermore, freeriders and exploiters impact negatively on leechers of every level of bandwidth and neither of the seeding strategies can completely eliminate this impact. Therefore, OSS should be employed if the number of selfish leechers is relatively small.

6.1.2 Distributed Arbitrary Segment Tree on DHT

In order to provide an efficient range query function for DHTs, we construct a DAST layer on top of DHTs. The DAST layer selects a hashing space and constructs an arbitrary segment tree. Every segment on the tree is a fixed range of hashing values. When the DAST layer receives a *put(key, value)* request, the *key* is abstracted and obtains a new hash in the hashing space. Afterwards, DAST will insert the {*key*, *value*} pair into every node whose fixed range contains the new hash of the *key*. When a range of keys are queried, DAST will look for the minimum number of nodes and the sum of all those nodes' ranges cover the query.

Although DAST requires more than one *insert* operation, with the consideration of the fact that a single {key, value} pair is always *put* once, but value retrievals are conducted any number of times, the overall operation overhead is lowered compared to normal DHT. Another advantage of DAST is that it does not modify the core of the underlying DHT and thus can be adapted easily by any DHT systems. Also, DAST has set up parameters such as *AoR*, *number of tree levels*, and so on. Applications can adjust the parameters according to different

application environments to obtain optimal range query performance. Finally, the DAST structure is deterministic once the range of the whole hashing space is decided and therefore does not need any maintenance work which brings simplicity and less overhead to the client applications.

6.1.3 A Parallelisation of BLAST over Peer-to-Peer network

The BitTorrent-like protocol and the DAST-DHT research provide two fundamental services, which are used in the implementation of ppBLAST. ppBLAST accepts BLAST jobs, dispatches them to peers and collects the results. A ppBLAST client submits a BLAST job through the locally running *service broker*. The service broker is not a centralised element in the overlay; instead, it is a component inside every BLAST client application. It accepts a BLAST job and divides the job into a number of tasks. Every task corresponds to a piece of database and has a taskId associated with it. Afterwards, the service broker will range query the overlay for all available peers and the sum of all peers' downloaded database pieces cover the whole database. Finally, the tasks will be dispatched to the peers. Since the BLAST job query has to be bound with every task, it is not efficient if the service broker sends the same copy of the query to every worker peer. Thus, for high efficiency, the service broker will store the query in the DAST-DHT layer once and all selected worker peers will retrieve the query from it. When a worker peer finishes a task, it puts the {taskId, result} into the DAST-DHT layer for users to retrieve.

6.2 Future Work

BitTorrent Seeding Strategies

Current popular BitTorrent clients employ either OSS or TSS, and none investigate the combination of these two seeding strategies. Therefore our future research is focused on how to combine these two seeding strategies so that the seeds can deliver enhanced service to unselfish leechers despite the existence of

selfish leechers. This will involve building a mechanism to detect the scale of selfish leechers in the overlay. Depending on the scale, we will direct the seeds to implement OSS or TSS dynamically.

DAST

In the current DAST design, although it effectively decreases the number of *get* operations when handling range queries, it still needs additional *put* operations for each insertion of *key*. Our next research step is to further reduce the *put* operation overhead for DAST while maintaining the same effectiveness of range queries in DHT.

ppBLAST

There are several directions for future work in ppBLAST. First, ppBLAST can create an additional BitTorrent-like layer to support the query distribution process. The current query distribution relies on the raw DAST-DHT operations which are proven to be relatively slow in our experiments. If we distribute queries in a BitTorrent-like manner, a large query may not need to be split and the performance of ppBLAST can be further improved.

The second potential area of work is to create a mechanism to encourage peers to join the network. In the current design, we assume that all peers that contribute to ppBLAST are self-motivated (just like ones in SETI@home and Folding@home) and they do not need to use BLAST themselves. However, some BLAST users can also be ppBLAST peers. How to encourage those users to contribute while they do not use their machines remains the topic of future work.

The current evaluations and experiments, which were carried out through real deployment, for ppBLAST are based on a small scale of peer setup. We are planning to adopt larger number of real peers for fuller evaluations of ppBLAST. This can be done via clusters (each node in cluster can be emulated as a peer) or joining Planetlab to obtain real peer resources over the Internet.

At last, what is QoS level of ppBLAST needs to be explored and how to maintain QoS is also our future work. The churn phenomenon (peers keep leaving or joining) should also be analysed for ppBLAST to investigate its stability.

Bibliography

[1] E. Adar and B. A. Huberman, "Free Riding on Gnutella", *First Monday*, Vol. 5, Issue 10, 2000.

[2] S. Saroiu, P. K. Gummadi and S. D. Gribble, "A Measurement Study of Peer-to-Peer File Sharing Systems", *MMCN*, pp. 2002.

[3] A. R. Bharambe, C. Herley and V. N. Padmanabhan, "Analyzing and Improving a BitTorrent Network's Performance Mechanisms", *IEEE INFOCOM*, pp. 2006.

[4] X. Chen and S. Jarvis, "ppBLAST: A Computational Service over Peer-to-Peer net-work for BLAST", *International Conference on Advances in P2P Systems (AP2PS 2009)*, Sliema, Malta, October 11-16, 2009.

[5] X. Chen and S. Jarvis, "Analysing BitTorrent's Seeding Strategies", *7th IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC-09)*, Vancouver, Canada, August 29-31, 2009.

[6] X. Chen and S. A. Jarvis, "Design and Implementation of Efficient Range Query over DHT Services", *1st International Conference on Signal Processing and Communication Systems (ICSPCS 2007)*, pp. 2007.

[7] X. Chen and S. A. Jarvis, "Distributed Arbitrary Segment Tree: Efficient Range Query Over Public DHT Services", *12th IEEE International Workshop on Computer Aided Modeling and Design of Communication Links and Networks, held as part of PIMRC07*, Greece, 2007.

[8] X. Chen and S. A. Jarvis, "Analysing Seeding Strategies and Fairness in BitTorrent-based Networks", *22nd Annual UK Performance Engineering Workshop*, pp. 2006.

[9] X. Chen, S. A. Jarvis, G. Tan, L. He, D. P. Spooner and G. R. Nudd, "An implementation of BLAST over peer-to-peer and its performance validation through simulation", *8th International Conference on Computer Modelling and Simulation*, pp. 2005.

[10] G. Tan, S. Jarvis, X. Chen, D. Spooner and G. Nudd, "Performance Analysis and Improvement of Overlay Construction for Peer-to-Peer Live

Media Streaming", *Simulation: Transactions of the Society for Modeling and Simulation*, 82:93-106(Feb 2006, pp.

[11] G. Tan, S. Jarvis, X. Chen, D. Spooner and G. Nudd, "Performance Analysis and Improvement of Overlay Construction for Peer-to-Peer Live Media Streaming", *13th IEEE Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sept. 2005.

[12] G. Tan, S. Jarvis, L. He, X. Chen, D. Spooner, G. Nudd and ", , , . "Modelling Web trans-fer Performance over Asymmetric Networks", *1st International Workshop on Per-formance Modelling in Wired, Wireless, Mobile Networking and Computing (PMWMNC-2005), 11th IEEE International Conference on Parallel and Distrib-uted Systems (ICPADS'05)*, Fukuoka Institute of Technology, Japan, 20-22 July 2005.

[13] L. He, S. Jarvis, D. Spooner, X. Chen and G. Nudd, "Dynamic Scheduling of Paral-lel Jobs with QoS Demands in Multiclusters and Grids", *5th IEEE/ACM Interna-tional Workshop on Grid Computing (Grid2004)*, Pittsburgh, USA, Nov 8, 2004.

[14] L. He, S. Jarvis, D. Spooner, X. Chen and G. Nudd, "Hybrid Performance-based Workload Management for Multiclusters and Grids", *IEE Proc.-Softw.*, 151(5):224-231(October 2004, pp.

[15] L. He, S. Jarvis, D. Spooner, X. Chen and G. Nudd, "Dynamic, Hybrid Perform-ance-oriented Scheduling of Moldable Jobs with QoS Demands in Multiclusters and Grids", *3rd International Conference on Grid and Cooperative Computing (GCC 2004)*, Wuhan, China, October 2004.

[16] L. He, S. Jarvis, D. Bacigalupo, D. Spooner, X. Chen and G. Nudd, "Queueing Net-work-based Optimisation Techniques for Workload Allocation in Clusters of Computers", *IEEE International Conference on Services Computing (SCC 2004)*, Shanghai, China, September 15-18, 2004.

[17] L. He, S. Jarvis, D. Spooner, X. Chen and G. Nudd, "Hybrid performance-oriented optimisation mechanism for scheduling QoS-requesting parallel jobs in multi-clusters and grids", *20th Annual UK Performance Engineering Workshop (UK-PEW' 2004)*, University of Bradford, July 7-8 2004.

[18] NCBI. http://www.ncbi.nlm.nih.gov, Accessed on 15/01/2009.

[19] A. Naruse, N. Nishinomiya, K. Kumon and M. Yamaguchi, "Hi-per BLAST: High Performance BLAST on PC Cluster System", *Genome Informatics. Vol. 13. pp 254–255 (2002)*, pp.

[20] GenBank. http://www.ncbi.nlm.nih.gov/Genbank/index.html, Accessed on 04/10/2008.

[21] TimeLogic. http://www.timelogic.com/products.html, Accessed on 02/02/2009.

[22] R. K. Singh, W. D. Dettloff, V. L. Chi, D. L. Hoffman, S. G. Tell, C. T. White, S. F. Altschul and B. W. Erickson, "BioSCAN: A Dynamically Reconfigurable Systolic Array for Biosequence Analysis", *CERCS96*, 1996.

[23] N. Camp, H. Cofer and R. Gomperts, "High-Throughput BLAST", *SGI White Paper*, September 1998.

[24] A. Darling, L. Carey and W. Feng, "The Design, Implementation, and Evaluation of mpiBLAST", *4th International Conference on Linux Clusters: The HPC Revolution 2003 in conjunction with ClusterWorld Conference & Expo*, June, 2003.

[25] W. Feng, "Green Destiny + mpiBLAST = Bioinfomagic", *10th International Conference on Parallel Computing (ParCo)*, September 2003.

[26] M. Bayer, A. Campbell and D. Virdee, "A GT3 based BLAST grid service for biomedical research", *All Hands Meeting (AHM04)*, pp. 2004.

[27] R. Braun, K. Pedretti, T. Casavant, T. Scheetz, C. Birkett and C. Roberts, "Parallelization of local BLAST service on workstation clusters", *Future Generation Computer Systems*, 17(6)(April 2001, pp. 745{754.

[28] TeraBLAST. http://www.timelogic.com/downloads/TeraBLAST_2009.pdf, Accessed on 02/02/2009.

[29] K. Pedretti, T. Casavant, R. Braun, T. Scheetz, C. Birkett and C. Roberts, "Three complementary approaches to parallelization of local BLAST service on workstation clusters", *Lecture Notes In Computer Science*, 1662 (1999, pp. 271 - 282.

[30] Napster. www.napster.com, Accessed on 01/08/2009.

[31] B. Cohen, "Incentives Build Robustness in BitTorrent", *First Workshop on Economics of Peer-to-Peer Systems*, pp. 2003.

[32] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells and B. Zhao, "Oceanstore: An Architecture for Global-Scalable Persistent Storage", *ASPLOS*, pp. 2000.

[33] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris and I. Stoica, "Wide-area Cooperative Storage with CFS", *18th ACM Symposium on Operating Systems Principles*, pp. 2001.

[34] A. Rowstron and P. Druschel, "Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility", *18th ACM Symposium on Operating Systems Principles*, pp. 2001.

[35] ppStream. www.ppstream.com, Accessed on 01/08/2009.

[36] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems", *18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, pp. 2001.

[37] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky and D. Werthimer, "SETI@home: An Experiment in Public-Resource Computing", *Communications of the ACM*, 45 No. 11(November 2002, pp. 56-61.

[38] E. Korpela, D. Werthimer, D. Anderson, J. Cobb and M. Leboisky, "SETI@home-massively distributed computing for SETI", *Computing in Science & Engineering*, 3, No. 1(2001, pp. 78-83.

[39] C. D. S. Stefan M Larson, Michael Shirts, Vijay S Pande, "Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology", *Computational Genomics, Horizon Press*, 2002.

[40] SETI@Home. http://setiathome.berkeley.edu/, Accessed on 07/2009.

[41] S. Altschul, W. Gish, W. Miller, E. Myers and D. Lipman, "Basic local alignment search tool", *Journal of Molecular Biology*, 215(1990, pp. 403 - 410.

[42] S. F. Altschul, T. L. Madden, A. A. Schaer, J. Zhang, Z. Zhang, W. Miller and D. J. Lipman., "Gapped BLAST and PSI-BLAST: a new generation of

protein database search programs", *Nucleic Acids Res.*, 25(1997, pp. 3389 - 3402.

[43] F. L. Fessant, S. Handurukande, A.-M. Kermarrec and L. Massoulié, "Clustering in Peer-to-Peer File Sharing Workloads", *The 3rd International Workshop on Peer-to-Peer Systems* 2004.

[44] I. Clarke, O. Sandberg, B. Wiley and T. W. Hong., "Freenet: A distributed anonymous information storage and retrieval system.", *The Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[45] N. Andrade, M. Mowbray, A. Lima, G. Wagner and M. Ripeanu, "Influences on Cooperation in BitTorrent Communities", *The Third Workshop on Economics of Peer-to-Peer System*, pp. 2005.

[46] B. Fan, D. M. Chiu and J. C. Lui., "The Delicate Tradeoffs in BitTorrent-like File Sharing Protocol Design", *International Conference on Network Protocols*, pp. 2006.

[47] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding and X. Zhang, "Measurements, Analysis, and Modeling of BitTorrent-like Systems", *ACM SIGCOMM Internet Measurement Conference (IMC'05)*, pp. 2005.

[48] M. Izal, G. Urvoy-Keller, E. W. Biersack, P. Felber, A. A. Hamra and L. Garces-Erice, "Dissecting BitTorrent: Five Months in a Torrent's Lifetime", *Passive and Active Network Measurement*, pp. 2004.

[49] S. Jun and M. Ahamad, "Incentives in BitTorrent Induce Free Riding", *ACM SIGCOMM*, pp. 2005.

[50] A. Legout, N. Liogkas, E. Kohler and L. Zhang, "Clustering and Sharing Incentives in BitTorrent Systems", *ACM SIGMETRICS*, pp. 2007.

[51] N. Liogkas, R. Nelson, E. Kohler and L. Zhang, "Exploring the robustness of BitTorrent Peer-to-Peer Systems", *Concurrency and Computation: Practice and Experience*, 2007, pp.

[52] T. Locher, P. Moor, S. Schmid and R. Wattenhofer, "Free Riding in BitTorrent is Cheap", *HotNets-V*, pp. 2006.

[53] J. A. Pouwelse, P. Garbacki, D. H. J. Epema and H. J. Sips, "The BitTorrent P2P File-Sharing System: Measurements and Analysis", *International workshop on Peer-To-Peer Systems*, pp. 2005.

[54] D. Qiu and R. Srikant, "Modeling and Performance Analysis of BitTorrent-Like Peer-to-Peer Networks", *ACM SIGCOMM*, pp. 2004.

[55] M. Sirivianos, J. H. Park, R. Chen and X. Yang, "Free-riding in BitTorrent Networks with the Large View Exploit", *IPTPS*, pp. 2007.

[56] Y. Tian, D. Wu and K. W. Ng, "Modeling, Analysis and Improvement for BitTorrent-Like File Sharing Networks", *Infocom*, pp. 2006.

[57] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica and H. Yu, "OpenDHT: A Public DHT Service and Its Uses", *ACM SIGCOMM*, pp. 2005.

[58] I. Stoica, R. Morris, D. Karger, F. Kaashoek and H. Balakrishnan, "Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications", *ACM SIGCOMM*, pp. 149-160, 2001.

[59] E. Sit, F. Dabek and J. Robertson, "UsenetDHT: A Low Overhead Usenet Server", *IPTPS*, pp. 2004.

[60] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker and J. Hellerstein, "A Case Study in Building Layered DHT Applications", *ACM SIGCOMM*, pp. 2005.

[61] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding and X. Zhang, "Measurements, Analysis, and Modeling of BitTorrent-like Systems", *ACM SIGCOMM Internet Measurement Conference (IMC'05)*, 2005.

[62] L. Massoulie and M. Vojnovic, "Coupon Replication Systems", *ACM Sigcomm*, pp. 2005.

[63] P. A. Felber and E. W. Biersack., "Self-scaling Networks for Content Distribution", *International Workshop on Self-* Properties in Complex Information Systems*, pp. 2004.

[64] A. Akella, S. Seshan and A. Shaikh, "An Empirical Evaluation of Wide-Area Internet Bottlenecks", *Internet Measurement Conference*, pp. 2003.

[65] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker, "A Scalable Content-Addressable Network", *ACM SIGCOMM*, pp. 161-172, 2001.

[66] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems", *18th IFIP/ACM*

International Conference on Distributed Systems Platforms (Middleware 2001), pp. Nov. 2001.

[67] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph and J. D. Kubiatowicz, "Tapestry: A Resilient Global-scale Overlay for Service Deployment.", *IEEE Journal on Selected Areas in Communications*, 2003, pp.

[68] C. Zheng, G. Shen, S. Li and S. Shenker, "Distributed Segment Tree: Support of Range Query and Cover Query over DHT", *IPTPS*, California, USA, pp. 2006.

[69] M. d. Berg, M. v. Kreveld, M. Overmars and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, 2000.

[70] V. Papadimos, D. Maier and K. Tufte, "Distributed Query Processing and Catalogs for Peer-to-Peer Systems", *Innovative Data Systems Research* Asilomar, CA, USA, pp. 2003.

[71] M. Abdallah and H. C. Le, "Scalable Range Query Processing for Large-Scale Distributed Database Applications ", *Parallel and Distributed Computing Systems*, Phoenix, AZ, USA, pp. 2005.

[72] D. Oppenheimer, J. Albrecht, D. Patterson and A. Vahdat, "Design and Implementation Tradeoffs for Wide-Area Resource Discovery ", *HPDC*, pp. 2005.

[73] A. R. Bharambe, M. Agrawal and S. Seshan, "Mercury: Supporting Scalable Multi-Attribute Range Queries", *ACM SIGCOMM*, pp. 2004.

[74] J. Aspnes and G. Shah, "Skip Graphs", *ACM - SIAM Symposium on Discrete Algorithms (SODA)*, pp. 2003.

[75] N. J. A. Harvey, M. B. Jones, S. Saroiu, M. Theimer and A. Wolman, "SkipNet: A Scalable Overlay Network with Practical Locality Properties", *Fourth USENIX Symposium on Internet Technologies and Systems*, pp. 2003.

[76] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker and I. Stoica, "Querying the Internet with PIER", *19th International Conference on Very Large Databases (VLDB)*, pp. 2003.

[77]

[78] R. D. Bjornson, A. H. Sherman, S. B. Weston, N. Willard, J. Wing and T. Inc, "TurboBLAST: A Parallel Implementation of BLAST Built on the TurboHub", *IPDPS*, 2002.

[79] H. Lin, P. Balaji, R. Poole, C. Sosa, X. Ma and W. Feng, "Massively Parallel Genomic Sequence Search on the Blue Gene/P Architecture", *IEEE/ACM SC2008: The International Conference on High-Performance Computing, Networking, and Storage*, November 2008.

[80] H. Lin, X. Ma, P.Chandramohan, A. Geist and N. Samatova, "Efficient Data Access for Parallel BLAST", *IEEE International Parallel & Distributed Processing Symposium*, April 2005.

[81] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", *Sixth Symposium on Operating System Design and Implementation (OSDI'04)*, San Francisco, CA, 2004.

[82] A. Hadoop. http://hadoop.apache.org/, Accessed on 18/07/2009.

[83] D. P. Anderson., "BOINC: A System for Public-Resource Computing and Storage", *5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, USA, pp. 2004.

[84] BambooDHT. http://bamboo-dht.org, Accessed on 04/09/2007.