uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Li Wu

AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Master of Computer Science

GRADE / DEGREE

School of Information Technology and Engineering

FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Distributed True String B-Tree Peer-to-Peer Overlay Networks

TITRE DE LA THÈSE / TITLE OF THESIS

Ramiro Liscano

DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

Stan Matwin

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

Shervin Shirmohammadi

Tony White

Gary W. Slater

LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES /
DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

Distributed True String B-Tree Peer-to-Peer Overlay Networks

by

Li Wu

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the Master degree in
Computer Science

School of Information Technology and Engineering
Faculty of Engineering
University of Ottawa

Your file Votre référence
ISBN: 978-0-494-14971-3
Our file Notre référence
ISBN: 978-0-494-14971-3

Abstract

A fundamental problem that confronts P2P Internet applications is to efficiently locate the physical (IP) node that stores a particular data item. To tackle this application-level routing problem, this thesis proposes a new P2P overlay network, called Distributed True String B-tree (DTSBT) P2P overlay network. Unlike popular Distributed Hash Table (DHT)-based P2P overlay networks which use DHT as their core data structure, the DTSBT P2P overlay network employs a new data structure, DTSBT as its core data structure. A DTSBT is a hybrid distributed data structure, in which all peers' routing tables make up a virtual B-tree and a Patricia Trie is plugged into each peer's routing table. The performance evaluation showed that the DTSBT P2P overlay network is more scalable, decentralized, resilient to failures, and self-organized compared to its competitors.

Acknowledgements

I would like to express my sincere gratitude and respect to my supervisor, Professor Dr. Ramiro Liscano, for his crucial guidance, non-stop patience, invaluable advice and great help. I feel extremely fortunate and honored to be his student.

Also, I would like to thank my co-supervisor, Professor Dr. Stan Matwin, for his generous help and advice.

Further, I would like to acknowledge School of Information Technology and Engineering, University of Ottawa for giving me such a wonderful opportunity to study for Master of Science, Computer Science program.

Finally, I would like to thank my parents and my sister for their constant love and encouragement.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Internet and Peer-to-Peer Networks

In recent years, the Internet has become a constant feature of our lives. Everyday, billions of people all over the world use the Internet and its various applications to communicate personally and professionally. At the same time, the Internet is used in practically every aspect of business, including advertising, production, shipping, planning, billing and accounting. The reason behind the Internet's success rests chiefly on its best effort datagram delivery service. Such a minimalist service leads the Internet to be the synonym of current computer networks.

However, as both Internet services and Internet customers continue to increase, the application-level routing scalability problem becomes more visible. Currently Internet applications such as email and web services employ completely centralized (mainframe model) or partially distributed (client/server model) system architecture. These architectures lead these applications to encounter severe scalability problems, such as network swamping. In certain cases, the scalability problem may lead to complete loss of application functionality for a period of time.

To overcome the drawbacks of the centralized system architecture, Internet application systems tend to employ a more distributed system architecture. This is known as the P2P system architecture. In P2P systems, computer resources (such as CPU cycles, storage and content) can be directly shared without requiring the intermediation of a centralized server [7]. An effective P2P system requires, among other things, scalability, decentralization, resilience to failures and self-organization. Many Internet applications have been developed and deployed based on P2P system architecture. Some such P2P

applications perform content distribution, such as Napster [4], Gnutella [2], Kazaa [3], Freenet [9], Publius [31] and Oceanstore [20]. Other P2P applications execute distributed computation, such as Seti@Home [5] and Genome@Home [1]. Others yet are distributed database systems, such as PIER [18] and Piazza [17].

1.2 Problem and Motivation

Centralized system architecture is relatively easy to manage with central index servers. This is not, however, the case with P2P system architecture. Since there are no centralized index servers, P2P Internet applications encounter the following application-level routing problem: what is the way to efficiently locate the physical (IP) node that stores a particular data item.

To address the application-level routing problem, a virtual network is introduced. This is known as the P2P overlay network. A P2P overlay network can be viewed as a new layer under the application layer, but over the network layer in the TCP/IP protocol stack. The application-level routing problem is thus seen as the fundamental problem of P2P overlay networks.

Certain P2P overlay networks, such as Chord [29], Pastry [27], Tapestry [34] and CAN [24], use Distributed Hash Tables (DHT) as their core data structures. In these overlay networks, data object (or value) location information is placed deterministically, at the peers with identifiers corresponding to the data object's unique *key* [21]. Each peer maintains a small routing table consisting of its neighbouring peers' key identifiers and IP addresses. The routing process can thus be executed sequentially according to the prefix digits of key identifiers. In general, DHT-based P2P overlay networks have the routing efficiency of $O(\log N)$ hops pathlength with requiring $O(\log N)$ neighbours storage space, where N is the size of the key identifier space.

Although DHT-based P2P overlay networks (including their routing protocols) have had some success, they still face certain challenges. The main problem is still that of scalability. The first step in creating a DHT-based P2P overlay network based on a random hash function is to map a text-based key identifier space to a digit-based key identifier space. Each digit-based identifier has a fixed length, and each bit in each such identifier has fixed representations. Once the length of each key identifier has been determined, and the number of representations of each bit has been confirmed, the size of the key identifier space is fixed. If, for instance, each identifier in a key identifier space has m-bit length and each bit has b representations, the size of the key identifier

space is b^m. In the design of a P2P Internet application system, however, it is difficult to determine just how many Internet application services the application system will provide. For example, when designing a P2P application system, designers estimate it will provide $1,000,000$ Internet application services, and they let the length of each key identifier be 6 and the number of representations of each bit be 10. It is reasonably possible, however, that customers would like to provide a $1,000,001^{th}$ service. This sort of scalability problem is similar to the recent high-profile Y2K problem or the extension from IPv4 to IPv6.

Second, for each DHT-based P2P overlay network, there is a tradeoff between the routing efficiency and the storage space required by the routing table. The greater efficiency a DHT-based P2P overlay network has, the more neighbours storage it will require. In general, most DHT-based P2P overlay networks have the routing efficiency of $O(\log N)$ hops pathlength with requiring $O(\log N)$ neighbours storage space, where N is the size of the key space. Can one DHT-based P2P overlay network achieve the routing efficiency of $O(\log N)$ hops pathlength with $O(1)$ neighbours storage space [25]? In other words, can one design a P2P overlay network which does not have this proportional relationship between its routing efficiency and its routing table storage space?

Furthermore, DHT-based P2P overlay networks suffer from the high cost of their system maintenance. Once a new key joins the system, or an old key leaves the system, many other nodes have to adjust their routing table. This adjustment is very costly.

Moreover, during the process of converting arbitrarily long text-based keys into fixed length digit-based keys, significant semantic information is lost. As XML-based Internet applications emerge, however, application-level routing protocols should be able to support semantic queries.

1.3 Main Contribution

The main contribution of this thesis is to provide a totally new approach to the application-level routing problem of Internet P2P overlay networks. In this approach, a virtual P2P overlay network is built as a new layer under the Internet application layer but over the Internet network layer in the TCP/IP protocol stack. This is known as the Distributed True String B-tree (DTSBT) P2P overlay network, which employs DTSBT data structure as its core data structure.

Unlike DHT-based P2P overlay networks, which convert their text-based key spaces into their digit-based key spaces, the key space of the DTSBT P2P overlay network

consists of all text-based keys. Each key is an arbitrarily long, alphabet-independent text string. As a result, the key space is *infinite*, *continuous* and *lexicographic*. Consequently, the DTSBT P2P overlay network is highly scalable.

Furthermore, for each key there is a routing table stored in the local IP node. This routing table contains the lexicographic relationship of a group of neighbouring keys, which are lexicographically close to the current key. All these routing tables make up a virtual B-tree-like P2P overlay network. Because each routing table employs a Patricia Trie as its inner routing tool, the routing table storage space has no relationship with the lengths of keys, but rather is proportional to the number of keys contained in the routing table. Therefore, in general, the DTSBT P2P overlay network has the routing efficiency of $O(\log_B N)$ hops pathlength with the B neighbours storage.

Routing tables in DTSBT P2P overlay networks only contain local routing information. As a result, as a new key joins the system or an old key leaves the system, only a small fraction of physical (IP) nodes need to modify their routing tables stored on them. Therefore, the system maintenance is very cost-efficient.

Since it dispenses with the process of converting an arbitrarily long text-based key space into a fixed length digit-based key space, the DTSBT P2P overlay network supports semantic query. As a result, XML-based Internet applications can easily be built based on it.

Another contribution of this thesis is to provide the procedure of simulating a P2P overlay network over the Internet. This thesis has conducted such a simulation in the NS-2 [6] simulation environment, and provides the results of several groups of experiments. The results of these experiments suggest that the DTSBT P2P overlay network provides a routing efficiency comparable to a DHT-based P2P overlay network, while performing considerably better than the latter in terms of storage space.

Chapter 2

Background and Related Work

In this chapter, the Internet network layer which is the underlying network of P2P overlay networks, is first introduced. Next, P2P systems and P2P overlay networks are discussed. Finally, some existing P2P overlay networks which attempt to address the application-level routing problem, are described.

2.1 The Underlying Network

As a packet-switching and datagram network, the Interent has come to be synonymous with current computer networks. The main reasons behind the Internet's success are its network lay's qualities of scalability and robustness. In providing best effort datagram delivery, the Internet allows routers to be stateless (except for the routing state, which is highly aggregated). Routers thus do not need to maintain detailed information regarding the state of the traffic. As a consequence, today's Internet is both highly scalable and robust. It is scalable because router complexity does not increase correspondingly to either the number of flows, or the number of nodes in the network. The Internet is robust because there is little state, if any, to update when a router fails or recovers.

2.2 P2P System Architecture

Currently most of Internet applications (*raisons d'etre* of the Internet) such as email and Web employ the client/server system architecture to provide various powerful Internet services. However, as both Internet services and Internet customers continue to increase, these client/server-based Internet applications encounter significant scalability problems.

Another problem of the client/server systems is that they are not robust. For example, should the central server be shut down, the whole system crashes. In terms of system management, the cost of the client/server systems is high because on the server side, there is a constant need for either a large dedicated hardware infrastructure or a significant human administration.

To overcome the drawbacks of the client/server systems, some Internet applications tend to employ a more distributed system architecture. This is know as the P2P system architecture. In P2P systems, computer resources (such as CPU cycles, storage and content) can be directly shared without requiring the intermediation of a centralized server [7]. P2P systems instinctively have many desired features: decentralization, resilience to failures, and self-organization. Most importantly, this system architecture has the potential to create wide-are Internet applications that are scalable.

The Client/server paradigm is relatively easy to manage with central index servers. This is not, however, the case with P2P system architecture. Since there are no centralized index servers, P2P Internet application systems encounter the following application-level routing problem: what is the efficient way to locate the physical (IP) node that stores a particular data object?

2.3 P2P Overlay Networks

To tackle this application-level routing problem when seamlessly intergrating Internet applications that use P2P system models, with the Internet, a virtual network is introduced. This is known as the P2P overlay network. A P2P overlay network can be viewed as a new layer over its underlying netwokr, the Internet network layer, but under the Internet application layer. In a P2P overlay network, there are two kinds of objects: data objects and location objects. Data objects, generally called keys, abstractly correspond to Internet application services while location objects are those IP nodes in the Internet.

According to P2P overlay network infrastructure, there are two classes of P2P overlay networks: *Unstructured* and *Structured.* In unstructured P2P overlay networks, such as Gnutella [2] and Napster [4], peers are organized in a random graph that has a flat or hierarchical topology. Such P2P overlay networks employ flooding or random walks to search for keys. In general, each such search needs to pass through a large proportion of the peers. As a result, this sort of P2P overlay networks are not suitable for large-scale Internet applications. A structured network is one in which the P2P overlay network topology is tightly controlled and data objects are placed not at random peers

but rather at specified locations in order to make subsequent queries more efficient. The salient feature of structured P2P overlay networks is that their application-level routing protocols are based on a core data structure.

2.4 Existing DHT-based P2P Overlays

In this section, some of the existing structured P2P overlay networks will be reviewed. All of them create their application-level routing protocol based on DHT data structure. Thus, these structured P2P overlay networks are called DHT-based P2P overlay networks.

2.4.1 Chord

By using *consistent hashing* [19], Chord [29] distributes the keys of the key space into all IP nodes existing in current underlying network. Consequently, among the current IP nodes randomly assigned with fixed length digit-based identifiers, all keys are distributed to make key identifer as close to that IP identifier as possible. All IP nodes are connected into a circular list. To speed up the routing efficiency, based on the key idenfiers, one figure table is inserted into each IP node's routing tables. Chord with N IP nodes can get the routing efficiency of $O(\log N)$ application-level hops pathlength with a routing table with the size of $O(\log N)$ neighbours.

2.4.2 Pastry

Each node in Pastry [27] has a unique identifier (nodeId). When presented with a message and a key, a Pastry node efficiently routes the messages to the node with a nodeId that is numerically closest to the key, among all currently live Pastry nodes. Also, Pastry takes into account network locality; it seeks to minimize the distance messages travel, according to scalar proximity metric like the number of IP routing hops.

2.4.3 CAN

CAN [24] chooses its keys from a d-dimensional torodial space. Each node is associated with a hypercubal region of this key space, and its neighbours are the nodes that *own* the contiguous hypercubes. Routing consists of forwarding to a neighbour that is closer to the key. *CAN* has a different performance profile than other algorithms in that nodes have

$O(d)$ neighbours and pathlengths are $O(dn^d)$ hops. Note, however, that when $d = \log n$, *CAN* has $O(\log n)$ neighbours and $O(\log n)$ pathlengths, similar to other algorithms.

2.4.4 Plaxton et al.

Plaxton et al. [23] developed perhaps the first routing algorithm that could be scalably used by DHTs. While not intended for use in P2P systems (because it assumes a relatively static node population), this algorithm does provide a very efficient routing of lookups. The routing algorithm works by *correcting* a single digit at a time. To do this, a node needs to have, as neighbours, nodes that match each prefix of its own identifier but differ in the next digit. For a system of n nodes, each node has on the order of $O(\log n)$ neighbours. Since one digit is corrected each time the query is forwarded, the routing path is at most $O(\log n)$ overlay (or application-level) hops.

This algorithm has the additional property that if the n^2 node-node latencies (or *distances* according to some metric) are known, the routing tables can be chosen to minimize the expected path latency. Moreover, the latency of the overlay path between two nodes is within a constant factor of the latency of the direct underlying network path between them.

2.4.5 Tapestry

Tapestry [34] uses a variant of the *Plaxton et al.* algorithm. The modifications are to ensure that the design, originally intended for static environments, can be adapted to a dynamic node population. Tapestry provides location-independent routing of messages directly to the closest copy of an object or service using only point-to-point links and without centralized resources. The routing and directory information within this infrastructure is purely soft state and easily repaired. The algorithm maintains the properties of having $O(\log n)$ neighbours and routing with path lengths of $O(\log n)$ hops.

Chapter 3

Motivation and Design Overview

3.1 Problems with DHT-based P2P Overlays

P2P file-sharing systems are now one kind of the most popular Internet applications and have become a major source of Internet traffic. Unfortunately, the initial designs for P2P application systems have significant scaling problems. To overcome these scaling problems, several research groups have (independently) proposed a new generation of scalable P2P application systems that ride over a virtual network. This virtual network is known as the P2P overlay network. Among them are Chord [29], Pastry [27], Tapestry [34] and CAN [24], which all employ a Distributed Hash Table (DHT) as their core data structures. These DHT-based P2P overlay networks abstractly regard each Internet application as a service and each such a service corresponds a unique key. One key is an identifier to seperate a servie from other services, such as a service description file. The cores of DHT-based P2P overlay networks are the routing protocols employed by them.

The general process of creating a DHT-based P2P overlay network, and then searching for a particular key by using its routing protocol can be depicted as below:

- **Hash Coding** Let $b \geq 2$ be a fixed integer. If $n \geq 0$ is any integer, then n can be written in the form:

$$n = r_m b^m + r_{m-1} b^{m-1} + ... + r_1 b + r_0, \tag{3.1}$$

  where $m \geq 0$ and $0 \leq r_i < b$ for all i. Further, these integers r_i and m are uniquely determined by n.

  Based on this equation, a DHT-based P2P overlay network can assign each key an m-bit digit-based identifier, in which each bit has b possible representations,

by using some random hash functions (for instance, Chord employs SHA-1 [30]). After this hash coding process, arbitrarily long text-based keys are turned into fixed length digital-based keys. The key identifier length m must be large enough to make the probability of two keys hashing to the same identifier negligible. For example, if there are n keys, and $b = 2$, then the key identifier length m must be at least $\lceil O(\log_2 n) \rceil$. Once b and m are assigned, it is clear therefore that the number n of keys is affirmed.

- **Creating Routing Tables** This step can be divided into two substeps: creating a sorted circle, and creating a finger table. In the first substep, each node creates two entries in its routing table. One entry points to its logic predecessor node, in which the key is the nearest key smaller than the key of the current node. The other entry points to its logic successor node, in which the key is the nearest key larger than the key of the current node. For the node which stores the largest key, its successor node is the node which stores the smallest key. After this substep, all nodes are connected in a virtual circle. From the view of data structure, this substep creates a circle link list. This substep is necessary because it makes routing protcols comprehensive. In other words, if a particular key is actually stored in a node, routing protocols can ensure they always find it. Without this substep, it would be possible that, even though a paritcular key does really exist in a node, routing protocols would not always find it. For example, Pastry has not created such a sorted circle. As a result of only using routing table without any other auxiliary routing information, there is the odd case, in Pastry where an existing key cannot be found.

  If one merely employed this sorted circle to find keys, the routing protocol would not be scalable because the sorted circle is a circle link list. As a result,the routing protocol has the routing efficiency of $O(n)$ hops pathlength, where n is the size of the keys space.

  In sum, both substeps are essential. In the second substep, each node creates an $m \times b$ entries figure table in its routing table, where m is the key identifier length, and b is the representation number of each bit. For example, if the key identifier length is 8, and each bit has 3 possible representations, then the finger table of each node can have $8 \times 3 = 24$ entries. Further, in the finger table, the $(b \times k)^{\text{th}}$ entry, $finger[k \times b]$, points to the node, which stores the b^{k-1}th key.

- **Routing** The routing process in DHT-based routing protocol can be regarded as the searching process in a b-ary search tree, where b is the possible representation number of each bit. It can be proved that for a balancing b-ary search tree, the height of the tree is $O(\log_b n)$, where b is the number of branches of each node in that tree. Thus, the conclusion can be drawn that the effciency of a DHT-based routing protocol can have the resulting pathlength of $O(\log_b n)$ hops, where n is the size of keys space and b is the representation number of each bit.

- **Maintenance** The space occupied by the keys is not static, but dynamic in that new keys can join this P2P overlay network, and old keys can leave this P2P overlay network. In the case of both nodes joining and nodes leaving, other nodes of the system should adjust their routing tables to make the routing process completely correct.

The DHT-based P2P overlay networks have had significant success in the two following ways. Firstly and theoretically, such overlay networks are scalable because the routing process has the efficiency of $O(\log n)$-level hops pathlength. Secondly and practically, such overlay networks are popular in that many Internet applications have been developed and deployed on their basis. However, this approach suffers from many drawbacks. These are examined below.

- **Is it really scalable? Yes, but its scalability is not extendible.** The DHT-based P2P overlay networks are scalable because they have the routing efficiency of $O(\log n)$-level hops pathlength. Conversely, it is not extendible because DHT-based P2P overlay networks employ m-bit digit-based key identifiers to represent a key. Once the length of key m and the possible representation number of each bit b are assigned, the size of key space is affirmed. In other words, the key space created by DHT is not **infinite**.

  However, Internet applications are extremely various and very versatile, and the number of the corresponding Internet services is usually underestimated. In other words, with high probability, the real number of keys needed is larger than the size of a particular key space because the size of a DHT-based key space is static, but the number of the corrresponding Internet services is dynamic. For example, if each key identifier has m-bit length, and the possbile representation number of each bit is b, then the size of the key space is b^m. Thus, in the face of a greater number of Internet services (such as the $b^m + 1$th Internet service) joining the current P2P network, how will that P2P network represent those services?

A simplistic solution to this problem is either to enlarge the length of the key identifier m, or to increase the representation number of each bit b. For a centralized system such as a client/server system, this relatively simple approach is not difficult in that after some required adjustments to the central server, the system may well be alright.

Such a solution is, by contrast, very difficult to implement in P2P systems. For one thing, each key should be changed according to the new rules. Correspondingly, each routing table should be changed based on the new key identifier representation. This approach is prohibitively costly. In fact, such cases have occurred several times in the history of computering. One recent example is the Y2K problem, and another one is the need to update from IPv4 to IPv6.

- **Is it adaptive? No.** A good application of P2P overlay networks is the P2P wireless sensor network. Such a network, however, is difficult to adapt to changed circumstances. Image a network, with 6-bit long key identifiers and the representation number of each bit being 2. In such a scenario, the performance of the P2P system can be very good. However, when this design is applied for another case, such as one in which only $1,000$ Internet services are needed to be provided for. In this second scenario, both the size of key space and the size of the routing table are too large to get the best performance.

- **Does it support semantics? No.** To keep keys load-balancing, random hash functions have been used in DHT-based P2P overlay networks. As a result, any two semantically closed keys can have large distance in the key identifier space. This leads XML-based semantic queries (such as a prefix query, range query and substring query) to be almost impossible. Additionally, this leads semantic service compositions to be almost impossible.

- **Can DHT-based P2P overlay networks, with both fixed m and b values, improve their routing efficency? No.** Even if the cost of both local memory and network maintenance is negligible, with both fixed m and b values. DHT-based P2P overlay networks can have at most $O(\log_b n)$ hops pathlength routing efficiency with each physical (IP) node's $b \times O(\log_b n)$ routing table storage, where n is the size of the key space. This is because DHT-based P2P overlay networks execute the routing process depending on the key identifier representation method. In other words, the routing efficiency is not independent of the key identifier representation

method. Once both m and b values are stable, the routing efficiency is affirmed.

- **Does it have high resilience to failures? No.** The nodes in P2P overlay network are notoriously transient, so that the resilience to failures is left significantly lacking. However, without auxiliary backup mechanisms, DHT-based P2P overlay networks themselves don't provide such backup mechanisms from the view of data structures. This is because generally each node has a different routing table, and all these nodes are connected in a sorted circle list. Once any node fails, the routing process cannot work. For example, there may be a key existing in the network, but the routing protocol is unable to locate it.

- **Is the cost of maintaining system very high? Yes.** In fact, DHT-based P2P overlay networks employ too much **global** information on both the key identifier space and the routing tables. On the key identifier space side, once a key is assigned an identifier with stable m and b values, the key itself recognises: (1) that the size of the key space is b^m; (2) that if given a pair of keys a and b, then the distance between them is $a - b$. On the routing table side, each key corresponds a finger table, which points the network addresses (such as an IP address) of the n/bth, $n/(b^2)$th,..., $n/(b^m)$th key in the key space. Thus, once either a new key joins the network, or an old key leaves the network, almost all other keys should change their own corresponding routing tables. In general, the cost of maintaining systems is $n \times O(\log_b n)$.

3.2 DTSBT P2P Overlay Network Design

To overcome DHT-based P2P overlay networks' drawbacks discussed above, this thesis proposes a new structured P2P overlay network, the DTSBT. The DTSBT is a virtual network built over the underlying network, the network layer of the Internet, and under various Internet applications. The key point of the DTSBT P2P overlay networks is that they use the DTSBT data structure as their core data structures.

3.2.1 Key Space

Instead of assigning each key a digit-based key identifier with both fixed key identifier length m and fixed representation number of each bit b, a DTSBT overlay employs the key itself as its identifier. Let Σ denote an arbitrarily large ordered alphabet of characters,

and let $\leq_L$ denote the lexicographic order between any two characters in Σ, and let $\triangle$ denote the set of strings which sequentially consist of arbitrarily long characters drawn from Σ. Then, the key space of DTSBT is $\triangle$.

As a consequence, the key space $\triangle$ is *lexicographic*, *continuous* and *infinite*.

- It is lexicographic because there is a lexicographic order $\leq_L$ between any pair of keys in $\triangle$.
- It is continuous because for any pair of keys in $\triangle$, there is a new key outside $\triangle$, which can lexicographically be inserted into the interval between them.
- It is infinite because for $\triangle$, there is a new key outside $\triangle$, which is lexicographically greater than all keys in $\triangle$, and which can be inserted into $\triangle$ as the lexicographically biggest key of $\triangle$; and there is a new key outside $\triangle$, which is lexicographically smaller than all keys in $\triangle$, and which can be inserted into $\triangle$ as the lexicographically smallest key of $\triangle$.

These three features are very desirable for a DTSBT P2P overlay network. The key space $\triangle$ is lexicographic leads the keys in $\triangle$ to be comparable and sortable. Without a sortable key space, a scalable structured overlay would not be established. The key space $\triangle$ is infinite and continuous leads $\triangle$ to be extendible and adaptive. The extension fulfills that any new key can easily join or leave the current system. The adaptation makes any overlay that performs well in a certain environment be seamlessly transferred to another environment. As a result, a DTSBT overlay is extremely scalable.

3.2.2 Key Nodes, IP Nodes and Routing Tables

Unlike DHT-based overlays whose basic overlay units are IP nodes, the DTSBT overlays employ those keys in the key space $\triangle$ as its basic overlay units. This basic overlay unit is referred to as a key node through out this thesis and all these key nodes make up a DTSBT overlay. In general, one key node corresponds to one IP node where the key is created and stored, and one IP node also corresponds to one key node. In the case where several different keys are created and stored at a same IP node, one IP node may correspond to several key nodes. Furthermore, this thesis assumes that the content (an arbitrarily long string) of a key is merely permanently stored at the local IP node where that key is created and cannot be permanently stored at any other IP nodes. This assumption implies that the DTSBT overlays have high degrees of decentralization and self-organization.

Routing tables play a critical role in any P2P overlay network. In a DTSBT overlay, a routing table that contains the routing information of a group of keys (including the current key and its neighbour keys) is stored at each key node. The design of the DTSBTs is thus different rather than the design of DHTs in that in DTSBTs a routing table directly belongs to a key node whereas in DHTs a routing table directly belongs to an IP node. In the event where one IP node corresponds to several key nodes, one IP node can indirectly contain several different routing tables. From the relationship between key node, IP node and routing table, the concept of key node is very important because after this abstraction, overlay and underlying network (IP layer) are clearly separated and the routing protocol can be directly created over that overlay network.

3.2.3 DTSBT, TSBT, String B-Tree

A DTSBT P2P overlay network's core is a new data structure, DTSBT data structure. DTSBT data structure is the distributed version of another new data structure, True String B-Tree (TSBT) (also provided by us) which is a variant of String B-Tree data structure [13] and is designed for an external memory algorithm.

TSBT is however clearly different from String B-tree in that a TSBT is a *true* B-tree whereas a String B-tree is B^+-tree. In a B-tree, keys can be allocated in every B-tree node, while in a B^+-tree keys can be merely contained in its leaves. As a result, TSBT data structure can be naturally and easily employed as the core data structure of a P2P overlay network, while String B-tree is not easily applied in the design of P2P overlay networks.

3.2.4 A Virtual Distributed B-Tree

A DTSBT data structure is a hybrid data structure, in which all key nodes make up a virtual distributed B-tree and a Patricia Trie is plugged into the routing table of each key node. A virtual distributed B-tree in a DTSBT overlay has the following properties:

- According to the lexicographic relationship between any pair of keys, all keys in the current overlay can be divided into a number of key groups. The keys contained in a group are lexicographically closed and in a lexicographic order. Each such a group is a B-tree node in the virtual distributed B-tree.

- Suppose a group or a B-tree node includes n keys. The rouing tables contained by the key nodes in this group are exactly the same, and contain n entries. In other

words, a group of key nodes logically share the same routing table that has a fixed number (n) of backups physically stored at each key node.

- Like a general B-tree, there are two kinds of B-tree nodes in a virtual distributed B-tree: internal nodes and leaves. Either internal nodes or leaves can store a group of keys. The routing tables of internal nodes (including the root) also contain $n+1$ IP addresses to its children groups. Leaf nodes have no children.

- The n entries in the routing table belonging to the key node x are used as dividing points separating the range of key nodes handled by x into $n+1$ subranges, each handled by one children group of x. When searching for a key node in such a virtual distributed B-tree, one makes an $(n+1)$-way decision based on comparisons with the n entries stored at the routing table.

- All leaves have the same depth, which is the virtual distributed B-tree's height h.

- There are lower and upper bounds on the number of keys that a B-tree node can contain. These bounds can be expressed in terms of a fixed integer $t \leq 2$ called the *minimum degree* of the virtual distributed B-tree:

  - Every B-tree node other than the root must have at least $t-1$ keys. Every internal node other than the root thus has at least t childrent. If the B-tree is nonempty, the root must have at least one key.

  - Every B-tree node can contain at most $2t-1$ keys. Therefore, an internal node can have at most $2t$ children. A B-tree node is *full* if it contains exactly $2t-1$ keys.

3.2.5 Patricia Tries and Routing Tables

Suppose a routing table contains n entries. Each entry of a routing table contains two parts: the content of a key, and the IP address of the IP node at which the key is created and stored. The inner routing process for such a routing table is thus executed as follows: compare the queried key with an entry of the routing table character-by-character. This is, however, extremely inefficient if repeated n times because its worst-case cost is proportional to the length of the two strings involved each time. This problem is called *rescanning*, due to the fact that the same input characters are (re)examined n times. Correspondingly, storing the contents of keys into a routing table makes the size

of that routing table proportional to the total length of the contents of those keys. As a result, the storage size is too large to be tolerated.

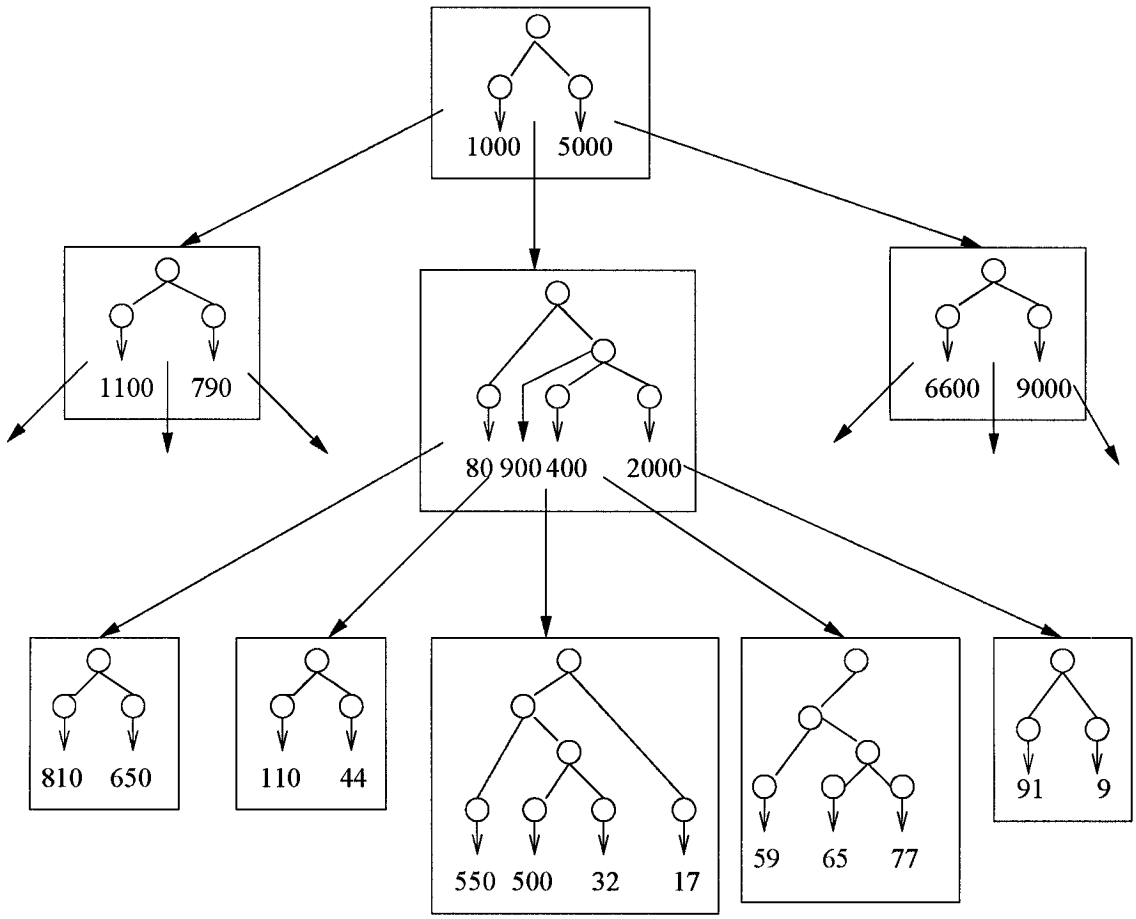
To overcome these two problems, the routing tables in DTSBT overlays do not rely on storing the content of keys. Rather, each routing table employs a Patricia Trie [22] to store the lexicographic relationship information of a group of keys. In other words, a Patricia Trie is plugged into a routing table to drive the inner routing. This makes the DTSBT data structure hybrid. As a result, searching for a key in a routing table merely needs a one time scan that travels from the root to a leaf in the Patricia Trie. More importantly, the size of a routing table is proportional to the number of entries (keys) contained in it rather than the total length of entries (keys) contained in it. Consequently, the resultant storage size is significantly reduced.

3.2.6 Network Computing Model

Although networks have already become very fast (100Mbps local networks are very popular, and optical networks are also emerging), general-purpose computer networks usually employ a hierarchy of computing capacity. On the local side, computing capacity is faster and more inexpensive. On the network side, however, computing capacity is slower and relatively more expensive [12]. This thesis accordingly employs a two-level model: faster local computing coupled with slower network computing. Because of this dichotomy, the analysis of the routing performance tends to neglect the running time of local computation and places emphasis on the total number of network accesses.

3.2.7 An Example of DTSBT P2P Overlay Network

Figure 3.1 illustrates an example of a DTSBT P2P overlay network. In this figure, each unit presents a key node's routing table, which does not include the contents of a group of key nodes. When the current B-tree node is an internal node, the routing table includes three parts: a Patricia Trie, a group of IP addresses that correspond to a group of key nodes, and a group of IP children addresses. For example, the routing table (ab, ac, acb, acd) includes a Patricia Trie that includes the lexicographic relationship between (ab, ac, acb, acd), a group of IP address (80, 900, 400, 2000), and 5 IP children addresses. When the current B-tree is a leaf such as (ace, acf), the routing table includes two parts: a Patricia Trie and a group of IP addresses.



IP	key	IP	key	IP	key
1000	aab	80	ab	6600	bba
5000	bbb	900	ac	9000	bbbd
1100	aaab	400	acb	810	aac
790	aaac	2000	acd	650	aad
110	aba	32	acaabb	77	acbabb
44	abc	17	acab	91	ace
550	acaaa	59	acbaa	9	acf
500	acaaba	65	acbaba		

Figure 3.1: An exmaple of the DTSBT overlay network

Chapter 4

DTSBT Design and Implementation

After a DTSBT overview design is given in chapter 3, the detailed DTSBT design and implementation are proposed in this chapter.

4.1 Routing Tables

In the DTSBT overlay, each key node contains four parts:

- the content of that key, an arbitrarily long string,
- the IP address where the key node is located,
- a routing table,
- a compressed routing table.

Routing tables play a critical role in any P2P overlay network. This is also the case with the DTSBT overlay. Especially, the size of the routing table in the DTSBT overlay is not relational with the total length of keys contained in the current routing table, but is proportional to the number of those keys. This desirable feature makes it possible that the routing table with a relatively small size can contain a huge number of keys. As a result, the scalability problem can be easily and elegantly solved. There are two types of routing tables in the DTSBT overlays: routing tables that are triggered by the arrival of messages and run in the main memory, and compressed routing tables that are stored in the external memory to save the cost of CPU cycles and the overhead of DTSBT packet.

A routing table mainly includes two parts: a Patricia Trie and some IP addresses. As Figure 4.1 shows, it has the following fields:

- *pat_root* points to the main memory address of the root of the Patricia Trie contained in the current routing table,
- *is_btree_leaf* checks whether or not the current routing table is the internal node or the leaf in the virtual distributed B-tree,
- *is_big* checks whether or not the all keys contained in the current routing table are lexicographically bigger than its parent key,
- *size* indicates how many keys are contained in the current routing table,
- *parent_address* records the IP address of the key node that contains the parent key,
- *addresses* records the IP addresses of the keys contained by the group where the current key locates,
- *children_addresses* records the IP addresses of the children groups that the current group links.

The Patricia Trie to which the field *pat_root* points is very useful because the routing table uses it as an inner routing tool to drive any routing process in a routing table. A Patricia Trie elegantly stores the lexicographic relationship of a group of keys with a small cost of space storage. As Figure 4.1 shows, a Patricia Trie includes the following field:

- *is_pat_leaf* checks whether or not the current node of the Patricia Trie corresponds to a key,
- *position* is the bit length of the substring or the string obtained by concatenating the edge label in the downward path leading to the current node,
- *inside_pos* presents the current node's sorted position in the lexicographic order among the keys contained by the current Patricia Trie,
- *pat_child* links the two children of the current node.

Also as Figure 4.1 shows, a compressed routing table can be depicted as the following fields: the fields *is_btree_leaf*, *is_big*, *size*, *parent_address*, *addresses*, and *children_addresses* are the counterparts of *is_btree_leaf*, *is_big*, *size*, *parent_address*, *addresses*, and *children_addresses* in its corresponding routing table. Furthermore, a compressed

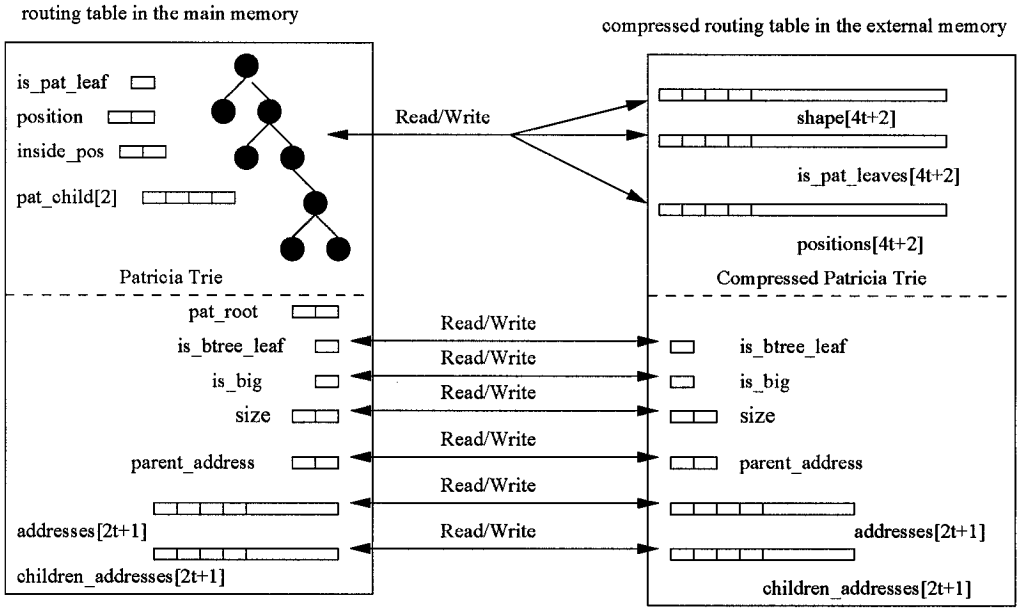


Figure 4.1: Routing tables

routing table uses a compressed Patricia Trie (including three parts the fields *shape*, *is_pat_leaves*, and *positions*) to store the corresponding Patricia Trie into the external memory.

There are two processes in DTSBTs: the routing table reading process and the routing table writing process. The former process performs the followings when a message is arriving at the current key node:

- reading the compressed routing table belonging to the current key node from the external memory into the main memory,
- decompressing the compressed Patricia Trie to create a Patricia Trie in the main memory,
- copying other counterpart fields from the compressed routing table to the routing table.

The latter process does the following when the event procedure is finished:

- compressing a Patricia Trie into several arrays in a compressed routing table,

- copying other counterpart fields from the routing table to the compressed routing table,
- writing the compressed routing table into the external memory.

4.2 Message-Passing Distributed System

A message-passing distributed system may operate in synchronous, asynchronous, and partially synchronous modes. In the synchronous mode, the execution is completely lockstep and each node proceeds in syschronous rounds, in which each node sends message to its outgoing neighbours, and then waits for the arrival of messages from its incoming neighbours, and finally performs some computation upon the receipt of the messages. On the other side, in the asynchronous mode, each node execute in arbitrary order at an arbitrary rate. The partially synchronous systems work at an intermediate degree of synchrony, where there are restrictions on the relative timing events [12].

To make the analysis of a DTSBT P2P overlay network simple and abstract, the message-passing distributed system of a DTSBT P2P overlay network employs synchronous mode. As a result, both joining and leaving operations have to use lock mechanism.

The message system is an event-driven system, which is triggered by the field *type* contained by the DTSBT packet header in the received packet. Packet Headers included in a packet for a particular protocol are provided by protocol developer to allow the protocol to perform packet-level operations. A DTSBT packet header or a DTSBT message at most includes four parts:

- *type* indicates the type of the current message,
- *content* records the content of the key contained in the current message,
- *saddr* presents the IP address of the source node that sends the current message,
- *crt* contains a compressed routing table.

As a DTSBT key node receives a DTSBT packet, one says that the current DTSBT key node receives a DTSBT message. As a DTSBT key node receives a DTSBT message, a relative event is triggered. And then the corresponding event procedure is executed at that key node. There are many types of messages in the DTSBT message system, which will be illustrated as the following.

Messages *i*, *e*, *m*

Whenever one intends to join a new key into the current DTSBT system from any IP node, it will send an i type message from that IP node to the system, or more exactly, to the nearest DTSBT node. There are two other messages involving this joining process: message e and message m.

A type of i DTSBT packet header includes three parts:

- the value of *type* is i
- *saddr* presents the IP address of the source node that intends to insert a new key into the current DTSBT system,
- *content* records the conent of the new key.

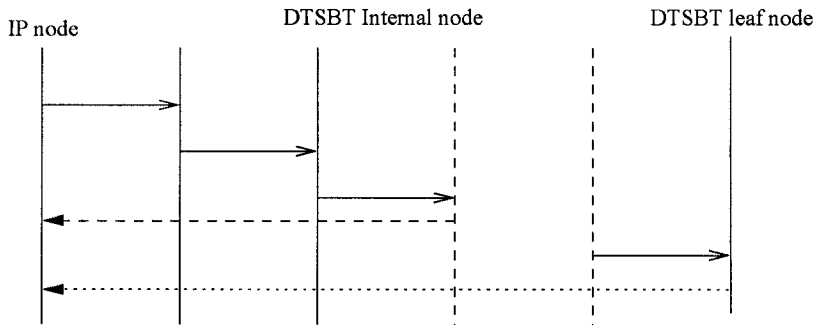
A type of e DTSBT packet header includes two parts:

- the value of *type* is e,
- *saddr* presents the IP address of the DTSBT key node whose key matches that new key.

A type of m DTSBT packet header includes two parts:

- the value of *type* is m,
- *crt* contains a compressed routing table.

The scenario of joining a new key into the current DTSBT system is depicted as Figure 4.2 (a). This scenario involves three types of messages: the solid horizontal line presents an i type message, the dash horizontal line presents an e type message, and the dot horizontal line presents an m type message. One i message starts at any IP node, ends at an internal node or a leaf node, and passes several internal nodes. From Figure 4.2 (b), one can know that if the current routing table contains a key which matches the new key, then the scenario is terminated and an e type message is directly sent to the source node. Figure 4.2 (c) presents the event procedure when a key node receives an e type message, and Figure 4.2 (d) presents the event procedure when a key node receives an m type message.



(a)

Event procedure of receiving an i type message in DTSBTs
1. if the current routing table contains a key that matches the new key
2. then the current key node sends an e type message to the source node
3. else if the current routing table is not a leaf
4. then forward that i type message to another node
5. else insert the new key into the current routing table
6 multicast the current compressed routing table to all neighbour key nodes

(b)

Event procedure of receiving an e type message in DTSBTs
1. print "The key is existing in the system!"

(c)

Event procedure of receiving an m type message in DTSBTs
1 write the compressed routing table contained in the current message into the external memory to replace the old compressed routing table or create a new compressed routing table

(d)

Figure 4.2: The scenario and procedures of joining a new key into a DTSBT system

Messages *s*, *f*, *u*

Whenever one attempts to search for a key in the current DTSBT system, it will send an s type message to the system, or more exactly, to the nearest DTSBT node. There are two other messages involving this search process: message f and u.

A type of s DTSBT packet header includes three parts:

- the value of *type* is s,
- *saddr* presents the IP address of the source node,
- *content* records the content of the queried key.

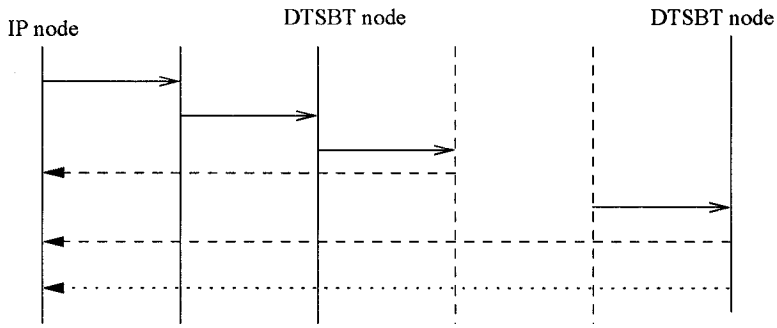
A type of f DTSBT packet header includes two parts:

- the value of *type* is f,
- *saddr* presents the IP address of the key node that sends this f message.

A type of u DTSBT packet header includes two parts:

- the value of *type* is u,
- *saddr* presents the IP address of the key node that sends this u message.

The scenario of searching for a key in the current DTSBT system is depicted as Figure 4.3 (a). This scenario involves three types of messages: the solid horizontal line presents an s type message, the dash horizontal line presents an f type message, and the dot horizontal line presents a u type message. One s message starts at any IP node, ends at an internal node or a leaf node, and passes several internal nodes. From Figure 4.3 (b), one can know that if the current routing table contains a key that matches the queried key, then the scenario is terminated and an f type message is sent to the source node. Otherwise, the process continues until a leaf. At a leaf, if the current routing table contains a key that matches the queried key, then an f type message is sent to the source node; otherwise a u type message is sent to the source node. Figure 4.3 (c) presents the event procedure when a key node receives an f type message, and Figure 4.3 (d) presents the event procedure when a key node receives an u type message.



(a)

Event procedure of receiving an s type message in DTSBTs
1. if the current routing table contains a key that matches
   the queried key
2. then the current key node sends an f type message to
   the source node
3. else if the current routing table is not a leaf
4. then forward that s type message to anothe key node
5. else send a u type message to the source node

(b)

Event procedure of receiving an f type message in DTSBTs
1. print "The queried key is found!' and the IP address of
   that key node

(c)

Event procedure of receiving a u type message in DTSBTs
1. print "The queried key is not found!'

(d)

Figure 4.3: The scenario and procedures of searching for a key in the DTSBT system

Messages x, y, r

After a new key is successfully inserted into the routing table of a key node, the key node will multicast the updated compressed routing table to all its neighbour key nodes. This case happens when the size of the current routing table is not more than $2 \times t - 1$, where t is *minimum degree* of the virtual B-tree. However, before insertion, if the size of the current routing table is more than $2 \times t - 1$, the splitting process should be done. The splitting process involves three types of message: message x, y, r.

Figure 4.4 (a) dash horizontal line presents an x type message, which is sent to the parent key node of the current key node when the key of the current key node is the middle key of the current routing table. The message x includes two parts:

- the value of *type* is x,
- the value of *content* is the content of the key of the current key node.

Figure 4.4 (a) dot horizontal line presents a y type message, which is sent to the the current key node's neighbour key node which contains the middle key of the current routing table. The message y includes two parts:

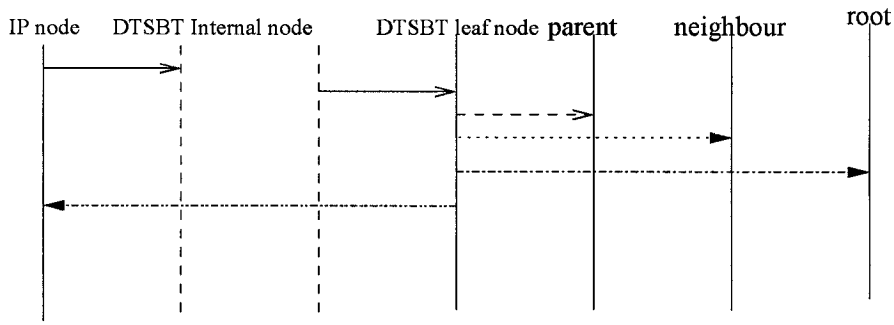
- the value of *type* is y,
- the value of *content* is the content of the new key.

Figure 4.4 (a) dash-dot horizontal line presents an r type message, which is sent to the current key node's neighbour key node which contains the middle key of the current routing table on condition that the current routing table is the root of the virtual B-tree. The message r includes two parts:

- the value of *type* is r,
- the value of *content* is the content of the new key.

4.3 Routing

The key idea of the DTSBT overlay is that its routing protocol employs a hybrid distributed data structure called DTSBT data structure, as its core data structure. DTSBT data structure employs a Patricia Trie as its inner routing construction inside a routing table, and uses a virtual B-tree as its outer routing construction among different routing



(a)

Event procedure of a leaf's receiving an i type message in DTSBTs
1. when the size of the current routing table is full before insertion
2. if the current routing table is the root of the virtual B−tree
   and the current key node contains the middle key of the routing table
3. then split the current routing table into two new routing tables
   and insert the message key into the left or right new routing table
4. the current key node become the root of the virtual B−tree
5. multicast the left and right new compressed routing tables
6. if the current routing table is the root of the virtual B−tree
   and the current key node does not contain the middle key of the routing table
7. then send a r type message with the new key to the middle neighbour key node
8. if the current routing table is not the root of the virtual B−tree
   and the current key node does not contain the middle key of the routing table
9. then send a y type message with the new key to the middle neighbour key node
10. if the current routing table is not the root of the virtual B−tree
    and the current key node contains the middle key of the routing table
11. then split the current routing table into two new routing tables
    and insert the message key into the left or right new routing table
12. send an x type message with the current key to the parent key node
13. multicast the left and right new compressed routing tables

(b)

Event procedure of receiving an x type message in DTSBTs
same as (b)

(c)

Event procedure of receiving a y type message in DTSBTs
same as (b)

(d)

Event procedure of receiving a r type message in DTSBTs
same as (b)

(e)

Figure 4.4: The scenario of splitting process in a DTSBT system

tables. As a result, each routing process generally contains two routing sub-processes. The first takes place in the Patricia Trie and comprises traversing from the root of the Patricia Trie to a node to which a key corresponds. The second, in a virtual distributed B-tree, involves moving downward from the current routing table to its children routing table, or moving upward from the current routing table to its parent routing table.

The key difference between the virtual distributed B-tree of the DTSBTs and a general B-tree is that for the latter each routing process begins from the root of the B-tree, whereas in the DTSBTs each routing process can start from any node of the virtual distributed B-tree. As a result, in the DTSBTs each routing process can move from anywhere to anywhere in the virtual distributed B-tree.

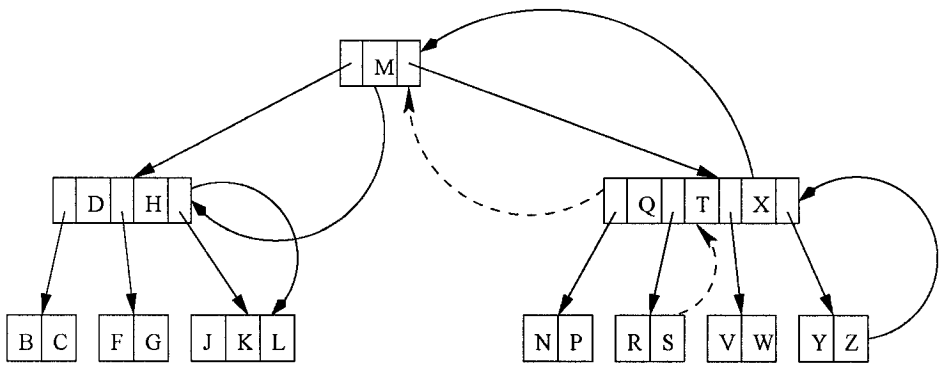


Figure 4.5: Routing in DTSBT

Figure 4.5 shows two routing cases in a DTSBT. In the case where key node Z wants to query the key L, as the solid circles show, the routing path is from the group (Y, Z) to the group (Q, T, X), then from the group (Q, T, X) to the group (M), then from the group (M) to the group (D, H), finally from the group (D, H) to the group (J, K, L). In the case where key node S wants to query the key M, as the dash circles show, the routing path is from the group (R, S) to the group (Q, T, X), then from the group (Q, T, X) to the group (M). Obviously, the first case is the worst case. Generally, the worst case is the routing process from the leaf of one side of the virtual distributed B-tree to the leaf of the other side of that B-tree. In general, the routing process in DTSBT can be described as the Table 4.1.

```
Procedure routing (Y)
1.  while the current key group is not a leaf of the virtual B-tree do
2.    compare the queried key Y with the parent key P
3.    if (routing tale.is_big = false and Y is lexicographically greater than P)
         or (routing table.is_big = true and Y is lexicographically smaller than P)
4.      then moving upward from current key group to the key group at which P locates
5.           go to 2.
6.      else search for Y in the current routing table using its Patricia Trie
7.           if a key X in the routing table = Y
8.             then jump from the loop and the routing is terminated
9.             else from the current routing table obtain the key Z
                    that is lexicographically closest to Y
10.                 obtain the children address corresponding to the key Z
11.                 move downward from the key group to the key group
                     to which the children address of the key Z points
```

Table 4.1: The routing procedure

4.4 Joining in the DTSBTs

There are two cases during the process of a new key's joining the DTSBT overlay. The first case is relatively easy. When a new key is inserted into a routing table, if the size of the current routing table is smaller than $2 \times t - 1$ then the new key is inserted into that routing table. After that, the current key node multicasts an m type message to all its neighbour key nodes in order to update their routing tables.Figure 4.6 (a), (b) show this process (assume the *minimum degree* of the virtual B-tree $t = 2$). Finally, the key group (R, T) becomes (R, S, T).

The second case is more complicated because splitting a B-tree node is involved. Before a new key is inserted into a routing table, if the size of the routing table is equal to $2 \times t - 1$, then the current key group or the current B-tree node should be split into two new key groups and the middle key should be inserted into its parent key groups. Such splitting process sometimes is recursively executed. More importantly, the splitting is the only means by which the virtual B-tree grows and the number of key groups increases.

Figure 4.6 (c), (d), (e), (f), (g) show this case.

- Figure 4.6 (c) shows the initial state that one wants to inset the key Q at the key group (R, S, T).

- Figure 4.6 (d) shows that first the key group (R, S, T) is split into two new key groups (R) and (T), then insert the key Q into the key group (R) and the key group (R) becomes (Q, R), finally, try to insert the middle key S into the parent

key group (P, X, Z).

- Figure 4.6 (e) shows that the middle key S is inserted into the key group (P, X, Z). First, the key group (P, X, Z) is split into two new key groups (P) and (Z), then insert the key S into the key group (P) and the key group (P) becomes (P, S), further, let the left children of the key S point to the key group (Q, R) and let the right children of the key S point to the key group (T), finally, try to insert the middle key X into the parent key group (C, E, M).

- Figure 4.6 (f) shows that the middle key X is inserted into the key group (C, E, M). First, the key group (C, E, M) is split into two new key groups (C) and (M), then insert the key X into two new key groups (C) and (M), then insert the key X into the key group (M) and the key group (M) becomes (M, X), further, let the left children of the key X point to the key group (P, S) and let the right children of the key X point to the key group (Z), finally, try to insert the middle key E into a parent key group.

- Figure 4.6 (g) shows that because the old key group (C, E, M) is the root of the virtual B-tree, the key E should become a new key group (E) and becomes the new root of the virtual B-tree. Further, let the left children of the key E point to the key group (C) and let the right children of the key E point to the key group (M, X).

The general joining procedure is illustrated as Table 4.2.

4.5 DTSBT Analysis

In this section, this thesis delves deeper and proves that the DTSBT is scalable, resilient to failures, decentralized, and self-organized.

4.5.1 Scalability

Theorem 1 *The routing efficiency of inserting or searching for a key in the DTSBT overlay is* $O(2 \times \log_B N)$, *regardless of key content length, where* B *is the number of the keys that each group can at most contain, and* N *is the size of the key identifier space.*

Proof: As a new key k is inserted or queried in the system at any IP node of the Internet, some bootstrap mechanism is employed to catch the IP node, which has the

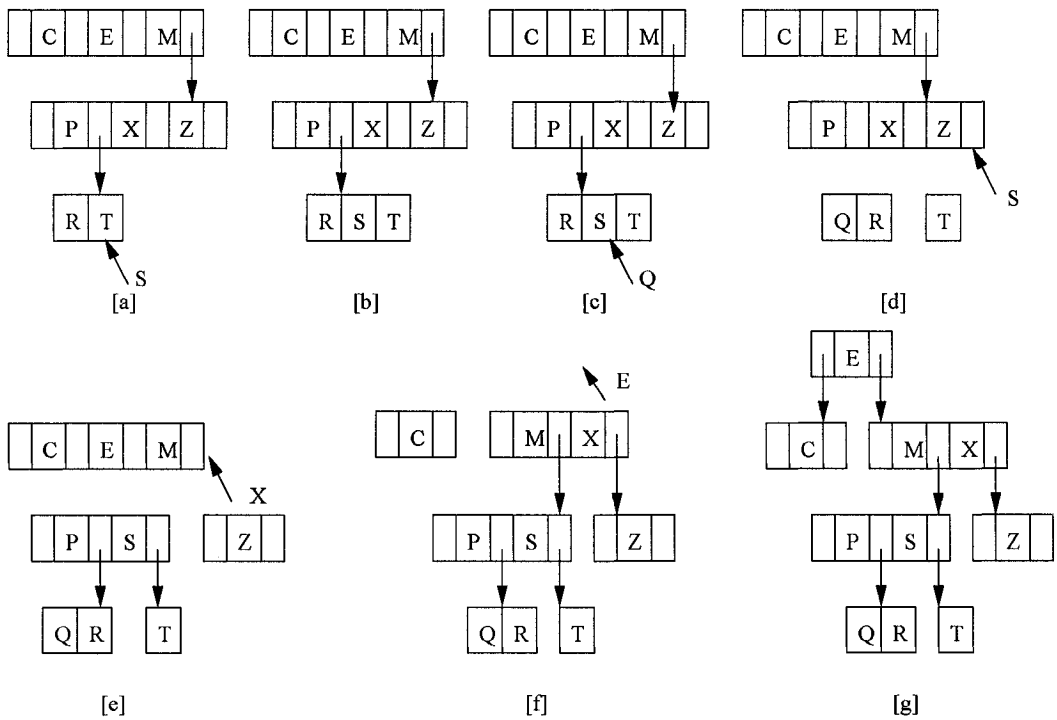


Figure 4.6: Joining in the DTSBTs

```
Procedure Joining (k)
1. try to find an IP node which contains one or more key nodes and has the
   shortest IP distance from the current IP node by using some bootstrap mechanism
2. move to a leaf of the virtual B-tree employing the DTSBT routing. During thisprocess,
   if one key in a key group matches the inserted key k, the joining process is terminated
3. if the size of the current routing table < 2t - 1
4.   then insert k into the current routing table
5.        multicast the current compressed routing table to all neighbours
6.        jump from the loop
7.   else split the routing table into two new routing tables
8.        insert k into one of those two new routing tables
9.        multicast those two new compressed routing table to all their neighbours
10.       insert the middle key m into the parent key group
11.       left the left and right children of m respectively point to its children group
12.       repeat step 3-11
```

Table 4.2: The joining procedure

closest distance from the current IP node, and which contains one or more DTSBT key nodes. In the worst case, the process of inserting or searching for a key k starts from the leftmost/rightmost leaf of the virtual distributed B-tree of the DTSBT overlay while the key will be inserted or is located in the rightmost/leftmost leaf of that virtual distributed B-tree. As a result, the routing path is like this: moving upward from the leftmost/rightmost leaf to the root layer-by-layer first, and then moving downward from the root to the rightmost/leftmost leaf layer-by-layer. Because the height of the virtual distributed B-tree is $\log_B N$, each such process needs $2 \times \log_B N$ hops pathlength.

Theorem 2 *The routing efficiency of inserting or searching for a key in the DTSBT overlay is* $O(2 \times (\log_B N)(1 + L/P))$, *with considering the key content length (assume that average key content length is* L, *and each packet can at most have a size* P*). The other parameters (*B *and* N*) are the same as defined in Theorem 1.*

Proof: The worst case is the *lcp* string does not exist in the current key node and it must be retrieved from another key node. Further, assume the average key content length L is far greater than the packet size P. Thus, for each key group or B-tree node, L/P network accesses are needed. As a result, the routing efficiency is $O(2 \times (\log_B N)(1 + L/P))$.

Theorem 3 *The storage efficiency of a DTSBT overlay is* $O((8 \times t + 5) \times C + (4 \times t + 3) \times I + (4 \times t + 3) \times A)$ *bytes, where* t *is the* minimum degree *of the virtual B-tree,* C *is the number of bytes that one character needs,* I *is the number of bytes that one integer variable needs,* A *is the number of bytes that one IP address needs.*

Proof: From the Figure 4.1, the field *shape* needs at most $4 \times t + 2$ characters storage, the field *is_pat_leaf* needs at most $4 \times t + 2$ characters storage, the field *is_btree_leaf* needs 1 character storage, the field *is_big* needs 1 character storage, these four fields need at most $8 \times t + 6$ characters storage. The field *positions* needs at most $4 \times t + 2$ integer variable storage, the field *size* needs 1 integer variable storage, these two fields totally need at most $4 \times t + 3$ integer variable storage.The field *parent_address* needs 1 IP address storage the field *addresses* needs at most $2 \times t + 1$ IP address storage, the field *children_addresses* needs at most $2 \times t + 1$ IP address storage, these three fields totally need $4 \times t + 3$ IP addresses storage. As a result, the total storage needed is $O((8 \times t + 6) \times C + (4 \times t + 3) \times I + (4 \times t + 3) \times A)$ bytes.

Theorem 4 *Without splitting the maintenance of a DTSBT P2P overlay network after an inserting process is* $O(B)$, *where* B *is the number of the keys that each routing table can at most contain.*

Proof: After a successful inserting process, the current key node should multicast its updated compressed routing table to all its neighbour key nodes contained in the current routing table. Because there are at most B keys contained in each routing table, for maintaining a DTSBT overlay network, the current key node should send $O(B-1) = O(B)$ m type messages to its $O(B-1) = O(B)$ neighbours.

Theorem 5 *With splitting the maintenance of a DTSBT P2P overlay network after an inserting process is $O(h \times B)$, where B is the number of keys that each routing table can at most contain, and h is the height of the virtual distributed B-tree to which the DTSBT P2P overlay network corresponds.*

Proof: In the worst case, an inserting operation can lead to h times splitting processes from the bottom to the top in the virtual distributed B-tree to which the DTSBT P2P overlay network corresponds. For one time splitting process, $O(B)$ is needed. As a result, with splitting, the total maintaining efficiency is $O(h \times B)$.

Based on above theorems, the DTSBT is highly scalable because of the following reasons:

- the storage of a routing table is not related to the total length of all keys strings in the key space, but is proportional to the number of the keys that the routing table contains. As a result, a routing table that needs very small size storage can store the routing information for a large number of keys.
- any basic operation (inserting, searching) has the routing efficiency of $O(2 \times log_B N)$ hops pathlength in the worst case,
- the maintaining operation has the maintenance efficiency of $O(h \times B)$ hops in the worst case,
- it is *extendible* because the key identifier space is *infinite* and any key that is lexicographically smaller or greater than all the keys in the current key space can be inserted into the key space,
- it is *adaptive* because the key identifier space is *continuous* and any key that is lexicographically between any pair of keys in the key space can be inserted into the key space,
- and the only relationship between any pair of keys is simply lexicographic. This means that the routing information in a routing table is local, but is not global. As

a result, any modified operation (such as inserting) can only need to update local information and consequently the cost is relatively low.

4.5.2 Other Features of the DTSBTs

For evaluating resilience to failure, one needs to evaluate whether the routing protocol can continue to work (with acceptable routing efficiency) as nodes fail without the need for other nodes to establish other neighbours to compensate; that is the neighbour nodes know that a node has failed but don't establish any new neighbour relations with other nodes [25]. In the DTSBTs, if each routing table stores B keys, then each routing table has B backups. This means that even if some key nodes fail, the DTSBTs can continue to work with almost the same routing efficiency using another neighbour key node's routing table. Consequently, the DTSBTs has high resilience to failures.

If there is a hotspot in the query pattern, with a certain key be requested very oftern, then the node holding that key may become overloaded. Do routing hotspots exist in the DTSBTs? No. Each key group that contains B keys, has exactly the same routing table with B backups. Thus, as a DTSBT key node in that key group becomes a hotspot, query can use the routing table of its neighbours.

The DTSBTs are highly decentralized because each key is merely stored at the IP node where that key is created. Furthermore, the routing table does not contain the content of the key string.

The DTSBTs has high self-organization because the data or the content of a key can merely be inserted, modified or released by/from the provider/owner.

Chapter 5

Patricia Trie Design and Implementation

Patricia Tries [22] play a crucial role in the DTSBTs because each key node's routing table contains a Patricia Trie that efficiently drives the subsequent search either for the keys in the same group or for the keys in both the children groups and the parent group. Moreover, the Patricia Trie makes the space occupied by a routing table proportional to the number of keys contained by the routing table instead of these keys' total length so that a routing table with a relatively small size can contain a huge number of keys. Since Donald R. Morrison first gave the description of Patricia [22] in 1968, many implementations for Patricia Tries have been proposed. One contribution of this thesis is to present a novel implementaion for a type of Patricia Tries in which either leaves or internal nodes can correspond to a key (string). Especially, a novel method of splitting such an extremely irregular Patricia Trie is first proposed in detail by this thesis.

5.1 Binary Patricia Tries in DTSBTs

In this section, from a high level of generality, the overview of what is the irregular Patricia Trie employed by the routing table in the DTSBTs, and how to create this binary Patricia Trie is presented.

5.1.1 Tries and Compressed Tries

A Trie (diminutive form of retrieval) [14] as a multi-way rooted tree T, in which each node has somewhere between 0 and N children, is designed to effectively store a set of

associative strings. All edges of T are assigned labels in the current character set such that all the edges leading to the children of a particular node receive different labels. All the descendants of any one node have a common prefix of the string associated with that node, and the root is associated with the empty string. Strings are stored as root-to-leaf paths in the Trie so that, if the string s is stored in T, there is a leaf v in T such that the sequence of edge labels encountered on the path from the root of T to v is precisely the string s. Figure 5.1 (a) shows an example of a Trie, in which the set of strings is $\{abaaaa, ababa, abacab, babbbb, babcb\}$ based on the character set $\{a, b, c\}$.

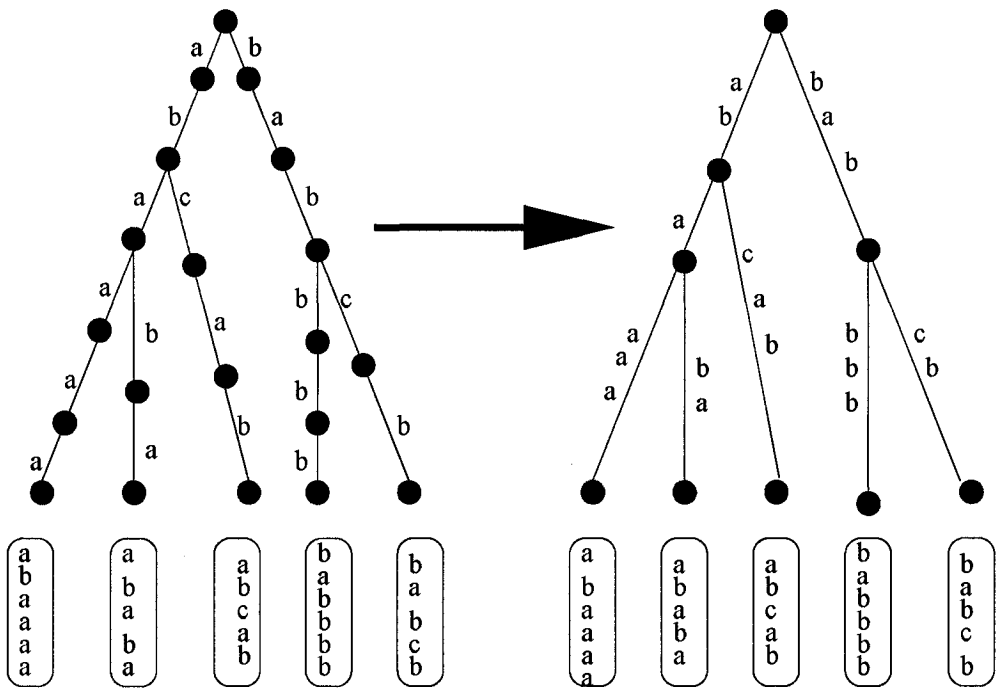


Figure 5.1: (a) an example of a Trie and (b) an example of a compressed Trie

A compressed Trie T is a simple variant on a Trie. In this particular Trie, all paths whose interior vertices have only one child are compressed into a single edge. To clarify this point, one now labels the edges of the Trie with substrings, so that the string corresponding to a leaf v is the concatenation of all the edge labels one encounters on the path from the root of T to v. Figure 5.1 (b) shows an example of a compressed Trie, which is converted from the Trie illustrated as Figure 5.1 (a).

5.1.2 Patricia Tries

A Patricia Trie [22] is a variation of a compressed Trie, in which each node is labeled by an integer that is equal to zero (the label of the root) plus the length of the substring obtained by concatenating the arc labels in the downward path leading to the current node, and for each edge, all characters are deleted except the first one, refered to as the branching character. Figure 5.2 (a) shows an example of a Patricia Trie, which is converted from Figure 5.1 (b), and includes an additional string (bab) that is a substring of the strings (babbbb) and (babcb).

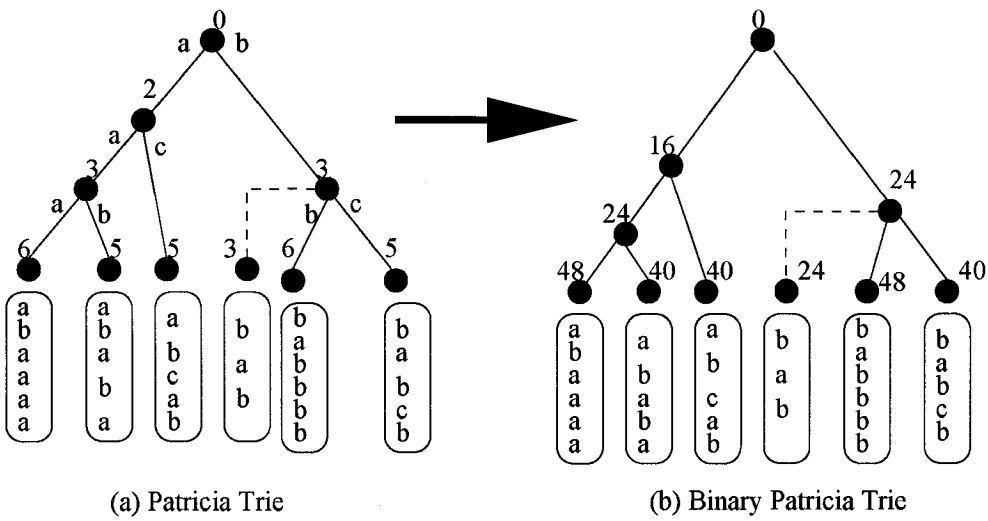


Figure 5.2: (a) an example of a Patricia Trie and (b) an example of a binary PatriciaTrie

5.1.3 Binary Patricia Tries in DTSBTs

Finally, based on the illustration described above, the method of creating a binary Patricia Trie in the DTSBT is demonstrated as below.

- First, let Σ denote an arbitrarily large ordered alphabet of characters, and let $\triangle$ denote the set of strings which sequentially consist of arbitrarily long characters drawn from Σ. he set $\{0, 1\}$ is employed to represent every character in Σ: following a rule similar to that representing the ASCII Code Table, each character in Σ can be encoded into an 8-bit long sequence of 0 or 1, so that any one string in $\triangle$ can be

turned into an $8 \times n$-bit long 0 or 1 sequence, where n is the length of the current string.

- Second, a binary Trie, in which each internal node only has two branches: the left one corresponding to the suffix substring whose first bit is 0 and the right one corresponding to the suffix substring whose first bit is 1, is created to store all strings in $\triangle$. Further, this binary Trie is converted into a compressed binary Trie, and this compressed binary Trie is continuously converted into a binary Patricia Trie in which each node is assigned a label with the value equal to the length of the substring obtained by concatenating the edge labels in the downward path leading to the current node, and the labels of all edges are deleted.

Such a binary Patricia Trie in the DTSBT has the following features:

- the root does not correspond to any string in $\triangle$,
- each leaf corresponds to a string contained in $\triangle$,
- some internal nodes called pure internal nodes, correspond to the substrings which are the common prefixes of some strings contained in $\triangle$,
- some internal nodes correspond to the strings contained in $\triangle$.

Figure 5.2 (b) shows an example of a binary Patricia Trie in the DTSBT, in which the internal node with the value 24 in the right side of this binary Patricia Trie corresponds to the string (bab).

5.2 Nodes in Patricia Tries

Nodes in the binary Patricia Trie in the DTSBTs play a very important role because they are the fundamental blocks of creating such a binary Patricia Trie. As Table 5.1 shows, this thesis employ a C programming language structure *pt_node* to describe the constructure of those nodes:

- The field *position* is the bit length of the substring or the string obtained by concatenating the edge label in the downward path leading to the current node. For the root, the field *position* is assigned the value 0.

```
typedef struct tag_pt_node {
  int position;
  int address;
  char leaf;
  int inside_pos;
  struct tag_pt_node* children[2];
} pt_node;
```

Table 5.1: The construction of nodes in the binary Patricia Trie

- The field *address* is the IP address of the physical IP node that stores the key (string) to which the current Patricia Trie node corresponds, when the current Patricia Trie node corresponds to a string (key). In the event that the current Patricia Trie node corresponds to a substring, the field *address* is assigned the value -1.
- The field *leaf* indicates whether or not the current Patricia Trie node corresponds to a key in the key space of the DTSBT.
- The field *inside_pos* presents the current node's sorted position in the lexicographic order among the strings contained by the current Patricia Trie, when the current node corresponds to a string (key). In the event that the current Patricia Trie node corresponds to a substring, the *inside_pos* is assigned the value -1.
- The field *children* has at most two children: the left child pointing to the node that corresponds to the substring or the string whose first bit is 0, and conversely, the right child pointing to the node that corresponds to the substring or the string whose first bit is 1.

5.3 Blind Search in Patricia Tries

Let Σ denote an arbitrarily large ordered alphabet of characters, let $\triangle = \{X_1, ..., X_d\}$ denote the ordered set of strings which sequentially consist of arbitrarily long characters drawn from Σ, and let $\leq_L$ denote the lexicographical order among the strings in $\triangle$. Given two strings X and Y that are not each other's prefix, one can define $lcp(X, Y)$ to be their *longest common prefix* length, that is, $lcp(X, Y) = k$, if and only if $X[1, k] = Y[1, k]$ and $X[k + 1] \neq Y[k + 1]$ [13]. The shorthand $max\_lcp(Y, \triangle)$ denotes the maximum among the *lcp*-values of Y and $\triangle$'s strings, that is $max\_lcp(Y, \triangle) = max_{X \in \triangle} lcp(Y, X)$.

Based on these definitions described above, this thesis proposes the procedure BLIND-SEARCH, which efficiently retrieves the IP address of the IP node storing the maximum *lcp* string of the string (key) set contained in the current routing table, for the queried string Y. This procedure takes a Patricia Trie PT and a queried string Y as its input parameters, and the IP address *Address* of the maximum *lcp* string in $\triangle$ as its output parameter, as Table 5.2 shows.

```
BLIND-SEARCH (PT, Y) --> (Address)
1  p <-- the root of PT, i <-- 0
2  while p != NIL and i < |Y| * 8
3    parent <-- p
4    if Y[i+1] = 0
5      then p <-- p.children[left]
6      else p <-- p.children[right]
7    if p != NIL
8      then i <-- p.position
9  if p = NIL
10   then p <-- parent
11 if p != NIL and p.leaf = 1
12   then Address <-- p.address, return Address
13   else while p != NIL and p.leaf = 0
14          p <-- p.children[left]
15        Address <-- p.address, return Address
```

Table 5.2: The BLIND-SEARCH procedure

From the procedure BLIND-SEARCH(PT, Y), one can find searching for the maximum *lcp* string for a queried string Y in a Patricia Trie PT is very efficient because such a search is a one time scan traversing PT from its root to one of its leaves in spite of how many strings are contained in PT. Moreover, in such a scan, comparisions between any one string in PT and Y only happen on some limited Patricia Trie nodes such that the scan is extremely speeded up.

5.4 Searching for a String in a Patricia Trie

The procedure PT-SEARCH is designed to search for a queried string Y in a given Patricia PT and return the result of whether or not there is a string in PT exactly matching Y. This procedure takes PT and Y as its input parameters, and has two output parameters: *found* indicating whethor or not a string matching Y exists in PT, and *pos_in_pt* which presents the matching string's sorted position in the lexicographic order among the strings containd by PT.

```
PT-SEARCH (PT, Y) --> (found, pos_in_pt)
1  BLIND-SEARCH (PT, Y) --> (Address), GET-STRING (Address) --> (X)
2  if X not in PT
3    then if Y[0] = 0
4           then return found <-- 0, pos_in_pt <-- 1
5           else return found <-- 0, pos_in_pt <-- |PT| + 1
6    else i <-- 0
7         while i < |X| and i < |Y|
8           if X[i+1] = Y[i+1] then i++ else break;
9         bit_position = i
10        i <-- 0, p <-- the root of PT
11        while p != NIL and i < bit_position
12          if Y[i+1] = 0
13            then p <-- p.children[left]
14            else p <-- p.children[right]
15          if p != NIL then i <-- p.position
16        if bit_position = |X| and bit_position = |Y|
17          then return found <-- 1, pos_in_pt <-- p.inside_pos
18          else if Y[bit_position + 1] = 0
19            then while p != NIL and p.leaf = 0
20                   p <-- p.children[left]
21                 return found <-- 0, pos_in_pt <-- p.inside_pos
22            else while p != NIL and p.leaf = 0
23                   p <-- p.children[right]
24                 return found <-- 0, pos_in_pt <-- p.inside_pos + 1
```

Table 5.3: The PT-SEARCH procedure

As Table 5.3 shows, the PT-SEARCH(PT, Y) procedure basically consists of two main phases. In the first phase, by using the BLIND-SEARCH procedurethe, maximum *lcp* string X is retrieved from another key node in the DTSBT. In the second phase, compare X with Y; if X matches Y, then return the corresponding information of X, otherwise, scan PT to find the string that is the lexicographically closest to Y (with high probability, that string is X).

5.5 Inserting a String into a Patricia Trie

The PT-INSERT procedure is designed to insert a new string Y into the current Patricia Trie PT. This procedure takes PT and Y as its input parameters, and returns an output parameter $SUCCESS$ to indicate whether Y has been successfully inserted into PT. It should be noted that PT-INSERT does not insert Y itself into PT, but rather inserts the IP address of the IP node that stores Y into PT and at the same time addes the lexicographic relationship information between Y and the existing strings contained by PT, into PT.

As Table 5.4 shows, there are several cases during inserting Y into PT:

- There is a string in PT which exactly matches Y. In this case, just return this information (lines 11-12).
- When all strings are lexicographically smaller than Y, insert Y into the leftmost position of PT; and conversely, when all strings are lexicographically greater than Y, insert Y into the rightmost position of PT (lines 4-6).
- If the maximum *lcp* string X and Y are not each other's prefix, create a new internal node and let this new internal node be the parent of X and Y (lines 20-29).
- If Y is a prefix of X, let X be the parent of Y (lines 30-35).
- If X is a prefix of Y, let Y be the parent of X (lines 36-41).

5.6 Splitting a Patricia Trie

Because a Patricia Trie with the parameter *minimum degree* t, can contain at most $2\times t-1$ strings and at least $t-1$ strings, as a new string is inserted into a full Patricia Trie which contains $2\times t-1$ strings, this full Patricia Trie should be split into three parts before the

```
PT-INSERT (PT, Y) --> (success)
1  BLIND-SEARCH (PT, Y) --> (Address), GET-STRING (Address) --> (X)
2  p <-- the root of PT
3  if X not in PT
4    then if X[0] = 0
5           then p.children[left].address <-- Y.address, p.children[left].position <-- |Y|
6           else p.children[right].address <-- Y.address, p.children[right].position <-- |Y|
7    else i <-- 0
8         while i < |X| and i < |Y|
9           if X[i+1] = Y[i+1] then i++ else break;
10        bit_position <-- i
11        if bit_position = |X| and bit_position = |Y|
12          then success <-- 0, return success
13        i <-- 0
14        while p != NIL and i < bit_position
15          parent <-- p
16          if Y[i+1] = 0
17            then p <-- p.children[left]
18            else p <-- p.children[right]
19          if p != NIL then i <-- p.position
20        if bit_position < |X| and bit_position < |Y|
21          then ALLOCATE-PAT-NODE () --> (internal), ALLOCATE-PAT-NODE () --> (leaf)
22               leaf.address <-- Y.address, leaf.position <-- |Y|
23               internal.position <-- bit_position
24               if p = parent.children[left]
25                 then parent.children[left] <-- internal
26                 else parent.children[right] <-- internal
27               if Y[bit_position + 1] = 1
28                 then internal.children[left] <-- p, internal.chidlren[right] <-- leaf
29                 else internal.children[left] <-- leaf, internal.children[right] <-- p
30        if bit_position < p.position and bit_position = |Y|
31          then ALLOCATE-PAT-NODE () --> (internal)
32               if p = parent.children[left]
33                 then internal.children[left] <-- p, parent.children[left] <-- internal
34                 else internal.children[right] <-- p, parent.children[right] <-- internal
35               internal.address <-- Y.address, internal.position <-- |Y|
36        if bit_position = p.position and bit_position < |Y|
37          then ALLOCATE-PAT-NODE () --> (leaf)
38               if Y[bit_position + 1] = 1
39                 then p.children[right] <-- leaf
40                 else p.children[left] <-- leaf
41               leaf.address <-- Y.address, leaf.position <-- |Y|
42 success <-- 1, return success
```

Table 5.4: The PT-INSERT procedure

insertion: a new Patricia Trie that contains the lexicograhically left $t-1$ strings of the old Patricia Trie, a string that is the lexicographically middle string of the old Patricia Trie, and another new Patricia Trie that contains the lexicographically right $t-1$ strings of the old Patricia Trie. This process is known as splitting a Patricia Trie, which is the most critical and difficult part of the DTSBTs. It is the most critical because without it, the process of inserting new keys into the DTSBTs cannot be correctly executed. It is the most difficult because a Patricia Trie contained by a routing table in the DTSBTs is an extremely irregular tree such that splitting that Patricia Trie is extremely complicated, and not a merely straightforward *cut and paste* process. Figure 5.3 illustrates an example of splitting a Patricia Trie, from which one can know the splitting process is by no means a easy *cut and past* process.

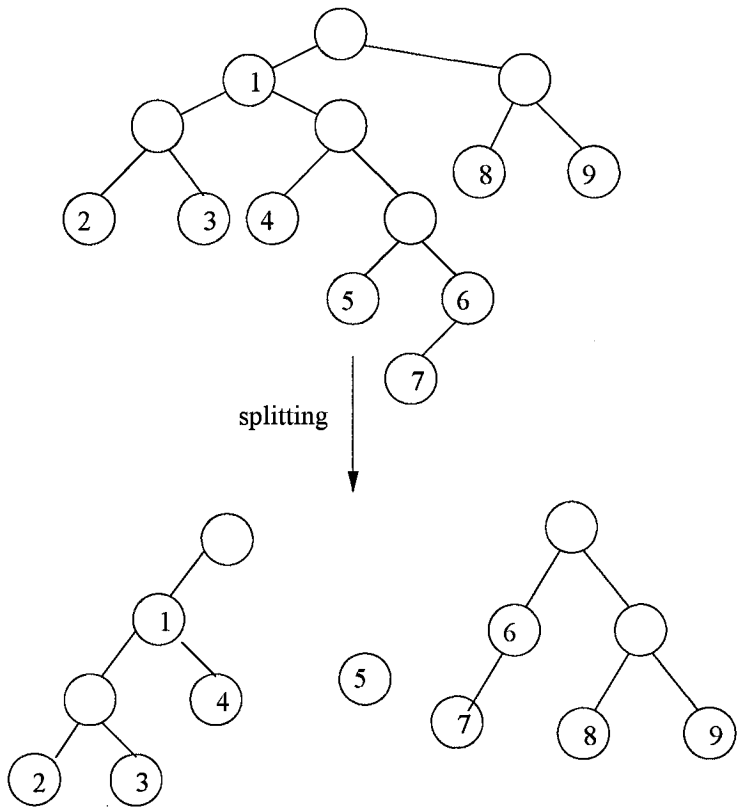


Figure 5.3: An example of splitting a Patricia Trie (minimum degree t = 5)

Like an *order-statistic tree* [11], this thesis uses similar augmenting and indicating information (such as *size*) to guide a one time scan through a Patricia Trie from the

root to the leaf containing the middle string and simultaneously split the Patricia Trie. Those augmenting and indicating information can be described using some augmenting and indicator variables as the following:

- *size* is stored into each node of the Patricia Trie as an augmenting variable, and records the number of the strings contained in the substree whose root is the current Patricia Trie node,
- *direction* indicates the next moving direction: the value of 0 for the left child direction and the value of 1 for the right child direction,
- *current_degree* indicates the splitting pointer's lexicographic position in the current subtree whose root is the current node.

In addition, some pointer variables are employed to store the current node information either in the old tree or in the new tree:

- *split_pointer* points to the current node in the split Patricia Trie,
- *parent_pointer* points to the parent node of *split_pointer*,
- *grand_pointer*, points to the parent node of *parent_pointer*,
- *root_pointer*, points to the root of the split Patricia Trie,
- *new_root_pointer*, points to the root of the new Patricia Trie,
- *new_split_pointer*, points to the current node in the new Patricia Trie.

Simply, splitting a Patricia Trie is a one time scan from the root to the leaf which stores the middle key and each step (or in each node of the split Patricia Trie) in this scan contains three phases:

- change the augmenting and indicating information for the next step,
- cut a part of the split Patricia Trie,
- create a part of the new Patricia Trie or paste the part of the split Patricia Trie into the the Patricia Trie.

Although it seems easy, since there are many possible cases in each step, splitting a Patricia Trie is really very complicated.

In the beginning, the initialization procedure works as the following: Assign *current_degree* the value of *DEGREE* and *direction* the value of 0. Let *split_pointer*, *parent_pointer*, *grand_pointer*, and *root_pointer* point to the root of the current Patricia Trie. Create a new Patricia Trie with its root only, and let *new_split_pointer* and *new_root_pointer* point to the root of the new Patricia Trie.

During the scan, when the current node has two branches, one should compare *current_degree* with the *size* of the child node on the *direction* branch to determine the direction of the next movement; on the other case, when the current node has only one branch (this node certainly contains a string) and the part of the Patricia Trie is a straight line or a zig-zag link, the comparison is not necessary.

Further, the procedure related to changing indicator variables works as below:

- for *direction*,

```
if the current node has only one child
  then do not change direction
  else if the size of the child on the direction branch < current_degree
         then if direction = 0
                then direction <-- 1
                else direction <-- 0
         else do not change direction
```

- for *current_degree*,

```
if the current node has only one child
   then if the size of the child on the direction branch > current_degree and direction = 0
        or the size of the child on the direction branch < current_degree and direction = 1
          then current_degree = current_degree - 1
          else do not change current_degree
   else if the size of the child on the direction branch > current_degree
          then if direction = 0
                 then current_degree = current_degree - 1
                 else do not change current_degree
          else if direction = 0
                 then current_degree = the size of the current node - current_degree
                 else current_degree = the size of the current node - current_degree - 1
```

- for *split_pointer*, *parent_pointer*, *grand_pointer*

```
let grand_pointer point to parent_pointer
let parent_pointer point to split_pointer
if the current node has only one child
  then let split_pointer point to that child
  else if the size of the child on the direction branch > current_degree and direction = 0
       or the size of the child on the direction branch < current_degree and direction = 1
         then let split_pointer point to the left child of the current node
         else let split_pointer point to the right child of the current node
```

5.7 Compressing and Uncompressing Binary Patricia Tries

Although binary Patricia Tries' properties are very promising, one will go a step further and uses a simple method for representing a binary Patricia Trie succinctly. In this way, one uses space more efficiently when storing Patricia Tries in the external memory. There are two procedures: PT-WRITE and PT-READ. This thesis uses the former to compress a Patricia Trie and write the compressed Patricia Trie into the external memory. The latter conversely, is used to read the compressed Patricia Trie from the external memory and uncompress it into a Patricia Trie in the main memory. Figure 5.4 shows these two procedures by an example.

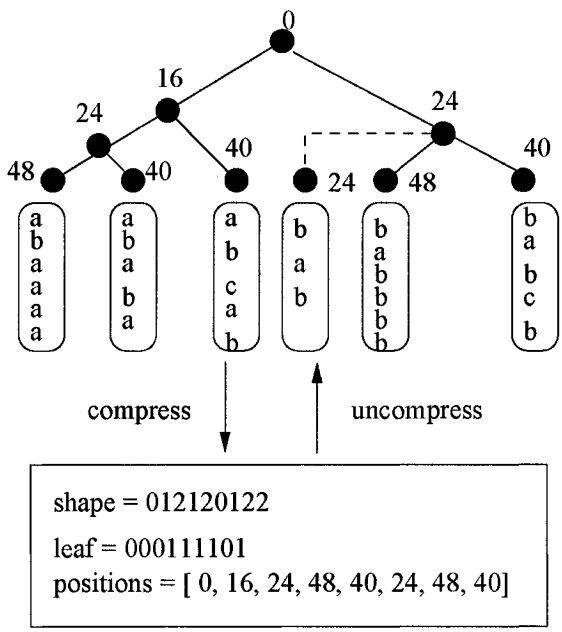


Figure 5.4: An Illustration of Compressing and Uncompressing a Patricia Trie

5.7.1 Structure of Compressed Binary Patricia Tries

As table 5.5 shows, one first introduces a constructure called *com_pt* to compress a Patricia Trie into several arrays:

- the field *shape* is used to record the shape of the current Patricia Trie,

```
typedef struct {
  char shape [DEGREE * 4 + 2];
  char leaf [DEGREE * 4 + 2];
  int positions [DEGREE * 4 + 2];
} com_pt;
```

Table 5.5: The constructure of a compressed Patricia Trie

- the field *leaf* is used to present the information of whether or not the nodes in the current Patricia Trie correspond to strings,
- the field *positions* is used to present the information of the bit lengths of the strings to which the nodes correspond.

5.7.2 Compressing Patricia Tries

Given a Patricia Trie *PT*, the PT-WRITE procedure compresses *PT* into a *com_pt* constructure, and then writes this constructure into a block stored in the external memory.

```
PT-WRITE (PT)
1  com_pt.shape[0] <-- 1, com_pt.positions[0] <-- 1, com_pt.leaf[0] <-- 1
2  PREORDER-WALK (PT) --> (com_pt)
3  POSTORDER-WALK (PT) --> (com_pt)
4  DISK-WRITE (com_pt)
```

Table 5.6: The procedure of compressing a Patricia Trie

As table 5.6 shows, for a given Patricia Trie *PT*, PT-WRITE works as follows:

- execute the initialization (line 1),
- by employing the procedure PREORDER-WALK (line 2), compress every node's *leaf* information into the array *leaf* and compress every node's *position* information into the array *positions*,
- by employing the procedure POSTORDER-WALK (line 3), compress the Patricia Trie's shape information into the array *shape*,
- Write the compressed Patricia Trie *com_pt* into a block stored in the external memory (line 4).

5.7.3 Decompressing Patricia Tries

The PT-READ procedure reads a compressed Patricia Trie *com_pt* from a block stored in the external memory, and uncompresses the compressed Patricia Trie into a Patricia Trie *PT*.

```
PT-READ() --> (PT)
1  DISK-READ () --> (com_pt)
2  PT-CREATE (com_pt) --> (PT)
3  PT-FILL (com_pt) --> (PT)
4  return PT
```

Table 5.7: The procedure of uncompressing a compressed Patricia Trie

As table 5.7 shows, PT-READ works as follows:

- read a compressed Patricia Trie *com_pt* from a block stored in the external memory (line 1),
- create the constructure of the Patricia Trie from *com_pt* by employing stack operations (stack-push-in and stack-pop-up) (line 2),
- by preorderly walking through the Patricia Trie, assign each node's *leaf* and *position* information to *PT* (line 3),
- return *PT* (line 4).

Chapter 6

Simulation and Experimental Results

6.1 NS-2 Simulation Environment

NS-2 [6] is a discrete event simulator targeted at networking research. NS-2 is an object oriented packet-level simulator and supports network protocol stack using protocol agents.

6.1.1 Dual Language and Object Oriented Simulator

NS-2 is an object oriented and dual language simulator, supporting two object oriented programming languages (C++ and OTcl), in which every network element (such as nodes, links, agents and packets) can be regarded as an object. The simulator supports a class hierarchy in C++ (also called the compiled hierarchy), and a similar class hierarchy within the OTcl interpreter (also called the interpreted hierarchy). The two hierarchies are closely related to each other, from the user's perspective, there is a one-to-one correspondence between a class in the interpreted hierarchy and one in the compiled hierarchy.

Why Two Languages?

NS-2 uses two languages because the simulator has two different kinds of things it needs to do. On one hand, detailed simulations of protocols require a system programming language, which can efficiently manipulate bytes, packet headers, and implement algo-

rithms that run over large data sets. For these tasks run-time speed is important and turn-around time (run simulation, find bug, fix bug, recompile, re-run) is less important [6].

On the other hand, a large part of network research involves slightly varying parameters or configurations, or quickly exploring a number of scenarios. In these cases, iteration time (change the model and re-run) is more important. Since configuration runs once (at the beginning of the simulation), run-time of this part of the task is less important [6].

NS-2 meets both of these needs with two languages, C++ and OTcl. C++ is fast to run but slower to change, making it suitable for detailed protocol implementation. OTcl runs much slower but can be changed very quickly (and interactively), making it ideal for simulation configuration. NS-2 (via tclcl) provides glue to make objects and variables appear on both languages [6]

Which Language for What?

Having two languages raises the question of which language should be used for what purpose. In general OTcl can be used for (1) configuration, setup and "one-time" stuff; (2) manipulating existing C++ objects. C++ can be used for (1) doing anything that requires processing each packet of a flow; (2) changing the behavior of an existing C++ class in ways that weren't anticipated.

6.1.2 Packet Headers

Objects in the class *Packet* are the fundamental unit of exchange between objects in the simulation. The class *Packet* provides enough information to link a packet onto a list, refer to a buffer containing packet headers that are defined on a per-protocol basis, and to refer to a buffer packet data. New protocols can define their own packet headers or can extend existing headers with additional fields [6].

New packet headers are introduced into the simulator by defining a C++ structure with the needed fields, defining a static class to provide OTcl linkage, and then modifying some of the simulator initialization code to assign a byte offset in each packet where the new header is to be located relative to others [6].

When the simulator is initialized through OTcl, a user may choose to enable only a subset of the compiled-in packet formats, resulting in a modest savings of memory during the execution of the simulation. Presently, most configured-in packet formats are

enabled. The management of which packet formats are currently enabled in a simulation is handled by a special packet header manager object. This object supports an OTcl method used to specify which packet headers will be used in a simulation. If an object in the simulator makes use of a field in a header which has not been enabled, a run-time fatal program abort occurs [6].

6.1.3 Agents

Agents represent endpoints where network-layer packets are constructed or consumed, and are used in the implementation of protocols at various layers. The class *Agent* has an implementation partly in OTcl and partly in C++. As a result, each existing network protocol over network layer, has the corresponding agent in the NS-2. Creating a new protocol should create an agent or extend an existing agent. The concept of *agent* makes NS-2 able to do simulation over different layers of network protocol stack.

6.2 P2P Overlay Network Simulation in NS-2

This section presents the procedure of doing network simulation for P2P overlay networks using NS-2. Ros and Ruiz's work [26], and Marc Greis's tutorial [16] have been refered.

6.2.1 Files and Directory

Once a new protocol is created, one directory should be built inside the NS2 base directory. For DTSBT, a directory, called dtsbt, is created under the ns-2.28 directory. Under this directory, the following should be created:

- *dtsbt.h* This is the header file where the routing agent, which performs the protocol's functionality, will be defined.
- *dtsbt.cc* In this file, the routing agent and Tcl hooks will actually be implemented.
- *dtsbt_pkt.h* In this file, all packets that the new protocol needs to exchange among nodes will be defined.
- *dtsbt_rtable.h* In this file, the routing table and the compressed routing table data structures will be defined.

- *dtsbt_rtable.cc* In this file, the basic operations about the routing table will be implemented.

6.2.2 Packet Headers

```
dtsbt/dtsbt_pkt.h
1  #ifndef __dtsbt_pkt_h__
2  #define __dtsbt_pkt_h__
3  #include <packet.h>
4  #define HDR_DTSBT_PKT(p) hdr_dtsbt_pkt::access(p)
5  struct hdr_dtsbt_pkt {
6    nsaddr_t saddr;   //Node which originally sends this packet
7    char type;  //Message type
8    char content[MAX_CHAR_LENGTH];
9    crt pkt_crt;
10   static int offset_;
11   inline static int& offset() { return offset_; }
12   inline static hdr_dtsbt_pkt* access(const Packet* p) {
13     return (hdr_dtsbt_pkt*) p->access(offset_);
14   }
15 };
16 #endif
```

Table 6.1: DTSBT Packet Header in NS-2

As Table 6.1 shows, lines 5-15 declare *hdr_dtsbt_pkt* structure which represents the new packet type that is defined for the DTSBT protocol. Lines 6-9 declare four raw attributes that the new packet has. Line 3 includes the file */common/packet.h* which defines *Packet* class. All packet headers are stored by *Packet* class, which uses an array of unsigned characters where packets' fields are saved. To access a concrete packet header it is necessary to provide the offset where it is located, as lines 10-14 do. Here a static offset, a member function to access it and a function which returns a *hdr_dtsbt_pkt* given a *Packet* are defined. Moreover, line 4 creates a macro to use this last function.

In addition, bind the new packet header to the Tcl interface should be done. This step is done by the following code as Table 6.2 shows.

6.2.3 Routing Agent

As Table 6.3 shows, a class *dtsbt* inheriting from class *Agent*, is declared for the DTSBT routing protocol. This class encapsulates its own IP address, the internal state, the content of key, and the compressed routing table.

```
/dtsbt/dtsbt.cc
1  int hdr_dtsbt_pkt::offset_;
2  static class DtsbtHeaderClass : public PacketHeaderClass {
3  public:
4    DtsbtHeaderClass() : PacketHeaderClass("PacketHeader/Dtsbt",
                                            sizeof(hdr_dtsbt_pkt)) {
5      bind_offset(&hdr_dtsbt_pkt::offset_);
6    }
7  } class_dtsbt_hdr;
```

Table 6.2: Packet Header Binding

```
1  class dtsbt : public Agent {
2    nsaddr_t  address;
3    int state;
4    crt local_crt;
5    char content[MAX_CHAR_LENGTH];
6  protected:
7    void send_dtsbt_pkt();
8    void recv_dtsbt_pkt();
9  public:
10   dtsbt(nsaddr_t);
11   int command(int, const char*const*)
12   void recv(Packet*, Handler*);
13 };
```

Table 6.3: Routing Agent

Tcl hooks

To bind this agent class to Tcl interface, a class inheriting from class *TclClass* must be declared as depicted in Table 6.4 to let the class *dtsbt* be able to be instantiated from Tcl.

```
1  static class DtsbtClass : public TclClass {
2  public:
3    DtsbtClass() : TclClass("Agent/Dtsbt") {}
4    TclObject* create(int argc, const char*const* argv) {
5      return (new Dtsbt((nsaddr_t)Address::instance().
               str2addr(argv[4])));
6    }
7  } class_dtsbt;
```

Table 6.4: DTSBT TCL class

Command Function

For every *TclObject* that is created, NS-2 establishes the instance procedure, *cmd*{}, as a hook to executing methos through the compiled shadow object. The procedure *cmd*{} invokes the method *command*{} of the shadow object automatically, passing the argument to *cmd*{} as an argument vector to the *command*{} method.

```
1  int dtsbt::command(int argc, const char*const*argv) {
2    if (argc == 2) {
3      if (strcmp(argv[1], "printnode") == 0 {
4        print_dtsbt();
5        return (TCL_OK);
6      }
7    }
     ......
8    return (Agent::command(argc, argv));
9  }
```

Table 6.5: Command Function

As Table 6.5 shows, the function is called with two arguments: the first argument (argc) indicates the number of arguments specified in the command line to the interpreter; the second arguments vector (argv) consists of (1) argv[0] contains the name of the method, "cmd", (2) argv[1] specifies the desired operations, (3) if the user specified any arguments, then they are placed in argv[2...(argc-1)]. The arguments are passed as strings; they must be converted to the appropriate data type.

If the operation is successfully matched, the match should return the result of the operation using methods described earlier. *command*{} itself must return either *TCL_OK* or *TCL_ERROR* to indicate success or failure as its return code. If the operation is not matched in this method, it must invoke its parent's command method, and return the corresponding result.

Receive Function

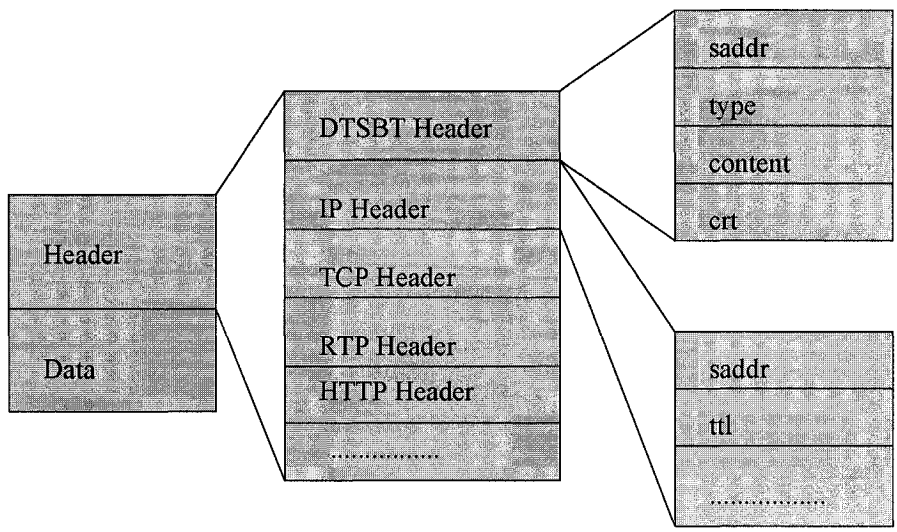


Figure 6.1: An example of packet header structure

As Table 6.6 shows, lines 2-4 respectively receive three types of packet headers: IP header, common header and DTSBT header. Lines 6-9 show that if there exists a loop, the packet must be dropped. Lines 10-18 show that if a packet includes DTSBT header, `recv()` function will execute DTSBT message system operations. Line 19 shows finally `recv()` should release the packet.

6.2.4 Needed Changes

In order to integrate the new routing protocol inside simulator NS-2, some changes must be done.

```
1  void dtsbt::recv(Packet* p, Handler* h) {
2    struct hdr_ip* ih = HDR_IP(p);
3    struct hdr_cmn* ch = HDR_CMN(p);
4    struct hdr_dtsbt_pkt* dh = HDR_DTSBT(p);
5    ......
6    if (ih->saddr() = address) {
7       drop(p, DROP_RTR_ROUTE_LOOP);
8       return;
9    }
10   if (ch->ptype() == PT_DTSBT) {
11     switch (dh->type) {
12       case 'i':
13       case 's':
14       case 'g'
15       case 'p'
16       ......
17     }
18   }
19   Packet::free(p);
20   return;
21 }
```

Table 6.6: Receive Fuction

Packet Type Declaration

In general, a constant is defined to indicate the new packet type in */common/packet.h.* First, try to find *packt_t* enumeration, where all packet types are listed. A new type *PT_DTSBT* must be added into this list as Table 6.7 shows.

```
/common/packet.h
1  enum packet_t {
2    PT_TCP,
3    PT_UDP,
4    ......
5    PT_DTSBT,
6    PT_NTYPE //This must be the last one
7  }
```

Table 6.7: First Change in Packet Header Configuration

Just below in the same file there is definition of **p_info** class. Inside constructor, a textual name for new packet type must be added as Table 6.8 shows.

```
/common/packet.h
1  p_info() {
2    name_[PT_TCP] = "tcp";
3    name_[PT_UDP] = "udp";
4    ......
5    name_[PT_DTSBT] = "dtsbt";
6    name_[PT_NTYPE] = "undefined";
7  };
```

Table 6.8: Second Change in Packet Header Configuration

Makefile

The object files entries must be added inside *OBJ_CC* variable in the *Makefile* for re-compile NS-2 as Table 6.9 shows.

```
Makefile
1  OBJ_CC = \tools/random.o tools/rng.o ... common/timer-handler.o \
2           ......
3           dtsbt/dtsbt.o dtsbt/dtsbt_rtable.o \
4           $(OBJ_STL)
```

Table 6.9: Change Makefile

6.3 Experiments and Results

Logic Path Length

The performance of any routing protocol depends heavily on the length of the path between two arbitrary nodes in the network. In the context of DTSBT, one defines the path length as the number of nodes traversed during basic operations (such as joining or query). From the Theorem 4.2, with high probability, the length of the path to perform a basic operation is $O(2 \times \log_B N)$, where N is the total number of keys, B is the number of keys that each group can at most contain. Figure 6.2 shows three path length curves for three different B values. These three curves are discontinuous because the path lengths are proportional to $\log_B N$. Experimental results show that DTSBT has high efficiency: as B value is large enough ($B = 1000$), the path length is very small (where $N = 1,000,000$, only 4 hops is needed).

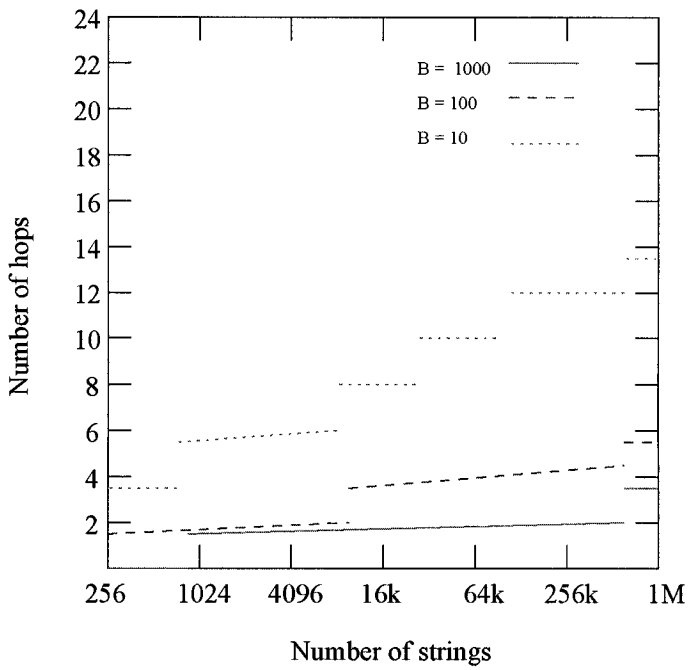


Figure 6.2: Effects of minimum degree on logic path length

Physical Path Length

The efficiency measure used in Figure 6.2 is the number of application-level hops taken on the path. However, true efficiency measure is the end-to-end latency of the path. In the simulation of this thesis, this thesis uses the number of IP node hops taken on the path. To simulate the practical Internet environment, the GT-ITM [33], [32], [8] topology model is used as the Internet topology. Figure 6.3 shows three path length curves for three different B values. These three curves are jump curves, too. Because DTSBT has not considered the network proximity yet, basically the physical path length is proportional to the corresponding logic path length.

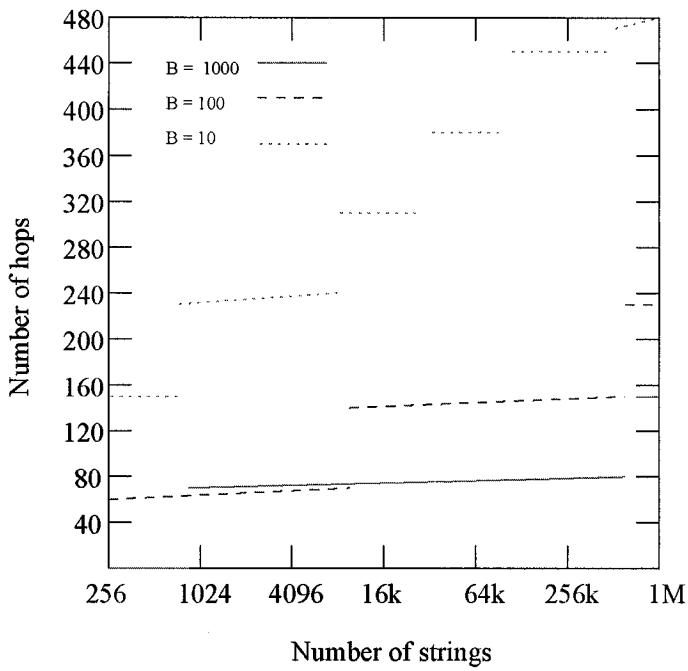


Figure 6.3: Effects of mininum degree on physical path length

Chapter 7

Conclusion and Future Work

In this chapter, this thesis is concluded by (1) summarizing its contributions, (2) exposing some limitations of the proposed approach, and (3) proposing several directions for future work.

7.1 Contributions

The Internet is a great success. The main reason behind the Interent's success is the stateless nature of its architecture. The fact that routers do not need to maintain vast quantities of information about traffic makes the Internet highly scalable. What is scalability? Jim Gray provides the following definition:

> Scalability: Devise a software and hardware architecture that scales up by a factor of 10^6. That is, an application's storage and processing capacity can automatically grow by a factor of a million; either doing jobs faster (10^6x speedup) or doing larger jobs in the same time (10^6x scaleup), by just adding more resources.

The benefit of this definition is that it gives a practical and numerical meaning of scalability. As a result, one can start working on the 10^6x scaleup problem with a view to addressing the larger problem.

The Internet is a world-scale computer system–it is a computer system of 100 million nodes. Currently, its scope is doubling annually. The issue of the scalability of the network and its servers is thus increasingly pertinent [15].

The Internet's high scalability basically refers to its network (IP) layer's high scalability. Internet applications, however, are not highly scalable with a centralized system

architecture. As a result, P2P system architecture emerges and a new layer, P2P overlay network, is plugged into the TCP/IP protocol stack. However, without an effective routing protocol, the P2P overlay network itself is not highly scalable. In response to this application-level scalability problem, DHT-based P2P overlay networks provide some remedy.

To a degree, such DHT-based P2P overlay networks are scalable, but they are not scalable enough. Due to the fact that they employ fixed length key identifiers, the key spaces of DHT-based P2P overlay networks are not *infinite* and *continuous*. Consequently, they are not highly scalable.

One contribution of this thesis is that, initially, the key space of a DTSBT P2P overlay network has *infinite* and *continuous* features so that it is much more scalable than the corresponding key space of a DHT-based P2P overlay network. Furthermore, a virtual network is built, employing a hybrid data structure DTSBT as its routing protocol. As a direct result, a DTSBT P2P overlay network has the routing efficiency of $O(\log_B N)$ hops pathlength with requiring only B-size routing table storage, where N is the size of the key space, and B is the number of keys contained in each routing table.

In addition to its high scalability, the DTSBT P2P overlay network has high decentralization, high resilience to failures, and high self-organization. It is highly decentralized because any key is only created and stored locally. It is highly resilient to failures because each routing table has B backups so that routing can keep the same efficiency even though some physical (IP) nodes that contain same a routing table fail. It is highly self-organized because only the owner/customer who creates a particular key can update the content of that key and release that key from the current system.

7.2 Limitations

The current design and implementation of DTSBT only supports synchronous models, but does not support parallel models. Consequently, before one operation (such as joining, lookup and releasing) has been completed, other operations cannot be executed. Internet application services, however, should be able to join, lookup and release simultaneously. In spite of this limitation, as those desirable features of the DTSBT P2P overlay network are demonstrated, DTSBT will be powerful enough to address the fundamental problem of P2P overlay network.

7.3 Future Work

As discussed above, the current DTSBT does not support the parallel model. With preserving those desirable virtues of DTSBT, one should introduce parallel mechanisms into the design and implementation of DTSBT. After that, one should analyse the revised DTSBT once more.

Like Chord, DTSBT focuses on P2P overlay networks without considering too much about network proximity. Optimization work should be conducted, keeping the load of the keys balancing on the one hand, and providing more appropriate routing efficiency of network (IP) hops pathlength on the other.

XML-based Internet applications can be easily developed and deployed over DTSBT P2P overlay network because DTSBT can effectively support semantic queries. Today, web services are very popular. Some work [28] based on DHT-based routing protocols havs been done to provide a P2P approach to web service discovery. The key problem of these applications is to convert a d-dimensional key space into a 1-dimensional key space while preserving the semantic locality. Thus, one can provide a P2P approach to web service discovery based on the DTSBT routing protocol.

7.4 Final Remarks

This thesis proposes a new approach to the application-level routing problem of the P2P overlay network. In this approach, DTSBT, a P2P overlay network, is designed and implemented using DTSBT data structure as its core data structure. As a result, DTSBT has desirable features: high scalability, high decentralization, high resilience to failures and high self-organization.

Bibliography

[1] The genome@home project web site. [Online]. Available: http://genomeathome.standford.edu/

[2] The gnutella web site. [Online]. Available: http://gnutella.wego.com/

[3] The kazaa web site. [Online]. Available: http://www.kazaa.com/

[4] The napster web site. [Online]. Available: http://www.napster.com/

[5] The seti@home project web site. [Online]. Available: http://setathome.ssl.berkeley.edu/

[6] The vint project. [Online]. Available: http://www.isi.edu/nsnam/ns

[7] S. Androutsellis-Theotokis and D. Spinellis, "A survey of peer-to-peer content distribution technologies," *ACM Computing Surveys*, vol. 36, pp. 335–371, Dec. 2004.

[8] K. Calvert, M. Doar, and E. W. Zeguar, "Modeling internet topology," *IEEE Communications Magazine*, June 1997.

[9] I. Clarke, O. Sandeerg, and B. Wiley, "Freenet: a distributed anonymous information storage and retrieval system," in *Proc. Workshop on Design Issues in Anonymity and Unobervability*, Berkeley, CA, 2000.

[10] D. Comer, "Ubiquitous b-tree," *ACM Computing Surveys (CSUR)*, vol. 11, pp. 121–137, June 1979.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 2nd ed. MIT Press and McGraw-Hill, 2001.

[12] H. El-Rewini and T. G. Lewis, *Distributed and Parallel Computing.* Manning Press, 1998, p. 150.

[13] P. Ferragina and R. Grossi, "The string b-tree: a new data structure for string search in the external memory and its applications," *Journal of the ACM (JACM)*, vol. 46, pp. 236–280, Mar. 1999.

[14] E. Fredkin, B. Beranek, and Newman, "Trie memory," *Communications of the ACM (CACM)*, vol. 3, pp. 490–499, Sept. 1960.

[15] J. Gray, "What next? a dozen information-technology research goals," *Journal of the ACM (JACM)*, vol. 50, pp. 41–57, Jan. 2003.

[16] M. Greis. Tutorial for the network simulator ns. [Online]. Available: http://www.isi.edu/nsnam/ns/tutorial/index.html

[17] A. Halevy, Z. Ives, P. Mork, and I. Tatarinov, "Piazza: data management infrastructure for semantic web applications," in *Proc. the 12th International Conference on World Wide Web*, Budapest, Hungary, 2003, pp. 556–567.

[18] R. Huebsch, J. Hellerstein, N. Lanham, and B. T. Loo, "Querying the internet with pier," in *Proc. the 29th VLDB Conference*, Berlin, Germany, 2003.

[19] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigraphy, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web," in *ACM STOC'97*, EI Paso, Texas, May 1997.

[20] J. Kublatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, S. Gummadi, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: an architecture for global-scale persistent storage," in *Proc. ACM ASPLOS'00*, 2000.

[21] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and S. Lim, "A survey and comparison of peer-to-peer overlay network schemes," *IEEE Communications Surveys and Tutorials*, vol. 7, Apr. 2005.

[22] D. Morrison, "Patricia-practical algorithm to retrieve information coded in alphanumeric," *Journal of the ACM (JACM)*, vol. 15, pp. 514–534, Oct. 1968.

[23] C. Plaxton, R. Rajaraman, and A. Richa, "Accessing nearby copies of replicated objects in a distributed environment," in *Proc. ACM SPAA'97*, Newport, Rhode Island, June 1997, pp. 311–320.

[24] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker, "A scalable content-addressable network," in *Proc. ACM SIGCOMM'01*, San Diego, CA, USA, Aug. 2001, pp. 161–172.

[25] S. Ratnasamy, I. Stoica, and S. Shenker, "Routing algorithms for dhts: some open questions," in *Proc. First Internal Workshop, IPTPS 2002, Revised Papers*, Cambridge, MA, USA, Mar. 2002, pp. 45–52.

[26] F. J. Ros and P. M. Ruiz, "Implementing a new manet unicast routing protocol in ns2," Tech. Rep., Dec. 2004.

[27] A. Rownstron and P. Druschel, "Pastry: scalable, decentralized object location and routing for large-scale peer-to-peer systems," in *Proc. IFIP/ACM International conference on Distributed Systems Platforms (Middleware)*, Heidelberg, Germany, Nov. 2001, pp. 329–350.

[28] C. Schmidt and M. Parashar, "A peer-to-peer approach to web service discovery," *World Wide Web*, vol. 7, pp. 211–229, June 2004.

[29] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Trans. Networking*, vol. 11, pp. 17–32, Feb. 2003.

[30] *Secure Hash Standard*, U.S. Department of Commerce/NIST, National Techinical Information Service Std. FIPS 180-1, Apr. 1995.

[31] M. Waldwan, R. Ad, and C. LF, "Publius: a robust, tamper-evident, censorship-resistant web publishing system," in *Proc. the 9th USENIX Security System*, 2000.

[32] E. Zegura, K. Calvert, and S. Bhattacharjee, "How to model an internetwork," in *Proc. IEEE Inform'96*, San Francisco, CA, USA, 1996.

[33] E. Zegura, K. Calvert, and M. Donahoo, "A quantitative comparison of graph-based models for internet topology," *IEEE/ACM Trans. Networking*, vol. 5, pp. 770–783, Dec. 1997.

[34] B. Zhao, J. Kubiatowicz, and A. Joseph, "Tapestry: an infrastructure for fault-tolerant wide-area location and routing," University of California at Berkeley, Computer Science Department, CA, Tech. Rep. UCB/CSD-01-1141, 2001.