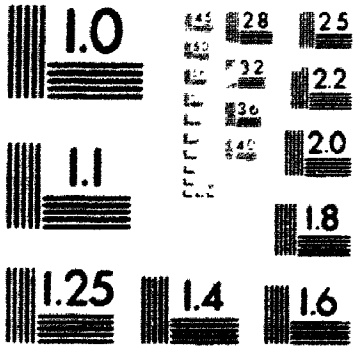
1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT



BTS: A New Distributed Data Structure

by
Yimei Yao

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

School of Computer Science
Carleton University
Ottawa, Ontario, Canada

April 1993

ISBN 0-315-84132-X

Name BTS: A New Distributed Data Structure

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science (Applied Science) — SUBJECT TERM

0984 — SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523
Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce-Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996
Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984
Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to

the Faculty of Graduate Studies and Research

acceptance of the thesis,

" BTS: A New Distributed Data Structure "

submitted by Yimei Yao, B.Eng.,

in partial fulfillment of the requirements for the degree of

Master of Computer Science

Thesis Supervisor

Director, School of Computer Science

Carleton University

19 May 1993

Date

Abstract

BTS: A New Distributed Data Structure

Yimei Yao

A *distributed data structure* is a data structure which is distributed among the sites of a communication network and may be accessed by many processes simultaneously. It is composed of a *data organization scheme* that specifies a collection of local data structures (or bins) storing copies of data item at various sites in the network, and of a set of *distributed access protocols* that enable processes to issue queries and modification instructions to the network and to get appropriate responses.

This thesis deals with the dictionary data structure, which supports the operations *Find*, *Insert* and *Delete*. The goal is to provide an efficient distributed data structure which can minimize the communication cost while requiring reasonable overall space and balancing the memory loads over the sites of the network. We have proposed the design, validation and implementation of a distributed data structure: *BTS* (*Binary-Tree-Structure*) by which the communication cost of the three dictionary operations is optimized and data load among the network is roughly balanced.

Concurrency and fault-tolerance are also discussed, and proper lock mechanisms are interpreted in the proposed distributed data structure.

Acknowledgments

First and foremost, I would like to thank my supervisor, Professor Nicola Santoro, for his guidance and valuable insight throughout this research. His support and patience were invaluable in the preparation of this thesis.

I am thankful for the financial support of the School of Computer Science and of my supervisor.

I thank the people at the office of the School of Computer Science, especially, Rosemary, and Marlene, for their help and patience.

Last but not least, I would like to express my deepest gratitude to my parents and my husband for their understanding and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

Distributed systems are widely used in many fields of computer applications and the networking enables systems to be designed to serve geographically located users. These systems are based on a set of separate computers that are capable of autonomous operation, linked by network. The advantages offered by distributed systems include resource sharing, fault-tolerance and parallel execution of a computation.

The resources, such as printer, file server, data, etc., can be shared by the computers among the network. Also, to avoid traffic and for the purpose of fault-tolerance, it is desired that the data be distributed among the network. The design of distributed system should be able to provide an efficient mechanism such that users of the system are given a single, integrated computing resource, although the resource is actually provided by more than one computer and the computers may be in different locations.

The shared data which are needed to provide an integrated computing service are provided by some of the computers in the network and are accessed by system software that runs in all of the computers, using the network to coordinate their work and transfer data between them. Thus, the design of system software plays the significant role in the management of these shared data. The software design of distributed systems is more complex than centralized ones due to the unavailability of the global state of the system. For example, in a dynamic system where resources (e.g., files, processes) can be migrated between nodes, a user must program an algorithm to find the current location of a resource needed by his or her computation. This can be avoided if users are provided with the abstraction of a unified system where the location of resources is transparent to them.

Ideally, a distributed system should look to its users like a conventional, centralized system. The user interface of a transparent distributed system should not distinguish between local and remote resources. That is, users should be able to access remote distributed system as though the later were local, and it should be the responsibility of distributed system to locate the resources and to arrange for the appropriate interaction. Resources are referred to by names and, at runtime, the system determines the current location of a named resource.

Any efficient algorithm in the software design must be inter-wined with a good data structure. This is also true in the distributed systems. In the distributed system, distributed system software need not only deal with how data is stored, but incorporate the control of data, i.e. how data is transferred. This introduces distributed data structure.

1.2 Distribution of Data and Control

1.2.1 Distributed Data

Resources, files or data structures can be represented as data. Data are needed to be distributed among the network due to various reasons, and consequently, the distribution of data can be done differently: replication or partitioning of the data.

Replication means that for a data item x, at each separate site there is a copy of x, denoted by $x_1, x_2, \ldots, x_n$. The problem then arises as to how to ensure consistency among all theses copies, i.e. at any instant $x_1 = x_2 = \ldots = x_n = x$, where x is the *correct* value of data item x at that instant. Partitioning of data means that the full set of data is partitioned among the sites: each site holds a partition and any data item is accessible through communication from any other site. Also in this case there is the need to ensure consistency. The two methods can also be combined, with partitioning of the full data items and replication of certain elements.

1.2.2 Distributed Control

In order to manipulate the distributed data, an independent control mechanism is needed at each site and a decision has to be made by means of agreement among a group of peer processors, this control and decision mechanism is called *distributed control* (or *access protocols*).

Since there is no central control in the distributed network, there must exist some methods to organize the data, and methods to access the data distributed among the network. These

methods form what are called *distributed data structures.*

Formally, a Distributed Structure is modeled as follows. The processors located at various sites of the system occasionally generate data that need to be stored in the structure. The data might later be needed by processors in other sites. The structure is thus composed of a *data organization scheme*, specifying how and where to store each data items in the system, and a set of *distributed access protocols* that enable processors to issue modification and query instructions to the system and get appropriate responses. In general, the repertoire of instructions includes both query instructions, specifying a request for information retrieval from the data structure, and modification instructions, requesting some change in the data structure.

The important fact is that such distributed data structures are in many cases constructed not as part of any particular algorithm but for direct use as building blocks of various permanent storage and retrieval mechanisms, such as distributed dictionaries, name servers in communication networks, resource allocation management etc. The function common to all of theses mechanisms is to supply facilities for storing accumulated information in the system and making it available to potential users throughout the system.

1.2.3 Complexity Considerations

The development of a distributed data structure involves several important design issues and performance aspects. There are different complexity measures for the data organization scheme and access protocols.

For the data organization scheme, the most important thing is to use reasonable overall

space requirements. Another is the need to balance the memory loads over the sites of the system. Future systems are expected to carry enormous loads of data, and a single site can hardly be expected to function as the sole storing site for a large data structure. It is therefore desirable to be able to balance the memory requirements of any particular data structure between all sites in the network.

For the access protocols we may consider response time and communication complexity. Given the available technology, a designer can do little to alleviate problems resulting from slow communication other than reduce the number of messages involved. Thus, our main concerns are to minimize the blocking time and minimize the number of messages passing for given query and modification operations.

1.3 Design Goals

When designing a distributed data structure, we must deal with the inherent complexity of asynchronized, supported operations in a potentially slow and error-prone environment. The applications that can make effective use of a distributed structure differ in their requirements. We concentrate on a class of applications which are concerned with the manipulation and preservation of long-lived, on line data. Examples of such applications are banking systems, airline reservation system, database systems.

For the kind of application above, we define our following design goals:

1. **Service**: A major concern is to provide continuous service of the system when there exits sites or network failure. An application can continue its task as long as the

particular sites it needs to communicate with are functioning and reachable.

2. **Autonomy**: Each site behaves as if itself is a central site. It can control the data stored at that site. Besides, it can also provide a service for manipulating data that is not stored at that site(e.g., for the query message to the possible site).

3. **Distribution and load balance**: The distribution of data can be a major impact on overall efficiency. Distribution also affects availability. It is desirable that the data at each site be almost balanced while maintaining overall efficiency and availability.

4. **Concurrency**: This is to take advantage of the potential concurrency of a distributed application, and thereby increasing efficiency and decreasing response time.

5. **Consistency**: The data structure must satisfy a set of assertions called *consistency constraints*. The only concurrency constraint for our application is *order preserving*(or strict serializable schedules): The structure is globally order preserved, if any sequence of the instructions (operations) originating form the same source and having the same destination are processed in the order which they are sent; the structure is also locally serializable, if the execution of an instruction at a site must be atomic, (i.e. uninterruptable).

1.4 Related Work

The study of distributed data structures involves topics from several large research areas, such as ordinary data structures, distributed databases and concurrency control theory. We

are going to mention some areas that are directly relevant to our investigation.

Concurrent data structure

One of the most important problem to be solved in a parallel or distributed programming system is the assignment of resources or tasks of a computation to processor nodes. Manber presented an efficient dynamic search structure to which many autonomous asynchronous processes can access. In [Man86], the design has achieved maximum concurrency assuming a single shared global memory, (i.e. the processes communicate only through the shared memory) and a single entry point to the structure.

Existing distributed structures

When there is no shared memory, the distributed system can be modeled as a number of logical processors communicating solely through port-based asynchronous message passing. A logical processor may encompass many processes that execute on the same physical processor and may generate data among themselves. In [Ell85], they developed a distributed version of an extendible hash file which is a dynamic indexing structure that could be useful in the design of a distributed database.

In [MS90], a special case was considered in which a large file (that does not fit into one disk) must be distributed among several disk. Each disk is controlled by a processor, and each pair, of disk and processor, resides in a different site; the sites are connected via a local broadcast network. Parallelism is used by distributing the tree among the processors. They distinguish conceptual tree(which is "wide" B^+ tree) and physical tree (which is the

distributed implementation). The method is based on the following idea: The entries (keys and pointers) of each node in the conceptual tree are partitioned among the processors. Each processor accesses only a part of each conceptual node and , in order to work properly, does not need to "see" the other parts of the conceptual node. They have assumed the physical environment consists of a host computer, a number of processors and the corresponding disk with each processor. They also assumed that each message sent by a processor, can be heard by all processors.

In [GP91] the problem handled is that of designing a deterministic dictionary structure on a distributed system. Their algorithm is an implementation of Willard's algorithm for sequential file (algorithm CONTROL-2 in [Wil86]) in distributed networks whose topology is a full binary tree(*FBT*). The structure is based on an *FBT* of processors B, where the leaves store the elements in sequential order. They developed a hybrid structure which consists of a main static tree B and "small" dynamic 2-3-tree structure "hanging from its leaves". The main tree B is an *FBT* that is embedded in a subtree of the given *FBT* network, while the "small" dynamic trees are embedded in clusters of nodes concentrated within a relatively small diameter. They also discuss a dictionary structure for general graphs from communication complexity view.

Resource Allocation

Resource allocation has been widely studied in both centralized control and decentralized control environment. In a centralized approach, all resources are controlled by a single central node. When a request arrives at a node, a "buyer" is commissioned, which travels,

via messages, to the central node to obtain a resource. The buyer then carries the resource back to the node where the request originated, so that the resource can be granted to the request at that location. In a decentralized control of the resources, the buyers must search for the resources. The choice can be made between sending only one buyer to search for resources for each request and sending several buyers in parallel to search different parts of network. Other strategies involve combining a decentralized search strategy with a dynamic resource redistribution strategy, letting resources search for requests (rather than vice versa), or giving nodes control of fractions of resources rather than whole resources. In [LGFG86] it presented the probabilistic analysis of a network resource allocation algorithm, for a large number of identical resources, to requests which can arrive anywhere in a distributed network. Resources, once allocated, are never returned. The algorithm searches sequentially, exhausting certain neighborhoods of the request origin before proceeding to search at greater distances. Choice of search direction is made non-deterministically.

Another kind of problem of resource allocation is name management. Resources are referred to by names and, at runtime, the system must determine the current location of a named resource. Conceptually, there exists a database that stores the associations between resource names and their location. This database can be partitioned and stored at one more nodes that are called *name servers*. When a remote resource, R, needs to be accessed, the request for its location should be sent to a name server that stores R's location. The system must also implement algorithms to update the information stored by the name servers when resources are created or deleted or when they are migrated. In the approach of [AAB88], the database is distributed in such a way that a name server at a node maintains a list of

only resources local to the node. In such a system a remote resource can be located by broadcasting its name, and having the node where the resource is located respond.

VLSI machine

The other related area concerns the design of special purpose VLSI machines for implementing data structures, mainly dictionaries. The model is that of a tightly-coupled machine of fixed topology which maintains the data structure. It has a specified I/O port, connected to a specially designated processor. This processor receives a stream of instructions for processing via the I/O port, and the machine has to execute the corresponding operations in a pipelined fashion and supply the replies to queries through the I/O port. One is usually interested in the latency of the machine, namely, the number of time units elapsed from the time the query arrived to the time the response is produced at the I/O port, and the period of the machine, namely the longest delay incurred by an arriving instruction until the next instruction can be issued to the machine [LP90] [FC91].

Load balance

In a distributed system environment, it is desirable to equalize the usage of resources (balance the load) in order to reduce the response time of jobs and improve the utilization of the resources. This can be achieved by migrating jobs to the lightly loaded sites. The load balancing problem in a distributed resource system can take different forms for different problems. For example, in a long haul packet network (where the resources are the channels) load balancing becomes the routing problem. The goal there is to find optimum paths on

which to distribute the packets, so that some well-defined performance criterion (overall delay, for instance) is optimized. In a distributed computer system, the load balancing problem may be formulated as the problem of distributing the execution of processes throughout the network in such a way that the overall user response time is minimized.

1.5 Motivation

When designing a distributed data structure, there are two types of problems we need to deal with: (1) those stemming from the introduction of concurrency, and (2) those caused by the distribution of the data and users.

For the former problems, solutions can be based on the techniques studied extensively in distributed database theory, that is concurrency control which include the approach called *serilizability*. In structures that are coordinated by a central processor, serilizability can be enforced by the coordinator simply by handling the requests in order of arrival, starting to process the next instruction only after completing the previous one. When there is no coordinator, the problem of inconsistence of the current contents and internal organization of the structure will arise because of different arrival orders of nearly simultaneously issued modification instructions or messages [Pel89].

For the latter problems, solutions must be sought which are efficient with respect to complexity issues, such as communication complexity and time complexity; at the same time, they must maintain the data structure using reasonable overall space requirements and other criteria.

Little work, however, has been done for the design, analysis and classification of

distributed structures based on a theoretical framework from a complexity-oriented point of view. Peleg [Pel89] considered these kinds of issues, and presented a new distributed data structure: *DIST-BIN*. The basic idea behind this structure is that a collection of bins $B_1, B_2, \ldots, B_p$ (each maintained in some vertex, $p \leq n$, where n is the number of vertices in the network) store the data in the ordered fashion. Vertices that do not store a bin are kept in a linked list of "free bins". Each bin has knowledge of other bins (such as the range of data items stored at that bin). Sometimes, it is necessary to update the knowledge. This is done by broadcasting the new information throughout the network. Thus, the worst-case performance of an operation is $O(n)$. An other characteristics of bin dictionary is that it does not make use of all the bins. Therefore, most of the times, some bins are full-loaded (even though it is load balanced among those bins), and others are just load-free. The third problem is the way to handle the concurrency. Since its *Find* operation is not an atomic part of the insertion operation, when a bin B_1 splits into B_1 and B_2, a processor v not yet knowing of the split may attempt to delete an item from B_1 while the item is already stored at B_2, or to insert into B_1 an item that should now be stored at B_2. These problems are overcome in a standard way. For example, when a bin B_i decides to split it first locks itself by broadcasting a message to all vertices, informing them of its intention to lock and collecting acknowledgments, including information about which processors have instructions to B_i that are currently outstanding. It then proceeds to "clear the table" by processing all outstanding instructions, and only then split, informs all processors of the new pointers and release the locks.

Noticing the cost of broadcasting due to the fact that each processor in *DIST-BIN*

behaves like a center, we propose a new distributed data structure: *Binary-Tree-Structure* (*BTS*) which takes advantage of the characteristics of trees. In this thesis, we present the design of *BTS*, study the verification of its manipulation protocols, and discuss its implementation issues.

1.6 Objective of This Study

In this thesis, we consider a more general situation of a distributed system. Each processor in the network can generate, access, and manage the data. Data can be moved among the network to meet the requirement of the requesting processor or to restore the load balance. The cost is the number of communication messages. Our goal is to provide an efficient algorithm to access data using minimum communication cost while maintain load balance.

In general, a distributed data structure is characterized by the set of instructions it supports. Depending on the set, we can distinguish Dictionary, Priority queue, Queue, and Stack, etc.. In this thesis, we discuss the distributed data structure which supports Dictionary instructions. The set of operations for a Dictionary structure is as follows:

Find(key, r): Find the data item x such that $key_x = key$, and return $Record_x$ to the querying process r. If no such items exist then return an "failure message".

Insert(x): Insert the data item x into the data structure. The instruction has to include only the items x (in addition to a short header), and is thus of length at most δ. No reply is sent to the requesting process in such a case.

Delete(key): Delete the data item x such that $key_x = key$ from the data structure. The

instruction has to include only the key (in addition to a short header), and is thus of length δ. No reply is sent to the requesting process in such a case.

Sometimes, we refer to *Insert* and *Delete* operations as *modification instructions*, and *Find* operation as *query instruction*. Both modification instruction and query instruction are called *request instruction* or *operations*.

For simplicity, we use *Find*(key, r), *Insert*(key, r), *Delete*(key, r) to represent the corresponding operation, and refer *key* as data item. When there is no confusion, we also use *Find*(key), *Insert*(key) and *Delete*(key) to represent a request operation.

The contributions of this thesis are summarized as follows:

1. The design of *BTS* is proposed. In *BTS*, since *Find* is an atomic part of a modification operation, query and modification operations can be issued concurrently without the need for synchronization.

2. The load of data items in the network is maintained roughly balanced. In particular, when some predefined threshold is paned, a balance process will start to restore the balance.

3. For the data access efficiency, we are concerned with two criteria: access time and the number of network message passing.

   When the system is balanced, query and modification operations are always performed without any blocking. The maximum number of messages for a request travelling from its source to its destination is $2 * (\log(n + 1) - 1)$, where n is the number of nodes(sites) in the network. An information gathering mechanism requires $\log(n +$

$1) - 1$ messages for each modification operation, which does not delay the response time to user's dictionary operation request.

When part of system is not balanced, it is only necessary to balance that particular part without influencing the performance of whole system.

4. The most important issue in the concurrency system is data consistency. In this thesis, we focus on the validation of proposed protocols. We use finite-machine to represent processes and implicit queues to represent channels. The queues modeling the channels having unbounded capacity to represent protocols allowing an arbitrary number of messages in transit.

   We classify global states that may occur, and for each given class of global states, we first verify that these global state are reachable. Then we prove that in such global state, the executions of request operations are order-preserved.

5. The fault tolerance issues are also discussed.

The remainder of this thesis is organized as follows. In Chapter 2 we describe in detail the model we used. In Chapter 3, we give the solution approach, present our *Binary-Tree-Structure* (*BTS*). Chapter 4 verifies the method we proposed. Chapter 5 gives evaluation of *BTS*, discusses the redundant structure to incorporate fault tolerance, summarizes the work and finally we present our conclusions.

Chapter 2

Preliminary

Although one usually speaks of a distributed system, it is more accurate to speak of a distributed view of a system. A hardware designer views an ordinary sequential computer as a distributed system, while a Pascal programmer views the same computer as non-distributed view of a distributed system, for example, to implement a distributed file system that allows the client programmer to ignore the physical location of his data.

A model can be used to denote a view or abstract representation of a distributed system. Our model is chosen in such a way that the components and properties of the system which are relevant to the analysis emphasized, and insignificant details are removed. In the following section, we will describe the model for our problem.

2.1 Model

The model of computation generally considered to be distributed is the *process* model, in which computation activity is represented as concurrent execution of sequential processes. Different process models are distinguished by the mechanism employed for interprocess communication, such as *message passing*, *shared variables* and *synchronous communication*. Shared variables and synchronous communication are not the case for our problem.

In the message passing model, a process sends a message by adding it to a message queue, and another process receives the message by removing it from the queue. In the following, we will describe in detail the message passing models used in this thesis from four separate concerns: *network topology*, *synchrony*, *failure*, and *message buffering*.

2.1.1 Network Topology

A computer network consists of a collection of sites or nodes, connected via communication links or channels. It can be described as an undirected graph

$$G = (V, E)$$

where a vertex $v \in V$ represents a node in the network, and the edge $e \in E \subseteq V \times V$ represents the neighbor relation, which describes the communication channels. The neighbor relation defines the set of pairs of nodes which have a means of directly exchanging external state information (passing message). For a vertex $v_i \in V$, the neighbor relation is defined as

$$Neighbor(i) = \{v_j \mid (v_i, v_j) \in E\}$$

Each node corresponds to a process which has no shared memory and can communicate only by sending and receiving messages to and from its neighbors, respectively.

2.1.2 Synchrony

A completely *asynchronous* model is used, which has no explicit and global concept of real time. Therefore, processes cannot access a global clock. The only events that a process can be aware of are those within itself and those relating to the sending or receiving of a message.

It is assumed that messages are eventually delivered and processes eventually respond, but no assumption is made about how long it may take.

Thus the behavior of the processes in the network can be described as "finite state" and event driven. i.e., the local process consists of a set of states, and each process at any time is in a particular state. When a predefined event occurs, (e.g. receiving a message), it will perform local computations, send messages to neighbors or change the current state according to the current state and event.

2.1.3 Failure

In the message-passing models, there can be two kinds of failure: process failure and communication failure. Both failures may cause loss of messages, or even worse, make it impossible for some neighbours to communicate with each other.

We first assume that there is no failure in the network. i.e., we describe the access algorithms in the absence of failure. In Chapter 5, we discuss the approach to incorporate

fault tolerance.

2.1.4 Message Buffering

There exists a delay between the time a message is sent and the time it is received. Such a delay implies there is some form of message buffering. We assume infinite buffers. That means there may be arbitrarily many unreceived messages in a link's buffer, and the sender can always send another message over the link. FIFO buffering is also assumed. In that way, messages are always received in the same order in which they were sent.

Summarizing, we have the following assumptions:

1. There exists a reliable underlying communication service, which enables a process to send a message to some other process for whom it has sufficient routing information (e.g. each node $v_i \in V$ has a distinct label $l(i,j)$ for every neighbor $v_j \in V \subset G$; it can send a message M to a neighbor j by the 'send M to $l(i,j)$' directive). In this way, we can build a logical view network in which the messages can pass the logical node by the channels.

2. The channels are reliable; they do not alter the messages, lose them or duplicate them. Any message transmitted along a channel by a node will reach the node to which the channel connects it after an arbitrary, but finite, period time. For transmission between any pair of nodes, messages are received in the order in which they were sent, i.e. there is no alteration in the sequencing.

3. Initial knowledge. Each node knows the structure of the network. This is necessary

only when some nodes decide to move a bunch of data to some other nodes (see the *BTS* structure).

4. We ignore the issue of fault-tolerance first, and assume that the system operates properly throughout its life time.

2.2 Local Storage: Bin

Suppose all the nodes have distinct names, and the name are taken from the integer $1, 2, \cdots, N$, where $N = |V|$. Each node (or process, site) i has its local storage, called bin B_i, which is used to store the data items. The data item x is taken from the universe of data U. Each data items $x \in U$ consists of two fields, $x = (key_x, record_x)$, where key_x is a key field taken from a totally ordered domain k and $record_x$ is a record field containing the relevant data. We assume that the size of a data item is at most δ bits of memory, where, δ is the maximum message size allowed. Thus, a data item can be transmitted in a single message. The data structure has to be able to store such data items in the system, and support various operations on this data. These operations are specified by instructions issued by processes. The process originating the instructions is refereed as *requesting* process(or source). The process containing the information on the request is refereed as *storing* process(or destination).

2.3 Describing A System

Having a computational model of the concurrent distributed system, we now describe its formal model, which forms the basis for our discussion of verification. The method of design and verification we have used is *event* or *message-based.*

A concurrent system is composed of a triple: a set of *states* S; a set of *actions* A; and a set of *behaviors* Σ, each behavior being a finite or infinite sequence of the form

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \cdots \tag{2.1}$$

where each s_i is a state, and each α_i is an action.

2.3.1 State

A system state describes the complete instantaneous state of the system. The state of the system is the combination of the states of all processes, variable, and channels. The state of a variable is its content; the state of a process is a tuple of the state of all of its variables and possibly a program counter. The state of an (unidirectional) asynchronous message channel is the multi-set of messages that are in transit in the channel. If the channel obeys the FIFO discipline, its state is the (ordered) sequence of messages in transit. If the channel guarantees a bound on the delay time of messages, the time each message is in the channel already is also part of the state of the channel.

2.3.2 Action

An action is a system operation that is described as event-driven. During the execution of the algorithm the state of the system changes. Messages are sent and received, and variables are assigned new values. Action is defined as any possible change of system state, and event(or message) is actual occurrence of an action. We choose the actions in such a way that it is possible to implement them as atomic, i.e. each action is invisible.

Assume there is a set of processes P, communicating only by exchanging messages. The processes in P cooperate to accomplish some task. The operation of the process is message driven. Only upon reception of a message, belonging to the computation, a process performs some computation and sends zero or more messages. The act of receiving a message, internal computation, and sending of messages is an atomic action of the system, which is also referred as a unit of a "transaction". It is described as follows(for the process i):

BA_i: {a message m arrives at process i }

begin *receive(m)*;

compute;

for all $q \in Q$ **do** $send\ m_q$ to q

end.

The local computation is represented by *compute*. This computation is finite and terminates always. Q is a finite sequence of processes; a message m_q is sent to each process q in this sequence. In each invocation of BA_i, a different Q can be generated. Process i can appear in Q, processes can appear more than once, and Q can also be empty. We refer to BA_i as

system's *basic action*, and any message can only be generated in the basic actions. To start the computation it is necessary that at least one message is created, other than by being sent during the execution of other *BA*. These messages are generated by some environment, which are not included in process itself. For example, a user typing a command on his terminal with the dictionary operation requests among the network.

Each *BA* occurs in a process. Since a process is localized, we can, in principle, observe its entire state at one time. We classify the local states with similar behaviors; most of the time, the occurrence of events (in *BA*) only affects the history of current process, therefore the actual state changes will occur only on its history. In this way, we can focus our observations on different class of local states. For example, for the class of *Active* state, system is in normal status, and request operations are manipulated in normal way. While in *Lock-In* state and *Lock-St*, request operation should be taken special care of. We will formally define a local state later.

2.3.3 System Behavior

A system behavior represents an *execution* of the system whose ith action α_i takes the system from state s_{i-1} to state s_i. The set Σ represents the set of all possible system executions. Given a set of states S and a set of actions A, we can specify the possible state transition $s \xrightarrow{\alpha} t$ (where $s, t \in S$) caused by each action α of A.

A behavior of the form Equation 2.1 is in Σ only if (i) each transition $s_{i-1} \xrightarrow{\alpha_i} s_i$ is a possible state transition of α_i; and (ii) it is either infinite or it terminates in a state in which no further action is possible.

Formally, we define a relation $\Gamma(\alpha)$ on S for each action α of A, where $(s,t) \in \Gamma(\alpha)$, if and only if executing the action α in state s can produce state t. The action α is said to be *enabled* in state s, if there exists some state t with $(s,t) \in \Gamma(\alpha)$. The behavior of the form (2.1) is in Σ only if (i) $(s_{i-1}, s_i) \in \Gamma(\alpha)$ for all i; and (ii) the sequence is either infinite or ends in a state s_l (i.e. there exists $l+1$ states) in which no action is enabled.

2.3.4 Views of System Behaviors

Given a set of process states and events, how can we know that it represents a possible system execution? We need two views to observe the system [Lam78].

The Space-Time View

The execution is viewed as a partially ordered set of events, where, for the distributed message-passing system, the partial ordering is determined by the following constraints:

1. All the actions performed by a single process are totally ordered;

2. The sending of a message must precede its receptions.

   Let the relation $\rightarrow$ be an irreflexive partial ordering. Then if e and f are distinct events in the same process and e precedes f in the sequencing of the events in that process, then $e \rightarrow f$. If e is in a *BA* with *'send m'* over a channel c, and f is in another *BA* with *'receive m'* from channel, then $e \rightarrow f$.

   If channel c is FIFO channel, we have the third constraint:

3. If e_1 and e_2 are two events in two basic actions in a process p that send message over the same channel c to a process q such that $e_1 \rightarrow e_2$, and f_1 and f_2 are two events in process q such that f_1 receives the message that e_1 sent, f_2 receives the message that e_2 sent. Then $f_1 \rightarrow f_2$. This means that messages over a single channel are received in the same order in which they are sent.

Each *basic sequence* seq begins with reception of an event. Let f_1 belong to $seq_x(p_{src})$ and f_2 belongs to $seq_y(p_{src})$, where x, y are sequence number and $x < y$. Therefore if $f_1 \rightarrow f_2$, then $seq_x(p_{src}) \rightarrow seq_y(p_{src})$, where p_{src} is a local state which will be defined in Chapter 4.

From the *Space-Time* View, we can derive the partial order of the events. Sometimes, we still need to look at how the order influence the states. This introduces the interleaving view.

The Interleaving View

The interleaving view begins with the same set of process and events. The $\rightarrow$ relation is extended to a total ordering of the events. Given this total ordering and an initial state, we can define the sequence of global states produced by this sequence of events, yielding an interleaving model. In general, however, there are many ways of extending the partial ordering $\rightarrow$ to a total ordering $\Rightarrow$. Thus, an interleaving obtained from a partial ordering is arbitrary. We use $e_1 \Rightarrow e_2 \Rightarrow \ldots \Rightarrow e_n$ to represent a total ordering on the set of events.

Now, let's look at our problem. Suppose, process src (source) sometime originates a *Insert*(key) request operation e_1. Later on, src originates a *Find*(key) request e_2 .

Obviously, they should have the same destination. If key belongs the local src, then there is no problem with the result(as long as *Insert* is processed first). Otherwise, the two requests should be delivered. We have to force them to arrive in the same order at their destination(in other process) as they are sent. We need to prove that they have one and only one path to get to the destination. On their way to the destination dst, the intermediate process k's local state is always changing. We will also prove that: at any state, the order cannot be changed.

Formally, let e_1 belong to $seq_x(s_{src})$, e_2 belong to $seq_x(s_{src})$, and they have the same destination dst. Then if $seq_x(s_{src}) \rightarrow seq_y(s_{src})$, then there exist a total ordering

$$\ldots seq_x(s_{src}) \Rightarrow seq_y(s_{src}) \ldots seq_v(s_{dst}) \Rightarrow \ldots \Rightarrow seq_w(s_{dst})$$

such that the two request are computed at process dst in the order $seq_v(s_{dst}) \rightarrow seq_w(s_{dst})$.

2.3.5 Safety and Liveness

Reasoning about a distributed algorithm is to determine if an algorithm does what it is supposed to do. To verify that a system satisfies a property C, the obvious way is to reason directly about behaviors, which can become quite complex with many cases to consider. Assertion methods may overcome this difficulty by drawing attention to states.

In [LL90], properties of the system are expressed by assertions about the set Σ. Any model is an abstraction that represents only some aspects of the system, and the choice of model restricts the class of properties one can reason about. Most formal reasoning about distributed systems has been aimed at proving two kinds of properties: *safety* and *liveness*. Intuitively, a safety property asserts that something bad does not happen, and a liveness property asserts that something good eventually does happen. A safety or liveness property

is an assertion about an individual behavior. It is satisfied by the system if it is true for all behaviors in Σ. A safety property is one that is false for a behavior if and only if it is false for some finite initial prefix of the behavior. A liveness property is one in which any finite behavior can be extended to a finite or infinite behavior that satisfies the property.

A behavior is considered to be a (finite or infinite) sequence of states having the form Equation 2.1 and properties are expressed in terms of state predicates — boolean-valued function on the set of states.

Let a state predicate P be an assertion about behaviors by defining P to be true for a behavior if and only if it is true for the first state of the behavior. We define $\Box P$ to be the assertion that is true for a behavior. So $\Box P$ asserts that P is "always" true. Traditional assertional methods prove that safety properties of the form $P \longrightarrow \Box Q$ for state predicates P and Q.

We say that a state predicate I is an invariant of a system if no action in A can make I false. More formally, I is an invariant if and only if, for each every action α in A and every pair (s, t) in $\Gamma(\alpha)$, if $I(s)$ is true then $I(t)$ is true [LL90].

2.4 Terminology

A distributed data structure is *compact* if it stores only one copy of each data item. Otherwise we say, it is *redundant*.

We define load-balance with respect to data only. A data structure is *load-balanced* if the amount of data stored at various sites in the system is roughly the same. Load balance depends on the insertion function used to specify the storing location of each data item.

This function is implicitly defined by the protocol implementing the insertion operation of the data structure.

The insertion is said to be *safe* if there is no duplicated key inserted in the data structure.

Static Structure and Dynamic Structure

Broadly speaking, *tree-based* design strategies fall into two main categories. The first is called *static trees*, which means the structure itself is explicit and fixed, i.e., it corresponding to some specific spanning tree of the network. It has N nodes, each corresponding to some vertex of the network and possibly storing many data items. The data allocation, however, is *dynamic*, i.e., data items can be moved from one node to another node to maintain memory balance.

Another is referred as *dynamic trees*. Node in the tree may join the tree or be deleted. split or be merged. The tree is virtual, and does not correspond to the topology of any particular subnetwork.

Static Algorithms and Dynamic Algorithms

A class of algorithms for message-passing models is referred as network algorithms, in which the behavior of the algorithm depends strongly on the network topology. Most of them can be divided into two categories: *static* and *dynamic*. Static algorithms are assumed to operate in fixed networks and to start with all their inputs available at the beginning; dynamic algorithms also operate in fixed networks but receive some of their inputs interactively. In other words, the static algorithms are based upon unchanging

information in the initial states of the processes, while dynamic algorithms use changing information from the changing state of the application processes. Our problem falls into the categories of dynamic algorithms.

Global Consistency

For any message-passing model, when accommodating concurrent accesses, one must involve the correctness aspect of the problem, i.e., the algorithm must ensure and maintain the global consistency of the data. For example, in a distributed banking system, one would like all branches of the bank to have a consistent view of the balance of any single account. In general, one would like to describe a distributed system in terms of its current global state.

2.5 Protocols and Load Balance

To support dictionary operations, it is necessary that sites among the network cooperate properly. Therefore, protocols are introduced. A protocol consists of rules and the transfer of supplementary control information between communication sites. Different dictionary operation requires different protocol.

The following protocols define the find, insertion, and deletion operations.

1. *Find*(key, r)

When *Find* protocol receive a querying operation, it decides if the querying item is local or not; that is, if local, it returns the found data item to the requesting site. If not local, it

forwards the query operation to the proper site to continue the search.

2. *Insert*(key, r)

To ensure that the insertion is a safe insertion, the insertion protocol must check if the data item is already in the system. This can be done by incorporating the actual insertion with a *Find* operation, which should be an atomic part of the insertion operation.

For the found case, the error message will be issued to the requesting site and the system state is the same as before. If not found, the data item should be sent to the site v, where it is supposed to go, and store data item at v. If no load balance is required, then the insertion is complete. Otherwise, the insertion of a single data item may cause the system to suddenly become unbalanced, even though it was balanced before.

3. *Delete*(key, r)

Delete(key, r) is similar to *Insert*(key, r) in which *Find* is also an atomic part of the deletion operation. The difference is that when the data is found at site v, the data can be deleted, otherwise, error occurs. The deletion also affects the balance status.

The criteria of balance depends on the definition of balance. Whenever the status of balance has been changed, the data will be shifted among some bins to restore the balance.

For a given site, if it is not the destination of an operation, the site simply forwards the operation, without change anything. Since only modification operations influence the balance status of system, in the following, we only observe *Insert* and *Delete* operations(messages). We say an operation m is *executed* at site i, if the data item in m is inserted or deleted at

site i. When an operation is executed at site i, *control* message is issued to inform other sites of the change.

The local state of a site (formal definition is given in Chapter 4) is always changing from one state to another because of the execution of operations or the reception of control message.

We define *local knowledge* to be the knowledge accumulated by the local executions of operations. Therefore, the initial knowledge about the site itself also belongs to the local knowledge. We also define *global knowledge* to be the knowledge accumulated by receiving control messages. Therefore, initial knowledge about other sites belongs to the global knowledge. Balance status will be changed based on accumulated local knowledge and global knowledge.

We use $history(s_i)$ to describe something that happened before at site i in its local state s_i. Formally, for each operation or control message m, if operation m is executed at site i or site i receives control message m during or before state s_i, then we say $m \in history(s_i)$.

Now, we can define balance function. *Balance function* is a boolean function such that, when the system is balanced it is true, and if at some point it is no longer balanced, it is false. Since there is no global state available, one can only observe the system status through a site. Thus, balance function is a function of a site's history, i.e. $balance(i) = balance(history(s_i))$.

Chapter 3

Binary-Tree-Structure BTS

3.1 DIST-BIN

Before presenting the *Binary-Tree-Structure*, we first briefly review the approach taken by *DIST-BIN* in [Pel89].

The *DIST-BIN* is based on each site serving as a "center", and a collection of bins $B_1, \ldots, B_p$ (each maintained in some site) storing the data in ordered fashion. Sites that currently do not store a bin are kept in a linked list of "free bins".

The *DIST-BIN*'s basic idea is to let each bin "remember" every other bin's information(e.g., storing the $Low(B_i)$ and the identity of the storing site of B_i for each bin B_i). This structure makes *Find* operations most efficient, since if the requesting data is not at local, the request can be directly sent to storing site.

However, insertions and deletions may require updating all sites, since the Low key of some bins may change. The scheme is therefore modified such that during some certain

stage, the pointers $Low(B_i)$ remain fixed between any two consecutive structural changes. In particular, it is possible that the item whose key is $Low(B_i)$ is deleted from B_i at some stage, but the pointer does not change until the next split or merge involving this bin.

The fact is that a bin has to be split or merged after certain amount of modification operations accessing it, and whenever that happens, it is indeed necessary to update all sites regarding the new boundary pointers of bins involved in the change. This is done by broadcasting these new pointers through the network.

When broadcast is expensive, we may avoid using this service. Observe the main reason to broadcast is that each bin has all other bin's information.

We choose another approach: each bin "remembers" only part of bins information in the network. A tree has such features: each internal node can store the information about its children. Then we introduce *Binary-Tree-Structure* (*BTS*).

3.2 Binary-Tree-Structure

The *Binary-Tree-Structure* (*BTS*) is a structure in which the network is logically viewed as a binary tree. Each site in the network is associated with a node in the binary tree, each site has its local storage called bin for keeping data. Each site can autonomously access data stored at local bin. When the requested data is not available in current site, the site can decide to send the request message to its neighbours.

We distinguish three kinds of neighbours: *parent*, *child*, and *adjacent* site. The parent and child are according to the common definition in a tree structure. The adjacent site of site i are site $i-1$ (if $i > 1$) and site $i+1$ (if $i < N$).

3.2.1 BTS Properties

In order to define the protocols in *BTS* , we first need to investigate the properties of *BTS* structure.

- The *BTS* is *static*, i.e. the Binary-Tree-Structure is fixed, each node in the binary tree corresponds to some vertex of the network.
- The da . items in each site are *ordered*. The smallest data in each internal node is bigger than the largest data of its left subtree, and the largest data is smaller than the smallest data of its right subtree.
- The data allocation is *dynamic*, i.e. data items can be moved to its adjacent site to keep the load balance.

We say the distributed data structure is *correct* if

- The binary tree is always consistent in the sense that the distribution of data items among the network must obey the following rules: If $i > j$, then the key values of all data items of site i are greater than the key values of all data items at site j.
- The execution of request operations is order-preserved. That is, for the request operations, originated from the same source and executed at the same destination, the execution should be order-preserved.
- There is no deadlock.

3.3 Policies

When designing the access protocol for a distributed data structure, there must be some policies to govern the data. Policies are used to decide what should be done in certain circumstance. Policies may vary, depending on different data structure or requirements. For the given set of operations (*Find*(key), *Insert*(key), and *Delete*(key)), we define the following policies for each site i:

1. **Transfer policy** : decides when the message containing the request operations should be transferred. Since the data is sorted in each bin, it only needs to compare the key with lower bound $lower(i)$ and upper bound $upper(i)$ of the data stored in current bin B_i. Denote

$$MyRange(i) = [lower(i), upper(i))$$

   So, when $key \notin MyRange(i)$, the request operation needs to be transferred.

2. **Location policy**: decides where the message containing the request operation should be transferred. When a message need to transfer, it can go up or down along the binary tree. According to *BTS* properties, the data in left children is smaller than that in current bin, and the data in right children is bigger than that in current bin. One more case should be considered: it is possible that the request data item is even smaller than the smallest data in the bin of leftmost children, that is, the request data does not belong to the subtree rooted at i. In this case, the request operation needs to go up. Denote

$$SubtreeRange(i) = [subtreelower(i), subtreeupper(i))$$

where *subtreelower*(i) (*subtreeupper*(i)) is the smallest (the largest) data in the left (right) most child's bin.

3. **Balance control**: decides when the system will be needed to re-balance. In practical systems, it is unnecessary to have a perfect load balance. We use a threshold to denote the the degree of unbalance that may be allowed in the system. Since *BTS* is a binary tree, we may want to restrict the unbalance between left tree and right tree. Denote

$$Thred1 = \frac{\mid LTreeData(i) - RTreeData(i) \mid}{(LTreeNode(i) + RTreeNode(i))} \leq Threshold \qquad (3.1)$$

where, $LTreeData(i)$ and $RTreeData(i)$ are the number of data items stored in site i's left tree and right tree, respectively; $LTreeNode(i)$ and $RTreeNode(i)$ are the number of nodes in site i's left tree and right tree respectively.

Intuitively, the value $Threshold$ indicates the average difference allowed between i's left tree and right tree in each site i. Therefore, when

$$Thred1 > Threshold$$

the system is considered unbalanced.

Sometimes, it is possible that a site is full or empty, but its ancestor is not aware yet of this situation. At this point, this site must block itself and send "full" or "empty" control message to its parent. When the parent receives this message, it considers the subtree not balanced, and it must start to restore to balance.

4. **Information gathering mechanism**: decides what kind of information the balance control needs in order to make decision and how this information is obtained. When

a site checks the current balance status, according to Equation 3.1, it needs to know the number of nodes in its left tree $LTreeNode(i)$ (and right tree). This information is static, and it can be obtained based on initial knowledge, and will not be changed (since the network is static).

Besides, site i also needs to know the number of data items currently stored in its left tree $LTreeData(i)$ and right tree $RTreeData(i)$. This information is dynamic information, and should be updated. Thus, when a data item is inserted in (or deleted from) some site, it is necessary to send a control message to its parent to inform it of this change. When the parent receives this control message, it updates $LTreeData(i)$ or $RTreeData(i)$. The parent will propagate this control message to its ancestors until the control message reaches root.

An insertion of a data item may also cause the current bin to become full. Then, besides sending the update control message, site i also sends "full" control message to its parent, and at the same time, blocks incoming modification operations, and enter into a new stage. The site receiving the "full" control message will decide to start balancing. Similar situation for a deletion of a data item.

5. **Concurrency control**: guarantees that the data structure is consistent. In *BTS*, we ensure that if two operations originate from the same site concerning about the same data key, then they are executed in the same order in their destination. We introduce lock mechanism to ensure consistency.

3.4 Descriptions of the Method

3.4.1 Approaches

The algorithm for the *BTS* is a complex protocol which provides the mechanism for the enforcement of the policies governing data resource use. That is to say how to do with the request operations. In this section, we will describe the approach taken by the protocol to support the three basic operations: *Find*, *Insert* and *Delete*.

3.4.2 Protocol for *Find* Operations

When an operation *Find*(key, r) is issued, the site r, which issues the operation, must first determine whether the element inquired is stored locally or not. To achieve this, each site i will maintain information

$$MyRange(i) = [lower(i), upper(i))$$

about the range of data stored in its own bin. This information is "local" knowledge of the site and it is sufficient to determine whether the requested item is stored locally. If this is not the case, then the location where the requested item is stored must be determined. To achieve this, every site i will maintain also the information

$$SubtreeRange(i) = [subtreelower(i), subtreeupper(i))$$

about the range of data stored in its subtree. This information is "global" knowledge.

So when a *Find*(key, r) is issued at site r, site r starts the process of locating the site i where the data item must be stored; note possibly $i = r$.

When a site i receives ***Find***(key, r), it does followings:

- if $key \notin SubtreeRange(i)$, then send ***Find***$(key, r)$ to i's parent;
- if $key \notin MyRange(i)$, then

  if $key < lower(i)$, then send ***Find***(key, r) to i's left child.

  if $key \geq upper(i)$, then send ***Find***(key, r) to i's right child.
- If $key \in MyRange(i)$

  if $key \in B_i$, then ***found***(i, key) is returned to the requesting site r.

  if $key \notin B_i$, then ***un-found***(i, key) is returned to the requesting site r.

Note: the ranges among the sites are consecutive, e.g., $upper(i) = lower(i+1)$. And the subtree range of root should be the domain of the data. Therefore any data must fall a site and only one site.

3.4.3 Protocol for *Insert* Operation

The protocol for processing an ***Insert***(key, r) operation is somewhat complex. The protocol needs not only to decide the location of the data, but also to update some global knowledge, take care of load balance, and interface with concurrency control.

When an operation ***Insert***(key, r) is issued at site r, the site must start the process of locating the site j where the data item must be stored; note that possibly $j = r$. The process is similar to the one for a ***Find*** operation. After the requests arrives at the site j, the item is stored locally. The global knowledge of the ancestors of j needs now to be updated; this process is started by j issuing an *Update-Glbl* message. If, after the operation, there is no

available space at j, the site also informs its parent by sending a *Children-Full* message, and locks itself.

Summarizing, an *Update-Glbl* message and, possibly, a *Children-Full* message will be sent by j to its parent. It is possible that the *Update-Glbl* message might cause a threshold to be paned. The site where this threshold is paned enters a *Lock-In* state and sends a *Lock-Inten* to its children; this site will be called a *lock-initiator*.

Any site receiving *Lock-Inten* sends *Lock-Intens* to its own children and becomes *Lock-In* state too. A leaf node, receiving *Lock-Inten*, will send *Acks* to its parent and enters state *Lock-St*. A site in state *Lock-In* will be collecting acknowledgements from its two children; it will then send *Acks* to its parent.

In state *Lock-In*, any modification request from the lock-initiator is blocked, while any query request is allowed to proceed. When the lock-initiator receives the two *Acks*, it enters state *Lock-St*, and the rebalance begins. During this period, every incoming message from the parent of the lock-initiator is put in a waiting queue.

3.4.4 Protocol for *Delete* Operation

It is similar to *Insert*'s. Instead of sending *Update-Glbl* to its parent, *DUpdate-Glbl* is sent. If, after deletion, the current site j becomes empty, site j informs its parent by sending a *Children-Empty*, and locks itself.

3.4.5 The Process of Rebalance

When it is necessary to start balance, there are several steps should be performed which are described as follows:

1. Decide in which direction the data should be moved. This can be done by checking the lock-initiator v. Clearly, data should be moved from lock-initiator's subtree with more data to the subtree with less data. We use $dir(v) \in \{$*left-bound, right-bound*$\}$ to denote the direction.

2. Calculate the amount of data needed to move for each node. This step is needed so to move the data in parallel. It requires that each site know exactly how much data should be moved out from each bin. Since the information about the data stored at a site is kept in its ancestors, the calculation will be performed in a top-down fashion on the tree.

   We start with the lock-initiator. After the rebalance, each site in the subtree rooted at v have roughly the same amount of data, denoted by Avg. Then we have

$$Avg = \frac{(LTreeData(v) + RTreeData(v) + MyData(v))}{(LTreeNode(v) + RTreeNode(v) + 1)} \tag{3.2}$$

   and

$$LTreeData'(v) = RTreeData'(v) = Avg * LTreeNode(v) \tag{3.3}$$

   where $LTreeData'(v)$ and $RTreeData'(v)$ are the amount of data in left tree and right tree after rebalance, respectively. Thus, the amount of data moving out and in

from v will be

$$out(v) = \begin{cases} LTreeData'(v) - LTreeData(v) & \text{if } dir(v) = \textit{left-bound}. \\ RTreeData'(v) - RTreeData(v) & \text{otherwise}. \end{cases} \tag{3.4}$$

$$in(v) = out(v) + Avg - MyData(v) \tag{3.5}$$

Lock-initiator v then sends *Start-Blnc* to its children with contents $in(v)$ and $out(v)$. Since the direction of moving is one of parameter in calculating the amount of data moved within the locked subtree, the lock-initiator is included in every message sending in *Lock-St* state.

Without lose of generality, consider a node i receiving *Start-Blnc* from its parent j with contents $in(j)$, $out(j)$ and $dir(v)$. After balance,

$$LTreeData'(i) = RTreeData'(i) = Avg * LTreeNode(i) \tag{3.6}$$

Notice that the data to be moved out from bin i must be equal to the data moving into its subtree. After rebalance, the data in each bin should be the same as Avg. We can therefore derive the following equations.

- If $dir(v)$ =*left-bound* and i is j's left child, then

$$\begin{cases} in(i) = RTreeData(i) - RTreeData'(i) + out(j) \\ out(i) = MyData(i) - Avg + in(i) \end{cases} \tag{3.7}$$

- If $dir(v)$ =*left-bound* and i is j's right child;

$$\begin{cases} out(i) = LTreeData'(i) - LTreeData(i) + in(j) \\ in(i) = Avg - MyData(i) + out(i) \end{cases} \tag{3.8}$$

- If $dir(i)$ =*right-bound* and i is j's left child:

$$\begin{cases} out(i) = RTreeData'(i) - RTreeData(i) + in(j) \\ in(i) = Avg - MyData(i) + out(i) \end{cases} \tag{3.9}$$

- If $dir(i)$ =*right-bound* and i is j's right child

$$\begin{cases} in(i) = LTreeData(i) - LTreeData'(i) + out(j) \\ out(i) = MyData(i) - Avg + in(i) \end{cases} \tag{3.10}$$

Figure 3.1 demonstrates how Equation 3.9 is derived.

3. Move the data in parallel. At this time, each bin can independently move to its $dir(i)$'s adjacent neighbor the amount $out(i)$ of data provided the data is available. Since sometimes it is possible that the amount of data to be moved is less than that present in the bin, the bin must wait until it is available. (i.e., transferred to it by a neighbour.

4. Execute the blocked messages in a order-preserved way. Lock-initiator starts balance itself while other sites have to wait *Start-Blnc* from their parents in order to decide how much data to move. After a site knows the amount of data to move, it moves it. It is possible that it has not enough at its own site(e.g. the data supposed to move in has not come yet); in this case, the site has to wait until the data arrives. It is also possible that a site first receiving moving data from an adjacent site, cannot yet decide how much data it has to move to the other adjacent site. In this case, it just waits until it receives a *Start-Blnc* message. Summarizing,

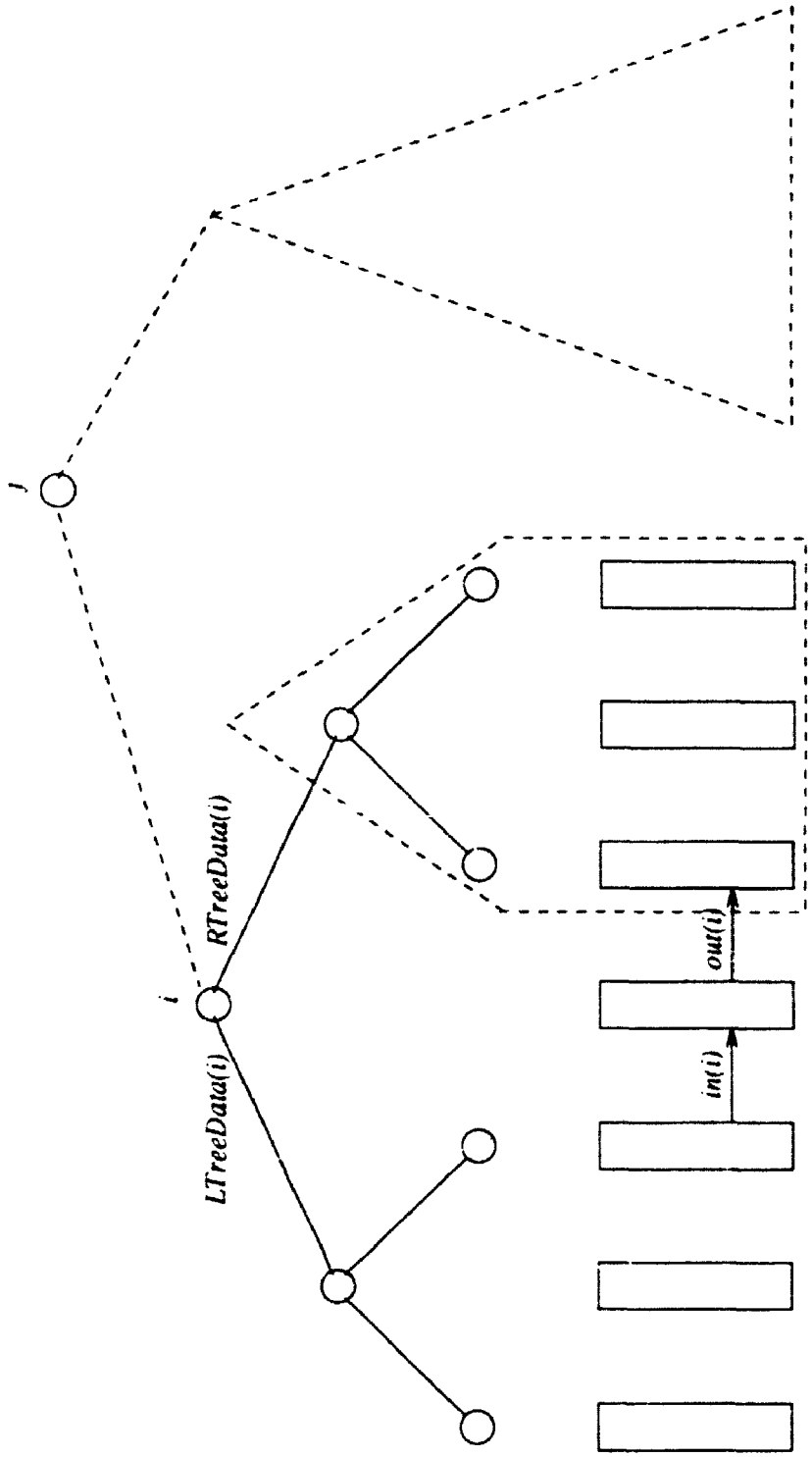


Figure 3.1: Demonstration of Equation 3.9

- if *Start-Blnc* arrives first, and the current site has enough data to move, then it moves the corresponding amount of data. Otherwise, it waits until it has enough data to move, and then moves it.
- if data arrives first, it cannot decide how much data to move. That means the current site must wait for *Start-Blnc*.

3.5 Examples

Example 3.1 Suppose there seven sites in the given network which is set up initially load balanced, and each site has the initial knowledge, see Figure 3.2. Now, there is a request *Insert*(360) originating from site 6. This request will travel along the path site 6→site 4→site 2 →site 3. After *Insert*(360) is executed at site 3, a control message *Update-Glbl* is sent to site 2 by site 3. Site 2 continues to propagate this message until it arrives at root which is site 4 in this example.

Example 3.2 Now, we look at how *BTS* manipulates the load balance process. Suppose the *Threshold* allowed in the current system is 2. That is, the average difference of the number of keys allowed is 2.

Consider a situation that a $key = 390$ is successfully inserted at site 3, see Figure 3.3.After this operation, site 3 updates $MyData(3)$, and then sends a *Update-Glbl* message to site 2. When site 2 receives this *Update-Glbl* from site 3, it updates $RTreeData(2)$ and sends *Update-Glbl* to site 4. When site 4 receives *Update-Glbl*, it updates $LTreeData(4)$, and checks the current balance condition by calculating $Thred1$ using Equation 3.1.

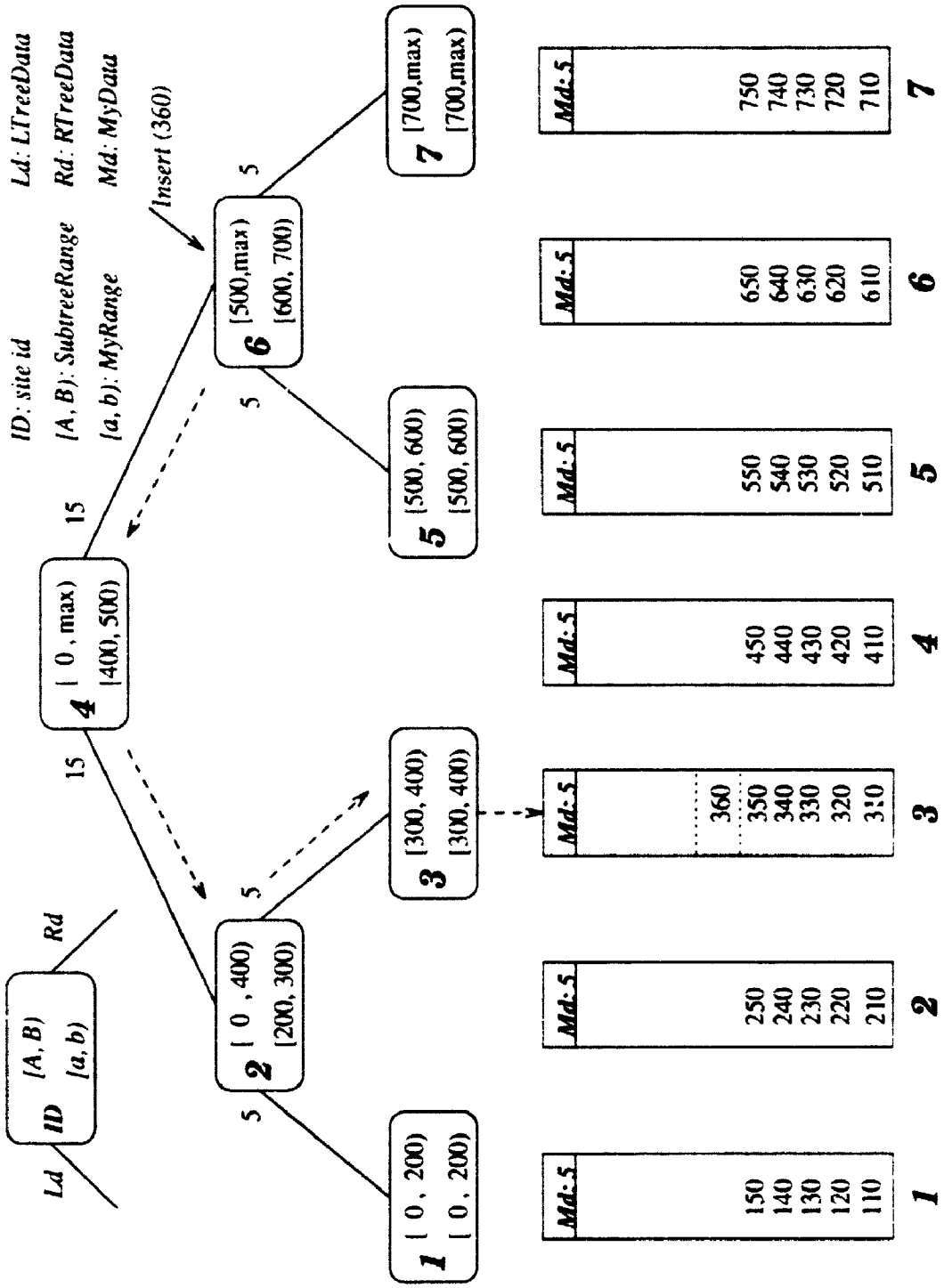


Figure 3.2: *Insert*(360) operation in *BTS*

$Thred1 = 2.3 > Threshold$, that means that the system is not balanced and needs to restore the balance.

After balance, the average amount of data in each site should be roughly the same(using Equation 3.2)

$$Avg = (29 - 15)/(3 + 3) = 7.3$$

Site 4 calculates $in(4)$ and $out(4)$ as follows:

$LTreeData'(4) = RTreeData'(4) = Avg * LTreeNode(4) = 7.3 * 3 = 21.9$

$out(4) = RTeeData'(4) - RTreeData(4) = 21.9 - 15 = 6.9 \simeq 7$

$in(4) = out(4) + Avg - MyData(4) = 6.9 + 7.3 - 7 = 7.2 \simeq 7$

Then, site 4 sends ***Start-Blnc*** with contents: $in(4)$, $out(4)$, and $dir(4)$=***right-bound*** to site 2 and site 6. When site 2 and site 6 receives ***Start-Blnc***, they calculate:

$LTreeData'(2) = RTreeData'(2) = Avg * LTrreNode(2) = 7.3 * 1 = 7.3$

$out(2) = RTreedata'(2) - RTreeData(2) + in(4) = 7.3 - 9 + 7.2 = 5.5 \simeq 6$

$in(2) = Avg - MyData(2) + out(2) = 7.3 - 9 + 5.2 = 3.5 \simeq 4$

$LTreeData'(6) = RTreeData'(6) = Avg * LTreeNode(6) = 7.3$

$in(6) = LTreeData(6) - LTreeData'(6) + out(4) = 5 - 7.3 + 6.9 = 4.6 \simeq 5$

$out(6) = MyData(6) - Avg + in(6) = 5 - 7.3 + 4.6 = 2.3 \simeq 2$

Similar calculation for site 1, site 3, site 5 and site 7. Notice subtree data and subtree node in these site are zero.

$out(1) = in(2) = 3.5 \simeq 4$

$in(1) = Avg - MyData(1) + out(1) = 7.3 - 11 + 3.5 \simeq 0$

$in(3) = out(2) = 5.5 \simeq 6$

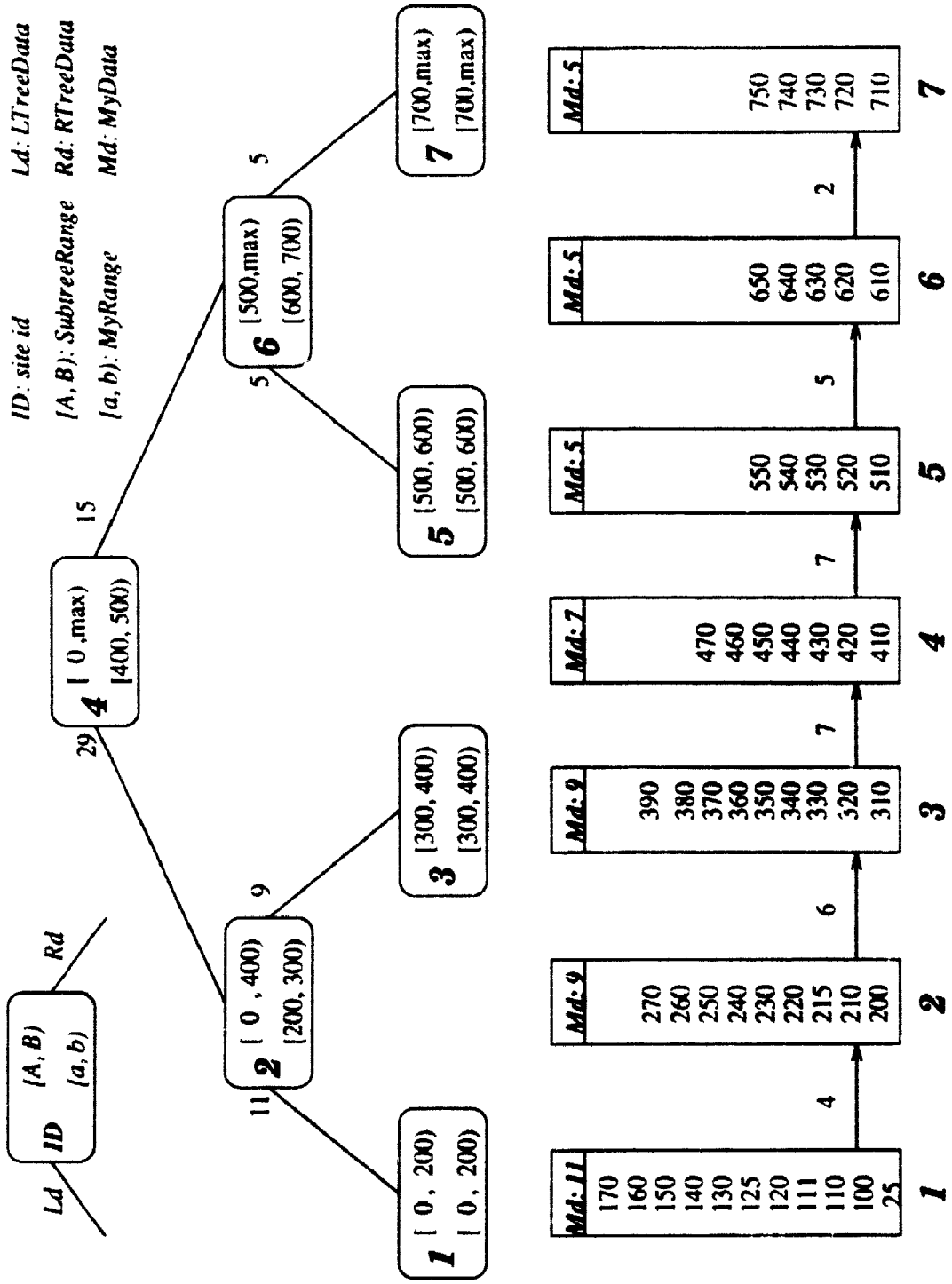


Figure 3.3: Balance computation in *BTS*

$out(3) = MyData(3) - Avg + in(3) = 9 - 7.3 + 5.5 = 7.2 \simeq 7$

$out(5) = in(6) = 4.6 \simeq 5$

$in(5) = Avg - MyData(5) - out(5) = 7.3 - 5 + 4.6 = 6.9 \simeq 7$

$in(7) = out(6) = 2.3 \simeq 2$

$out(7) = MyData(7) - Avg + in(7) = 5 - 7.3 + 2.3 = 0$

Figure 3.4 shows the status of *BTS* after balance from Figure 3.3.

The complete algorithm will be described in the following chapter.

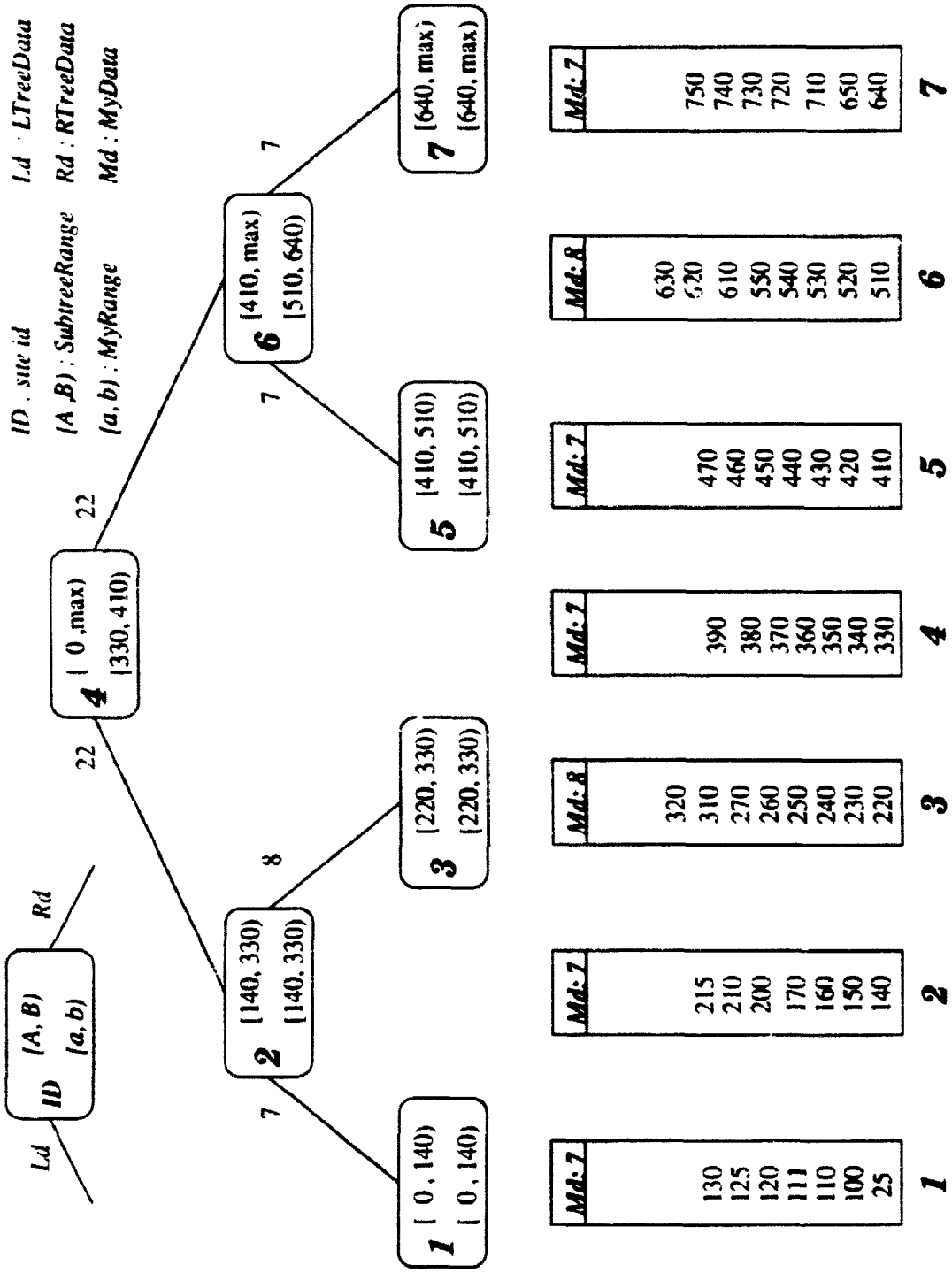


Figure 3.4: After balance from Figure 3.3

Chapter 4

Correctness Of BTS

4.1 Descriptions Of Protocol

As mentioned before, a state describes the complete instantaneous state of the system, and a local state of a node is part of system state, therefore a different local state can be used to represent what different stage the system is in. In each local state of a node, the node has the corresponding functionality which is responsible for the local management and remote cooperations. We distinguish four kinds of states for each node, and we will give a description of the protocol.

4.1.1 State *Initial*

This is initial state of each node. Assume each node is set up with same amount of data, therefore, the system is balanced initially. We also assume a node has its initial knowledge: local knowledge, global knowledge, and the network structure,

Function	*Find*	*Insert*	*Delete*
Issue	yes	yes	yes
Process	yes	yes	yes
Block	no	no	no

Table 4.1: Functionality of a node in *Active* state

- Local knowledge—*MyRange*: $[lower, upper)$;
- Global knowledge—*SubtreeRange*: $[subtreelower, subtreeupper)$;
- Network structure—the network is a binary tree structure, each node knows tl structure and its own location.

4.1.2 State *Active*

The system is normally in this state. Occasionally, some node issues query instruction *Find*(key) or modification instruction *Insert*(key) or *Delete*(key) among the network. When a request arrives at a node i, node i processes the request by processing at local site or sending to remote site. The request is never blocked. Table 4.1 describes the functionality of a node in *Active* state.

Since the situation of deletion is similar to that of insertion, in this chapter, we focus on *Find*(key) and *Insert*(key) operations.

According to different state of system, a node can receive various kinds of messages. A

node in *Active* state may receive the following messages: *Find*(key), *Insert*(key), *Update-Glbl*(*rqst-id*), *Children-Full*, and *Lock-Inten*(lock-init).

Upon reception of *Find*(key) **or** *Insert*(key)

The received message will be processed locally, go up, or go down according to the transfer policy and location policy. *Find*(key) or *Insert*(key) will finally arrive at the appropriate site i in whose local range the data belongs to.

Find will not change the system status, while *Insert* may cause local full, global unbalance, etc.. In the following, we will focus on the effect when the insertion is executed.

- If key exists in site i, then error message is returned(safe insertion);
- If site i is not full, then the data key is inserted in site i, *Update-Glbl* is sent along the path of its ancestors;
- If, after insertion, site i is full, it sends *Children-Full* to its parent. When site i receives *Update-Glbl*, it always checks load-balance condition. At the same time, *Update-Glbl* continues going up (until it gets to root).

Whenever the unbalance condition is detected, it is necessary to begin to balance starting from this site. Otherwise the insertion is finished. In the former case, the *Lock-Intens* are sent to its children, and itself becomes *Lock-In* state.

Upon reception of *Children-Full*

Since site i is in state *Active*, reception of such a message means that site i has not detected the global unbalance before receiving *Children-Full* (otherwise it would have entered *Lock-In* or *Lock-St*, which will be described later on). It sends *Lock-Intens* to its children, closes link from self, and becomes *Lock-In*.

Upon reception of *Lock-Inten*

Site i sends *Lock-Intens* to its children, and collects acknowledgments *Acks*, and becomes *Lock-In*. If it is a leaf node, it sends *Ack* to its parent, and becomes *Lock-St*.

4.1.3 State *Lock-In*

In this state, each site collects *Acks* from its children, *Update-Glbl* keeps going up. *Find*(key), *Insert*(key) are processed in the same way as in state *Active*, and *Lock-Inten* (the later message comes from the ancestor of this lock-initiator, which can be processed after lock is released) is sent to itself and is blocked. There is no new request to be issued in this state. For any request from parent of the lock-initiator, *Insert* or *Delete* is blocked at lock-initiator while *Find* can get through. Table 4.2 describes the functionality of a node in *Lock-In* state.

Upon reception of *Ack*

If site i has never received *Ack*, then it does nothing. Suppose site i has already received two *Acks*. If site i is not the lock-initiator, then it sends *Ack* up, and becomes *Lock-St*.

Function	*Find*	*Insert*	*Delete*
Issue	no	no	no
Process	yes	yes	yes
Block	no	yes	yes

Table 4.2: Functionality of a node in *Lock-In* state

Otherwise(i is lock-initiator) site i checks load-balance conditions again. This is necessary since at the time that lock-initiator decides to balance, the information about its subtree is not up-to-date; when the lock-initiator has received two *Acks*, it guarantees all *Update-Glbls* have been received(after leaves send *Acks* to their parents, no insertion is processed). Each site i checks the balance condition according to the most up-to-date information. If the threshold is triggered, then site i sends *Start-Blnc* and become *Lock-St*. Otherwise it sends *No-Shift* to its children, opens link from self, and becomes *Active* again.

Upon reception of *Update-Glbl*

Site i updates corresponding information, but does not check the load-balance condition. This is reasonable since its ancestor has already decided to balance.

Upon reception of *Children-Full*

The received message would be ignored. Since the current site has already decided (lock-initiator) or taken part in (the decedent of lock-initiator) the load balance.

Function	*Find*	*Insert*	*Delete*
Issue	no	no	no
Process	no	no	no
Block	yes	yes	yes

Table 4.3: Functionality of a node in *Lock-St* state

4.1.4 State *Lock-St*

In this state, *Start-Blnc* from parent decides how much data will be needed to be moved out and moved in, the message *Moving* from neighbor delivers the moving data items to this site. But those neighbour site. are restricted each other by some conditions due to the sequence of moving data (e.g. a site has received *Start-Blnc*, and it knows how much data should be moved out, but it does not have enough data to move or it receives *Moving* first, but it does not know how much to move). In this state, no new request is issued; any old request must have arrived at its destination(see proof); any request from parent of lock-initiator is blocked at lock-initiator. Table 4.3 describes the functionality of a node in *Lock-St* state.

Upon reception of *Start-Blnc*

After data is moved in and moved out, the range in each site in this state is changed to a new range. F r the leaf node, it sends *New-Range* to its parent to update parent's subtree range, opens link from self, and becomes *Active* again.

Upon reception of *New-Range*

The site i updates information about *subtree(SubtreeRange, LTreeData, RTreeData)*. If it has received two *New-Ranges*, and i is not lock-initiator, then it sends *New-Range* to parent, opens link from self, and becomes *Active*.

Upon reception of *No-Shift*

The site i knows that the lock-initiator has decided not to balance at the present time. Site i passes the *No-Shift* to its children.

Upon reception of *Children-Full*

The received message would be ignored , since the current site has already decided (lock-initiator) or taken part in (the decedent of lock-initiator) the load balance.

4.2 Proof Of Partial Correctness

4.2.1 Some Definitions

Properties of a protocol are defined in terms of its execution. An execution is a sequence of global states, where each global state gives the state of all the processes and contents of the channels. An execution should start from the initial global state, where each process i is in its initial state and all channels are empty. The execution can thus proceed; at each step process i performs a basic action BA_i:

BA_i: {a message m_{ki} from process k arrives at process i }

begin *receive*(m_{ki});

compute;

for all $q \in Q$ **do** *send* m_{iq} to q

end.

where Q is finite sequence of processes generated by *compute*.

1. receives a message m_{ki} by removing it from channel C_{ki} at the side of process i.

2. sends a message m_{iq} by appending it to the channel C_{ik} at the side of process i.

We define basic sequence

$$seq(s_i) = (+m_{ki})(-m_{ix}m_{iy}m_{iz})\ldots$$

which means that process i at state s_i receives m_{ki}, and decides to send $m_{ix}m_{iy}m_{iz}\ldots$

We distinguish two kinds of messages: request operations generated by the outside environment and control messages generated by request operations or by other control messages. The life time of a request operation is from the time when it is generated by the outside environment messages(at source) and to the time when it is executed at a local process (at destination). So, for a request operation, on the way to its destination, the intermediate processes just forward the request to the next possible destination and the request does not influence the status of processes on the path. When the request arrives at its destination i, process i has a computation that changes the status of the process (for the modification request). Further more, the control messages will be issued to inform other sites about this change. We say process i has a *history* if it has a local computation or

received control message in the past. Process i's ***history*** at current state p_i is denoted by $history(p_i)$.

Definition 4.1 A *local-state* s_i of process i is a pair

$$< p_i, history(p_i) >$$

Where

- p_i is the current state of process i, and $p_i \in P = \{$ ***Initial, Active, Lock-In, Lock-St*** $\}$.
- $history(p_i)$ is the set of histories accumulated at p_i.

The $history(p_i)$ includes initial knowledge about the network structure, denoted by ***Init-Know***. Initially, for each $i \in [1, N]$, $p_i = Initial$, $history(Initial) = \{InitKnow\}$. An example of $history(p_i)$ may be the set of executions at site i during p_i: { 'I am leaf', receive $m_{ki}, \ldots,$}.

Definition 4.2 Process i is a *finite-state machine*, defined by a quadruple

$$< S_i, I_i, M_i, Next >$$

Where

- S_i is the set of local states of process i;
- I_i is an element of S_i, $I_i \in S_i$, i.e. $I_i =< Initial, InitKnow >$;
- M_i is the set of messages that can be sent or received by process i;

- $Next$ is a partial mapping function $S_i \times M_i^+ \rightarrow S_i$.

  $Next(s_i, seq(s_i)) = s_i' = < q_i, history'(p_i) > \in S_i$.

  where

  $$s_i =< p_i, history'(p_i) >$$

  $$seq(s_i) = mm_1m_2 \ldots m_k$$

  $$q_i = \begin{cases} p_i' \in P & \text{if } f(history'(p_i)) = \text{true} \\ p_i & \text{otherwise} \end{cases}$$

  $history'(p_i)$ is the same as before except: If (1) m is a modification operation, and i is its destination; or (2) m is a control messages, then i change corresponding *history*. Otherwise, nothing is changed.

  p_i' is the state changed from p_i after process i receives m, at current process state p_i, i.e. $history'(p_i) = history(p_i) \cup \{receive\ m\}$.

  $f(history'(p_i))$ is a boolean function such that for the given state p_i and $history'(p_i)$, it is true, otherwise it is false. For example, a node (or process) i in state *Active*, if $\exists$ $history(Active)$, such that $balance(history(Active)) = false$, then $f(history(Active)) = true$.

Definition 4.3 A *global state* is a pair

$$< S, C >$$

where

- S is a N-tuple of local states $s_1, s_2, \ldots, s_N$;

- C is an N^2-tuple $(c_{11}, c_{12}, \ldots, c_{1N}, \ldots, c_{NN})$;
- each c_{ij} is the contents of the channel from j to i;
- c_{ii} is considered as a channel from self to self. It is used as a waiting queue at process i. The initial state $S_0 =< Initial, Initial, \ldots, Initial >$, and each channel is empty, $C_0 =< \Phi, \ldots, \Phi >$, i.e., initial global state is $< S_0, C_0 >$.

Definition 4.4 Let $\vdash^*$ be the irreflexive and transitive closure of $\vdash$, and $< S, C >$ and $< S', C' >$ be the global states. If there exists a basic sequence seq that can change the global state from $< S, C >$ to $< S', C' >$, we say that S' is reachable in one step from S , denoted by $< S, C > \vdash < S', C' >$. For the given initial global state $< S_0, C_0 >$ and sequences $seq_1, \ldots,$, if there exists a sequence of system states: $T = S_1, S_2, \ldots, S_l$,

$$< S_i, C_i > \vdash < S_{i+1}, C_{i+1} > (for\ all\ i < l)$$

we say S_l is reachable, denoted by

$$< S_0, C_0 > \vdash^* < S_l, C_l >$$

In fact, when a basic action *BA* occurs at process i (let $seq(s_i) = mm_1m_2 \ldots m_k$ be the corresponding basic sequence), there exists a relation

$$< S, C > \vdash < S', C' >$$

where $< S, C >$ is original global state, whereas $< S', C' >$ is the changed state. All are the same except that

- $s'_i = Next(s_i, seq(s_i))$

- $c'_{ji} = c_{ji}$ after removing m at the head of the sequence of messages in this channel (process j sent m through this channel).

Definition 4.5 *Sp-state* is a class of global states

$$\{< s_1, c_1 >, < s_2, c_2 >, \ldots, < s_N, c_N >\}$$

$\exists i, j \in [1, N], i < j$, such that

$$p_i = p_{i+1} = \ldots = p_j = \textbf{\textit{p-state}} \in P$$

where

- $s_i =< p_i, history(p_i) >$;
- $c_i = \{c_{i1}, c_{i2}, \ldots, c_{iN}\}$;
- *Sp-state*(i) is a global state, such that $p_i =$ *p-state*. What other process's state is does not matter.

Initially, each process is in its *Initial* state. So $S_{Initial}$ is an initial global state such that

$$p_1 = p_2 = \ldots = p_N = Initial$$

For the given *BTS*, we consider a subtree such that i, j is left most and right most node, respectively, and rooted at $r = (i + j)/2$. Therefore, when we say *Sp-state*($i - j$) is reachable, then the whole subtree rooted at $r = (i + j)/2$ is in *p-state* state.

4.2.2 Approaches

As stated in Chapter 2, the *partial* correctness of an algorithm means that the program does nothing wrong. Equivalently, the program yields no wrong results. Partial correctness properties are usually referred to as *safety* properties. Formally, a safety property is expressed in terms of an invariant, an assertion that must always hold during the execution of the algorithm.

Let A be an assertion of which we want to prove invariant, and E an action of the system. We say A is preserved under E if A holds after E is applied to a system state. If A is preserved under all actions of the system, we say A is *stable*. It implies that if A holds on some system state S, it also holds in all system states reachable from S. If A holds initially and A is stable, then A is an invariant of the system. Since an action may be non-deterministic, an action in one state can change in one step to all of the possible states. The entire proof will be quite long. In the literature, the following heuristic approaches are used:

1. The "variable disjointness rule".

2. Use earlier results: If invariance of an assertion $A1$ has been proved, we know that every reachable system state satisfies $A1$. Sometimes the proof that an action E preserves an assertion $A2$ can be shortened by using this for the state before or after the action of E.

3. Prove that a class of assertions is preserved under an actions: Sometimes it is possible to express properties of an action by identifying a class of assertions which it preserves.

4. Prove that an assertion is preserved under a class of actions: Sometimes it is possible

to express an properties of an assertion by identifying a class of actions which preserve the assertions .

We are going to use the combinations of 2, 3 and 4.

4.2.3 Assertions

First, according to the system requirement, we define the following assertions A:

$A1$: At any time, the data in the logical binary tree is maintained consistent.

$A2$: At any time, the system is balanced. That means as long as a non-leaf node has noticed that its subtree is not balanced, or a node's children is full. it is always blocked and begins to balance.

$A3$: The execution of the operations satisfies concurrency constraints.

During *Lock-In* and *Lock-St* state, for any instruction from outside of the balance tree, *Insert* is blocked while *Find* is proceeded, we cannot guarantee they arrive in the same order, but if we give the same result, we can say it is consistent.

As long as we can verify $A1$, $A2$, and $A3$, we say A holds and therefore the system is consistent. So we need to prove:

1. that A holds in the initial state.

2. that for each action E, A is preserved under E, or that any $A_i \in A$ is preserved under a class of actions.

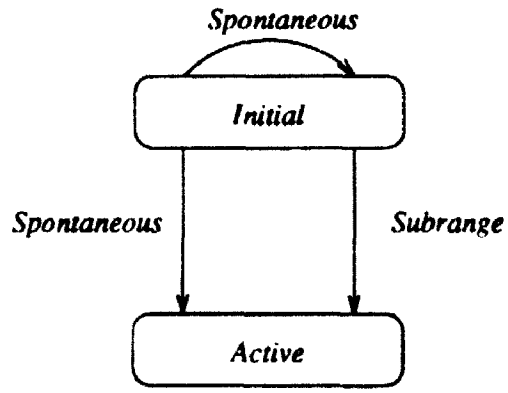


Figure 4.1: State transmission from state *Initial* to state *Active*

4.2.4 Proof

Property 4.1 Initially, the system is set up with balanced load, and each node keeps proper local and global information. The range of each node is consecutive in the sense that any data item in the domain must fall into one and only one bin.

Theorem 4.1 A holds in $S_{Initial}$ state.

Lemma 4.1 $S_{Initial} \overset{*}{\vdash} S_{Active}$, i.e. S_{Active} is reachable.

Proof: Initially, p_k = *Initial*, for $\forall k \in [1..N]$. We need to prove that $\exists\, i, j \in [1..N]$ such that $\forall k \in [i..j]$, $S_{Initial}(k) \vdash S_{Active}(k)$. In fact, during $S_{Initial}$ state, every process k can only receive *Spontaneous* or *Subrange* event in a basic action BA_k. The transition of state is shown in Figure 4.1.

Leaf nodes and internal nodes have different chance to become *Active*:

1. Leaf node becomes *Active* by receiving *Spontaneous*:

$$\exists\, seq(p_k) = +m_0(-m_1)$$

where $m_0 = Spontaneous$, $m_1 = Subrange$ such that

$$history'(p_k) = history(p_k) \cup \{receives\ m_0\}$$

$$Next(< p_k, history'(p_k) >, seq(p_k)) = q_k = Active$$

2. Internal node becomes *Active* by receiving two *Subrange* messages. Each internal node has two children, by induction, if each child has become *Active* , then it must have sent *Subrange* to its parent. Therefore, each internal node will receive two *Subrange* from its children.

Let

$$seq(p_k) = +m_0(-m_1)$$

where

$$m_0 = Subrange, m_1 = Subrange$$

$$history'(p_k) = history(p_k) \cup \{receives\ m_0\}$$

then

$$Next(< p_k, history'(p_k) >, seq(p_k)) = q_k = Active$$

If receiving *Subrange* appears in *history*(p_k) two times, then $S_{Initial}(k) \vdash S_{Active}(k)$.

For some instant, if a internal node becomes *Active* , then the whole subtree under this node must have become *Active*, and let the corresponding leftmost node be i, and let the right most node be j. □

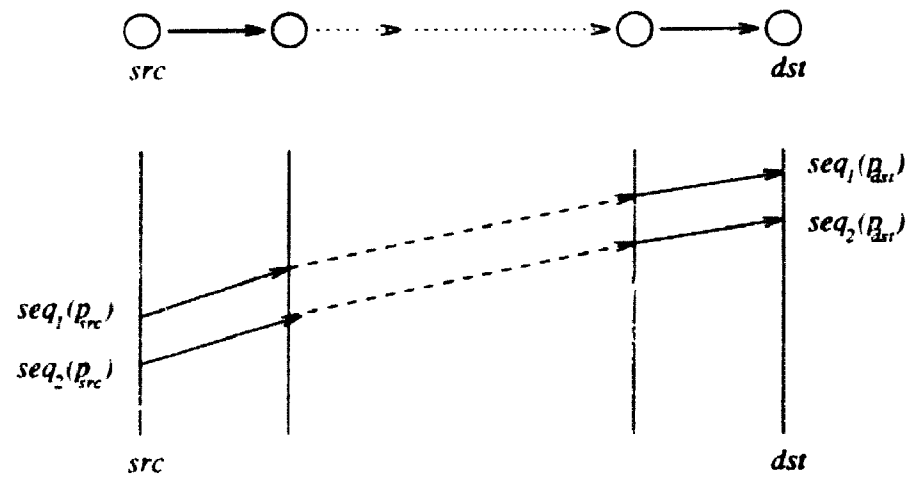


Figure 4.2: Space view for two events

Property 4.2 *Insert* and *Find* request operations behave in the same way except: when *Insert* arrives its destination, an *Update-Glbl* is issued and propagated to its ancestors.

Lemma 4.2 Each request operation will finally get to the destination, and execution of operations at the same destination from the same source is order-preserved.

Proof: Let src be a process issuing $op_1(key), op_2(key)$ request operations such that they are in two basic sequence: seq_1, seq_2, see Figure 4.2. Where

$$seq_1(p_{src}) = +op_1(-op_1), seq_2(p_{src}) = +op_2(-op2), op_1 \rightarrow op_2$$

i.e. $seq_1(p_{src}) \rightarrow seq_2(p_{src})$, then there exist a total ordering

$$\ldots seq_x(p_{src}) \Rightarrow seq_y(p_{src}) \ldots seq_v(p_{dst}) \Rightarrow \ldots \Rightarrow seq_w(p_{dst})$$

such that the two request are computed at process dst in the order $seq_v(p_{dst}) \Rightarrow seq_w(p_{dst})$. In fact, a process delivers the request to the next process according to the range. By Lemma

4.1, S_{Active} is reachable, we can assume that on the way to the destination, each process is in *Active* state. Since the range does not change during *Active* state, $op_1(key)$ and $op_2(key)$ should have the same destination with a single path.

By induction, in each step in intermediate process k, the receiving order is kept as $seq_1(p_k) \rightarrow seq_2(p_k)$, therefore, at their destination dst, $seq_1(p_{dst}) \rightarrow seq_2(p_{dst})$. □

Theorem 4.2 A holds in S_{Active} state.

Proof: from Lemma 4.1, Lemma 4.2. □

Theorem 4.3 $S_{Active} \overset{*}{\vdash} S_{Lock-In}$, i.e. $S_{Lock-In}$ is reachable from S_{Active} state.

Proof: We divide the proof into two parts:

First, one node(called lock-initiator v) can decide to balance through observing the *history*, initiates *Lock-Inten* to its subtree, and itself becomes *Lock-In* state.

Second, all the descendants of lock-initiator will become *Lock-In* state. See Figure 4.3.

1. Let ST be a subtree rooted at v, $v = (i + j)/2$, where i, j is v's leftmost node and rightmost node, respectively. Then $S_{Lock-In}(v)$ is reachable. This can be true as long as v receives *Update* message with some *history*, such that *balance(history(p_v))* = *false*. And thus sends *Lock-Inten* to its children, itself becomes *Lock-In*. i.e.

$$\exists\, seq(p_v) = +m_0(-m_1)(-m_2)$$

such that

$$m_0 = Update, m_1 = m_2 = \textit{Lock-Inten}$$

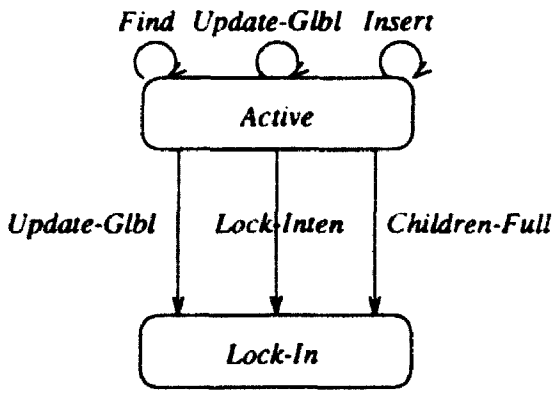


Figure 4.3: State transmission from state *Active* to state *Lock-In*

$$history'(p_v) = history(p_v) \cup \{seq(p_v)\}$$

$$Next(p_v, history'(p_v)) = q_v = \textit{Lock-In}$$

2. Suppose there is no node under v deciding to balance itself(if does, it is blocked). i.e. $\exists\, k \in ST$, it can only become *Lock-In* by receiving $m =$ *Lock-Inten*, and it will propagate the *Lock-Inten* to its subtree to inform its children to become *Lock-In*. By induction, all nodes in the subtree ST will become *Lock-In* state when all leaf nodes become *Lock-In*. □

Property 4.3 i) Lock-initiator is the first node entering *Lock-In* state for the whole subtree intending to balance; ii) Any request operation is issued during *Active* state. i.e. in *Lock-In* state, no new request operation is generated; iii) In *Lock-In* state, lock-initiator can receive *Insert*, *Find*, *Lock-Inten* (from another lock-initiator) from its parent, *Insert* request and *Lock-Inten* are blocked at lock-initiator while *Find* requests can get through.

The *Lock-In* is only an interim stage from *Active* to *Lock-St* which will influence the order-preserving of request operations, therefore, it is essential to guarantee that:

1. Each request operation must have arrived at its destination dst before dst becomes *Lock-St*;

2. Each node in balance tree ST has the most up-to-date information about its subtree before it begins to balance, i.e. before receiving ***Start-Blnc*** from its parent. This is necessary since when a node starts to balance, it needs to know how much data in its subtrees to decide the amount of data to be moved in or/and moved out.

In order to observe the behavior of the system during *Lock-In* state, we now look at the possible paths of a request operation with at least one of its source or its destination belonging to the balance subtree ST. See Figure 4.4.

1. **Downward**: two kinds of sources to a destination in ST : i) from outside of ST, denoted by PT_1; ii) from inside of ST, denoted by PT_2;

2. **Upward**: two kinds of destination from a source in ST i) to outside of ST, denoted by PT_3; ii) to inside of ST, denoted by PT_4;

3. **Upside-down U**: i) whole path in ST, denoted by PT_5; ii) part of path in ST, denoted by PT_6, PT_7.

That is to say, if a request operation $op \in \{$ *Insert, Find*$\}$ has a path with at least one source or destination belonging to ST, then the path must belong to PT_i, denoted by $op \propto PT_i, i = 1 \ldots 7$.

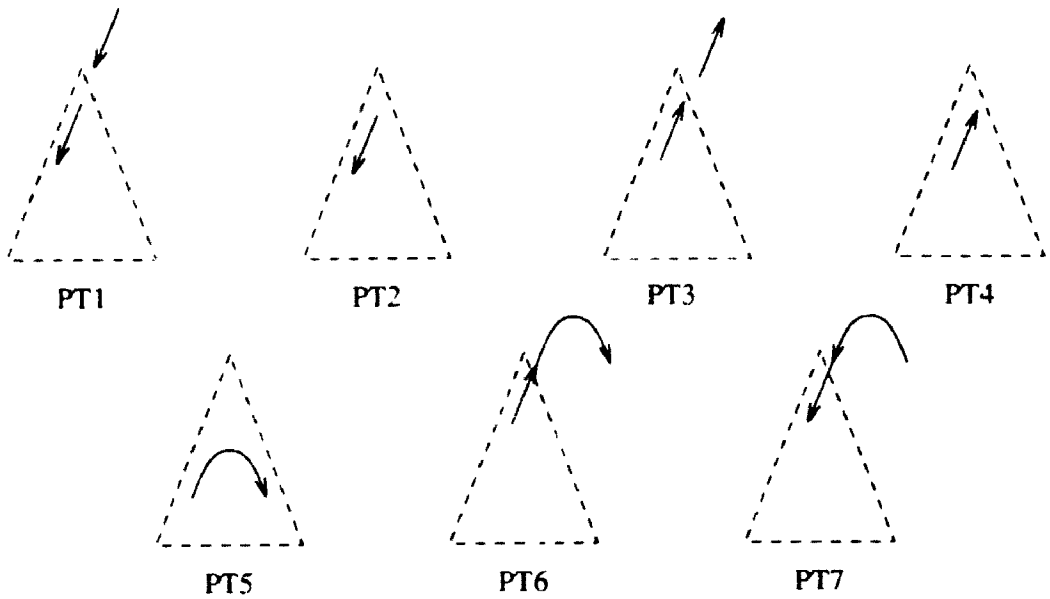


Figure 4.4: The possible paths of a request operation

Let

$$I = \{Insert(ikey_1), Insert(ikey_2), \ldots, Insert(ikey_m)\}$$

be *Insert* requests blocked at lock-initiator during *Lock-In* state, and

$$Ikey = \{ikey_1, ikey_2, \ldots, ikey_m\}$$

And let

$$F = \{Find(fkey_1), Find(fkey_2), \ldots, Find(fkey_n)\}$$

be *Find* requests received from parent of lock-initiator. $op(j)$ represents the event that process j receives a request. $op(src)$ represents the event that op originates at source src, $op(dst)$ represents the event that destination dst receives op. Then we have the following Lemmas:

Lemma 4.3 If $op \propto PT_1$ or $op \propto PT_7$, then $op(dst) \rightarrow$ ***Start-Blnc***(dst).

Proof: Since all *Insert* operations are blocked at lock-initiator, we just need to look at how the *Find* operations are executed.

1. For all *Find*$(fkey_k)$ from parent of lock-initiator, such that $fkey_k \in Ikey$, they will be executed at lock-initiator in *Lock-In* state.

   This is true since the *Find* request with $fkey_k \in Ikey$ does not have to get to its destination. Rather, when it gets to the lock-initiator, and it finds out that there is a *Insert* request blocked there, it can be sure that the key does exist in the system and the *result* (e.g. "found") can be returned.

2. if $fkey_k \notin Ikey$, then they will be executed before data is moved. This is true since during *Lock-St*, every message from parent of lock-initiator is blocked, then any op from parent of lock-initiator happens before *Start-Blnc*, i.e. $op \rightarrow$ *Start-Blnc*. op(in fact only *Find*) can only goes down, and same as *Start-Blnc*. Therefore, $op(dst) \rightarrow$ *Start-Blnc(dst)*. □

Lemma 4.4 If $op \propto PT_4$, then $op(dst) \rightarrow$*Ack*(dst).

Proof: This is obvious. When dst receives *Ack*, it must have become *Lock-in* state, and no request is issued in this state. □

Lemma 4.5 If $op \propto PT_3$ or $op \propto PT_6$, the state of node in ST has no influence on the execution of op.

Proof: op must have arrived at lock-initiator before lock-initiator becomes *Lock-St*, and then lock-initiator just pass the op outside ST. This is true since at any node k, $op(k) \rightarrow Ack$, and lock-initiator can become *Lock-St* only by receiving two *Acks*. □

Lemma 4.6 If $op \propto PT_2$, then $op(dst) \rightarrow$ *Lock-Inten*(dst) or $op(dst) \rightarrow$ *Start-Blnc*(dst).

Proof: If op originates from a node in ST or op has already arrived at lock-initiator before it decides to begin to balance, then $op(dst) \rightarrow$ *Lock-In*(dst). If op is a request received by lock-initiator during *Lock-In* state(op can only be *Find* and can only go down), then $op(dst) \rightarrow$ *Start-Blnc*(dst). □

Lemma 4.7 If $op \propto PT_5$, then $op(dst) \rightarrow$ *Start-Blnc*(dst).

Proof: At any node k in ST, since $op(k) \rightarrow Ack(k)$, op must have reached the highest point h before h receives *Ack* (which may cause h enter *Lock-St*). And on the down road of op, $op(k) \rightarrow$ *Start-Blnc*(k). □

Lemma 4.8 Lock-initiator is the last node to enter *Lock-St*, and it has the most up-to-date information about its subtrees.

Proof: After lock-initiator sends *Lock-Intens* and becomes *Lock-In* state, it collects *Acks* from its children. Any *Update* message and *Insert* must be prior to *Ack*, thus, when a node receives *Ack* from a subtree, it must have received all *Update* and *Insert* messages from this subtree before. □

Lemma 4.9 In *Lock-In* state, if node i receives *Children-Full*, i or i's ancestor must have already decided to begin to balance.

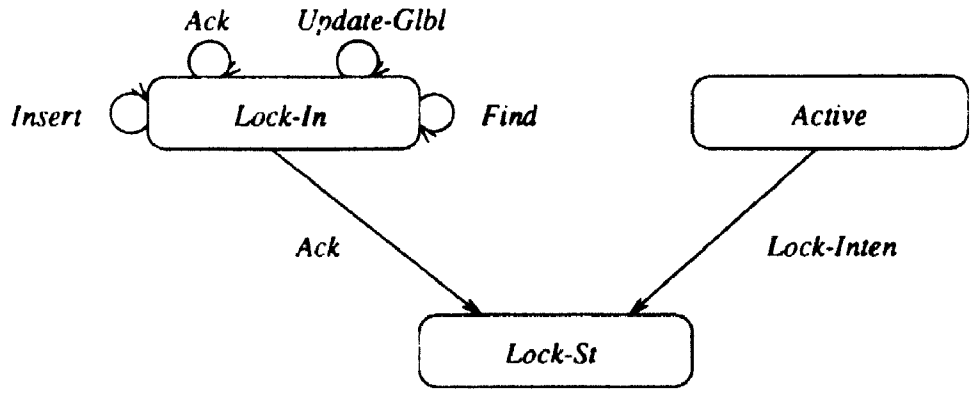


Figure 4.5: State transmission from state *Lock-In* to state *Lock-St*

Proof: This is obvious. Since a node can become *Lock-in* state only when itself decides to start to balance or it receives *Lock-Inten* from its ancestor. □

Lemma 4.10 The execution of request operations in *Lock-In* state satisfies order preserving.

Proof: From Lemma 4.3 to Lemma 4.9. □

Theorem 4.4 A holds in $S_{Lock-In}$ state.

Proof: From previous Lemma 4.3 to 4.10. □

Theorem 4.5 $S_{Lock-In} \overset{*}{\vdash} S_{Lock-St}$, i.e. *Lock-St* state is reachable from *Lock-In* state.

Proof: We divide the proof into two parts: first, a leaf node sends *Ack* to its parent by receiving *Lock-Inten*, and becomes *Lock-St*; second, all descendants of lock-initiator will become *Lock-St*.

1. For a leaf node k in state *Active*, when it receives *Lock-Inten*, it realizes that one of its ancestors (i.e. lock-initiator) has decided to start to balance. It sends acknowledgment

Ack to its parent, closes link from self, and itself becomes state *Lock-St*. i.e.,

$$\exists seq(p_k) = +m_0(-m_1)$$

such that

$$m_0 = \textbf{Lock-Inten}, m_1 = Ack$$

$$history'(p_k) = history(p_k) \cup \{receives\ m_0\}$$

therefore

$$Next(< p_k, history'(p_k) >, seq(p_k)) = q_k = \textbf{Lock-St}$$

2. For a internal node, when it receives *Ack*, it must have received *Lock-Inten* before and must have become state *Lock-In*. When it receives two *Acks* from its children, it sends *Ack* to its parent and itself becomes state *Lock-St*. By induction, when lock-initiator receives two *Acks*, all of its descendants must have become *Lock-St*, and itself becomes *Lock-St*. □

Theorem 4.6 $S_{Lock-St} \overset{*}{\vdash} S_{Active}$, i.e. *Active* state is reachable from *Lock-St* state.

Proof: A node in *Lock-St* has four cases to become *Active*, those are:

1. When a lock-initiator receives two *Acks*, it has the most up-to-date information about its descendants (since *Update-Glbl* $\rightarrow$ *Ack*), this information is different from that in the instant when it decides to send *Lock-Inten* to its children. It is possible that the subtree *ST* is balanced again without the need of data movement. In this case, it is unnecessary to balance. The lock-initiator sends *No-Shift* to its children, and

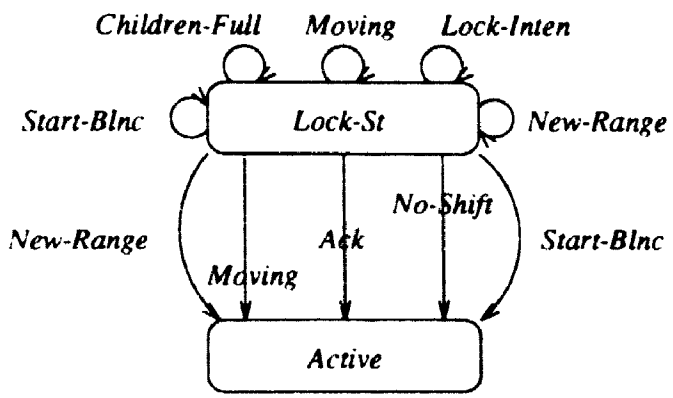


Figure 4.6: State transmission from state *Lock-St* to state *Active*

becomes *Active* again, i.e.

$$\exists seq(p_k) = +m_0(-m_1)$$

such that

$$m_0 = Ack, m_1 = \textit{No-shift}$$

$$history'(p_k) = history(p_k) \cup \{receives\ m_0\}$$

$$balance(history'(p_k)) = true, f(history'(p_k) = true$$

therefore

$$Next(< p_k, history'(p_k) >, seq(p_k)) = q_k = Active$$

When a node receives *No-shift*, it knows its parent has decided not to balance. It continues to propagate this *No-shift* message to its children, and itself becomes *Active*. When all the leaf nodes has received *No-Shift*, the subtree must have reached S_{Active}.

2. For a leaf node i, if *Start-Blnc* comes first, i sends *Moving* to the corresponding adjacent node. If i does not expect the moving-in data, i.e., $in(i) = 0$, then it sends *New-Range* to its parent and itself becomes *Active*.

3. For a leaf node i, if *Moving* comes first, i sends *Moving* to the corresponding adjacent node, then sends *New-Range* to its parent and itself becomes *Active*.

4. For a internal node i, when it has received two *New-Range*s, it continues to propagate *New-Range* to its parent, and itself becomes *Active*. When the lock-initiator r has received two *New-Range*, all of its descendants must have become *Active*. r updates its *MyRange* and becomes *Active*. □

Property 4.4 i) Each message from parent of lock-initiator is blocked in *Lock-St* state; ii) a process begins to move data only when it receives *Start-Blnc*; iii) a process changes state from *Lock-St* to *Active* only if it gets information about new-range from two subtrees.

Lemma 4.11 In *Lock-St*, when a node enters *Active* state, it has the most up-to-date information about itself and its subtree.

Theorem 4.7 A holds in $S_{Lock-St}$ state.

Proof: No request operation is executed during this stage, and data is moved in such a way that the data consistency is kept. □

Chapter 5

Conclusions

5.1 Redundant Distributed Structure RBTS

In all discussion so far, we have assumed that the distributed data structure *BTS* is compact, i.e., the data items stored in the structure have one and only one copy. In the real world, however, an essential requirement from a practical distributed data structure is that it supports fault-tolerance. In this section, we discuss some issues involved in fault tolerance for *BTS*.

5.1.1 Issues Concerning Fault-Tolerance Models

Incorporating fault-tolerance is the nature of distributed systems, and fault-tolerance is usually enforced through replication or redundancy. That is, through multiple copies of data items. This introduces the strict consistency for each items. That means all copies of a data item should have the same value, ensuring that answers to user queries have a

consistent value across the distributed system.

The schemes that use redundancy to support fault-tolerance can be classified into three categories: the *primary-stand-by* approach, the *modular redundancy* approach, and the *weighted voting* approach.

The primary-stand-by approach selects one copy from the back-ups and designates it as primary, whereas the others are stand-bys. Then all subsequent requests are sent to the primary copy only. The stand-by copies are not responsive for the service, and they only synchronize with the primary copy periodically. In the case of a failure, one of the stand-by copies will be selected as new primary one, and the service goes on from the point synchronized most recently.

The modular redundancy approach makes no distinction between the primary copy and stand-by ones. Requests are sent to all the back-ups simultaneously, and service is performed by all the copies. Therefore, it is fault tolerant provided that there exists at least one correct copy. In contrast to the primary-stand-by approach, the service continues instantaneously after the fault occurs, but it is costly to maintain the synchronization between the duplicated items, especially when there are many of them.

The weighted voting approach is much complex and we are not going to discuss here.

5.1.2 Redundant Structure RBTS

The scheme we use for the redundancy is something between primary-stand-by and modular. We allow each site to have three same size bins: one is called *Master-bin*, the other two are called *Left-bin*, *Right-bin*, respectively. The function of *Master-bin*s is the same as

before. Any operation instructions will refer only to *Master-bins*, *Left-bin* and *Right-bin* in site i which are used to store the copy of data in i's left adjacent bin and right adjacent bin, respectively.

Thus, the access protocol is modified as follows: when a data item key is inserted at site i in its *Master-bin*, the control message *Insert-Backup*(key) is also forwarded to i's adjacent site $i-1$ (if $i > 1$) and site $i+1$ (if $i < N$). Notice that this activity can actually be postponed, or deferred, over a long period of time, and can be performed in the background. This demonstrates that no extra cost is necessary to maintain the redundant structure except the storage requirement.

The performance of *RBTS* can be improved. In *BTS*, to restore the balance, data items are moved among adjacent bins. Since in site i, it already has its adjacent bin's copies of data, it is only necessary to copy from *Left-bin* or *Right-bin* to *Master-bin* locally without occurring communications. The actual data movement from adjacent *Master-bin* to local *Left-bin* or *Right-bin* can be done in the background.

The user query and modification instructions can also be improved. Suppose each *Left-bin* or *Right-bin* also stores the range of data in that bin. When *Find*(key) arrives at site i, if the key does not belong to local *Master-bin*, the *Left-bin* and *Right-bin* is checked. If data belongs to the data range of *Left-bin* or *Right-bin*, the message will be directly forwarded to the corresponding site.

5.2 Evaluation

We evaluated our protocols mainly from several aspects: load balance, concurrency, efficiency, correctness and fault tolerance. Data load among network is balanced, and storage requirement is lowest. For the *BTS*, the structure is compact, the only extra storage needed is for the information about current site's subtree, that is much less comparing with *DIST-BIN*, in which each site needs to "remember" every other's information.

Find becomes an atomic part of *Insert* and *Delete* This guarantees that any modification operation is a safe operation.

In our protocols, modification operations can be issued concurrently with no need of synchronizing each other; and in normal state, their executions can be proceeded to the end without being blocked.

When a part of network is not balanced, the protocols can detect its unbalance, and automatically start to restore the balance regarding to only necessary subtree without affecting the rest part of network. This has been achieved by allowing from outside of subtree, query operation able to get through the subtree while blocking modification operations at lock-initiator (the root of the subtree).

The concurrency is handled in an appropriate way such that the data items stored in *BTS* are consistent.

We have taken advantages of special characteristics of tree structure. In particular, each site "remembers" the information about its subtree and *control* message is used to update this information. In this way, the maximum number of messages for a query or modification operation travelling from its source to its destination is $2 * (log(n+1) - 1)$ (i.e. two times of

the height of the tree). When there is an execution at some site, a *control* message is sent to its parent which in turn continues propagate this message to the root. It requires maximum $log(n+1)-1$ messages which does not delay the response time to the modification request.

We have been focusing on the validation of *BTS*. We defined several classes of global states. For each class of predefined global states, we have proved that it is reachable; in each such kind of global state, the execution of request operation is order-preserving. Our model of protocols have employed a simple and commonly used representation of the communication process, namely, each process is a finite-state machine. Each pair of communicating process is connected by a FIFO channel. Our approach of validation has two advantages: first, we do not have to enumerate all possible executions of system behavior (that is impossible), thus we can focus on several classes of global states that are reachable, and observe the behavior in such global states; second, for a given received message, current process can become different class of global state based on the knowledge that it has ever received other messages before. We introduced the concept *history*(p_i) which records process i having received some kinds of messages before during state p_i. Therefore, it is always determinable by the current state and its *history* which class of global state process i should enter when receiving a message. This method will be useful for the validation of some complex protocols which have too many global states to consider. *RBTS* has the characteristics of dealing with fault tolerance.

5.3 Summary

Distributed data structures play an important role in the distributed systems. In this thesis, we have discussed the motivation, formal model, correctness, and simulation of an efficient data structure *BTS*. To denote a view of a distributed system, a general model is chosen. In this model, processors distributed among network communicate through message passing, i.e., event-driven. The asynchronous processors can perform local computation, send message to neighbours or change the current state according to the current state and event. It is assumed that failure is absent in the *BTS*.

During the first stage of design, a concept called bin is introduced. Each site has a local storage bin which is used to store the data items in the current site. Based on the given computational model, a formal description of system behaviors is presented, which forms the basis of the discussion of verification. Dictionary operations *Find*, *Insert* and *Delete* are defined, and these operations are supported by the proposed *BTS*.

When we designed the distributed data structure, our main concern was the communication cost. At the same time, we try to maintain the load balance of data among the network. Thus, balance function is defined which is the most important mechanism affecting system performance.

Distributed applications have various requirements for the distributed data structure. We identified a kind of application, and proposed the design goal. The most difficult part we found is the consistency during balance stage. Thus, some policies such as *Transfer* policy, *Location* policy, *Balance control*, *Information gathering mechanism* and *concurrency control* are carefully described. Based on these policies, we described *how to do* for

each query or modification request. During process of modification request, load balance should be considered. we allow the data movement as long as the current site has enough information and data is available.

When designing a distributed algorithm, it is essential that the algorithm be correct. We first described what the protocol does in each pre-defined state. Then we gave the partial correctness, i.e., the protocol does nothing wrong during run time. Finally, we introduced *RBTS* to incorporate fau tolerance.

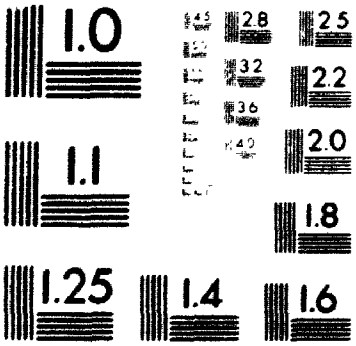
5.4 Discussions and Future Work

In this thesis, we were only concerned with dictionary operations. In fact, there are some other operations required from a distributed structure, such as *Modify*(), *range searches*, *batch insertion*, and *batch deletion*. The requirement of *Modify*() is slightly more involved than *Find*() but less stringent than *Insert*() or *Delete*(). Since *BTS* only uses the keys of data items to organize the distributed data structure, the protocol for *Modify*() can be exactly the same as *Find*() except that when the request arrives at its destination t, site t changes the contents of records with regarding key, instead of returning the found record. *BTS* can also be extended to support *range searches*, *batch insertion*, and *batch deletion*.

BTS structure proposed in this thesis is designed to work on network whose topology is a logical balance tree. Thus, it is much suitable to a simple bus architecture and a hypercube. For an arbitrary graph, the first stage is to construct a suitable logical tree, and then to embed it into the network. Some work has been done by Gilon and Peleg, refer to [GP91] for detail. In *BTS*, we used binary tree as the frame of network. Notice that the

2 OF/DE 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT



height of the tree has main effect of the communication cost for each request operation. It is possible to use B tree or its invariant instead of binary tree. These and related work are left for the future work.

Appendix A

BTS Protocols

BTS has been simulated based on"DC-PARASOL", a C-based system using the PARASOL distributed systems simulator. It was developed by Robert Black [Bla92]. It can be used to simulate a distributed algorithm with a minimum of effort. The model of distributed computation is that a distributed system which is a collection of N processing nodes interconnected by communication links. Nodes are equipped with local non-shared memory and clock, and can communicate with neighbouring nodes by sending and receiving messages.

We start with defining the topology of network as complete network. The reason, why the complete but not binary topology is used, is that a node needs not only communicate to its neighbours, but also to its adjacent nodes (during data movement to restore the balance). As described in Chapter 4, there can be 5 states during run time. In the following, we present the implementation issues for each state.

A.1 State *Initial*

Since we assume, initially, the system is balanced, and with proper knowledge, we must guarantee the system is set up properly before any node enters other states. There are two kinds of messages: *Spontaneous* and *Subtree-Range*. When receiving *Spontaneous* message, a site initializes itself. For the leaf node, after initialized with data, it sends *Subtree-Range* to its Parent, and become *Active*. For the internal node, after it receives two *Subtree* messages, it sends *Subtree-Range* with its new range up, and becomes *Active*. After root receives two *Subtree-Range* messages, it becomes *Active* and initialization has been finished.

On Reception of *Spontaneous*

```
{
    if not initialized
      initialize;
      if (not root) and (has not sent Subtree-Range)
        send Subtree-Range to Parent;
        become(Active);
      fi
    fi
}
```

On Reception of *Subtree-Range*

```
{
```

```
    if (not initialized)
        initialize;
    fi
    if (counter < 2)
        update-new-range;
        counter = counter + 1;
    fi
    if (counter = 2) and (not root)
        send Subtree-Range to Par. nt;
        become(Active);
    fi
}
```

Note: Probability = 1 guarantees that every site will get *Spontaneous* message before all the nodes become *Active*.

A.2 State *Active*

To simulate system occasionally generates some instructions, we use *Spontaneous* message to randomly generate *Find*(key, r) or *Insert*(key, r).

When receiving *Find*(key, r), *Insert*(key, r) , it will be processed locally, go up, or go down according to the transfer policy and location policy. *Find*(key, r) or *Insert*(key, r) will finally arrive the appropriate site i in which the data belongs to its local range. Find

will not change the system status while Insertion may cause local full, global unbalance, etc..

On Reception of *Find*(*key*, *r*) /* request from site *r* */

```
{
    condition = check-condition( key, r );
    switch (condition) {
            case in local:
                return result;
            case not in subtree:
                send Find to Parent;
                break;
            case is-leaf:
                return ("not found"); /* return to request site r */
            case in left :
                send Find to Left;
                break;
            case in right:
                send Find  to Right;
                break;
    }
}
```

On Reception of *Insert*(*key*, *r*)

```
{
    condition = check-condition(key, r);
    switch (condition) {
            case in local:
                if (full)
                  close-link-from(i); /* block the incoming message */
                  send Insert to i;
                  become(Lock-In);
                else if (not exists)
                      store locally;
                      send Update-Glbl to Parent;
                    else
                      return("Error"); /* for the safety insertion */
                    fi
                fi
                break;
            case not in subtree:
                send Insert to Parent;
                break;
            case is-leaf:
                store-data;
                break;
```

```
            case in the left:
                send Insert to Left
                break;
            case in the right:
                send Insert to Right
                break;
    }
}
```

On Reception of *Update-Glbl*

```
{
    do-update;
    if (trigger threshold)
      send Lock-Inten to Left, Right;
      if (not root)
        send Update-Glbl to Parent;
      fi
      close-link-from(i);
      become(Lock-In);
    else if (not root)
          send Update-Glbl to Parent;
        fi
    fi
```

```
}

On Reception of Lock-Inten
{
    if (leaf)
        send Ack to Parent;
        close-link-from(i);
        become(Lock-St);
    else
        send Lock-Inten to Left, Right;
        close-link-from(i);
        become(Lock-In);
    fi
}

On Reception of Children-Full
{
    send Lock-Inten to Left, Right;
    close-link-from(i);
    become(Lock-In);
}
```

A.3 State *Lock-In*

In this state, each internal node sends *Lock-Intens* to their children. When a leaf receives a *Lock-Inten*, it sends to *Ack* to its Parent and itself becomes *Lock-St*. After an internal node receives two *Acks*, it sends *Ack* to its parent and becomes *Lock-St*. When lock initiator receives two *Acks*, it sends *Start-Blnc* to its children and becomes *Lock-St*.

On Reception of *Ack*

```
{
    num-ack = num-ack + 1;
    if (num-ack = 2)
      if (lock-initiator)
        if (trigger threshold)
          send Start-Blnc to Left, Right;
          do data movement when available;
          become(Lock-St);
        else
          send No-Shift to Left, Right;
          open-link-from (i`:
          become(Active);
        fi
      else
        send Ack to Parent;
```

```
        become(Lock-St);
      fi
    fi
}
```

On Reception of *Update-Glbl*

```
{
    do-update;
    if (not root)
      send Update-Glbl to Parent;
    fi
}
```

On Reception of *Spontaneous*

```
{
    send Spontaneous to (i);
}
```

On Reception of *Find*

```
{
    same as in Active state
}
```

On Reception of *Insert*

```
{
    same as in Active state
}
```

A.4 State *Lock-St*

We use variables: owe-in, owe-out to indicate which comes first, *Start-Blnc* or *Moving* and the current site status.

Initially, owe-in = -1, owe-out = 0. There are two kinds of buffer: local buffer 'buf', and link buffer 'linkbuf'. When the data items are to move out, they are put in linkbuf; when data items are relieved, they are put in local buf.

On Reception of *Start-Blnc*

```
{
    calculate (in, out);
    if (not leaf)
      send Start-Blnc(lock-init, in, out) to Left, Right;
    if owe-in = 0; /* means the moving data has come before*/
      if (self has enough)
        self → linkbuf(out, lock-init);
        buf → self(out, lock-init);
      fi
      if  (self not enough)
```

```
        self → linkbuf;
        buf → linkbuf;
        buf-left → self.
    fi
    move-ptr = linkbuf;
    if (out ≠ 0)
      range = GetRange();
      if (left-bound(lock-init))
        send Moving(move-ptr, out, range) to LeftAdj;
      else
        send Moving(move-ptr, out, range) to RightAdj;
      fi
      change status: owe-in = -1; (everything has been done)
      owe-out = 0; (not owe any more)
    fi
  if owe-in = -1; /* means data has not come yet */
    if (self has enough)
    move-ptr = linkbuf;
      self → linkbuf;
      if (out ≠ 0)
        range = GetRange();
        if (left-bound(lock-init))
```

```
        send Moving( move-ptr, out, range) to LeftAdj;
      else
        send Moving( move-ptr, out, range) to RightAdj
      fi
    fi
    if (in = 0)
      reset status: owe-in = -1;
      owe-out = 0;
      num-ack = 0;
      if (leaf)
      NewLeafRange(lower, upper, subtotal);
      send New-Range(lock-init, i, lower, upper, subtotal) to Parent;
      open-link-from(i);
      become(Active);
      fi
    else
        owe-in = 1; /* wait for the moving in data */
        owe-out = 0; /* own data has been moved out */
      fi
    fi
  fi
  if (self not enough)
```

```
        change status: owe-in = 1;
        owe-out = 0; /* not move out now */
      fi
    fi
}
```

On Reception of *Moving*(move-ptr, out, range)

```
{
    GetData(move-ptr, out);
    ChangeMyRange(lock-init, range);
    if (owe-in = -1); /* Start-Blnc not come yet */
      owe-in = 0; /* indicates Moving comes first */
    else
      if (owe-out = 0)
        buf → self;
      else
        buf → linkbuf;
        self → linkbuf;
      fi
      move-ptr = linkbuf;
      if (owe-out ≠ 0)
        if (left-bound(lock-init))
          send Moving(move-ptr, out, range) to LeftAdj;
```

```
        else
          send Moving(move-ptr, out, range) to RightAdj;
        fi
        owe-in = -1; /* everything has been done */
        owe-out = 0;
        num-ack = 0;
        if (leaf)
          NewLeafRange(lower, upper, subtotal);
          send New-Range(lock-init, i, lower, upper, subtotal) to Parent;
          open-link-from(i);
          become(Active);
        fi
      fi
    fi
}

On Reception of New-Range(new-range-id, lower, upper, subtotal)
{
    ChangeSubRange(lower, upper);
    ChangeSubNum(new-range-id, subtotal);
    rg-ack = rg-ack + 1;
    if (rg-ack = 2)
      if ( i ≠ lock-init)
```

```
        send New-Range(lock-init, i, lower, upper, subtotal) to Parent;
        rg-ack = 0;
        open-link-from(i);
        become(Active);
      fi
    fi
}

On Reception of No-Shift
{
    if (not leaf)
      send No-Shift to Left, Right;
    fi
    open-link-from(i);
    become(Active);
}

On Reception of Lock-Inten(lock-init)
{
    send Lock-Inten(lock-init) to (i); /* block it */
} /*———————— End ————————*/
```

References

[AAB88] M. Ahamad, M.H. Ammar, J.M.Bernabeu-Auban and M.Yousef Khalidi, "Using Multicast Communication to Locate Resources in a LAN-Based Distributed System", *Proc. of IEEE 13th Conf. on Local Computer Networks*, pp. 193-202, Oct. 1988.

[ABLP89] B. Awerbuch, A. Bar-Noy, N. Linial and D. Peleg, "Compact Distributed Data Structures for Adaptive Routing", *Proc. 21st ACM Symp. on Theory of Computing* pp. 479-489, May 1989.

[ADD82] G.R. Andrews, D.P. Dobkin and P.J. Downey, "Distributed Allocation with Pools of Servers", *Proc. 2th ann. ACM Symp. on the Principle of Distributed Computing(1982)*, pp. 73-83, 1982.

[Ber85] A.J. Bernstein, "A Loosely Coupled Distributed System for Reliably Storing Data", *IEEE Transactions on Software Engineering*, vol. SE-11, pp. 446-454, Nov. 1985.

[BFB91] C. Barmon, M.N. Faruqui and G.P. Battacharjee, "Dynamic Load Balancing Algorithm in A Distributed System", *Microprocessing and Microprogramming 29*, pp. 237-285, 1990/91.

[BG81] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems", *Computing Surveys*, vol.13, no.2, June 1981.

[Bla92] R. Black, "The DC/P Simulator: A System for Running DC Program under PARASOL", a user's guide, School of Computer Science, Carleton University, Canada, March 1992.

[CK90] T.L. Casavant and J.G. Kuhl, "A Communicating Finite Automata Approach to Modeling Distributed Computation and Its Application to Distributed Decision-Making", *IEEE Transactions On Computers*, vol. 39, no. 5, pp. 628-639, May 1990.

[CK87] T.L. Casavant and J.G. Kuhl, "Analysis of Three Dynamic Distributed Load-Balancing Strategies with Varying Global Information Requirements", *Proc. 7th Int. Conf. on Distributed Comput. Syst.*, pp. 185-192, Sept. 1987.

[CL85] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Transactions on Computer Systems*, vol. 3, no. 1, pp. 63-75, Feb. 1985.

[CS91] H. Cheung and J. Sheu, "Design and Implementation of a Distributed File System", *Software-Practice and Experience*, vol. 21(7), pp.657-675, July 1991.

[Ell85] C.S. ELLis, "Distributed Data Structures: A Case Study", *IEEE Transactions on Computers*, vol. c-34, no. 12, pp. 1178-1185, Dec. 1985.

[FC91] Zhenqiang Fan and Kam-Hoi Cheng, "A Generalized Simultaneous Access Dictionary Machine", *IEEE Transaction on Parallel and Distributed Systems*, vol.2, No.2, April 1991.

[GG89] B. Gliss and J. Gorski, "Specification and Implementation of Concurrency in A Simple Database System", *Proc. IFAC/IFIP Workshop*, pp. 16-18, Oct. 1989.

[GP91] K. Gilon (Ziegler) and D.Peleg, "Compact Deterministic Distributed Dictionary" (extended Abstract), *Proc. 10th ann. ACM Symp. on the Principle of Distributed Computing(1991)*, pp.81-94, 1991.

[JNSM90] X. Jia, H. Nakano, K.Shimizu and M.Maekawa, "Highly Concurrent Directory Management in the GALAXY Distributed System", *IEEE 10th Int. Conf. on Distributed Comput. Syst.*, pp. 416-423, 1990.

[JRW90] S. Jablonski, T. Ruf and H. Wedekind, "Implementation of a Distributed Data Management System for Technical Applications—a Feasibility Study", *Information Systems*, vol. 15, no. 2, pp. 247-256, 1990.

[Lam78] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", *Communications of the ACM*, vol.21, pp.558-565, July 1978.

[Lam82] L. Lamport, "An Assersional Correctness of A Distributed Algorithm", *Science of Computer Programming* 2, pp. 175-206, 1982.

[LGFG86] Nancy A. Lynch, Nancy D. Griffeth, Michael J. Fischer and Leonidas J. Guibas "Probabilistic Analysis of a Network Resource Allocation Algorithm", *Information and Control 68*, pp.47-85, 1986.

[LL90] L. Lamport and N. Lynch, "Distributed Computing: Models and Methods", *Handbook of Theoretical Computer Science*, Elsevier Publishers, B.V., 1990.

[LP90] Hon F. Li, David K. Probst, "Optimal VLSI Dictionary Machines Without Compress Instructions", *IEEE Transactions on Computers*, vol.39, no.5, pp.676-693, 1990.

[LP91] A. Leff and P.S. Yu, "An Adaptive Strategy for Load Sharing in Distributed Database Environments with Information Lags" , *Journal of Parallel and Distributed Computing* 13, pp. 91-103, 1991.

[LY87] T.H. Lai and T.H. Yang, "On Distributed Snapshots", *Information processing Letters* 25, pp. 153-158, 1987.

[Man84] U. Manber, "Concurrent Maintenance of Binary Search Trees", *Transaction on Software Engineering*, vol. se-10, no. 6, Nov. 1984.

[Man86] U. Manber, " On Maintaining Dynamic Information in a Concurrent Environment" , *SIAM Journal. on Comput.*, 15:1130-1142, 1986.

[ML84] U. Manber and R. E. Ladner, "Concurrency Control in a Dynamic Search Structure", *ACM Transaction on Database Systems*, vol.9, no. 3, pp. 439-455, Sept. 1984.

[MS90] G. Matsliach and O. Shmueli. "Distributing a B^+-Tree in a Loosely Coupled Environment", *Information Processing Letters 34*, pp. 313-321, May 28, 1990.

[Pel89] D. Peleg, "Complexity Considerations for Distributed Data Structures", *Technical Report CS89-31 of The Weizmann Institute of Science*, Israel, Dec. 1989.

[Pet88] J.G. Peters, "Distributed C: An Environment for Designing and Testing Algorithms for Distributed Systems—Preliminary System Specifications", *Technical Report*, CSS/LCCR TR 88-27, Simon Fraser University, Canada, Nov. 1988.

[PLNZ90] C.B. Pinkerton, E.d. Lazowska, D. Notkin, and J. Zahorjan, "A Heterogeneous Distributed File System", *The 10th International conference on Distributed Computing Systems*, 1990.

[PW91] F.J. Prusker, "The Siphon: Managing Distant Replicated Repository", *Technical Report*, DEC Paris Research Laboratory, May, 1991.

[Ray88a] M.Raynal, "Distributed Algorithms and Protocols", *John Wiley and Sons Ltd.*, 1988.

[Ray88b] M.Raynal, "Network and Distributed Computation", *MIT Press*, 1988.

[SEDB91] M. Schaar, K. Efe, L. Decambre and L.N. Bhuyan, "Load Balancing with Network Cooperation", *IEEE 11th Int. Conf. on Distributed Comput. Syst.*, pp. 328-335, 1991.

[SK85] N.Santoro and R. Khatib, "Labelling and Implicit Routing in Networks". The Computer Journal, vol. 28, no. 1, pp. 5-8, 1985.

[TB90] P. Triantafillou and M. Bauer, "Distributed Name Management in Internet Systems:A Study of Design and Performance Issues" , *Journal of Parallel and Distributed Computing* 9, pp. 357-368, 1990.

[Tel91] G.Tel, "Topics in Distributed Algorithms" , *Cambridge University Press*, 1991.

[Ver87] J.P. Verjus, "On the Proof of a Distributed Algorithm", *Information Processing Letters* 25, pp. 145-147, 1987.

[Wil86] D.E. Willard, "Good Worst-case Algorithms for Inserting and Deleting Records in Dense Sequential Files" , *ACM on SIGMOD*, pp. 251-260, 1986.

END

15-12-93

FIN