EFFICIENT TRANSACTION PROCESSING IN BROADCAST-BASED ASYMMETRIC COMMUNICATION ENVIRONMENTS

By

YAN HUANG

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2001

To my family

ACKNOWLEDGMENTS

First and foremost, I would like to express my deep gratitude to my advisor, Dr. Yann-Hang Lee, for giving me the opportunity to work with him and to study under his guidance. I want to thank him for giving me continuous and excellent guidance and advice during my years of Ph.D. study. I also want to sincerely thank him for being so kind in his care for me and his other students. His strong support and efforts to make everything easier in our lives are unforgettable. My full appreciation also goes to my committee members, Dr. Randy Chow, Dr. Paul Chun, Dr. Richard Newman, and Dr. Beverly Sanders, for their valuable and constructive advice.

I also thank all of the graduate students in our Real-Time Systems Research Lab: Vlatko Milosevski, Ohekee Goh, Daeyoung Kim, Yoonmee Doh, and Youngjoon Byun. I thank my good friends Zecong, Weiying, Youzhong, and Hongen for their friendship and encouragement while working and studying together.

I also want to express my gratitude to Dr. Bob O'Dea, Dr. Jian Huang, and the members of the Florida Communication Research Labs of Motorola Labs for giving me an opportunity to visit their lab and finish my degree in such a strong research atmosphere.

Finally, I want to give my special thanks to my dear mother, Xiaojun Zhang; my father, Jingcheng Huang; and my husband, Ji Xiao. Without their strong support, care and encouragement, I could never have completed this work. I also thank my parents-in-law for encouraging me and caring for me as I wrote this dissertation.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

EFFICIENT TRANSACTION PROCESSING IN BROADCAST-BASED ASYMMETRIC COMMUNICATION ENVIRONMENTS

By

Yan Huang

December 2001

Chairman: Dr. Yann-Hang Lee
Major Department: Computer and Information Science and Engineering

Asymmetric communication based on data broadcasting is an appropriate model for Internet and wireless computing. Research problems in such an environment have drawn much attention in the past few years. While many have been addressed, unresolved issues exist and efficient solutions are needed.

In this dissertation, we introduce the concept of broadcast-based asymmetric communication. We survey current research in this type of environment, including broadcast scheduling, cache strategies, indexing, concurrency control, fault tolerance, and real-time scheduling. Then we report the research efforts on several topics applicable to the broadcast environment. Compared with the existing research, which mainly considers individual data operations, the purpose of our research is to support efficient multi-operation transaction processing in broadcast-based asymmetric communication environments. The researched topics focus on efficient transaction concurrency control protocols, transaction processing in hierarchical broadcasting models, and performance

issues for transaction processing. New algorithms and improvements to existing algorithms are sought in these topics. We also describe the design and implementation of a simulation framework that can be used to evaluate the performance measures of the proposed protocols and environment. The simulation results show very successful performance of our solutions. These results and efficient solutions can contribute to various broadcast-based transaction processing applications.

CHAPTER 1
INTRODUCTION

1.1 Broadcast-based Asymmetric Communication Environment

1.1.1 Asymmetric Communication

The advent of wireless and Internet computing introduced a large number of applications that run in an asymmetric communication environment. Communication asymmetry implies that bandwidth and traffic are unbalanced among communication links in an application environment. As an example, in a wireless application, the downstream bandwidth from server to clients can be much larger than the upstream bandwidth from clients to server. Some wired Internet service methods, such as ADSL and DirectPC [STA97], also have the property of different downstream and upstream bandwidths.

Communication asymmetry can be caused by the physical communication medium and can also be caused by the difference arising from the amount of data supplied by a server and the volume of client requests. For example, many Internet applications have large populations of clients requesting a limited set of information from a small number of servers. Thus, data sets in high demand will have a large number of simultaneous requests. In such an application, if the servers send a reply to each client individually, it will cause two performance problems. The first problem occurs when the client requests exceed an acceptable number, resulting in a high probability of server overload. The second problem is the high network traffic that wastes a large portion of

bandwidth by repeatedly sending the same data to different clients. Based on the observation that one server downstream reply can satisfy a large number of simultaneous upstream requests for the same data, we describe this type of application environment as asymmetric. That is, the amount of information provided is asymmetric to the volume of requests for it.

1.1.2 Broadcast-based Asymmetric Communication

The problem of unbalanced downstream/upstream link bandwidths and information/request volumes of asymmetric communication motivated us to adopt data broadcasting as a method of information communication in such an environment. In data broadcasting, the server sends all information sequentially and repeatedly to all clients at the same time through the downstream channel. The clients do not send requests to the server. When a client needs data, it monitors the broadcast channel until those data go by. The upstream link is used only in applications that allow clients to perform updates on the server side. Although this broadcasting data model can be used in both wireless and wired network environments, we call it on-air broadcast because of its similarity to radio or television broadcasting (e.g. news, weather channels etc).

Data broadcasting is an appropriate model for an asymmetric communication environment. For applications that have asymmetry between downstream and upstream link bandwidth, especially for wireless applications that use very slow cellar upstream links, the upstream channel can be freed from sending requests and downloading information. All needed data will be available on the high bandwidth downstream link. More importantly, a single broadcast of each data unit can meet the requirements of many simultaneous client requests at the same time. For applications that have an imbalance between information and requests, the data broadcast model eliminates all simultaneous

requests from the clients and the repeated replies from the server. As a result, network traffic and bandwidth consumption are greatly reduced. Moreover, data broadcasting improves the scalability of applications. In a model where clients send requests then download data separately, the server will be overloaded if the number of requests is too large. By broadcasting data, the size of the client population no longer matters.

We call an application environment that has the property of communication asymmetry and uses data broadcasting as its communication model a “broadcast-based asymmetric communication environment.” As many new applications are launched in wireless and Internet domains, efficient processing algorithms have been investigated for applications in such an environment. They are focused mainly on the design of broadcasting schedules, cache strategies to memorize data for faster data read, index schemes to reduce power consumption, concurrency control for broadcast transaction processing, fault tolerance, and real-time issues. In fact, most attention has been paid to accelerate individual data operations. For the few that address multi-operation transactions, performance is limited. Moreover, transaction processing on such new models as hierarchical broadcasting has not been taken into consideration.

The limited research on transaction processing in broadcast environments motivated us to explore efficient transaction processing schemes in a broadcast-based asymmetric communication environment. In this dissertation, we detail our motivation, solution, design, and performance results in four categories of transaction processing problems that we believe to be important in such an environment. All of these value a transaction’s performance over single operations, and offer the solutions to improve transaction processing efficiency.

1.1.3 Organization of the Dissertation

This dissertation is organized as follows. First, we discuss how to support broadcast operations in a wireless or Internet environment and examine potential applications. We also compare data broadcasting to other information communication models. In Chapter 2, we survey the existing research work in a broadcast-based asymmetric communication environment and introduce what is novel in this dissertation research. Chapters 3 to 6 explore the problems of an efficient transaction currency control protocol, the problems introduced by the proposed concurrency control protocol, transaction processing in hierarchical broadcasting models, other performance issues for broadcast transaction processing, and offer solutions to each. Chapter 7 discusses the implementation of the simulation framework for the performance evaluation of our solutions and designs. Finally, Chapter 8 summarizes this dissertation and gives an overview of possible applications based on the problems and solutions we studied.

1.1.4 Broadcast Support and Examples of Applications

In an asymmetric communication environment, data broadcasting can be supported in different ways. Satellite broadcasting can be used as a high-bandwidth downstream link to wireless or wired end users with receivers [DAO96, LEO98]. Moreover, a wireless service-provider's base station can also broadcast data to the users in its cell with much higher bandwidth than any upstream cellular link. Broadcast can also be implemented in the wired Internet. For example, a server can use a high-bandwidth downstream ADSL link to broadcast data to clients. Another example is the Hughes DirectPC, which downloads user-requested data through the user’s satellite receiver in response to the requests sent through a low-bandwidth telephone link. An application also can use a download satellite link to broadcast data without user requests.

Data can be broadcasted in an Ethernet and almost any other local area network. In a wide-area Internet separated by subnets, broadcast can also be emulated using similar technologies developed for multicast. This dissertation mainly focuses on research problems, algorithms, and protocols using an established data broadcasting model. Therefore, the problem of how data broadcast is implemented on the physical level is not our major concern.

News, traffic, weather, and stock-quote information-distribution systems are the most obvious examples of data broadcasting [ACH95b, ACH96b, ALM00, FRA96, FRA97, HUA99, PEN98, SUB99a, SUB99b]. These applications are popular in both wireless and Internet environments and have the property of communication asymmetry. Some less obvious applications are virtual classrooms, hospitals, insurance agents, airline ticket agents, and intra-company information distribution systems. Data broadcasting can also be used to broadcast real-time information, support real-time client data requirements, or distribute information in a hard real-time system. The applications supported can be single-data operation-based, or transaction-based. When the bandwidth is large enough, it has the potential to be used in any traditional transaction-based database systems, such as those for banks, or library management systems, even though these database applications do not have the property of communication asymmetry. When we explore problems in data broadcasting environments, we do not limit the possible applications for which it can be used. If the information communication model has enough bandwidth and is well designed, any kind of application [BOW92, HER87] can operate on it. However, our interest is in a model that is more efficient when used in an asymmetric communication environment.

1.2 Comparison to Other Communication Models

Many other information communication models, such as request/response and publish/subscribe, are widely used in wireless and Internet applications. All communication models, including data broadcasting, can be categorized into pull-based and push-based [ALT99, FRA96, FRA97]. In the pull-based communication model, the client always actively sends a request to the server to retrieve information. In the push-based communication model, the client is passive most of the time and data are pushed to it. The pull-based model has the advantage of knowing the user's interests, but it can overload the server and waste bandwidth by duplicating responses. The push-based model has the advantage of reducing the number of client requests sent to the server. However, it involves filtering of useful pushed data. These models can also be categorized as point-to-point and one-to-many. Point-to-point means data are communicated between one source and one other machine. One-to-many allows multiple machines to receive data sent by one data source. One-to-many has two types: multicast and broadcast. With multicast, data are sent to a particular subset of clients, while broadcast sends information to an unidentified and unbounded set of clients. Broadcast differs from multicast in that the clients who may receive the data are not known *a priori*.

By combining the two ways of categorizing communication described above, all communication models can be classified as pull-based, point-to-point; pull-based, one-to-many; push-based, point-to-point; and push-based, one-to-many. Data broadcast is classified as push-based one-to-many. Some existing models mapping to these classifications are shown in Table 1-1. The following subsections compare data broadcasting with some of the other models shown in Table 1-1.

Table 1-1 Classification of communication models

Pull-based		Push-based	
Point-to-point	One-to-many	Point-to-point	One-to-many
Request/response, periodic polling	Request/response or periodic polling with snooping	Triggers	Publish/subscribe, data broadcasting

1.2.1 Request/Response

Request/response is the most common information communication model used between data resource server and clients. It is an aperiodic, pull-based, point-to-point model. A client sends a request to a server for data whenever necessary and the server replies to each client separately. As discussed earlier, this model can overload servers and waste network bandwidth for applications with information/request asymmetry. For wireless applications, the request/response model can be extremely inefficient when upstream bandwidth is low and is used for both request and response. In comparison, data broadcast can overcome the difficulties a request/response model has in an asymmetric communication environment.

1.2.2 Periodic Polling

Some applications, such as remote sensing or controlling, pull data from a server periodically using another pull-based, point-to-point model called periodic polling. As the name suggests, in periodic polling, each request and its corresponding response are sent periodically. However, if an application with periodic requests for data is communication asymmetric, data broadcasting is more efficient because it can provide data repeatedly and periodically on the channel and, at the same time, periodic requests can be eliminated during data broadcasting.

1.2.3 Request/Response or Periodic Polling with Snooping

When snooping is added to such pull-based, point-to-point models as request/response and periodic polling, they become a pull-based, one-to-many model. In

these models, although data are pulled by a single client, one-to-many can still be achieved when many other clients snoop on the communication link and read the pulled data. This model can also reduce network traffic by avoiding some simultaneous requests and duplicated responses. However, data broadcasting usually has a well-organized periodic and repeated broadcasting scheduling that considers overall client data requests. As a result, it can provide data matching client requests in a more efficient and organized way instead of random snooping. Again requests from clients are eliminated when data broadcasting is used.

1.2.4 Publish/Subscribe

While the push-based, point-to-point model can be realized using triggers, data pushing is mostly associated with one-to-many multicast or broadcast. Publish/subscribe is a popular push-based, one-to-many mechanism. Using publish/subscribe, clients subscribe to given classes of information by providing a set of expressions that describe the data of interest. These subscriptions form a profile that is, in fact, a continuous query. The server publishes information that is tagged in a way that allows data to be matched to the subscriptions of the users. When new data items are created or existing ones updated, the items are disseminated to the subscribers whose profiles match the items.

Compared with data broadcasting, the publish/subscribe model pushes data to a pre-identified subset of clients based on their data interests. Moreover, it is aperiodic since it only pushes data when new data or an update is available. Publish/subscribe is also a good solution for communication asymmetry. However, it is not as scalable as data broadcast because the latter does not need to maintain a client profile or classify information based on client subsets. Clients still need to pull from the server when they need new data that they have not subscribed to in the publish/subscribe model. In

contrast, in the data broadcasting model with its repeated and periodic broadcasting, clients never need to pull.

In conclusion, data broadcasting is a push-based, one-to-many information communication model that has many advantages over other types of models used in an asymmetric communication environment. Therefore, this communication model has attracted much research attention in the past few years. After the survey of these research efforts, this dissertation will detail our results and solutions to many problems of supporting efficient transaction processing using this model.

CHAPTER 2
RESEARCH BACKGROUND

The characteristics of data broadcasting and communication asymmetry in broadcast-based asymmetric communication present many new research problems that need to be solved when designing applications. First, because data are sequentially disseminated to clients instead of in response to client requests, a broadcast schedule needs to be designed where data appear on the schedule in a sequence that ensures the most requests can be satisfied in the shortest waiting time. To achieve the goal of short request waiting time, proper cache strategies should be designed for clients with caching ability to locally memorize some data already broadcast so that some requests for cached data can be served without waiting. To save battery power, when data broadcast is used in wireless applications with mobile equipment, energy saving is very desirable. This makes continuous on-air impossible. As a solution, index schemes are being researched that would wake up data receivers only when necessary. Concurrency control protocols are also studied in order to support transaction-processing applications. Moreover, fault tolerance is another research topic in broadcast-based asymmetric communication. Some approaches attempt to reduce broadcasting error and improve recovery efficiency. Finally, because research in data broadcast can also be used in real-time applications, research has been conducted using different types of real-time requirements.

In this chapter, we survey existing research conducted in broadcast-based asymmetric communication environments. After the survey, we describe how the research topics studied in this dissertation are innovative.

2.1 Related Research

2.1.1 Broadcast Scheduling

As mentioned earlier, in the data broadcasting model the information provider broadcasts information on a channel sequentially and repeatedly to an unlimited set of clients. If a request is issued right after the needed data unit has been broadcasted and that data unit is sequenced on the schedule, the request may have to wait a long time to be satisfied. Moreover, if a large number of clients request some data very frequently while these data are broadcast in long sequence, the probability for many clients to wait a long time for their data would be high. Since request response time (wait time) is a major performance metric in broadcast-based applications, an efficient schedule of data broadcast sequence and frequency (how it is repeated) is an important topic. This section surveys research done on broadcast scheduling.

Currently, the major performance metric used to evaluate broadcast scheduling algorithms is overall average response time [ACH95a, STA96, STA97, VAI97a], which is the average waiting time for all client requests. Because client requests are unbounded and unidentified, the evaluation of this performance metric is always dependent on the access (limited to data read) possibility of each data unit, which is the portion of overall requests that are targeted for particular data. Moreover, it is also dependent on the average response time for all requests for each data unit.

A flat broadcast program [ACH95a, STA96, STA97, VAI97b] (sometimes called a flat disk) is the simplest broadcast-scheduling algorithm. In such a program, all data units are broadcast sequentially from the first to the last. After that, the broadcast starts again at the first, using the same sequence, and repeats the same action infinitely. The period to broadcast all data once is called a cycle. The advantage of this algorithm is its

simplicity. When new data are inserted in the server, they are simply added to the end of the current (or next) cycle. However, this method does not consider the differing interests of clients. All requests would have the same average response time no matter whether a data unit is requested very frequently or very rarely.

A multi-disk algorithm [ACH95a] is a scheduling algorithm that broadcasts hot data with higher frequency than colder data. This scheme is very similar to putting data on several disks that spin at different speeds. Hot data are assigned to a fast disk while cold data are on a slow disk. This algorithm overcomes the inefficiency of flat broadcast since it makes the space between hot data on a schedule shorter than that between cold data. Consequently, the overall average response time can be reduced.

It is important to mention that many scheduling algorithms are based on the rule that the inter-arrival rate of each data unit on a schedule should be fixed [ACH95a, HAM97, VAI96, VAI97a]. That is, all data units should be evenly spaced whenever possible. This rule is the result of the following observation: If the inter-arrival rate of a data unit is fixed, the average delay for any request arriving at a random time is one-half the even space between successive broadcasts of the data unit. In contrast, if there is variance in the inter-arrival rate, the space between broadcasts will be of different lengths. In this case, the probability of a request arriving during a long time duration is greater than the probability of the request arriving during a short time duration. Thus, the average waiting time must be longer than would be the case when inter-arrival rate is fixed, under the condition that the total broadcast rate of one data unit in a given period of time (such as a cycle) is the same.

A multi-disk algorithm is based on even-spaced data broadcasting. It first orders the data from hottest to coldest and then partitions these data into multiple ranges ("disks") where each range contains pages with similar access possibilities. All data on one disk are assigned the same integer relative frequency value. Each disk is again divided into a number of chunks (num_chunks; the notation C_{ij} refers to the jth chunk in disk i). The num_chunks is calculated by dividing the least common multiple of all disks' relative frequency (max_chunks) by relative frequency of disk i. Then a broadcast schedule is segmented into max_chunks. In the jth segment, disk i's $C_{i,\ (j \bmod num\_chunks(i))}$ chunk is broadcast sequentially. This method reduces average response time of hot data by sacrificing the broadcast rate of cold data and overall average response time may be reduced. The limitation to this method is that the relative frequency of all data is estimated (it has to be an integer and the number of disks should be limited). The consequence is, though performance is improved compared with flat disk, it might not guarantee optimal performance. Moreover, multi-disk assumes that all data units are the same size.

To achieve an optimal (minimum) overall average response time, researchers [HAM97, VAI97a, VAI97b] derived a data broadcast frequency assignment scheme based on each data unit's access possibility and length. These works also assume even-spaced broadcasting for each data unit. Given M data items with length L_i and access possibility P_i for each data item ($1 \le i \le M$), researchers concluded that the minimum overall average response time is achieved when frequency F_i of each item i is proportional to $\sqrt{P_i}$ and inversely proportional to $\sqrt{L_i}$. That is:

$$F_i \propto \sqrt{\frac{P_i}{L_i}}\,. \tag{2-1}$$

The even space S_i can be calculated based on the above result of optimal data frequency assignment:

$$S_i = \left(\sum_{i=1}^{M} \sqrt{P_i L_i}\right) \sqrt{\frac{L_i}{P_i}}\,. \tag{2-2}$$

However, if all data were simply spaced evenly using these values in a broadcast schedule, some of them would overlap. Consequently, the absolute minimum response time cannot always be achieved. Therefore, these researchers also proposed a fair-queuing scheduling algorithm based on the calculated S_i value of each data unit. This algorithm is based on the observation that each data unit occupies a fixed portion (L_i/S_i) of a broadcast cycle and the data units are evenly spaced. Since this is similar to conditions imposed on the packet fair-queuing algorithm [BEN96a, BEN96c] for bandwidth allocation, this fair-queuing algorithm is adopted to broadcast scheduling based on S_i values.

This broadcast fair-queuing algorithm maintains two variables B_i and C_i for each item i. B_i is the earliest time when the next instance of data i should begin transmission and is initialized as 0. $C_i = B_i+S_i$ and is initialized as S_i. At first, the algorithm determines optimal spacing S_i for each data unit based on Formula (2-2). It uses T to refer to current time, which is initialized as 0. Set S contains data items for which $B_i \le T$. Variable C_{min} denotes the minimal value of C_i over all items in set S. The algorithm always chooses the item $j \in S$, such that $C_j = C_{min}$ to broadcast. After broadcasting a data item j at time T, it sets $B_j = C_j$, $C_j = B_j+S_j$, and $T = T+L_j$ sequentially. Then it updates S based on the broadcasted data item's B_i, C_i value, and the new current time T. Then the data with least

C_j is chosen, and above steps are repeated infinitely. Although all data items cannot be exactly evenly spaced, some simulation results [HAM97, VAI97a, VAI97b] show that the performance is very close to the optimal result. The advantages of the broadcast fair-queuing algorithm are: it supports different lengths of data, it achieves nearly optimal overall average response time, and it has dynamic scheduling (deciding the sequence at the time of scheduling instead of deciding it in advance, as in a multi-disk algorithm).

Multi-disk and broadcast fair-queuing algorithms are based on pre-known access possibility and are completely independent of the dynamic client-access requests. There also exist some on-demand broadcast schedule algorithms, which assume data requests can still be sent to the server through a backup upstream link [ACH97, ACH98, AKS97, AKS98, STA96, STA97, FER99, JAI95, XUA97]. Some of these algorithms [ACH97, ACH98, AKS97, AKS98] only use the backup channel to inform the server of the current data demand of clients, while all requests are still satisfied by broadcasting. This is suitable for applications where the data access possibility can dynamically change. In order to save broadcast bandwidth, some others also use the backup channel to send very cold data back to the requesting clients [STA96, STA97, FER99].

For example, in addition to the passive downstream channel, one team of researchers [ACH97] proposed using a backup channel to send requests. When requests are queued at the server, another related team [ACH98] proposed a new metric called stretch (ratio of response time to service time) and used it to evaluate several request scheduling algorithms, including FCFC, LWF, SSTF, PLWF, SRST, LTSF, BASE, and an on-line BASE algorithm, which decides what data to broadcast first based on the current requests. Aksoy and Franklin [AKS97, AKS98] proposed another request

scheduling algorithm called RxW and evaluate it based on average request response time. The models proposed in [STA96, STA97, FER99] are also called hybrid broadcasting, where data can be sent both on a broadcast channel and backup channel. Stathatos et al [STA96, STA97] divide data based on their temperature and exclude cold data from broadcasting. Not just a hybrid model, [FER99] also can shift data between broadcasting and single response based on current demand, which is estimated by using a cooling factor.

2.1.2 Cache

Although optimal overall average response time can be achieved by using proper scheduling algorithms, it is not efficient enough for all client applications to depend completely on server broadcasting. When the database is relatively large, the data access delay can be intolerable. To solve this problem, research on cache strategies is conducted to allow the client to memorize some data already broadcast earlier in a local cache. Consequently, if the data are cached efficiently, most client requests can be satisfied by reading from the cache directly instead of always waiting on the air.

Cache strategies are mainly addressed in research articles [ACH95a, ACH95a, BAR94, HU99, JIN97, LEO97, ZDO94]. In all, basic cache strategies are the same. They work by memorizing part of the data locally at the client end. However, when it is necessary to replace cached data with new data, their replacement policies differ.

Acharya et al [ACH95a, ACH95a, ZDO94] propose P (Possibility), PIX (Possibility Inverses Frequency), and LIX (LRU Inverses Frequency), respectively, as cache replacement policies. P replaces the data with a least access possibility. PIX replaces data with the least value of access possibility inverse broadcast frequency. LIX is a simplified version of PIX and uses LRU sequence to represent access possibility.

Among these solutions, PIX performs best because it considers the broadcast frequency feature of a data unit in addition to whether it is hot or not. If a hot data unit is broadcast very frequently, it is not necessary to cache it. However, for a data unit with relatively high client interest but low broadcast frequency, it is more worthwhile to cache it.

In addition to caching a data unit when it is requested and as it goes by on the air, pre-fetch, which pre-caches some data before it is requested based on some criteria, has also been investigated [ACH95a, ACH95a, ZDO94]. Proposed criteria include P (Possibility) and PT (Possibility Time Time-left). Again, P pre-fetches a data unit as it goes by on the air and its access possibility is higher than any other in the full cache. Using PT, data with a higher value of access possibility multiplied by time left until the next broadcast is pre-fetched. Based on simulation results [ACH95a, ACH95a, ZDO94], PT performs better, and it is even better than, PIX. The reason is it uses a dynamic broadcasting scheme that caches the data with highest access rate and longest time left until next broadcasting.

2.1.3 Index

The spreading use of wireless applications motivates the analysis on how important energy-saving is in mobile applications [HU01, IMI94a, IMI94b, IMI94c, KHA98]. Based on these analyses, the ratio of power consumption in mobile equipments' active mode to doze mode can be as large as 5000. Therefore, keeping mobile equipment in doze mode can greatly prolong battery life, which aids convenience and reduces pollution. However, the data broadcasting model usually requires the client to be active all the time in order to monitor the data units that go by. This makes power-saving infeasible in broadcast-based applications.

To help save power in broadcast models, indexing schemes are proposed in [DAT97, IMI94a, IMI94b, IMI94c]. The basic idea of indexing is to insert a pointer for data broadcast in a future schedule in a broadcast cycle. Consequently, a client application can go to doze mode after it accesses this pointer, and only wakes up at the time the requested data unit is on the air.

However, randomly inserting an index in the broadcast cycle can influence the average response time of data access because inserted information can prolong a basic cycle. Therefore, index schemes need to be designed efficiently to provide good performance for both access time and tuning time [IMI94b]. Access time determines how long on average a request needs to wait until the data unit is retrieved. Tuning time determines how long a client needs to be active on one data unit request.

Several index schemes have been proposed in [DAT97, IMI94a, IMI94b, IMI94c]. Among them, access_opt provides the best access time and the worst tuning time, which uses no index. Tune_opt provides the best tuning time with a large access time, which only inserts one index tree at the beginning of each broadcast cycle. The large access time of tune_opt is caused by a client always waiting for the beginning of the next cycle to access the index information.

Since achieving good performance of both access time and tuning time is very important, (1,M) index and distributed index schemes are proposed to achieve this goal. Using a (1,M) index, an index tree of all data in a broadcast cycle is inserted using even space in the cycle for M times. Pointers to each real data units are located at the leaves of the index tree while a route to a specific leaf can be found following the pointer from the tree root. Each normal data unit also has a pointer to the start of the next index tree. As a

result, each data access process can go to doze mode when it obtains the pointer to the beginning of the next index tree. Then the client can wake up at the tree root, follow the pointers until it finds the leaf referring to the needed data. According to the analysis, the tuning time for the (1,M) scheme is $2+\lceil \log_n Data \rceil$, where Data is the number of equal-sized data items broadcast, n is the number of pointers a data unit can have, and the optimal access time is achieved when M has the value of $\sqrt{\frac{Data}{Index}}$, where Index is the number of nodes in the index tree.

The (1,M) index is somewhat inefficient because of the M redundant replications of the whole index tree in a broadcast cycle. A distributed index has been proposed to reduce this inefficiency [IMI94b]. This is a technique in which the index is partially replicated. It is based on the observation that there is no need to replicate the entire index between successive data segments--it is sufficient to have only the portion of index that indexes the data segment that follows it. Non-replicated distribution, entire path replication, and partial path replication are different distributed index options. Non-replicated distribution does not broadcast an index not pointing to the subsequent data segment. As a result, if the needed data unit is not in the next data segment, the client has to wait until the start of the next cycle. On the other hand, entire path replication replicates all index nodes on the route from the root to the leaves pointing to the following data segments. This method wastes broadcast bandwidth by replicating some useless index tree nodes, despite the fact that it is already more efficient than the (1,M) index. Lastly, partial path replication only replicates a part of the necessary nodes needed from the root to the leaves pointing to the following data segments. The replicated branch

nodes are all necessary nodes that help to locate other parts of the index tree when requested data is not in a subsequent data segment.

Performance analysis result shows that the partial path replication distributed index scheme has a tuning time bounded by $3+\lceil \log_n Data \rceil$ and achieves much better access time than other methods.

2.1.4 Concurrency Control

Data broadcasting can also be used in transaction-based applications, where client data requests are issued by multi-operation transactions. When transactions are involved and there exist data updates at the server, concurrency control is necessary to maintain server database consistency and the currency of data read by client transactions.

On the other hand, there is no direct communication between any client and server in a broadcast-based asymmetric communication model for read operations. This implies traditional schemes such as 2-phase locking, time-stamp ordering, and optimistic concurrency control cannot be applied in this type of environment to control concurrency in transaction processing because all these methods require extensive communication among clients and server for all operations. In order to take full advantage of data broadcasting and still achieve correctness and consistency, new concurrency control protocols suitable for the broadcast-based asymmetric communication environment need to be developed. In this section, we survey existing research on broadcast-based concurrency control strategies.

Update First Ordering (UFO) [LAM99] is a concurrency control scheme used for applications with transaction processing where all updates are at a database server and all transactions from mobile clients are read-only. Moreover, it assumes a maximum drop

period n_c (in cycles) for each read-only transaction and the read operations can be out of order. UFO achieves serialization by always guaranteeing an UT (Update Transaction) → MT (Mobile Transaction) sequence. To achieve this objective and based on the fact that it is always true that BT → MT, UFO always lets UT → BT (Broadcast Transaction). Because there is a maximum drop period for all read-only transactions, at the end of an update transaction, the data set broadcast within n_c cycles before the completion of the update transaction is checked. If there is an overlap between the broadcasted set and the updated set, the overlapped items will be rebroadcasted immediately. In this way, UT → BT is achieved. So, if there is any read-only transaction that reads a data unit earlier while the same data unit is updated before it commits, the same item will be broadcasted again. Because of the execution can be out of order, data can always be reread, and serialization is achieved. UFO has the advantage of simplicity and low overhead. However, it can only support read-only mobile transactions and has such assumptions as out of order read operations, maximal transaction spanning time, small number of update transactions, and low data conflict probability.

A Multi-Version broadcast scheme has been proposed which has strong serialization properties [PIT98, PIT99a]. Instead of broadcasting the last committed value for each data item, this method broadcasts all values of an item at the beginning of the last x broadcast cycles along with a version number that indicates the broadcast cycle to which each version corresponds. The value of x equals the maximum transaction span among all read-only transactions. Any transaction started in a specific cycle only reads the data version corresponding to that cycle. The advantage of Multi-Version broadcast is its support of strong serialization and disconnections. However, the disadvantages, big

broadcast overhead and inflexibility of transaction order, overshadow the advantages. Moreover, this solution depends heavily on the broadcast cycles and also only supports read-only transactions.

Another method supporting strong serialization is based on Serialization Graphs [PIT98, PIT99a, PIT99b]. In this method, the server and clients all maintain a serialization graph of a history H. Each client maintains a copy of the serialization graph locally. At each cycle the server broadcasts any updates of the serialization graph. Upon receipt of the graph updates, the client integrates the updates into its local copy of the graph. The serialization graph at the server includes all transactions committed at the server. The local copy at the client end also includes any live read-only transactions. The content of the broadcast is augmented with the following control information: an invalidation report that includes all data written during the previous broadcast cycle along with an identifier of the transaction that first wrote each data item and an identifier of the transaction that last wrote each data item, and the difference from the previous serialization graph. Each client tunes in at the beginning of the broadcast cycle to obtain the information. Upon receipt of the graph, the client updates its local copy of the serialization graph to include any additional edges and nodes; it also uses the invalidation report to add new precedence edges for all live read-only transactions. The read operations are accepted if no cycle is formed, while a cycle is formed if there exists an update transaction that overwrote some items in the read set of the read-only transaction which precedes the update transaction in the graph. The Serialization Graph method suffers from complicated maintenance and computation of serialization graphs. At the same time, the complicated control information also causes excessive overhead for

broadcasting. As is true for UFO and Multi-Version, Serialization Graph only supports read-only mobile transactions. To simplify the operations, an Invalidation-Only Broadcast scheme [PIT98, PIT99a] only broadcasts an invalidation report that includes all data items that were updated during the previous broadcast cycle. A live read-only transaction is aborted if any item that is in the read set of this transaction appears in the invalidation report. This reduces processing overhead but raises the probability of increased transaction abort rate.

BCC-TI [LEE99] (Broadcast Concurrency Control with Time Stamp Interval) also only supports read-only transactions. It supports adjustable transaction serialization orders by using a time interval strategy. In BCC-TI, the server records the time TS(U) that any update transaction U commits. It also traces the write set WS(U) of all update transactions committed in one cycle. Each data item d has a write time stamp WTS(d), which is the time that data d is last updated, attached to it. At the beginning of each cycle, TS(U) and WS(U) of all committed update transactions in the last cycle are broadcasted as control information. Each data item d and its time stamp WTS(d) are broadcasted together. Each read-only transaction is assigned a time interval (LB, UB), which is (0, $+\infty$) at the beginning. Whenever it reads an item d from the air, it change its LB to be the maximum of the current LB and WTS(d). At the beginning of each cycle, an alive, read-only transaction’s read set in the last cycle is compared with each WS(U), which is the write set of a committed update transaction in the last cycle. If there is an overlap with a WS(u), the UB of this read-only transaction is changed to the minimum of the current UB and TS(u). If at any time a read transaction's LB is larger than or equal to its UB, this transaction is aborted. Otherwise, it can be finally committed. BCC-TI has flexibility and

relatively low overhead, though it only supports read-only transactions and is dependent on cycles.

To our knowledge, F-Matrix [SHA99] and Certification Report [BAR97a] are the only existing methods that support both mobile client read-only and update mobile client transactions. F-Matrix [SHA99] uses a weaker update consistency other than global serializability [BER87, OZS91] as its correctness criterion. Update consistency guarantees all update transactions are serializable [BER87, OZS91] and all read-only transactions are serializable to all update transactions that they read from. F-Matrix suffers from the overhead of N×N matrix control information broadcast and computing, which is especially large when the size of the database is big and the size of the data unit is relatively small.

Compared with F-Matrix, Certification Report [BAR97a] is global serializability-based. In Certification Report, each transaction maintains a ReadSet and a WriteSet of itself. At the beginning of every cycle, the server broadcasts the ReadSet and WriteSet of the transactions that have been committed in last cycle. The server also broadcasts the transactions IDs of the accepted transactions and rejected transactions of last cycle. A locally executing transaction compares its own ReadSet and WriteSet with the broadcasted ReadSet and WriteSet. If there are any conflicts, the executing transaction is aborted. Otherwise, this transaction will be submitted to the server in a package by sending its ID, its ReadSet/WriteSet information and the newly written values. In order to avoid any conflict among submitting transactions and validating transactions when transactions are on their way to be submitted, each submitted transaction is again compared with the ReadSet/WriteSet of all transactions committed from the time it was

finished and submitted at the client to the time it is validated at the server. A transaction can only be committed when no conflict is detected in the comparison. To achieve this, a Certification Report history needs to be kept. This might be longer than the Certification Reports of the last and current cycle. At the same time, a local submitting timestamp of each transaction also needs to be maintained and a synchronization scheme is needed to decide the time period in which the history of the Certification Report needs to be compared. A transaction's ID is inserted into the accepted transaction set or rejected transaction set and is broadcast in the next broadcast cycle to let the owner of the transaction know if it is committed at the server. A big drawback is that the architecture of the protocol also needs all read-only transactions to be submitted to the server. If the read-only transactions are disproportionately large, this solution cannot fully utilize the advantage of asymmetric communication. Moreover, the Certification Report has a relatively large overhead of control information, and is very dependent on cycles, transaction ID maintenance, and synchronization of client and server times. It is also unable to avoid aborting some transactions unnecessarily.

2.1.5 Fault Tolerance

Errors can happen in data broadcasting. Therefore, fault tolerance is also an important issue in data broadcasting based applications. Currently, this problem is only addressed in [BAR97b, BES94, BES96]. These researchers assume data objects are broadcast on-air. With each data object divided into several blocks, any error in one block will mean the client has to wait for another broadcast cycle for the block to be re-broadcasted so that the whole object can be constructed.

An information dispersal scheme called AIDA is proposed in these papers as a fault tolerance and recovery algorithm used for simplex information distribution

applications. Assume a file F of length m blocks is to be broadcasted by sending N independent blocks. Using the AIDA algorithm, F can be processed to obtain N distinct blocks in such a way that recombining any m of these blocks, $m \leq N$, is sufficient to retrieve F. The process of processing F and distributing it over N blocks is called the dispersal of F, whereas the process of retrieving F by collecting m of its pieces is called the reconstruction of F. The dispersal and reconstruction operations are simple linear transformations using irreducible polynomial arithmetic. The dispersal operation amounts to a matrix multiplication that transforms data from m blocks of the original object into the N blocks to be dispersed. The N rows of the transformation matrix $[X_{ij}]_{Nxm}$ are chosen so that any m of these rows are mutually independent, which implies that a matrix consisting of any such m rows is not singular and thus invertable. This guarantees that reconstructing the original file from any m of its dispersed blocks is feasible. Indeed, upon receiving any m of the dispersed blocks, it is possible to reconstruct the original object through another matrix multiplication. The transformation matrix $[Y_{ij}]_{mxm}$ is the inverse of a matrix $[X'_{ij}]_{mxm}$, which is obtained by removing N - m rows from $[X_{ij}]_{Nxm.}$ The removed rows correspond to dispersed blocks that were not used in the reconstruction process.

By using AIDA, when one block is broadcast with an error, a client only needs to wait until one more block is broadcast in order to accumulate the m blocks necessary to reconstruct an object. This is more efficient than waiting for another broadcast cycle to retrieve the same block. Analysis of AIDA shows the following results: If the broadcast period of a flat broadcast program is t, then an upper bound on the worst-case delay incurred when retrieving an object is rt units of time, where r is the number of block

transmission errors. However, if the maximum time between any two blocks of a dispersed file in an AIDA-based flat broadcast program is t', then an upper bound on the worst-case delay incurred when retrieving that file is rt' units of time. Based on this result, if an object is broadcast evenly in N blocks in a cycle, by using AIDA the delay is only 1/N of normal delay caused by any r errors.

2.1.6 Real-time Requirement

A data broadcasting model can also include real-time issues. Existing real-time based research mainly addresses real-time data broadcast scheduling [BAR97b, BES94, BES96, FER99, JIA99, LAM98].

In many models, each broadcast data unit has a deadline [BAR97b, BES94, BES96]. Flat organization, rate monotonic organization, and slotted rate monotonic organization are proposed to schedule a group of broadcast data with deadlines. If the broadcast cycle is less than or equal to the least deadline among all data, flat organization broadcasts each data unit once in a flat schedule. This solution wastes bandwidth by broadcasting most data too frequently compared with their required deadline. Rate monotonic organization broadcasts each data unit periodically with the deadline as the period. A group of data is schedulable if its bandwidth utilization is less than or equal to the available channel bandwidth. Despite the fact that it achieves optimal utilization of downstream bandwidth, this method is not practical in a large system with thousands of data items, each with a different timing constraint. Flat organization trades off bandwidth utilization for ease of broadcast programming, whereas rate monotonic organization trades off ease of broadcast programming for bandwidth utilization. The slotted-rate monotonic organization strikes a balance between these two extremes. The basic idea is to coalesce some data items together into groups so that they can be broadcast in the same

period. The real broadcast period of each data unit should be less than its deadline. Such a method can improve bandwidth utilization better than flat organization, and reduce broadcast program complexity better than rate monotonic organization.

Fernandez-Conde and Ramamritham [FER99] address the real-time requirements of client requests. In this research, each client request for a specific data item has to be satisfied within a deadline. Assume there are N data items (each with length S_k, each data item is broadcasted in S_k units and any S_k units can construct data k) in the server, and the deadline for any request on data k is D_k. A pfair schedule is used to broadcast server data such that in any D_k time period at least S_k data units are used to broadcast data k. A pfair schedule first decides whether required bandwidth $\sum_{k-1}^{N}(S_k+1)/(D_k+1)$ is less than 1. If not, the given data are not schedulable. Otherwise, add a dummy pair (S_{N+1}, D_{N+1}) such that $\sum_{k-1}^{N}(S_k+1)/(D_k+1) = 1$. Then, it schedules the N+1 data items by assigning each broadcast unit to the contending item with the smallest pseudo deadline. Ties can be broken arbitrarily. A data item is said to be contending at time unit t if it may receive the broadcasting unit without becoming overallocated. That is, item k is not overallocated at time t if $\left\lceil \frac{allocated(k,t)}{(S_k+1)/(D_k+1)} \right\rceil \le t$, where allocated (k, t) = number of time units allocated to item k in the interval of [0,t)). Pseudo-deadline of item k at time unit t is defined as: $\left\lceil \frac{allocated(k,t)+1}{(S_k+1)/(D_k+1)} \right\rceil$.

Lam et al [LAM98] addresses real-time data scheduling problems for multi-request client transactions. This research assumes each transaction (read-only) has

multiple data requests and these requests are sent to the server together. At the same time, each transaction has a deadline. A request portion and deadline-based algorithm are proposed for scheduling these transactions. Request portion assigns scores to each data unit based on how many current transactions need this data unit and the stage of these transactions where the data unit is requested. The score is higher and the data unit is scheduled earlier if the data unit is requested in an earlier stage of any transaction or is requested by a large number of transactions. Moreover, a deadline-based algorithm assigns each data unit a deadline that is the minimal deadline value derived from all transactions that need this data unit. If a transaction has a deadline D and has N data request, then each data unit requested by it has a deadline of D/N based on this transaction. A deadline-based algorithm schedules the data unit with minimum deadline value first.

2.2 Novelty of this Research

The purpose of this dissertation research is to solve problems not covered or in need of improvement in existing research in a broadcast-based asymmetric communication environment. At the same time, the main objective of our research is to explore schemes to support efficient transaction processing in broadcast-based applications.

The following observations motivated us to emphasize broadcast-based transaction processing. First, only the performance of a single operation is the major concern in most existing efforts. As an example, Multi-disk [ACH95a] scheduling tries to minimize average response time of single access operations. PIX and PT [ACH95a, ZDO94] for cache or pre-fetch are also only concerned with an individual operation's

behaviors. Some real-time scheduling algorithms like pfair [FER99] only consider a single operation's deadline. On the other hand, although research on broadcast-based transaction-processing issues, such as concurrency control and real-time scheduling, does exist, it only covers particular assumed models. This means there are transaction-processing problems with different system assumptions and requirements without solutions. At the same time, some of these transaction-processing schemes have limitations. Finally, transaction processing has never been addressed in some new broadcast-based asymmetric system architectures, such as hierarchical broadcasting models.

Broadcast-based applications are very likely to be transaction-based. For example, stock-exchange applications can involve transactions with multiple price-readings and buying/selling operations. Insurance agents might utilize consecutive transactions to collect insured-customer information, process an accident claim, or retrieve insurance policy information. Optimal research results favoring single operations might not be optimal for complete transactions in these types of applications. When transaction processing is involved, some other problems appear that also need to be solved. Therefore, it is very desirable for us to examine or reexamine many research topics in a broadcast-based environment focusing on transaction behaviors. This transaction-based research should aim at supporting efficient transaction processing, i.e., supporting good performance in broadcast applications from the transaction point of view.

In conclusion, the novelty and research goal of this dissertation is to explore efficient transaction-processing schemes in a broadcast-based asymmetric communication environment. The problems covered are divided into the following four

categories: an efficient concurrency control protocol, problems evolved from the proposed currency control protocol, transaction processing in hierarchical broadcasting models, and other performance issues for broadcast-based transaction processing.

2.2.1 Efficient Concurrency Control Protocol

The earlier research survey introduced some concurrency control strategies designed for transaction processing in broadcast applications. In the first part of this dissertation, we will discuss the limitation of these existing strategies and the reasons why new correctness criteria with their corresponding new concurrency control protocol are needed for transaction processing in a broadcast environment. Then the new criteria, called Single Serializability and Local Serializability, and the newly designed concurrency control protocol, called server timestamp update broadcast supported concurrency (STUBcast) are presented. Simulation results of STUBcast are discussed in this chapter after the correctness proof of this protocol.

2.2.2 Problems Introduced by the Proposed Concurrency Control Protocol

Performance evaluation shows STUBcast is an efficient concurrency control scheme. This makes it desirable for use in real transaction-based applications. However, it is necessary to adjust the applications' caching, scheduling, and indexing schemes to the new protocol's solution model because existing methods do not fit into STUBcast directly, or do not work efficient enough with STUBcast. The primary STUBcast protocol also needs to be modified to accommodate these new components. Therefore, this category of research reexamines all these issues and coordinates them to work together in transaction processing applications.

2.2.3 Transaction Processing in Hierarchical Broadcasting Models

Most broadcast research topics assume one-level broadcasting, where one top server broadcasts data to all clients. In this category, we investigate transaction-processing issues in hierarchical broadcasting models where more than one level of server and client communication exists in an application and at least one level of communication is broadcast.

2.2.4 Other Performance Issues for Transaction Processing

The last category addresses some miscellaneous transaction-performance issues found in a broadcasting environment. They are called "miscellaneous" because these issues are independent. They have different system assumptions, solve different transaction-based problems, and have different performance requirements. In fact, we study four independent problems related to transaction processing in a broadcast environment and design solutions to meet different transaction performance requirements for each problem in this category.

The first is a scheduling problem, the solution to which aims at providing minimal fair response-time among transactions. Second, a real-time scheduling problem is studied with the objective of making all in-phase, periodic, client transactions meet their deadlines by scheduling data properly.

The third problem is to find suitable cache replacement policies to allow long transactions to be favored, because long transactions have longer restart numbers than short ones when concurrency control is required. A good cache scheme can enhance overall transaction performance on response time and restart number. The fourth problem in this category uses efficient cache-replacement policies to improve the percentage of transactions that meet their soft real-time requirements.

CHAPTER 3
EFFICIENT CONCURRENCY CONTROL PROTOCOL

In this chapter, we first present the motivation to design a new concurrency control protocol for transaction processing in a broadcast-based asymmetric communication environment. We then describe the solution, including some new concurrency correctness criteria and an efficient currency control protocol called server timestamp update broadcast supported concurrency (STUBcast), followed by its correctness proof and simulation results.

3.1 Motivation and Problem Definition

In Chapter 2, we introduced several existing methods for broadcast-based transaction concurrency control, such as UFO [LAM99], Multi-Version Broadcast [PIT98, PIT99a], Serialization Graph [PIT98, PIT99a], BCC-TI [LEE99], F-Matrix [SHA99], and Certification Report [BAR97a]. All these methods have some limitations, however. For instance, UFO, Multi-Version, Serialization Graph, and BCC-TI only support read-only transactions, which introduces the necessity to develop applications with both read-only and update transaction capabilities. In addition, the processing overhead in Multi-Version and Serialization Graph is potentially substantial if they implement the process to backup/broadcast/compute old-version data items or serialization graphs. Among the solutions supporting both read-only and update transactions, F-Matrix has large computing and broadcast overhead by using N×N matrix control information each cycle. The Certification Report approach also suffers from

inefficiency because it needs to synchronize server and client time clocks, send both read-only and update transactions to server, and backup the server transaction history for verification.

In terms of the correctness criteria, UFO, Multi-Version, BCC-TI, and Certification Report are designed to support global serializability, which requires all existing transactions on any clients and server to be serializable [BER87, OZS91, SHA99]. The execution of transactions is serializable if it produces the same output and has the same effect on the database as some serial execution of the transactions [BER87]. However, among these methods, BCC-TI and Certification Report abort some transactions even though they are actually serializable to all other transactions. Figures 3-1 and 3-2 show examples of how BCC-TI and Certificate Report reject serializable transactions. The examples are based on their protocol polices described in Section 2.1.4.

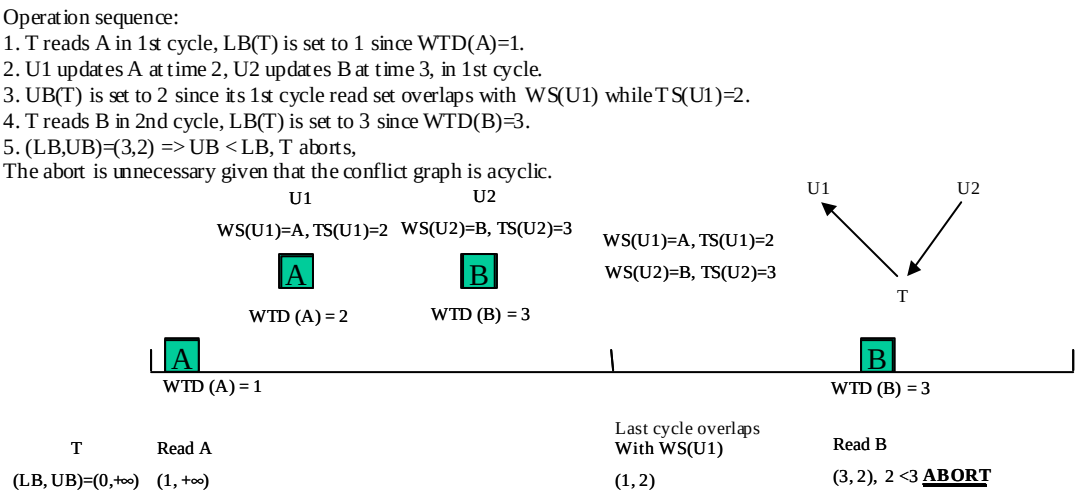


Figure 3-1 Example of a rejected transaction under BCC-TI protocol

In these figures, conflict graphs are used to verify whether transactions are serializable based on the Serializability Theorem [BER87]. The conflict graph of a set of

transactions is a directed graph where the transactions are linked with edges that represent the conflict relationship among transactions. A conflict means two transactions execute operations on the same data unit and at least one of them is a write. Thus, the operations must be ordered in a certain sequence, usually in their execution sequence. An edge is drawn from one transaction to another when there is at least one conflict between them and the first transaction's conflict operation is sequenced (executed) earlier than the second one. According to Serializability Theorem [BER87], a group of transactions is serializable if and only if their conflict graph has no cycles.

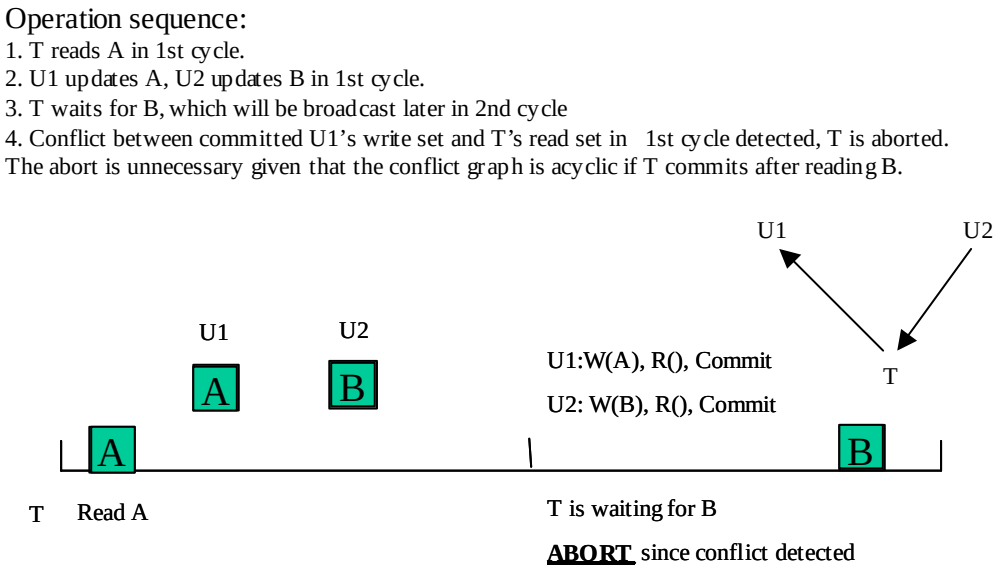


Figure 3-2 Example of a rejected transaction under certification report protocol

Rejecting transactions that are actually serializable, as happens in BCC-TI and Certification Report, reduces the efficiency of the transaction-processing system. However, if the information on transaction execution is insufficient or when certain optimizations must be carried out, some serializable transaction may be rejected. When not enough information is available to judge whether cycles exist, a protocol may reject

the transaction or send inquiries to verify the relationship. In Figures 3-1 and 3-2, the conflict graphs among transactions T, U1, and U2 have no cycles. Despite this, BCC-TI and Certificate Report reject them because the protocols have no information about whether there is an edge from U1 to U2.

The various limitations of existing broadcast-based concurrency control solutions suggest we need to design a new protocol that can overcome these limitations. This introduced the following problem:

Design a new, efficient concurrency control protocol for a broadcast-based asymmetric communication environment that supports both client read-only and update transactions, low broadcast overhead, low protocol computation overhead, simplicity, and low unnecessary abort rate caused by maintaining global serializability.

3.2 Single Serializability and Local Serializability

Before we propose a new concurrency control protocol to solve the problem above, we analyze the necessity for and define some new concurrency correctness criteria. This new protocol is designed to support all required features stated in the problem and these new criteria are essential to avoid unnecessary aborting of acceptable transactions.

Unnecessary aborting of serializable transactions happens because a pessimistic view of the transaction conflict graph is applied to achieve global serializability and it becomes problematic to establish a precise conflict dependency in such a distributed system. Nevertheless, strong global serializability may not be necessary and relaxed correctness criteria can still be practical in real applications. For instance, the F-Matrix approach ensures consistency and serialization among all update transactions [SHA99].

While update consistency is essential to guarantee data integrity, we argue that weaker criteria than global consistency is practical in many distributed applications. For instance, the reports of two independent stock-price updates do not need to be consistent if they are taken from two different sites of the systems. That is, one informs the old price of stock A and the newly updated price of stock B in New York, whereas the other presents the new price of stock A but the old version of stock B in Chicago. In other word, the updates are concurrent and the viewing operations do not need to (or cannot) be atomic as they occur in disparate locations and instances.

The relaxed criteria require us to look into two types of serializability: single serializability *(SS)* and local serializability *(LS)*. Single serializability insists that all update transactions and any single read-only transaction are serializable. Local serializability, conversely, requires that all update transactions in the system and all read-only transactions at one client site are serializable. Obviously, *SS* and *LS* are weaker than global serializability since they do not consider all other read-only transactions or read-only transactions on different client sites. Similar to update consistency, *SS* and *LS* guarantee the consistency and correctness in a server's database. However, they are easier to achieve in a broadcast environment because the read-only transactions ignore the potential dependency cycles caused by different read-only transactions (or read-only transactions on different client sites) in their conflict graphs. They can retrieve the broadcasted data, have no need to contact the server, and need not worry about how other read-only transactions view the updates. Even if several views of updates are required in decision-making, often the transactions involve the same client site, a site where *LS* can be applied.

Motivated by the requirements of the problem and the applicability of the relaxed serialization criteria, in next section, we propose, the design of STUBcast, a new broadcast-based concurrency control protocol. This protocol supports *SS* and *LS* in a broadcast-based asymmetric communication environment and meets the requirement defined above.

3.3 STUBcast – An Efficient Concurrency Control Protocol

3.3.1 System Model

The system model for STUBcast is shown in Figure 3-3. Database items are broadcast continuously by a server to a large population of clients, where both read-only transactions and update transactions are performed. Read-only transactions are completed independent of the server. Update transactions can modify the data in the server by sending a request initiated by the client through a low bandwidth or separate channel. Any transactions issued by the server are also treated as client-side transactions. STUBcast is designed for transactions that execute their operations sequentially. The number and sequence of transaction operations are not known in advance. There can be inter-arrival time between any two subsequent operations. We assume neither temporary cache nor index is used in the STUBcast. How to adapt STUBcast to cache and to index are discussed as separate problems in Chapter 4. Since our concern in this section is concurrency control, we make no assumption as to how the server schedules the broadcast cycles.

The broadcast operations can be divided into two parts. One disseminates the existing data items to the clients. We denote this as primary broadcasting *(Pcast)* during which the current value and the identifier (*ID*) of each data item will be put on the air.

The server can use any broadcast algorithm, such as flat, multi-disk, or dynamic scheduling (like fair-queuing scheduling), as its *Pcast* schedule. The second type of broadcast operation, denoted as update broadcasting (*Ucast*) sends out newly updated data items (with value and ID) by committed transactions. Corresponding to each committed update transaction, a *Ucast* is inserted into the on-going *Pcast* once the transaction commits at the server. As shown in Figure 3-4, to mark a *Ucast* stage, each *Ucast* begins with a update broadcast beginning (*UBB*) tag and ends with a update broadcast ending (*UBE*) tag. In addition to the broadcast of updated data items, the IDs of the data items read by the committed update transaction are attached using a *UBE* tag. Note that *Ucast* operations are done immediately after the update transactions commit and we require the *Ucast* operations to be sequenced in the same order of conflict relations among transactions.

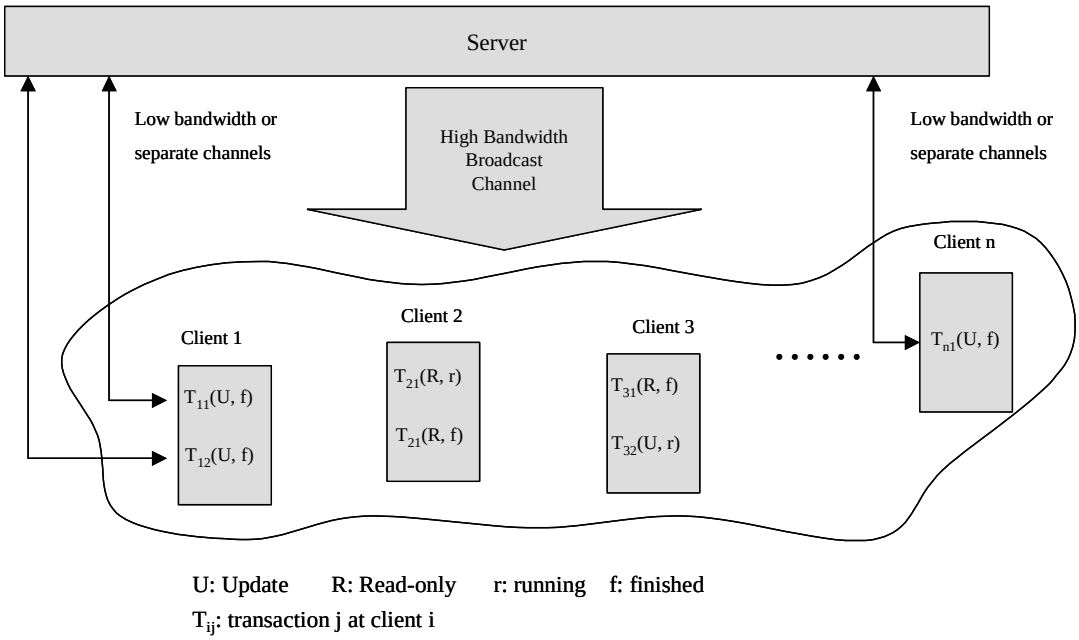


Figure 3-3 The system model for STUBcast

3.3.2 Protocol Components

STUBcast has three main components: client side Read-only Serialization Protocol (RSP), client side Update Tracking and Verification protocol (UTVP), and server side Server Verification Protocol (SVP). RSP is defined in two versions: RSP_{SS} (single serializability supported RSP) and RSP_{LS} (local serialization supported RSP).

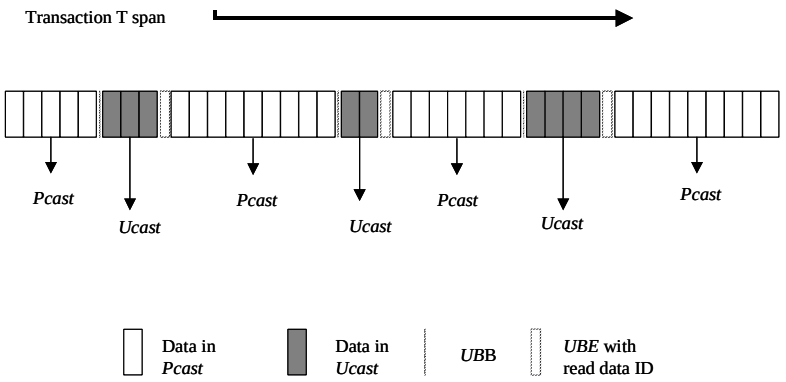


Figure 3-4 Broadcast operations in STUBcast protocol

RSP_{SS} accepts a read-only transaction if and only if it is serializable with respect to all acceptable update transactions. Similarly, RSP_{LS} accepts a read-only transaction if and only if it is serializable with respect to all acceptable read-only transactions for the same client and all acceptable update transactions. However, in order to simplify the protocol and guarantee correctness, UTVP and SVP track and verify update transactions and only accept some of the serializable ones. Together, RSP, UTVP, and SVP guarantee single and local serializability in all accepted transactions. In the following sections, we give the details of the protocols and explanations of how they work.

3.3.3 Protocol Essentials

The three protocol components of STUBcast achieve concurrency control by inserting *Ucast* operations into a server's primary broadcasting and by maintaining some essential data structures, such as server timestamp, timestamp array, no-commit flag, read ahead flag, conflict array, and RECarray. To simplify the description, all client-side transactions mentioned in this protocol essentials subsection are read-only transactions and the objective of concurrency control is single serializability. However, the strategies and data structures introduced in this subsection will also be used in update transactions and local serializability. In this subsection we will discuss the case that all update transactions committed at the server are serializable and the corresponding *Ucast* operations are sent out in the order of their conflict relations.

As each client listens to the broadcasting channel and is aware of whether it is in *Ucast* or *Pcast* stage, an executing client transaction knows about its conflict relationship with each update transaction that commits within the client transaction's spanning time. A *Server Timestamp* (the time the last update of a data unit is committed) is always saved and broadcast with each data unit in the server. At the same time, each transaction has a *Timestamp Array* in which we record the data items read by the transaction and the timestamp of the data items. When a transaction reads a data unit for the first time, it records the server timestamp attached to that data unit in the corresponding unit in the array. Since a *Ucast* on the broadcast channel implies an update transaction committed on the server, by comparing the timestamp attached with each updated data unit in that *Ucast* and the corresponding timestamp in the timestamp array, STUBcast can decide whether a client transaction read any data unit before it was changed by the update

transaction. This identifies the conflict relationship and serialization order between a read-only transaction and a committed update transaction.

Given that the *Ucast* operations are sequenced in the same order as the conflicting updated transactions, it can be asserted that all update transactions committed outside a client read-only transaction's spanning time cannot initiate any cycle in the conflict graph of that read-only transaction. This is illustrated in Figure 3-5, where no cycle can include U1 or U4. Any potential cycles can only be initiated by the update transactions that have a *Ucast* inside a read-only transaction's spanning time. We enumerate all possible cycles in Figure 3-6.

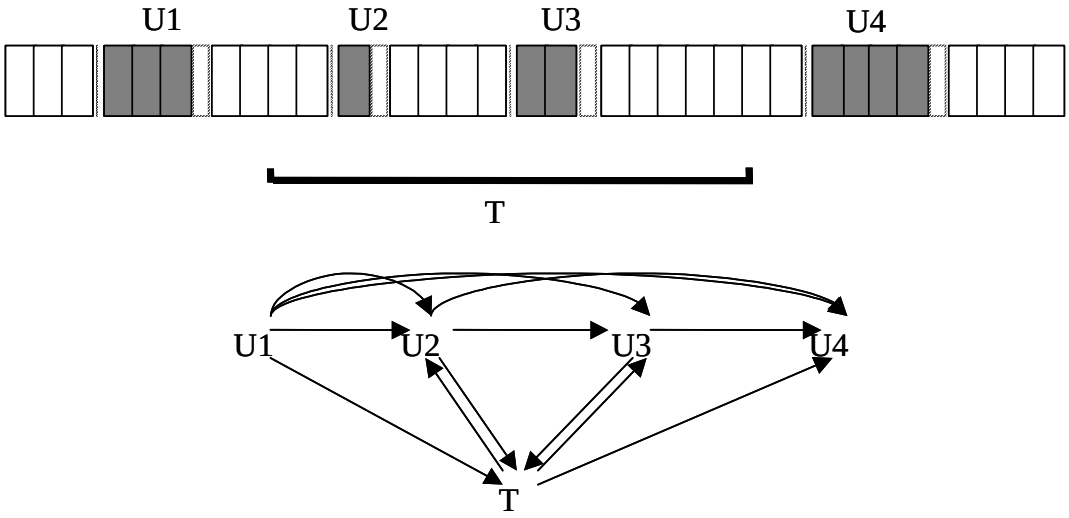


Figure 3-5 All possible conflict dependencies with a read-only transaction T

Figure 3-6 (a) and (b) illustrate the cases where a client transaction T reads data item 1 from the *Ucast* of the update transaction U, and has already read a version of data 2 earlier than U updates it (we call this a *Read Ahead*). A cycle is formed between U and T due to these two read operations. STUBcast uses a *Read Ahead Flag* to indicate whether a read ahead status is detected within a *Ucast* stage. For example, in (a), when data 2 is broadcast in U's *Ucast*, it must have a timestamp larger than the one recorded

for data item 2 in T's timestamp array. STUBcast sets T's read ahead flag to true whenever such a situation is detected. Note that the only difference between (a) and (b) is that the read ahead flag is set after in (a) or before in (b) T reads data from the *Ucast*. Suppose that in case (a) T reads data item 1 and commits, then the read ahead status would never be detected. To avoid this, a *No-Commit Flag* is set until the end of a *Ucast* to prevent T from committing if T ever reads any data within an on-going *Ucast*. STUBcast aborts a client read-only transaction if its no-commit flag and read ahead flag are both set.

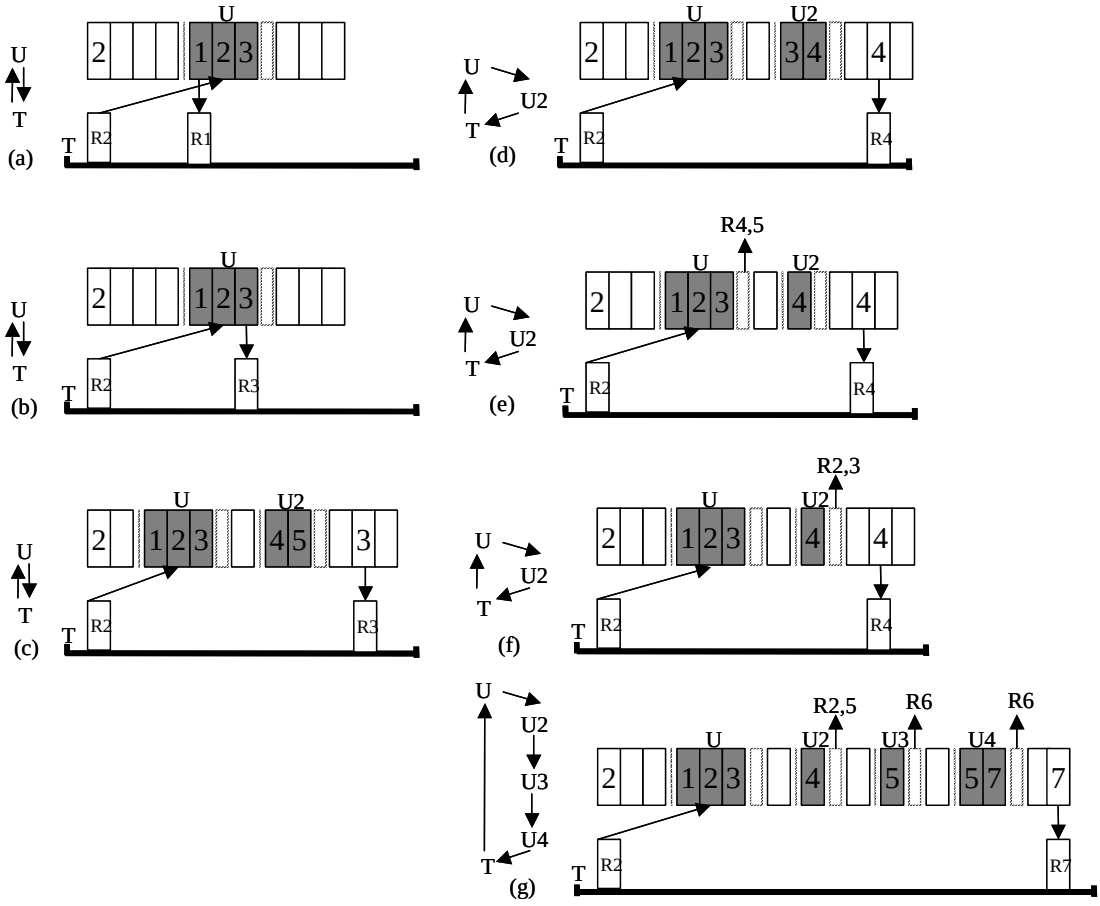


Figure 3-6 Cycles in the conflict graphs including read-only transaction T

If T reads ahead of an update by U and does not read data from U within its *Ucast*, a cycle can still exist if T reads any data U updates from a later *Pcast* (Figure 3-6 (c)). So in STUBcast, once the read ahead flag is set for T in U's *Ucast*, the data U updates are recorded in T's *Conflict Array* where each unit has two fields: update and read fields. All update fields mapping to U's *Ucast* items are set to true if a read ahead is detected for T within this *Ucast*. So later if T reads any data whose update field in T's conflict array is true, a cycle is implied and T will be aborted.

Figure 3-6 (d), (e), (f), and (g) show the cases where T reads ahead of U but never reads anything U updates. A cycle can still happen because of the indirect conflicts among update transactions. Figure 3-6 (d) shows the case where U and U2 have a write-write conflict, and then T reads from U2. Figure 3-6 (e) shows the case where U and U2 have a read-write conflict, and then T reads from U2. Figure 3-6 (f) shows the case where U and U2 have a write-read conflict, and then T reads from U2. We call these indirect conflicts a *Conflict Chain*. Finally, Figure 3-6 (g) shows the case where T only reads from a data unit updated by U4, but there is a more complicated conflict chain among U, U2, U3, and U4. The conflict array is again used to solve these conflicts.

Conflict array is based on the idea of a conflict chain, by which a chain is constructed by the time sequence of the *Ucast* of server update transactions:

- An update transaction of where a client transaction T reads ahead is added to T's conflict chain.
- An update transaction in which T does not read ahead is added to T's conflict chain if the update transaction has a write-write, read-write, or write-read conflict with the transactions in T's existing conflict chain.

A conflict array uses both update and read fields to record the data that transactions in T's conflict chain have written or read and to detect whether a new

transaction should be put in the chain. If an update transaction is in the chain, the update fields mapping to the data items in *Ucast* and the read fields mapping to the data items with the read IDs attached to the *UBE* are set to true. An update transaction of the current *Ucast* is put in the chain if T reads ahead of it. Otherwise, if any update or read field in the existing conflict array mapping to any data item in its *Ucast* is true (write-write/read-write conflicts) or if any update field mapping to any data item in its read IDs is true (write-read conflicts), an update transaction is also put in the chain.

In conclusion, timestamp array, no-commit flag, read ahead flag, and conflict array are used by STUBcast to detect whether or not a read-only transaction is serializable with respect to any accepted update transactions. It will be shown in the protocol that timestamp array and conflict array are also used to maintain local serializability. On the other hand, conflict array is not needed to serialize update transactions. Note that the arrays do not need to be physical arrays of all database items. They need only keep the information about the data units that T reads and the data units in T's conflict chain.

Finally, STUBcast uses a *RECarray* to submit update transactions to server. A RECarray is a list of records that tracks the ongoing update transaction operations. These operations are submitted to the server for additional verification and update operations.

3.3.4 RSP_{SS} (Single Serializability Supported RSP)

Figure 3-7 shows the control flow graph of RSP_{SS}. The notations used in the graph are also defined. The left part of the graph shows the policies applied to each read-only transaction's execution based on RSP_{SS}. The right part of the graph shows that each client has a *Transaction Manager* and that the policies used by the transaction manager based on RSP_{SS} are to check the serialization of each running read-only transaction.

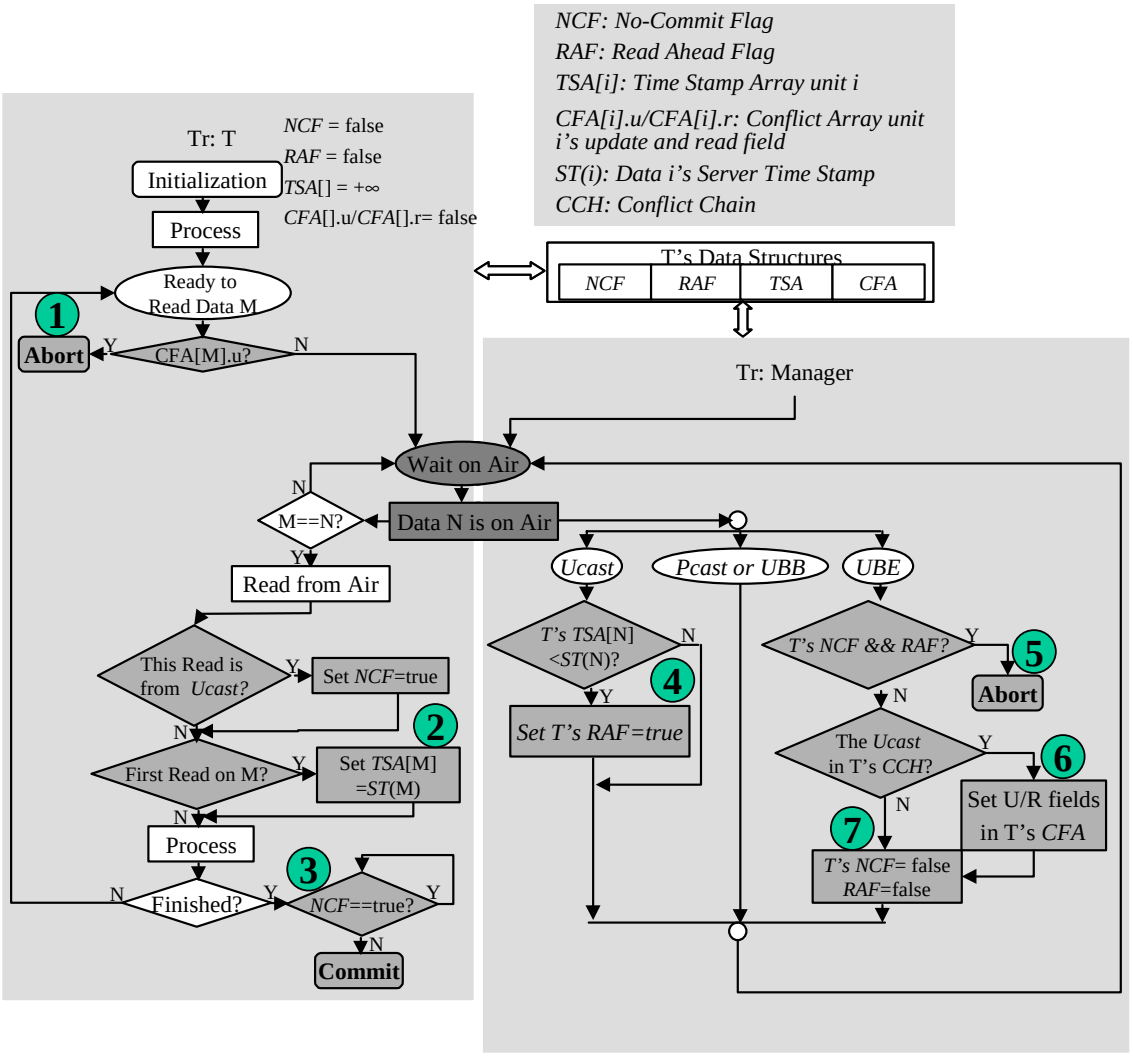


Figure 3-7 RSP_{SS} protocol

A read-only transaction consists of a sequence of read operations and it processes the data between each read. In addition to reading on-air, a read-only transaction uses its conflict array to detect non-serialization (*State 1* in Figure 3-7) statuses shown in Figure 3-6 (c), (d), (e), (f), and (g). It also sets a no-commit flag or timestamp array when necessary (*State 2*). A read-only transaction cannot commit until its no-commit flag is reinitialized to false *(State 3).*

The transaction manager always tunes in to the broadcasting channel such that it can know which stage the channel is in. It can detect and set a transaction's read ahead status when the broadcasting is in *Ucast* (*State 4*). It tests whether any non-serialization status, shown in Figure 3-6 (a) and (b), exists for each running transaction using a no-commit flag and read ahead flag when *UBE* is on air (S*tate 5*). If no non-serialization status is found for a transaction at this time, the manager updates the transaction's conflict array if the current update transaction that the *Ucast* belongs to is in the transaction's conflict chain (*State 6*). Finally, the manager reinitializes the transaction's no-commit and read ahead flag (*State 7*).

3.3.5 RSP_{LS} (Local Serializability Supported RSP)

The major difference between RSP_{LS} and RSP_{SS} is that it needs to detect any non-serialization status caused by combining all local read-only transactions to all update ones. Figure 3-8 shows all possible types of conflict cycles caused by multiple read-only transactions. To detect these cycles, RSS_{LS} keeps the history of timestamp arrays and conflict arrays of all executing or executed read-only transactions on the same client. We call these histories *TSAList* and *CFAList*. Moreover, each conflict array unit extends to have a timestamp field. When a data unit's update field in the array is set as true, the timestamp attached to the data unit in the corresponding *Ucast* is put in its timestamp field. The additional strategies and steps used for RSP_{LS} on a transaction T are given in Table 3-1. Similar notations of Figure 3-7 are used in the table.

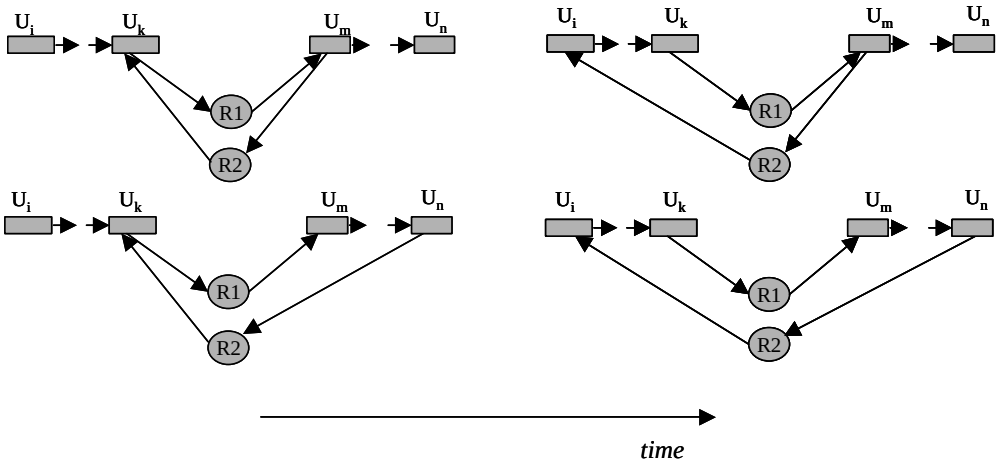


Figure 3-8 Possible conflict cycles caused by multiple read-only transactions

Table 3-1 RSP_{LS} protocol

Condition	T reads a new data unit k successfully and at least one update field of T's *CFA* is true (similar to transaction R2 in Figure 3-8 and $U_m \rightarrow T \rightarrow U_i$ /U_k or $U_n \rightarrow T \rightarrow U_i$ / U_k)
Strategy (Used by the transaction, not the transaction manager)	1. Search in *CFAlist*, check if there is any *CFA*'s (other than T's *CFA*) *CFA*[k].u is true. (Check whether there is a read-only transaction R that $R \rightarrow U_m \rightarrow T$ or $R \rightarrow U_m \ldots \rightarrow U_n \rightarrow T$) 2. Continue transaction if no true value is detected in Step 1. (R does not exists) 3. Otherwise (R exists), for each *CFA* that has a true *CFA*[k].u field, locate its owner transaction's *TSA* in *TSAList* (here name it *TSA1*). Then for each true update field *CFA*[j].u of T's *CFA*, compare to see if *CFA*[j].timestamp of T ≥ *TSA1*[j]. If any such comparison results are true ($T \rightarrow U_k \rightarrow R$ or $T \rightarrow U_i \ldots \rightarrow U_k \rightarrow R$ exists, so R is R1 type), abort T.

Note that RSP_{LS} does not actually need to keep all executed transactions timestamps and conflict arrays history. A variable *MinCFATime* can be used to record the minimum of all timestamp field values of all conflict arrays that belong to the executing transactions. Then each executed transaction's timestamp array maintains a variable *MaxTSATime* that records the maximum timestamp value in the array. If any executed

transaction's *MaxTSATime* becomes less than *MinCFATime*, the timestamp and conflict arrays of that transaction are removed from the history.

3.3.6 UTVP (Update Tracking and Verification Protocol)

Update transactions execute at client sites and all write operations are done locally at the client execution stage. The update tracking and verification protocol (UTVP) tracks local written values and other important operations in a data structure *RECarray* and submits the array to the server for updates at the server and for commit operations. It also detects possible non-serialization status at the client site. After RECarray is submitted to the server, it waits for the server's reply to see whether or not the server accepts it.

Table 3-2 Track operations for update transactions

Operation (on data item m)	Policy
First write	Add (m, value, "W") to RECarray
Non-first write	Update (m, value, W) in RECarray to (m, newvalue, "W")
Local read	Do nothing
First non-local read	Add (m, timestamp of m, "R") to RECarray
Non-first non-local read	Do nothing

Table 3-3 Update transaction verification in UTVP

Transaction:	Transaction Manager:
• Track first write/non-first write/first non-local read as in Table 3-2 • If non-local read in *Ucas*t, set *NCF* = true • If first non-local read at m, set *TSA*[m] = *ST*(m) • When finished and *NCF* is set, wait until *NCF* is reset • When finished and *NCF* is reset, submit RECarray to the server	• *Pcast*: No action • *UBB*: No action • *Ucast* (n is on air): If *TSA*[n]< *ST*(n), set *RAF*=true and abort • *UBE*: reset *NCF*

A write operation on one data item in an update transaction can be a *first write* or a *non-first write*. Only the latest written value of each data update by the transaction is recorded and submitted to the server. A read operation on a data item can be a *local read* (read that data unit's locally written value), a *first non-local read* (first time read on that data unit from on-air) , or a *non-first non-local read* (read of that data unit from on-air again). Table 3-2 gives the tracking operations in constructing RECarray.

The verification part of UTVP is based on the framework for RSP shown in Figure 3-7, where the protocol is applied to both the update transaction and the transaction manager. On the transaction side, tracking steps are taken when first write, non-first write, or first non-local read happens. The no-commit flag is still set when a non-local read receives the data value in a *Ucast* stage. On the transaction manager side, an update transaction's timestamp array and read ahead flag are also used to detect read ahead status. Different from RSP, once read ahead is detected, an update transaction is aborted, so conflict array and conflict chain are not needed for update transactions. Because of this, UTVP only accepts part but not all of serializable transactions. This strategy simplifies the solution and is necessary to guarantee the correctness of STUBcast [HUA01b, HUA01c]. Table 3-3 gives the major update verification operations.

3.3.7 SVP (Server Verification Protocol)

A transaction submitted by UTVP to the server is serializable to all update transactions committed before its completion [HUA01b]. However, because of the communication and queuing delays, whether or not a transaction is also serializable to other update transactions committed after its completion also needs to be checked. This is why a server verification protocol (SVP) is needed in addition to just updating the server database using the locally written values. The SVP verifies a received update transaction

by a *First-Come-First-Served* sequence. Verification is based on update transaction's RECarray. The policies used by SVP are as follows:

Step 1. Check each *first non-local read* operation (m, timestamp of m, “R”); if the timestamp has a value less than the one currently saved in the database, send an "Abort" reply to the client.

Step 2. If Step 1 is completed without aborting, update the server database using the value in each (m, value, “W”) record. Attach the current server timestamp (same for all operations) to each data item. Send a "Commit" reply to client.

Step 3. Insert a *Ucast* based on the write tracks into *Ucast*, attach the IDs of all first non-local reads to *UBE*.

All replies sent to an update client are sent over the separate channel and are not broadcast.

3.4 Correctness Proof

In this section, we prove the correctness of STUBcast on supporting single and local serializability. The proof is based on the Serialization Theorem [BER87, OZS91] and the theorem that a group of transactions are serializable if and only if their conflict graph has no cycles [BER87, OZS91]. Our objective is to prove the following theorems.

Theorem 1. All transactions accepted by RSP_{SS} (RSP_{LS}), *UTVP*, and *SVP* conform to single (local) serializability.

Theorem 2. A read-only transaction is accepted by RSP_{SS} (RSP_{LS}) *if and only if* it conforms to single (local) serializability.

These theorems can be proved based on the following lemmas.

Lemma 1. All update transactions accepted by *UTVP* and *SVP* are serializable.

We use induction to prove it. When there is only one transaction T_1 accepted by *UTVP* and *SVP*, it is serializable. Assume when n-1 transactions $T_1 \ldots T_{n-1}$ are accepted, they are still serializable. We now prove that when T_n is accepted, $T_1 \ldots T_n$ are still serializable.

Let C_{n-1} be the conflict graph of $T_1 \ldots T_{n-1}$. C_{n-1} is acyclic because $T_1 \ldots T_{n-1}$ are assumed to be serializable. Construct C_n (conflict graph of $T_1 \ldots T_n$) by adding node T_n and drawing edges based on T_n's operations to C_{n-1}.

First, add edges for all T_n's write operations. For each write operation, edges can only be drawn from any transaction in $T_1 \ldots T_{n-1}$ to T_n because they commit earlier than T_n based on *UTVP* and *SVP* and they must read/write any conflict data unit before T_n writes to it in the database. Thus all edges added based on T_n's write operations are toward T_n and cannot initiate loops.

Next, add edges for all T_n's read operations that read from on-air. For each such read, edges also can be drawn only from transactions in $T_1 \ldots T_{n-1}$ to T_n because they must have written the data unit before T_n reads it from on-air if any conflict exist. Assume some transactions in $T_1 \ldots T_{n-1}$ do write some data unit after T_n reads it from on-air, then *UTVP* or *SVP* must have rejected T_n because a read ahead status would be detected by *UTVP* (*TSA[n] < ST(n)*) or SVP (the *first non-local read* record (m, timestamp of m, "R") has a timestamp less than the current one in the database). So all edges added based on T_n's reading from on-air are toward T_n and cannot initiate loops.

Finally, add edges for all T_n's read operations that read locally. No such edges are actually added because once a transaction in $T_1 \ldots T_{n-1}$ conflicts with T_n's local read, it

must also conflict with T_n's write operation that writes to that local data unit and the edge has already been drawn. Therefore, based on local read operations, no loop appears.

In conclusion, no loop appears when constructing C_n, so $T_1 \ldots T_n$ are serializable, and Lemma 1 is proved.

Lemma 2. A read-only transaction accepted by RSP_{SS} is serializable with all update transactions accepted at server.

Assume $T_1 \ldots T_n$ are all accepted update transactions and read-only transaction R is accepted by RSP_{SS}. Let C_n be the conflict graph of $T_1 \ldots T_n$. Based on Lemma 1, C_n must be acyclic. Now construct C_nR, that is, the conflict graph of R and $T_1 \ldots T_n$, by adding node R and drawing edges in C_n. C_nR remains acyclic when node R is added. For each read operation in R, draw edges for all its conflicts (only read-write, write-read conflicts) with any transaction in $T_1 \ldots T_n$. Now we use induction to prove Lemma 2.

First, assume the first read operation of R read data item M and divide $T_1 \ldots T_n$ into three sets. $Tset_1$ is the set of transactions that update M before R reads it; $Tset_2$ is the set of transactions that update M after R reads it; $Tset_3$ is the set of transactions that do not write to M. Edges from all $Tset_1$ transactions to R and edges from R to all $Tset_2$ transactions are added to C_nR. These new edges can only initiate loops when there already exists at least one edge from any transaction in $Tset_2$ to some transaction in $Tset_1$. But this is not possible, because this edge and edges from all $Tset_1$ transactions to all $Tset_2$ transactions (these edges are in this direction because transactions in both sets all write to M, and transactions in $Tset_1$ obviously write earlier than transactions in $Tset_2$ since R reads between them) would have made C_n cyclic earlier. Therefore, C_nR remains acyclic after drawing edges for R's first operation.

Next, assume C_nR still remains acyclic until after the edges for R's next-to-last operation are added. Now we prove C_nR is acyclic after drawing edges for R's last operation. As in the first step, assume this operation reads data item K and divide T_1 … T_n into $Tset_1$, $Tset_2$, and $Tset_3$. New edges from $Tset_1$ to R and from R to $Tset_2$ are added. There are seven possibilities (Figure 3-9) for a loop to appear because of these new edges.

Possibility 1. This requires an edge from a transaction in $Tset_2$ to a transaction in $Tset_1$ to form a cycle. This is not possible because, as stated above, this would have made C_n cyclic earlier.

Possibility 2. This requires an edge from R to a transaction in $Tset_1$ in C_nR. This implies R reads some other data units earlier than at least one transaction in $Tset_1$ that updates it. Assume R reads data unit L earlier than T in $Tset_1$ updates it. Since R reads K after T's update, based on RSP_{SS}, R will not commit until T's *Ucast* finishes. This is true because *NCF* is set when R reads K within T's *Ucast* (Case 1), or because K is requested by R after T *Ucast* K (Case 2). The R' *RAF* is set within T's *Ucast* since R reads L earlier than T updates it ($TSA[L] < ST[L]$). So, in Case 1, R will be aborted because both *NCF* and *RAF* are true, which can be detected by RSP_{SS}. In Case 2, since *RAF* is set and K is broadcast in T's *Ucast* (T writes to K), *CFA[K].u* is set as true based on RSP_{SS}. When R tries to read K later, it will detect *CFA[K].u* is true and abort R. So R will not be accepted by RSP_{SS}, which makes Possibility 2 impossible. Because of the same reasons, Possibility 3 (when there is any edge from any transaction in $Tset_2$ to R in C_nR) also cannot happen.

Possibility 4. This requires an edge from to a transaction in $Tset_3$ and from the same transaction in $Tset_3$ to some transaction in $Tset_1$. If this is true, assume there is an

edge from R to a transaction T_3 in $Tset_3$ and then an edge from T_3 to a transaction T_1 in $Tset_1$. An edge from R to T_3 mean R reads some other data unit earlier than T_3 updated it. There are two possibilities between T_1 and T_3: T_1 committed before T_3, or T_3 committed before T_1. In the first case, the edge from T_3 to T_1 cannot be caused by a write-to-write or write-to-read conflict because of the commit sequence. The only possibility is a read-to-write conflict, which is also not possible because T_3 would have been aborted by *UTVP* or *SVP* because a read ahead can be detected. In the second case, the conflict between T_3 and T_1 can be write-to-write, write-to-read, or read-to-write. In RSP_{SS}, once R reads ahead of T_3, i.e., T_3's update to some data units that R has read is detected, T_3's update and read sets are recorded in R's *CFA*. The update sets of any other later update transactions that have a conflict with T_3, such as T_1, must also be recorded in R's *CFA*. So if R later reads K from T_1, R's *CFA[K].u* must be true (i.e., R would have been aborted). This contradiction shows Possibility 4 is not possible. Based on the same reasons as for Possibility 4, Possibility 5 (when there is any edge from $Tset_2$ to any transaction in $Tset_3$ and from the same transaction in $Tset_3$ to R) is also not possible.

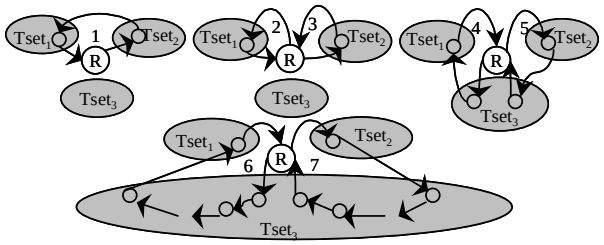


Figure 3-9 Possible loops caused by a read operation

Possibility 6. This requires an edge from R to any transaction in $Tset_3$ and from the same transaction in $Tset_3$, going through some other transactions in $Tset_3$ and then to

a transaction in $Tset_1$. This is similar to Possibility 4, except that there are the edges through the intermediate transactions between T_3 and T_1 that participated in the loop. However, R's *CFA* keeps track of these edges because RSP_{SS} records all update transactions' update and read sets if they conflict with R's old *CFA*. Then when R reads K, RSP_{SS} would have aborted R since *CFA[K].u* must be true. Therefore, Possibility 6, as well as Possibility 7 (when there is any edge from $Tset_2$, going through some other transactions, to a transaction in Tset3 and then from the same transaction in $Tset_3$ to R) with the same reasons, is not possible,

Since there is no possibility for a loop when constructing C_nR, R is serializable with $T_1 \ldots T_n$. Lemma 2 is proved by induction.

Lemma 3. All read-only transactions from one client accepted by RSP_{LS} are serializable with all update transactions accepted at the server.

The only difference between proving Lemma 2 and proving Lemma 3 is, for lemma 3 we have to prove the loops in Figure 3-8 cannot happen under RSP_{LS}. The existence of both R_1 and R_2 types of transactions (based on their relation to update transactions) is the reason for the loops in Figure 3-8. The RSP_{LS} protocol detects whether R_1-type transactions exist whenever R_2-type transactions are detected. RSP_{LS}'s condition holds whenever an R_2-type transaction is found. By using *CFAList*, Step 1 of RSP_{LS} can decide whether there are edges from any other read-only transactions directly to, or going through other update transactions to, the one R_2 reads from, because *CFA* can keep track of any conflicts among update transactions. Not detecting such edges in Step 1 means no R_1-type transaction exists. However, if such edges appear for some other read-only transaction, R_1-type transactions do exist. We call any such transaction R_1. Then, in Step

2, the system checks whether or not there are edges from any update transactions, from which R_2 reads some data ahead directly or indirectly, to any update transactions an R_1 reads from. The RSP_{LS} achieves this goal by using R_2's *CFA* and R_1's *TSA* to detect conflicts among them and using their timestamps to decide the direction of the conflicts. If such edges are found in Step 2, R_2 is aborted. So by using RSP_{LS}, any possible loops shown in Figure 3-8 can be prevented, which implies all read-only transactions accepted from the same client are serializable with all update transactions. Lemma 3 is proved.

Lemma 4. All read-only transactions conforming to *SS* (*LS*) are accepted by RSP_{SS} (RSP_{LS}).

From the protocol description, we can conclude that RSP_{SS} and RSP_{LS} only abort a read-only transaction when a loop exists in the conflict graph of one read-only transaction (or all read-only transactions from one client) and all other serializable update transactions. Since all read-only transactions conforming to single (local) serializability do not cause such loops, they will all be accepted by *RSP*. Lemma 4 is proved.

Finally, Theorem 1 can be proved by Lemmas 1, 2, and 3. The *if* part of Theorem 2 is proved by Lemma 4, while the *only if* part can be proved by Lemmas 2 and 3.

3.5 Simulation

3.5.1 Simulation Configuration

This section presents our simulation results for STUBcast with RSP_{SS}, UVP, and SVP. Average response time and average restart times of transactions are the major performance measurements. Table 3-4 shows the important configuration parameters and their definitions.

Read-only and update transactions are simulated based on the ratios defined in the table. Each transaction in the simulation consists of several data access operations and computation operations before or after each data access. A transaction's length is the number of access operations in it. MAX_TRAN_LEN represents the maximal transaction length in the simulated transactions; the transaction length is uniformly distributed. Therefore, the average length of all transactions is MAX_TRAN_LEN/2. The computation time between two access operations is the operation inter-arrival time. Time between the starts of two consecutive transactions in a simulated system is denoted as transaction inter-arrival time. Each simulation run uses only one client. Large numbers of clients are emulated by using a small average transaction inter-arrival time from one client. As one of the two data access distribution modes, NON_UNIFORM divides the database into n parts and simulates a situation where some parts of data are accessed more often than others. To see how the protocols influence the performance, for each configuration the average response time is also recorded without any concurrency control and compared with the value obtained under STUBcast.

A set of simulations is run under different combinations of PCAST_SCHEME, DB_SIZE, ACCESS_DISTR, TR_INTARR_TIME, and MAX_TRAN_LEN. When ACCESS_DISTR is set to NON_UNIFORM, the simulations divide the database into n parts, and the access possibility ratio conforms to:

$$\frac{P_i}{P_{i+1}} = 4 \quad 1 \le i \le n . \tag{3-1}$$

Consequently, when MULTIDISK [ACH95a] is used as the PCAST_SCHEME, we use the optimal frequency assignment scheme derived from [HAM97, VAI97a,

VAI97b], which concludes that the minimum overall average single data response time can be achieved when:

$$F_i \propto \sqrt{\frac{P_i}{L_i}}\,. \tag{3-2}$$

Table 3-4 Simulation Configuration

Parameter name	Definition	Value
BcastUnit	The time used to broadcast a data unit (fixed size)	20
UBBunit	The time used to broadcast *UBB*	1
UBEunit	The time used to broadcast *UBE*	10
READ_TO _UPDATE	Ratio of read-only to update transactions	2:1
READ_TO _WRITE	Ratio of read-to-write operation in update transactions	2:1
NONLOC_TO _LOC	Ratio of non-local to local read in update transactions	4:1
DATA_TO _ID	Ratio of data value's length to the ID/timestamp's length	32:1
DOWN_TO_ UP	Ratio of downlink bandwidth to uplink bandwidth	8:1
OP_INTARR_ TIME	Mean operation inter-arrival time (Poisson distribution)	1
NUM_OF _TRANS	Number of transactions simulated	5000
PCAST_ SCHEME	Primary schedule scheme	FLAT, MULTIDISK
DB_SIZE	Number of items in the database	100, 1000 (D)
MAX_TRAN _LEN	Maximal transaction length	4,8,12,16,20,24 (L)
TR_INTARR _TIME	Mean transaction inter-arrival time (Poisson distribution)	50, 100 or 500 (Tr)
ACCESS_ DISTR	Distribution mode of data accesses	UNIFORM or NON_UNIFORM [n = 4, P_i:P_{i+1} = 4:1, $1 \le i \le n$]

Because we assume the same length among all data, the optimal broadcast frequency in the simulations is obtained when:

$$\frac{F_i}{F_{i+1}} = 2 \quad 1 \le i \le n\,. \tag{3-3}$$

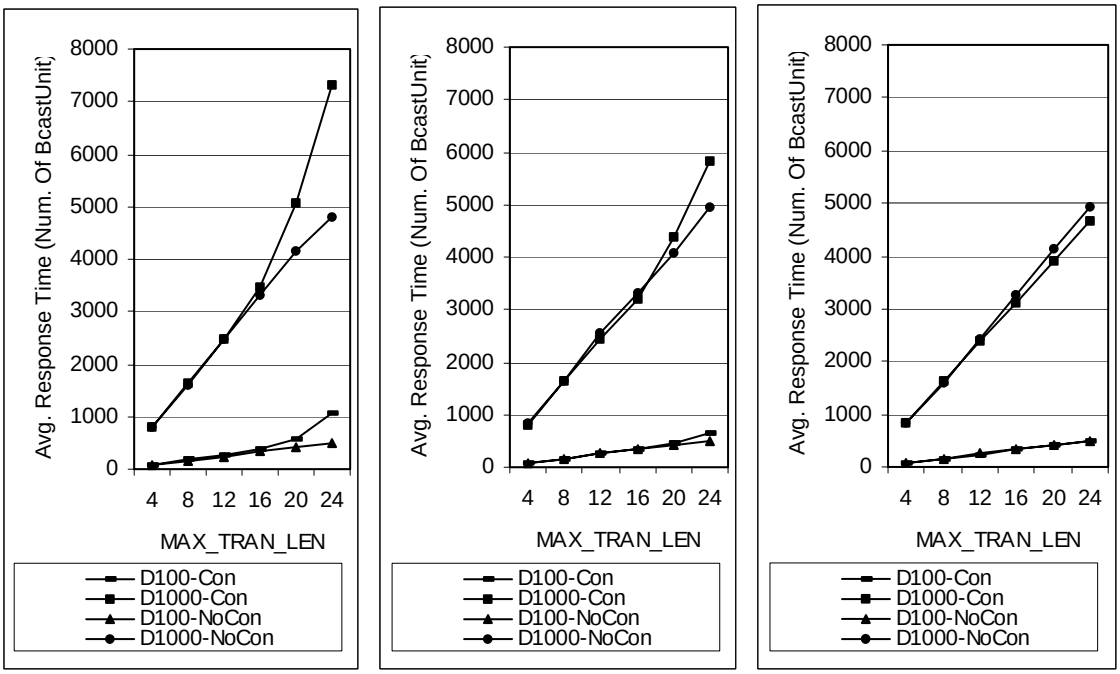
We then use this result to assign relative frequencies among all the disks [ACH95a] when MULTIDISK scheduling is used.

3.5.2 Simulation Results

3.5.2.1 Average response time

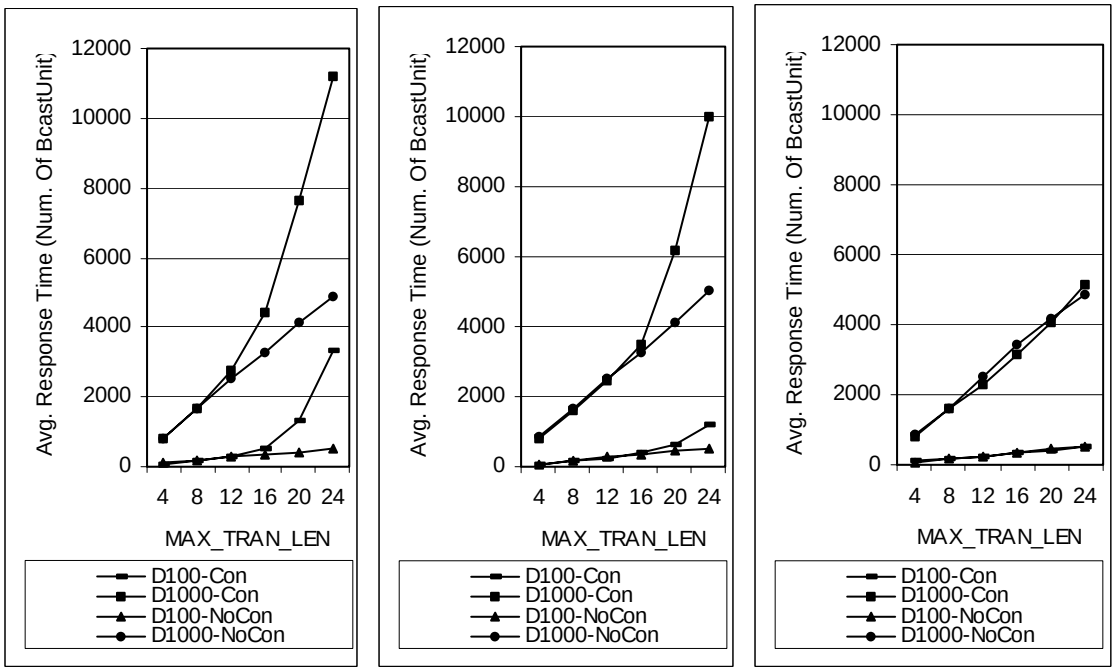
Figures 3-10, 3-11, and 3-12 show the performance of average response time compared to MAX_TRAN_LEN among simulated transactions under both STUBcast and no concurrency control. Each simulation run records the response time of all transactions, which is the time period between a transaction's invoked time and the time it commits. It can include the duration of multiple runs of a transaction if the transaction has ever been aborted and restarted. Average response time is the average response value among all transactions without considering the transaction lengths.

These figures show the results under UNIFORM/FLAT, NON_UNIFORM/FLAT, and NON_UNIFORM/MULTIDISK with different transaction inter-arrival times. It shows that the response time increases with DB_SIZE, MAX_TRAN_LEN and decreases with the TR_INTARR_TIME under both STUBcast and no concurrency control. It also shows that when FLAT scheduling is used with NON_UNIFORM mode, the response time becomes greater than with other configurations because hotter data is broadcast at the same rate as cold data. The performance is greatly improved when MULTIDISK is used with NON_UNIFORM mode. Table 3-5 shows the performance distribution according to MAX_TRAN_LEN of all simulated cases considering the difference between using STUBcast and no concurrency control using the same configurations. We consider the average response time under no concurrency control as optimal since it ensures that no restart can happen.



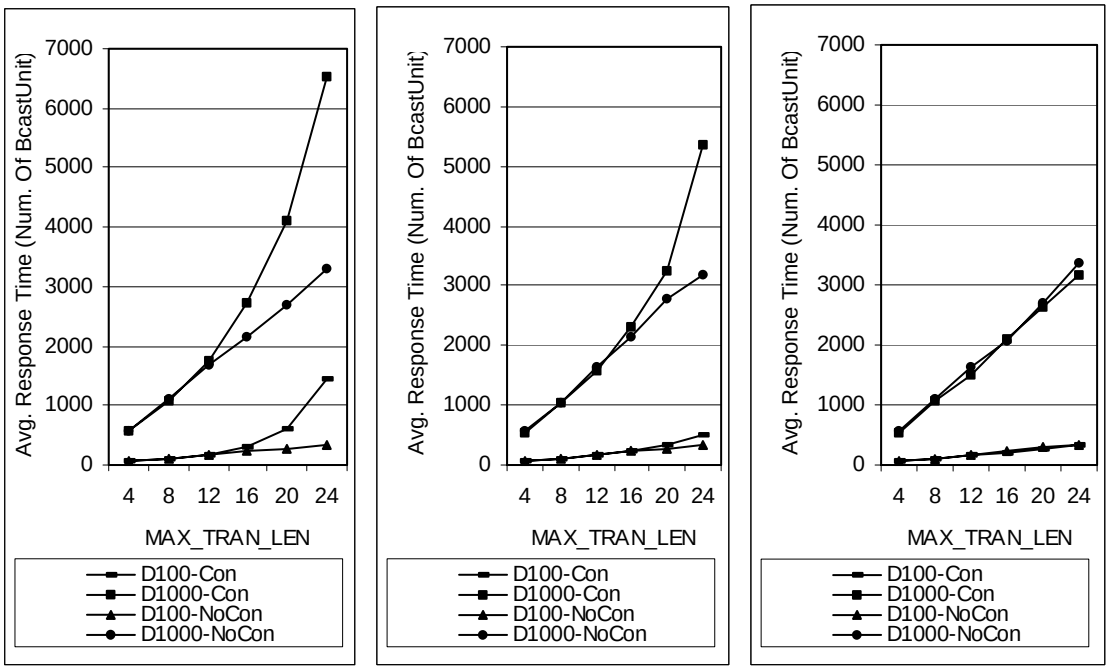
(a) (b) (c)

Figure 3-10 Average response time under UNIFORM data access and FLAT scheduling. a) Tr50; b) Tr100; c) Tr500



(a) (b) (c)

Figure 3-11 Average response time under NON_UNIFORM data access and FLAT scheduling. a) Tr50; b) Tr100; c) Tr500



(a) (b) (c)

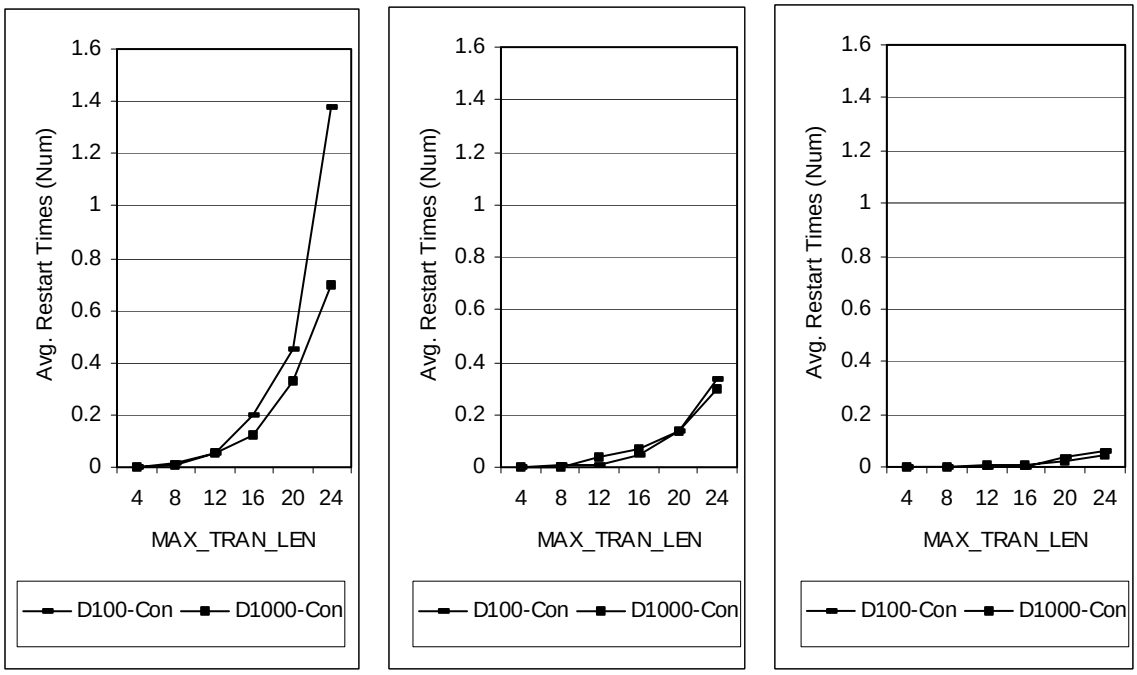
Figure 3-12 Average response time under NON_UNIFORM data access and MULTIDISK scheduling. a) Tr50; b) Tr100; c) Tr500

Table 3-5 Average response time performance distribution among simulated cases

Performance Trans. len	Same (Difference < 5%)	Similar (Difference < 10%)	Very well (Difference < 50%)
≤ 12	100%	100%	100%
≤ 16	92.6%	95.2%	100%
≤ 20	81.2%	87.8%	96.7%
≤ 24	72.3%	79.8%	88%

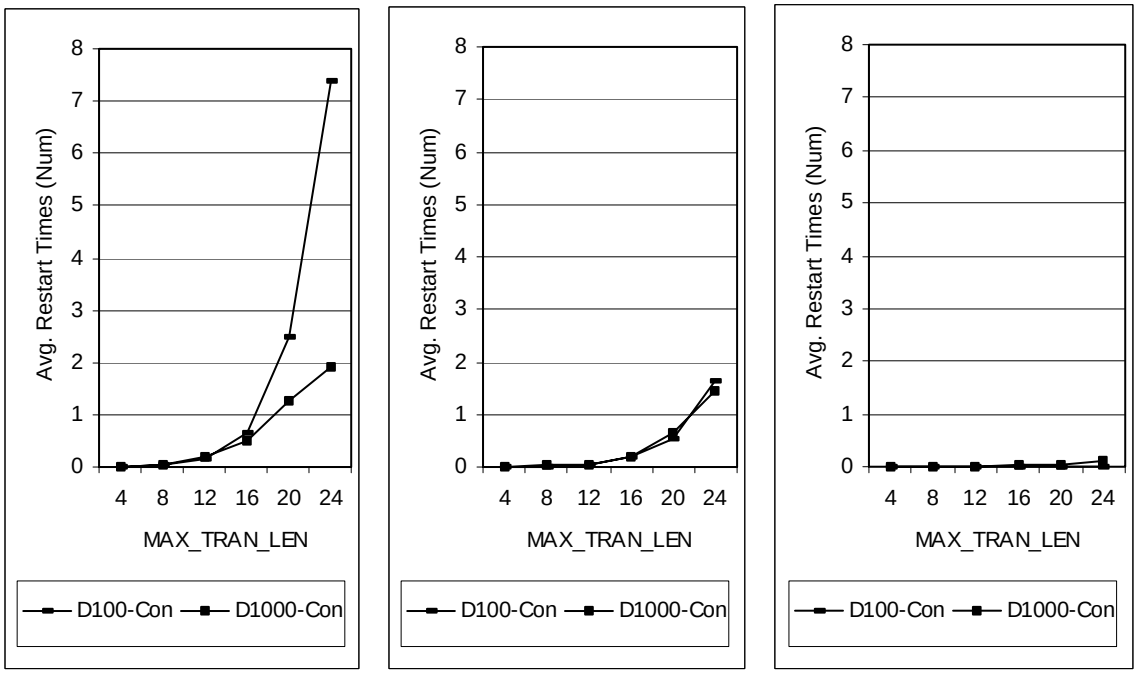
3.5.2.2 Average restart times

Figures 3-13, 3-14, and 3-15 show the average restart times among all transactions when STUBcast is used. This performance measurement counts the average number of time a transaction needs to restart under certain configuration using STUBcast.



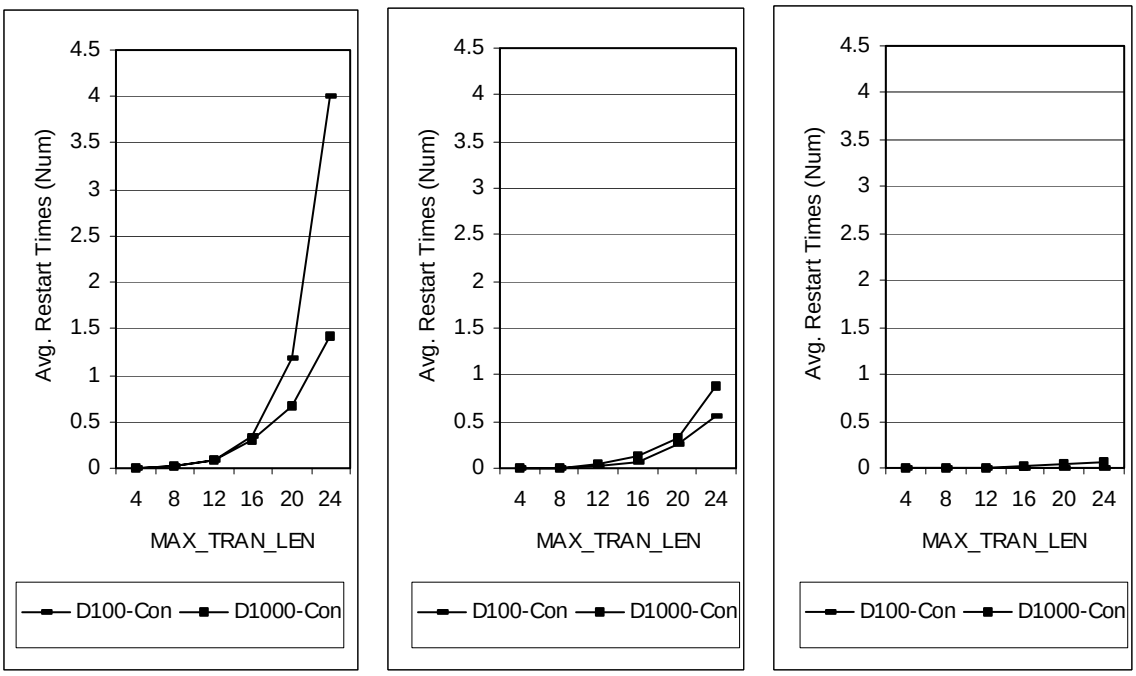
(a) (b) (c)

Figure 3-13 Average restart times under UNIFORM data access and FLAT scheduling. a) Tr50; b) Tr100; c) Tr500



(a) (b) (c)

Figure 3-14 Average restart times under NON_UNIFORM data access and FLAT scheduling. a) Tr50; b) Tr100; c) Tr500



(a) (b) (c)

Figure 3-15 Average restart times under NON_UNIFORM data access and MULTIDISK scheduling. a) Tr50; b) Tr100; c) Tr500

These figures reflect the results given in Table 3-6, which implies that, in most cases, STUBcast introduces low transaction restart rates in the simulations. We also conclude that a higher number of restarts happen using the NON_UNIFORM mode. This is true because NON_UNIFORM mode introduces higher data conflict rates especially when transaction inter-arrival rate is relatively high.

Table 3-6 Average restart times performance distribution among simulated cases

Avg. Restart Times	Percentage of Cases
≤ 2	97.3%
≤ 0.5	83.4%
≤ 0.2	76%

3.5.2.3 Analysis of the results

We can conclude the following based on the simulation results and the information in Tables 3-5 and 3-6.

- When the MAX_TRAN_LEN is less than or equal to 12, STUBcast performs the same as using no concurrency control (optimal) under all configurations.
- STUBcast performs similarly to no concurrency control (near optimal) in nearly all configurations when MAX_TRAN_LEN is less than or equal to 16.
- STUBcast performs very well in nearly all configurations when MAX_TRAN_LEN is less than or equal to 20.
- STUBcast performs less well when transaction length is very large, transaction inter-arrival rate is very large, and when the scheduling algorithm design is not based on the data requests (e.g., when FLAT is used with NON_UNIFORM).
- STUBcast performs excellently when TR_INTARR_TIME is relatively large. In our simulations, it has optimal performance in all cases when Tr is set at 500 BcastUnits. In most cases it also has optimal performance when Tr is set at 100 BcastUnits.

We observe that having a MAX_TRAN_LEN less than or equal to 12, 16, or 20 is very applicable in many transaction-based applications in a broadcast environment. For example, a stock quote/exchange or auction application is most likely to have a maximal transaction length of ten. Therefore, STUBcast can have near-optimal performance in these applications even without considering other configurations. On the other hand, for applications with relatively large transaction inter-arrival time (such as Tr100 or Tr500 in our simulations), a near-optimal performance can also be achieved with long transactions under any other configurations by using STUBcast.

In conclusion, STUBcast can have very good performance in real applications because systems having a maximal transaction length less than or equal to 12 (or 20), or transaction inter-arrival time larger than 100 (or 500) are very common.

3.6 Conclusion

STUBcast protocol outlined in this chapter meets the requirements of the defined problem because it is designed for broadcast-based transaction processing; it supports both client read-only and update transactions; it has low broadcast overhead since data ID and server timestamp are the only control information; it needs very little computation overhead and is very simple. Theorem 1, that STUBcast correctly supports the criteria of single and local serializability, was proved. The use of these two relaxed criteria can reduce unnecessary aborts in read-only transactions. Theorem 2, that STUBcast guarantees a read-only transaction's acceptance *if and only if* the transaction conforms to single or local serializability, was also proved. STUBcast also has an advantage that different client sites may run RSP_{SS} or RSP_{LS} in the same system without interference, depending on the local needs. Simulation results also show that STUBcast can perform very efficiently in real applications.

CHAPTER 4
PROBLEMS INTRODUCED BY STUBCAST

Chapter 3 outlined the design and performance of STUBcast, showing it to be a new and efficient concurrency control protocol for transaction processing in broadcast-based asymmetric applications. Chapter 2 noted that broadcast-based applications need efficient scheduling, cache, and index strategies to improve performance. Existing designs might not fit into STUBcast directly or work most efficiently with it, however. In order to allow applications to use STUBcast and take advantage of its efficiency, these strategies should adapt to STUBcast's solution model instead of simply using existing solutions. Conversely, the primary protocol of STUBcast also needs to be adjusted to work with these components. Therefore, in this chapter we address problems associated with STUBcast protocol; these adjust the design of cache, data scheduling, index, and STUBcast to allow them work together in transaction-processing applications.

4.1 STUBcache

STUBcast protocol introduced in last chapter does not assume the use of cache. Consequently, a transaction always needs to wait on the channel for the next needed data unit, which can introduce intolerable response time. Therefore, integrating STUBcast with cache is very desirable.

4.1.1 Basic Rule of STUBcache

We call the following cache designed to work under STUBcast a STUBcache. As is true of any general cache scheme, STUBcache provides data to a transaction’s read

operations directly. However, when a requested data unit is not in the cache, STUBcache tries to fetch it from on-air. For every data unit accessed from on-air by some transaction, STUBcache considers whether it should be inserted into the cache. If the cache is not full, this data unit is put into the cache directly; otherwise STUBcache uses a replacement policy to judge whether any data in the current cache should be replaced by this new data unit.

In addition to the above general scheme, we designed the following policies of STUBcache to adjust itself to STUBcast protocol (Figure 4-1). These STUBcache policies allow STUBcast to handle transactions as if they always read on-air.

- STUBcache saves the timestamp for each cached data unit.
- The cache is always checked for each data unit in the *Ucast* stage.
    - If a data unit in *Ucast* is already in the cache, it will be updated by the new data unit with the new timestamp.
    - All cached data appearing in this *Ucast* are marked.
    - If any transaction reads a marked data unit before the *Ucast* finishes, the transaction's no-commit flag is set.
    - All marks are removed after the *Ucast* finishes.

Figure 4-1 STUBcache policies

4.1.2 Replacement Policies for STUBcache

STUBcache requires a replacement policy to replace cached data when necessary in order to provide a good transaction performance response time. Chapter 2 discusses such schemes as P, PIX for broadcast data accessing, which we can use in STUBcache.

However, we observe that STUBcast has some special features that we should consider when using cache.

First, read ahead status is always the direct or indirect reason for a transaction's abort. A data unit updated frequently triggers a read ahead status easily if it is cached. If such a data unit is cached, a transaction will read it directly from the cache first. At the same time, since this data unit is updated frequently, it will appear in an upcoming *Ucast* with higher probability. This results in a higher probability of triggering a read ahead status for the transaction that reads the same data unit from the cache using STUBcast. Thus, caching a frequently updated data unit makes it easier to initiate read ahead status, which can easily lead to aborting a transaction.

Moreover, caching a data unit already in some transaction's update fields of conflict chain is meaningless for that transaction. If this transaction needs this data unit at a later stage (by the transaction model, the operations are not known in advance), STUBcast will abort this transaction. If it does not need this data unit, caching the same data unit is useless. Therefore, the involvement of a data unit in a transaction's conflict chain can influence the performance of the cache.

Corresponding to the possible influence of read ahead status and conflict chain of STUBcast, we give the following two designed replacement policies (RxU and CiR).

4.1.2.1 Read and update most (RxU)

This policy requires that STUBcache replace the data in the cache if it has the highest value of RxU and this RxU value is larger than the one on-air.

$$\text{RxU} = \text{Average Number of Requests (R)} \times \text{Average Number of Updates (U)}.$$

“Average number of requests” (R) of a data unit is the accumulated number of requests for transactions on the data unit in a given time period. “Average number of updates” (U) of a data unit is the accumulated number of updates on the data unit from *Ucast* in a given time period.

This policy suggests replacing a data unit that is likely to trigger read ahead status in transactions because it is both read and updated more often. Avoiding read ahead status can help reduce transaction aborts.

4.1.2.2 Conflict-chain inverse read most (CiR)

This policy requires STUBcache to replace the data unit in the cache if it has the highest value of CiR and this CiR value is larger than the one on-air.

CiR = Number of Conflict Chains Involved (C) / Average Number of Requests (R).

"Number of conflict chains involved" (C) of a data unit is the number of current transactions whose conflict array's update field of the data unit is true.

CiR can be considered as the percentage of read operations on a data unit that will cause an abort of the transaction because of the involvement of this data unit in the transaction's conflict chain. Note that, although C is a dynamic value based on the data and all current transactions’ conflict arrays, R is an evaluated value based on past operations (current transaction's future operations are not known in advance). However, we assume R can reflect the current requests for one data unit by obtaining the average number of requests in the most recent given time period.

A data unit with a high CiR value should be replaced because, as noted in the previous section, it is meaningless to keep this data in the cache. Therefore, CiR policy is based on the idea that data units that are less useful for caching should be replaced with more useful data units.

4.1.3 Performance of STUBcache

We conducted simulations on STUBcache with replacement policies PIX, RxU, and CiR. These simulations used the same configuration parameters as the primary STUBcast simulations except for the ones shown in Table 4-1. Results shown in this section are all under Tr50.

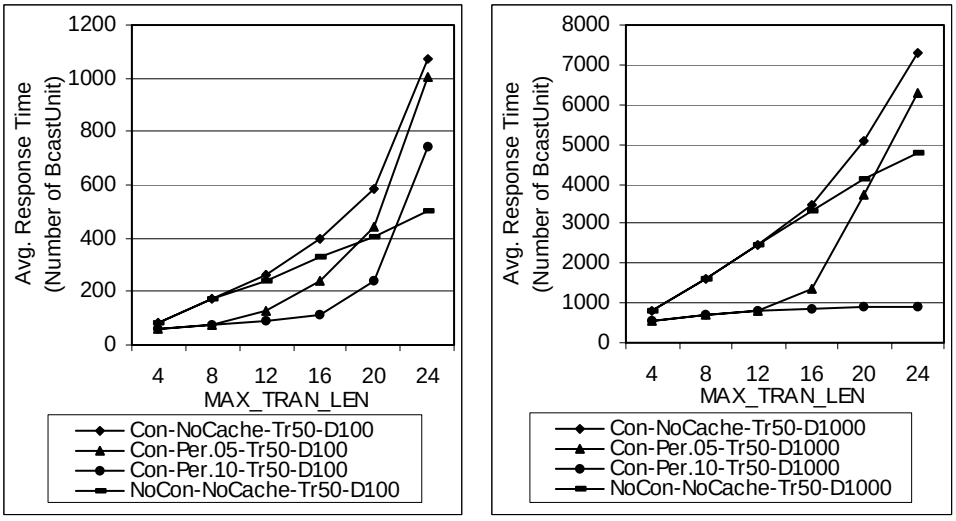
Table 4-1 Additional parameters for STUBcache simulation

PERCENTAGES .03 (3%), .05 (5%), .10 (10%), (Per)	MACHINE 1, 4 (M)
The cache size of a client is set as a given percentage of the DB_SIZE.	The number of clients (each client is on a different machine, so uses a separate cache) in the simulation. All transactions are distributed uniformly for all clients, but TR_INTARR_TIME still applies in all transactions.

4.1.3.1 With and without cache

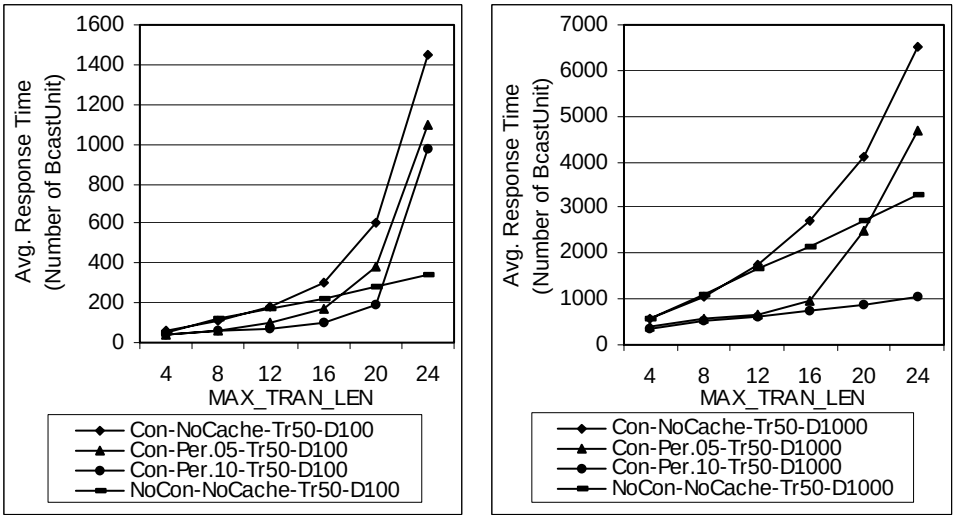
Figure 4-2 shows the performance improvement of STUBcache (with PIX) compared with primary STUBcast with configuration Tr50, D100 (or D1000), Per.05 (or Per.10), UNIFORM, and FLAT. All transactions are run for one client. The results show that using caching results in a much shorter response time than not using it. When MAX_TRAN_LEN is not too large (≤20), using cache achieves much better performance compared with no concurrency control. However, when MAX_TRAN_LEN exceeds 20 and the cache size is small (D100, and Per.05-D1000), the aborts caused by concurrency control and long transactions overrides the benefit of caching compared with no concurrency control. However, STUBcache still performs better than STUBcast. We predict that RxU and CiR might improve this situation because they consider the concurrency control behavior of STUBcast. We can draw the same conclusions as above

from the results in Figure 4-3, which shows the performance of STUBcache under NON_UNIFORM and MULTDISK. Our later simulations will skip NON_UNIFORM and FLAT unless necessary because it was found that these configurations were not reasonable.



(a) (b)

Figure 4-2 Average response time with and without cache under UNIFORM data access and FLAT scheduling. a) D100; b) D1000

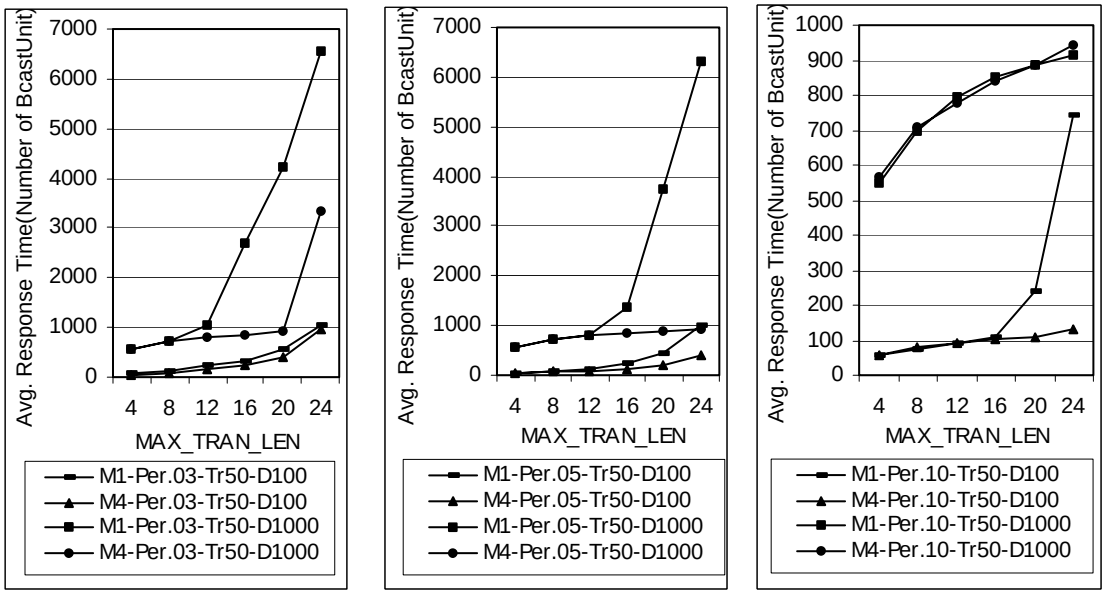


(a) (b)

Figure 4-3 Average response time with and without cache under NON_UNIFORM data access and MULTIDISK scheduling. a) D100; b) D1000

4.2.3.2 Number of clients

The minimal influence of STUBcache (with PIX) is given when all transactions are run using one client. How STUBcache performs if the same cache size is adapted for each client while these transactions are distributed across multiple clients is shown in Figure 4-4. Although real applications usually have a scalable number of clients and transactions, these results using a limited number of machines and transactions can help predict the their performance. Based on the results of M1 (compared with M4), Per.03 (or Per.05, Per.10), Tr50, D100 (or D1000), UNIFORM, and FLAT, we reach the following conclusions:



(a) (b) (c)

Figure 4-4 Average response time with different number of clients under UNIFORM data access and FLAT scheduling. a) Per.03; b) Per.05; c) Per.10

- Response time becomes smaller when cache size increases under both distributed and centralized configurations, but the distributed ones improve faster.
- When cache size is very small (Per.03-D100), there is no advantage to response time of more distributed transactions and caches compared to the centralized ones (they are both large).

- Response time of more distributed configurations have an increased advantage over centralized ones when the database size or cache percentage increases (Per.03 to Per.10, or D100 to D1000).
- The advantage of response time for more distributed caches and transactions becomes trivial (they are both small) when the cache size is very large (Per.10-D1000). This implies that STUBcache can perform very well as long as the cache size for one client is large enough, even though there is a heavy transaction load on that client.

Based on data collected from simulations with the same configurations as in Figure 4-4 except for replacing the access mode with NON_UNIFORM and the scheduling algorithm with MULTIDISK, we can actually draw the same conclusions as above when comparing the performance using different number of clients. In the later part of this dissertation, we may show the data figures of only one of the UNIFORM/FLAT and NON_UNIFORM/MULTIDISK configurations if they have similar performance behaviors using other certain parameters.

4.1.3.3 Cache percentage and database size

Figures 4-2, 4-3, and 4-4 clearly reflect the influence of cache percentage and database size on STUBcache. First, for the same database size, STUBcast's performance improves more when cache percentage is greater. Second, a high cache percentage leads to more performance improvement when the database size is larger. Therefore, we conclude that the performance of STUBcache is a function of absolute cache size.

4.1.3.4 Compare PIX with RxU and CiR

The results in 4.1.3.1. show that STUBcache cannot efficiently improve the response time at very large maximal transaction length and smaller cache size using the PIX replacement policy. This is because longer transactions can cause more aborts under concurrency control. Therefore, we simulated RxU and CiR using the same

configurations as PIX because these policies are designed based on STUBcast's concurrency control features.

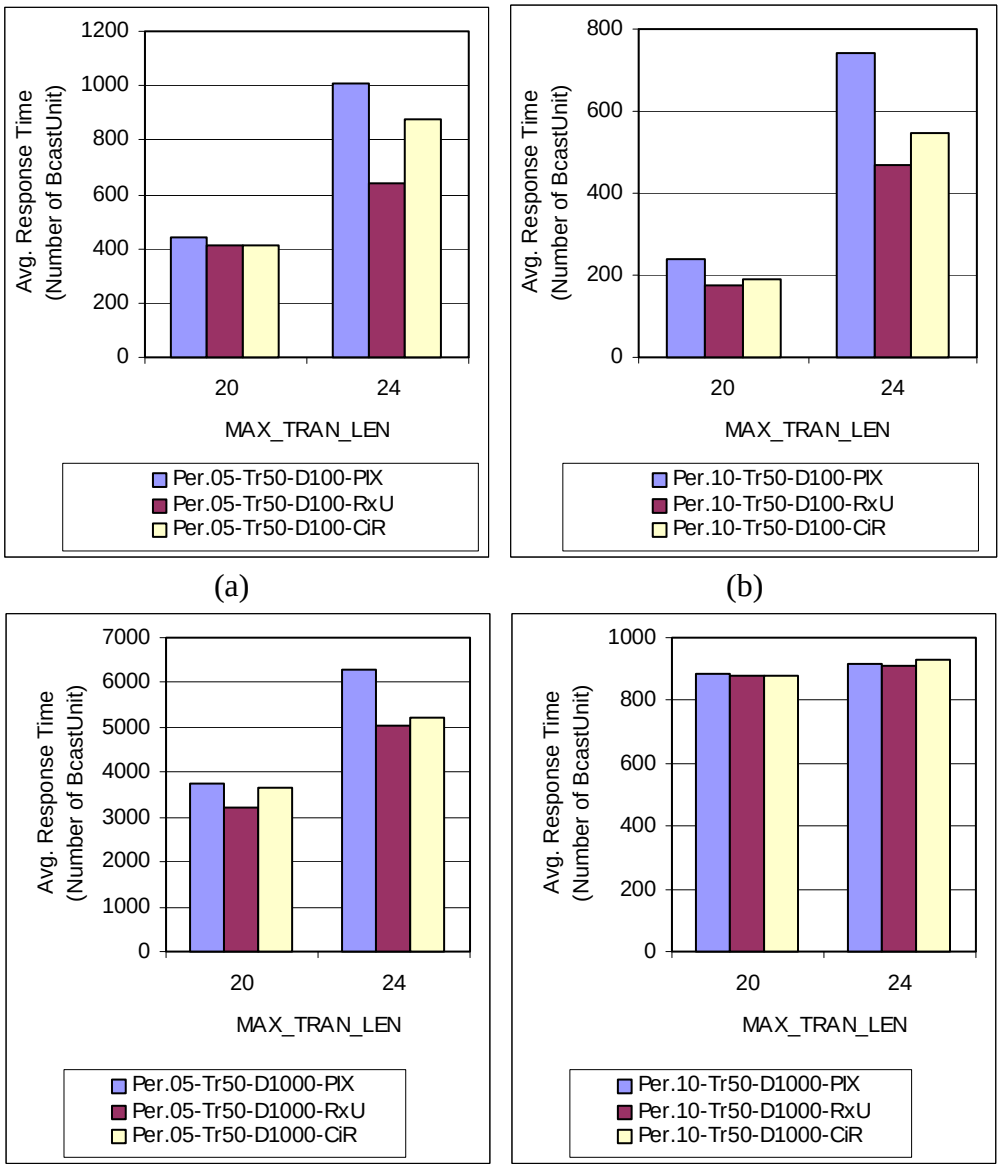
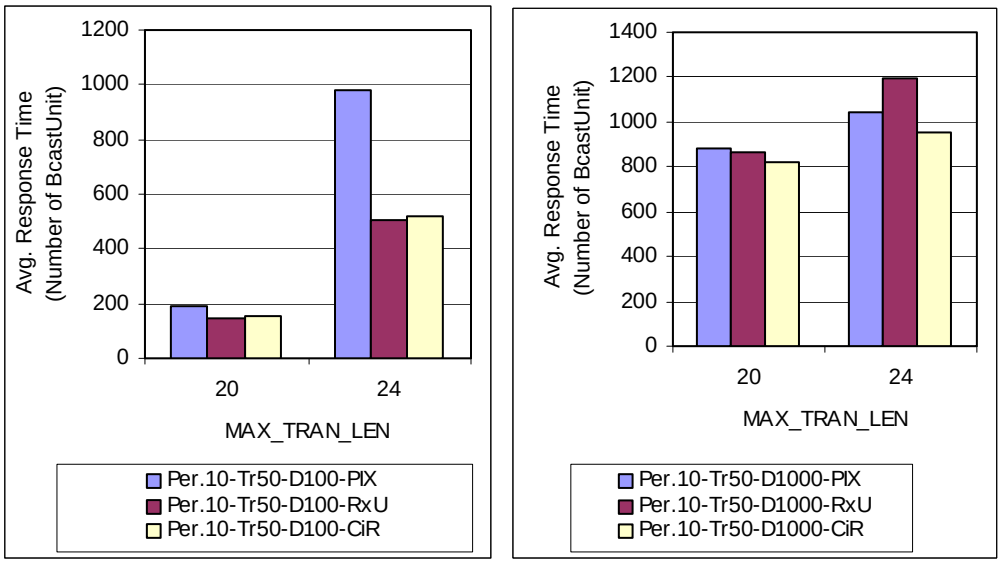


Figure 4-5 Average response time with PIX, RxU, CiR under UNIFORM data access and FLAT scheduling. a) Per.05-D100; b) Per.10-D100; c) Per.05-D1000; d) Per.10-D1000

Simulated data showed no improvement of RxU or CiR over PIX when MAX_TRAN_LEN was under 20, because PIX already performs well under conditions

of small abort rates. However, as expected, RxU and CiX perform much better than PIX at L20 and L24, and when the cache size is relatively small. For example, we can observe a 20% to 40% improvement of RxU and a 20% to 30% improvement of CiR at L24 as shown in Figure 4-5a to 4-5c (UNIFORM/FLAT). On the other hand, Figure 4-5d shows that these policies' performances are nearly the same when the cache size is very large because PIX already performs very well.



(a) (b)

Figure 4-6 Average response time with PIX, RxU, CiR under NON_UNIFORM data access and MULTIDISK scheduling. a) Per.10-D100; b) Per.10-D1000

A similar conclusion can be drawn using NON_UNIFORM/MULTIDISK. Figure 4-6 only shows the cases under Per.10. Note that Case a) shows an even larger improvement of RxU and CiR (50%) at L24. However, with the largest cache size configuration (P.10-D1000), PIX performs better than RxU. We believe this is because PIX shows its advantage sufficiently under NON_UNIFORM data access when the cache is large enough.

Based on above simulation results, we can conclude that STUBcache can efficiently improve the performance of STUBcast. Moreover, RxU and CiR, designed based on the protocol's special features, can greatly improve the performance of general replacement policy like PIX at large maximal transaction length and smaller cache size.

4.2 STUBcast Adapted Scheduling

STUBcast inserts *Ucast* into a primary schedule unpredictably whenever there is an update transaction committed at the server. Although it can easily be implemented with any existing data scheduling algorithm, it might degrade the average response time of a given algorithm (such as multi-disk [ACH95a] and fair queuing scheduling [VAI97b]) because of its dynamic behavior. This section discusses a server scheduling problem, which adapts *Ucast* into an evenly data-spaced scheduling algorithm to provide satisfactory and predictable performance.

Moreover, STUBcast verifies all submitted update transactions in a FCFS order without considering the relations among transactions. This processing sequence is disadvantageous when considering the fairness among different transaction sources. This section also discusses a problem of rescheduling the verification sequence of submitted transactions to guarantee server service fairness among all clients.

4.2.1 Scheduling *Ucast* among Evenly Spaced Broadcast Algorithms

4.2.1.1 *Ucast* and evenly spaced broadcasting

Problem. As discussed in 2.1.1., most existing data scheduling algorithms depend on evenly spacing the instances of each data unit within a schedule [ACH95a, HAM97, VAI97a, VAI97b] to achieve optimal total average response time. Although these algorithms are not designed specifically for transaction processing, it is easy to conclude

that this evenly spaced scheme is still optimal for individual transaction operations in order to achieve minimal transaction response time.

However, when STUBcast is used for concurrency control with any evenly spaced scheduling algorithms, the pre-assigned distances among data units in *Pcast* will be enlarged randomly by the insertion of *Ucast*. Although the data broadcast in *Ucast* might actually shorten the spaces of some data unit instances, it is unpredictable, and it is the reason for prolonging the distance between broadcast of other data. Finally, the total performance might differ greatly from the original objective of the primary schedule. This introduces a new problem , scheduling *Ucast* among evenly spaced scheduling algorithms.

Find a solution to schedule random Ucast with evenly spaced broadcast algorithms and guarantee a predictable worst-case overall average access time.

Assumptions. Some assumptions of this scheduling problem are as follows:

- We focus on scheduling STUBcast with the multi-disk or fair-queuing scheduling algorithms, which require even spacing of data units.
- Each server data unit has a known access possibility p_i, $1 \leq i \leq$ DB_SIZE. Each normal data unit and a data unit in *Ucast* have the same size. The size of *UBB* and *UBE* with data-read ID is included in the length of each *Ucast* (the number of *Ucast* data units in it).
- *Ucast* requests are uniformly distributed on the timeline.

4.2.1.2 Solution--periodic *Ucast* server

We solve this problem by scheduling aperiodic *Ucast* as normal data units in primary scheduling. When the frequency and *Ucast* length conform to a specific model, each *Ucast* can be treated as one or more normal data units under any evenly spaced algorithm. Consequently, evenly spaced time slots are assigned to aperiodic *Ucast*. Each *Ucast* can only be inserted into these time slots instead of inserting to the schedule

randomly. If no *Ucast* happens at the beginning of a time slot, the slot is randomly assigned to a normal data unit. This is very similar to a periodic server for real-time aperiodic tasks. Thus, we give it a name, periodic *Ucas*t server. Based on this type of scheduling, other normal data units can be pre-scheduled evenly in the time slot not assigned to *Ucast*. Even space among the data unit instances in this algorithm would already include the time occupied by *Ucast* time slots. Consequently, the maximal space among each data unit's instances is predictable, though the broadcast in *Ucast* and non-used *Ucast* time slot might shorten some data unit's instance spaces.

4.2.1.3 Minimal *Ucast* response time

The periodic *Ucast* server method can schedule all *Ucast* as normal data units with different length and frequency (or different period and budget). For example, if the frequency of *Ucast* is F in a given broadcast period and each *Ucast* has length L, we can schedule them as one length F×L data with frequency 1, or one length L data with frequency F, or even one length L/2 (L>1) data unit with frequency 2×F.

A different data-unit view of these *Ucast* can result in different spacing and size of the time slots. This further results in different average response time for a *Ucast* request, which is the average time from when an update transaction commits to the time the transaction's *Ucast* finishes broadcasting. On the other hand, a different data-unit view of *Ucast* has no influence on other normal data units because the bandwidth occupied by all *Ucast* is fixed. Since it is more practical to finish broadcasting an update transaction's *Ucast* as early as possible, we need to analyze how to map *Ucast* to normal data unit optimally in order to achieve least average response time for any *Ucast* request in a periodic *Ucast* server.

***Ucast* frequency and length models.** By analysis and simulation, we have found the optimal way to map *Ucast* to normal data units for the following two *Ucast* frequency and length models.

- Model 1: *Ucast* has frequency F in a fixed time period and all with same length L.
- Model 2: *Ucast* has frequency F in a fixed time period and each *Ucast* can have length from 1, 2, … to L with the uniform probability.

Analysis results of optimal mapping of *Ucast* to data units, without considering queuing delay of *Ucast* requests. The analysis procedure [HUA00] is omitted here.

For Model 1, the total number of time slots required in all *Ucast* in the fixed time period is as follows:

$$F \times L.$$

By analysis, the least average *Ucast* response time is achieved when each *Ucast* containing L data items is treated as one normal data unit with length L. Consequently, the periodic server should pre-assign evenly spaced time slots for data-unit mapping to *Ucast* with length L and frequency F.

For Model 2, each *Ucast* might have i ($1 \le i \le L$) data units with possibility 1/L. So the total number of time slots required in all *Ucast* in the fixed time period is (assume F divides L):

$$1 \times \frac{F}{L} + 2 \times \frac{F}{L} + ... L \times \frac{F}{L} = \frac{F \times (L+1)}{2}.$$

By analysis, the least average response time of any *Ucast* is achieved when each single data unit in any *Ucast* is treated as one normal data unit with length 1.

Consequently, the periodic server should pre-assign evenly spaced time slots for data-unit mapping to *Ucast* with length 1 and frequency $\frac{F \times (L+1)}{2}$.

The result of this case requires all *Ucas*t data units to be separate from each other and evenly spaced. The concurrency control protocol can be modified to adjust to this change. In the modified protocol, each data unit broadcast within a *Ucast*'s response time can be easily marked as a normal data unit or a *Ucast* data unit (for example, based on the data unit's timestamp; a normal data unit's timestamp must be less than the timestamp of the current *Ucast*). If a data unit is both scheduled as a normal one and an updated one within the response time of a *Ucast*, the newest timestamp will be attached with all this data unit's instances. No other changes are necessary to the protocol.

Simulation results of optimal mapping of *Ucast* to data, considering queuing delay of *Ucast* requests. Earlier analysis results assume no queuing of *Ucast* at the server, such that any *Ucast* can insert its next data unit into the earliest available time slot assigned to *Ucast*. However, in real applications, a *Ucast* may be queued and wait for earlier *Ucast* to finish using pre-assigned time slots. Therefore, we also need to find the optimal mapping with *Ucast* queuing.

Instead of using the analysis method, we simulated the two *Ucast* frequency and length models, using different F/L configurations and some possible lengths of normal data units that a *Ucast* can map to. Detailed simulation configurations are shown in Tables 4-2 and 4-3. The conclusions are given in Table 4-4.

Figures 4-7 and 4-8 show some cases from all collected results that support the conclusion. Figure 4-7a shows the cases for Model 1, a small portion of *Ucast*, *Ucast*_F5, and *Ucast*_L (from 1 to 5), where optimal mapping is always achieved when Mapped_L

is equal to *Ucast*_L (i.e., Map_L is equal to L). Figure 4-7b shows the cases for a large portion of *Ucast* and the same other configurations as in 4-7a. Their optimal mapping is achieved when Mappd_L is equal to *Ucast*_L+1 (i.e. Map_L is equal to L×2). Moreover, Figure 4-8 shows cases of Model 2, with either (small portion of *Ucast*, *Ucast*_F5, *Ucast*_L2) or (large portion of Ucast, *Ucast*_F5, *Ucast*_L3). Note that in Figure 8 some bars with very large values are truncated in order to show the smallest value clearly. They both show that optimal mappings are achieved when Mapped_L = 2^{Ucast_L}+1, (here when Mapped_L = 5 or Mapped_L = 9, i.e., Map_L is equal to L+1). The same conclusions as in Table 4-4 and from Figures 4-7 and 4-8 can be drawn from all simulated cases.

4.2.1.4 Optimal normal data frequency and spacing

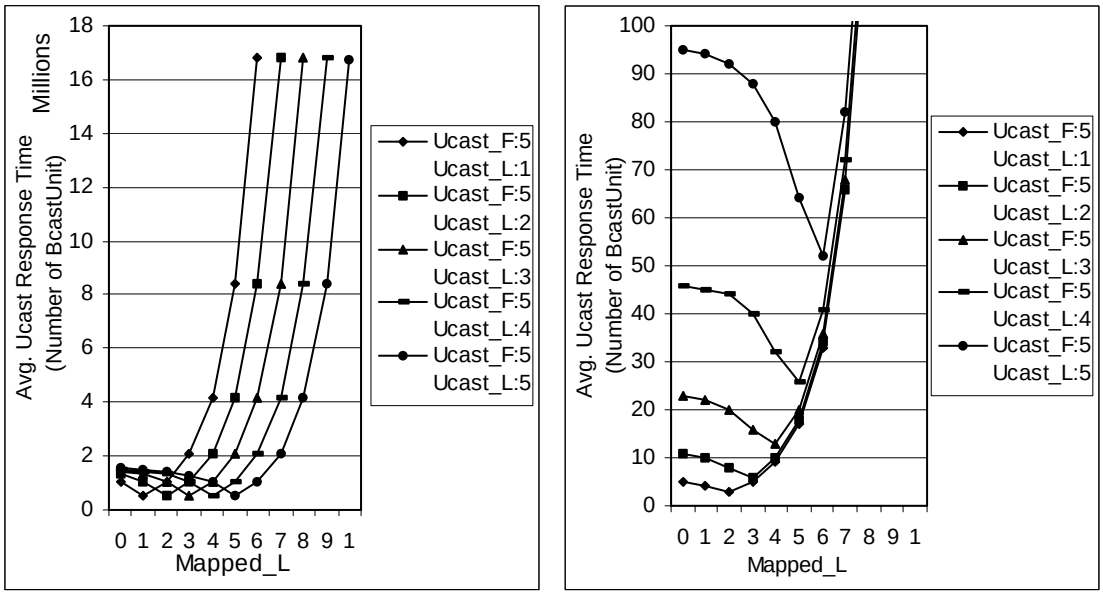
Analysis and results. Data broadcasting frequency and distance assignment for minimal overall average access time are presented in [HAM97, VAI97a, VAI97b] and in Section 2.1.1. Based on these, we now discuss what is the optimal assignment of these parameters for a normal data unit using a periodic *Ucast* server with notations defined in Figure 4-9. The major difference from the earlier analysis is that here the length of all normal data units is 1.

Table 4-2 F and L configurations

Names	Explanation	Value
F	The frequency of *Ucast* in a fixed period	2^{Ucast_F}, MIN_F ≤ *Ucast*_F ≤ MAX_F MIN_F = 5, MAX_F = 10
L	The length of *Ucast* Model 1: L for all *Ucast* Model 2: [1, L] with possibility 1/L	2^{Ucast_L}, MIN_L ≤ *Ucast*_L ≤ MAX_L MIN_L = 1, MAX_L = 5
Map_L (Map_F)	Possible normal data length (corresponding frequency) that *Ucast* data can map to	Model 1: 2^{Mapped_L} (Map_F = F×L/Map_L), 0 ≤ Mapped_L ≤ *Ucast*_F + *Ucast*_L Model 2: Mapped_L (Map_F = F×L/Map_L), 0 ≤ Mapped_L ≤ MAX, and a divisor of MAX, MAX = 2^{Ucast_F-1}×(2^{Ucast_L}+1).

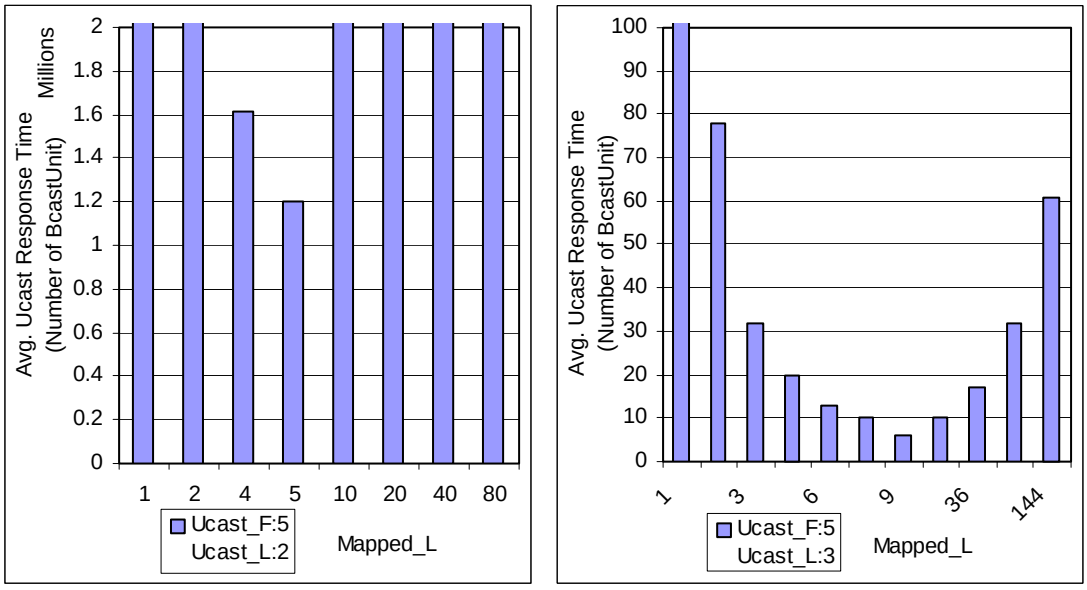
Table 4-3 Scheduling cycle configurations

Names	Explanation	Value with small *Ucast* portion		Value with large *Ucast* portion	
		Model 1	Model 2	Model 1	Model 2
Data cycle	The total number of normal data in a fixed period	$2^{MAX\_F}\times 2^{MIN\_L}\times 2^{MIN\_L+1}\times..\times 2^{MAX\_L}$	$2^{MAX\_F-1}\times (2^{MIN\_L}+1)\times (2^{MIN\_L+1}+1)\times..\times (2^{MAX\_L}+1)$	$2^{Ucast\_F}\times 2^{Ucast\_L}$	$2^{Ucast\_F-1}\times (2^{Ucast\_L}+1)$
Ucast cycle	The total number of data in all *Ucast* in a fixed period	$2^{Ucast\_F}\times 2^{Ucast\_L}$	$2^{Ucast\_F-1}\times (2^{Ucast\_L}+1)$	$2^{Ucast\_F}\times 2^{Ucast\_L}$	$2^{Ucast\_F-1}\times (2^{Ucast\_L}+1)$
Total cycle	The total number of normal data and data in *Ucast* in a fixed period	Normal data / *Ucast* data ratio ≥ 1024:1	Normal data / *Ucast* data ratio ≥ 2295:1	Normal data / *Ucast* data ratio = 1:1	



(a) (b)

Figure 4-7 Sample results for Model 1. a) Small *Ucas*t portion; b) Large *Ucast* portion



(a) (b)

Figure 4-8 Sample results for Model 2. a) Small *Ucast* portion; b) Large *Ucast* portion

Table 4-4 Optimal mappings from *Ucast* to normal data with *Ucast* queuing

Models	Small *Ucast* portion	Large *Ucast* portion
Model 1	Map to frequency F and length L	Map to frequency F/2 and length 2×L
Model 2	Map to frequency F/2 and length L+1	

N: The length of a broadcast cycle.
N': The number of normal data units in a broadcast cycle.
P: The length of the fixed time period where F length L *Ucast* requests appear.
f_i, s_i, p_i: Broadcast frequency (in a broadcast cycle), the even broadcast distance, and access possibility of data i ($1 \le i \le$ DB_SIZE), respectively.
r_i: Defined as $\frac{f_i}{N'}$, i.e., the percentage of bandwidth that data i ($1 \le i \le$ DB_SIZE) occupies among the bandwidth occupied by normal data in a broadcast cycle.

Figure 4-9 Notations for optimal data frequency and spacing analysis

Formula (4-1) derives the relation among N, N', P, F, and L.

$$N = N' + \frac{N}{P} \times F \times L \;\Rightarrow\; N = \frac{P}{P - F \times L} \times N' \;. \tag{4-1}$$

Using the same methods found in [HAM97, VAI97a, VAI97b], we can conclude Formula (4-2), which implies that the optimal frequency of each normal data unit is proportional to $\sqrt{p_i}$. This result is the same as in the references cited above.

$$\sum_{i=1}^{DB\_SIZE} f_i = N' \;\Rightarrow\; \sum_{i=1}^{DB\_SIZE} r_i = 1 \;\Rightarrow\; f_i \propto \sqrt{p_i} \;. \tag{4-2}$$

Furthermore, Formula (4-3) represents the optimal results of s_i. From the result of Formula (2-2), we can see that this distance is enlarged by a factor of $\frac{P}{P - FL}$, which is caused by the insertion of F length L *Ucast* in every fixed time of P.

$$f_i = \left(\frac{N'}{\sum_{j=1}^{DB\_SIZE} \sqrt{p_j}}\right)\sqrt{p_i} \;\Rightarrow\; s_i = \frac{N}{f_i} = \left(\frac{P}{P - FL}\right)\left(\sum_{j=1}^{DB\_SIZE} \sqrt{p_j}\right)\frac{1}{\sqrt{p_i}} \;. \tag{4-3}$$

The above results imply the following.

- The total number (N') and optimal broadcast frequency of normal data units in a broadcast cycle under periodic *Ucast* server remains the same as it would be without *Ucast*. They are not influenced by the values of P, F and L.
- On the other hand, the total length of a broadcast cycle (N) and the optimal even space of each normal data unit increases when the value of F×L in P time (can be defined as the portion of *Ucast*) increases.

Application of results. These observations allow us to use the frequency assignment of multi-disk and fair-queuing scheduling applied in normal scheduling directly. For example, we can use the same relative frequency assignment in MULTIDISK as in the simulations in Chapter 3 when simulating the same configurations under a periodic *Ucast* server. However, the distances between a data unit's instances are

enlarged because we periodically (Map_F times in time P) inserting Map_L slots for *Ucast* data in the schedule. The differences between using multi-disk and fair queuing scheduling are as follows.

- Case 1: If Map_L equals 1, we can always use multi-disk (because it schedules data with same length) by considering *Ucast* data units as a separate disk with relative frequency calculated from Map_F and P.
- Case 2: If Map_L is larger than 1 and the original multi-disk schedule can be divided by all length Map_L *Ucast* data units such that all instances of any normal data are still separated by the even distance calculated from Formula (4-3), then multi-disk can still be used.
- Case 3: In any situation, we can always use fair-queuing scheduling [HAM97, VAI97a, VAI97b] because it allows different lengths of data units. Normal data unit's distance s_i is set by Formula (4-3). *Ucast* data unit's distance S is set as $\frac{P}{Map\_F}$.

The predictable and optimal worst-case normal data overall average access time. With the assigned optimal frequency and distance among normal data, this objective performance value is obtained in Formula (4-4). It is also the optimal overall average access time of normal scheduling [HAM97, VAI97a, VAI97b] multiplied by the factor of $\frac{P}{P-FL}$.

$$t_{optimal} = \frac{1}{2}\sum p_i s_i = \frac{1}{2}(\frac{P}{P-FL})(\sum_{i=1}^{DB\_SIZE} p_i)^2 . \quad (4\text{-}4)$$

We call Formula (4-4) predictable and optimal worst case because such an average access time is guaranteed and is optimal if no data unit is broadcast in *Ucast* and non-used slots. The actual performance should be better because some data units are broadcast extra times in *Ucast*, which, though not evenly spaced, can clearly provide a shorter response time than the assigned even space for these data.

4.2.1.5 Other even space adapted scheduling method

Here we design another solution, called dynamic space adjustment scheduling, to schedule *Ucast* with fair queuing algorithm [HAM97, VAI97a, VAI97b]. This scheme does not avoid randomly inserting *Ucast* into a fair queuing scheduling and does not guarantee an even space for each data unit. It is based on the idea of counting each data unit broadcast in a *Ucast* as one instance in a primary schedule. The algorithm is given in Figure 4-10.

Step 1. Calculate the optimal even space value S_i ($1 \le i \le DB\_SIZE$) for each data item using the method in [HAM97, VAI97b].
Step 2. Start scheduling data units by fair queuing algorithm, based on the value of S_i, time T, length of data, B_i and C_i, as defined in [HAM97, VAI97b].
Step 3. When a *Ucast* starts, for all updated data in the transaction but not broadcasted in this *Ucast* yet, choose the one with the least value of C_j to broadcast. Arbitrarily choose one if there is any tie.
Step 4. After the broadcasting, set $B_i = C_i$ and then $C_i = B_i + S_i$, and adjust time accordingly by the length of data (one broadcast unit). Return to Step 2.

Figure 4-10 Dynamic space adjustment scheduling algorithm

The algorithm itself is basically the same as fair queuing, except that fair queuing as primary scheduling ignores data in *Ucast* (does not adjust B_i, C_i for data in *Ucast*), while this algorithm counts any data unit in *Ucast* as one in primary broadcasting (adjust B_i, C_i afterward). If a data unit is moved ahead in the broadcasting by a *Ucast*, the space between this instance and next one is enlarged from the calculated even space. Then no matter how earlier instances of a data unit move ahead in the schedule, the broadcast location of any later instance is not influenced. Therefore, it works like the moved-ahead instance in *Ucast* exchanged locations with delayed data. This can avoid any data unit occupying extra bandwidth by appearing in *Ucast*. Although the distances among all data

instances are alternated, the bandwidth occupied by all data stays the same. Moreover, this scheme also tries to broadcast a *Ucast* data unit as near as possible to the next time slot it should be in by choosing the data unit with minimal C_i value.

4.2.1.6 Performance results

We conducted simulations by adapting all solutions and results discussed in earlier sections on a similar simulation framework to Chapter 3. Some additional configurations and explanations are given in Table 4-5. Figure 4-11 shows how data mapped from *Ucast* are inserted to a multi-disk scheduling with D256 and cycle length of 960.

Based on all the performance results, we made following observations. Some example results are shown in Figure 4-12.

- Performance of different methods differ little when either F or L is small (i.e., F smaller than 40 per cycle or L smaller than 8 for both D256 and D1024). This implies inserting *Ucast* randomly into an even spaced primary schedule does not largely influence the performance under equally simple and simpler conditions.
- Performance difference becomes great when F and L are relatively large, i.e., F is 120 or larger per cycle and L is 12 or larger for both D256 and D1024.
- When cold read data appears (Figure 4-12a) in update transactions (UPDATE_ACCESS_MODE is Cold), randomly inserting *Ucast* into the schedule is more harmful when F and L are large. Aperiodic scheduling thus shows worse performance than W_case, even under the condition that W_case does not make use of any slots pre-assigned to *Ucast*. At the same time, Dynamic scheduling has no advantage either and it has the worst performance among all methods. Since cold data instances should occupy very small bandwidth, their appearing in *Ucast* very often causes Dynamic scheduling not able to show its benefit. Finally, the P_U_server shows the best performance among all methods. It improves Aperiodic scheduling by 15 to 30% and it is always better than W_case among all simulated cases where F and L have large values.
- When hot read data appears (Figure 4-12b) in update transactions (UPDATE_ACCESS_MODE is Hot) while F and L have large values, W_case scheduling has the worst performance for not using any pre-assigned slots for *Ucast*. Aperiodic scheduling performs better than W_case in this case, but P_U_server still shows its advantage over it (about 15 to 30% improvement). On the other hand, we observe that Dynamic scheduling shows the best performance when L is 12 and 16. It is logical to see this performance with a limited number of

hot data moved ahead in Dynamic scheduling. This advantage disappears when L becomes larger than 16 and Dynamic scheduling again performs worst. This is also logical, because in these cases even hot data have over-occupied the bandwidth because of the large proportion of *Ucast*.

Table 4-5 Scheduling *Ucast* among evenly spaced algorithms simulation configurations

Parameters	Values
*UCAST_* SCHED_ALG Algorithms used to schedule *Ucast*	W_case: Optimal worst case, i.e., all slots for *Ucast* are spared Aperiodic: Insert *Ucast* randomly Dynamic: Dynamic space adjustment P_U_server: Periodic *Ucast* server, with unused slot spared
READONLY_ ACCESS_MODE	NON_UNIFORM [n = 4, $P_i : P_{i+1}$= 4:1, $1 \le i \le n$] The data access mode in all read-only transactions
UPDATE_ ACCESS_MODE	Cold: Only write to data that are read least (data Part 4) Hot: Only write to data that are read most (data Part 1) Data access mode in all update transactions, which only write and happen with a uniform distribution mode on the time line
PCAST_ SCHEME	Multi-disk with optimal frequency assignment Note: a fair queuing algorithm can find the same schedule as multi-disk in our simulation configurations, so Dynamic scheduling can also be used with this multi-disk scheduling
READONLY_ LEN	1 Client read-only transactions' length, all set to 1 to avoid abort
DB_SIZE	D256, D1024
L	4, 8, 12, 16, 20, 24 Length (Model 1) or maximal length (Model 2) of a *Ucast*. Mapped to L in Model 1 and L+1 for Model 2
F Number of *Ucast* (i.e. update transactions) in a broadcast cycle.	240, 120, 40, 24, 8 (D256 with multi-disk cycle size 960) 960, 480, 240, 120, 40, 24, 8 (D1024 with multi-disk cycle size 3840) The chosen numbers are some divisors of the total number of normal data units in a broadcast cycle. These values allow any length L (Model 1) or L+1 (Model 2) data units mapped from *Ucast* to be inserted F (Model 1) or F/2 (Model 2) times with even space and normal data units under a multi-disk scheduled sequence that have their even spaces calculated from Formula (4-3), as in Case 2 of Section 4.2.2.3.

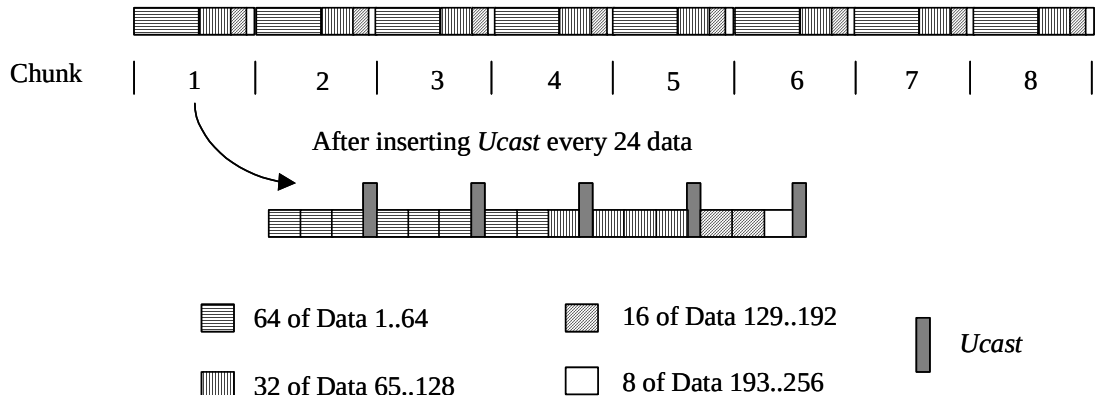
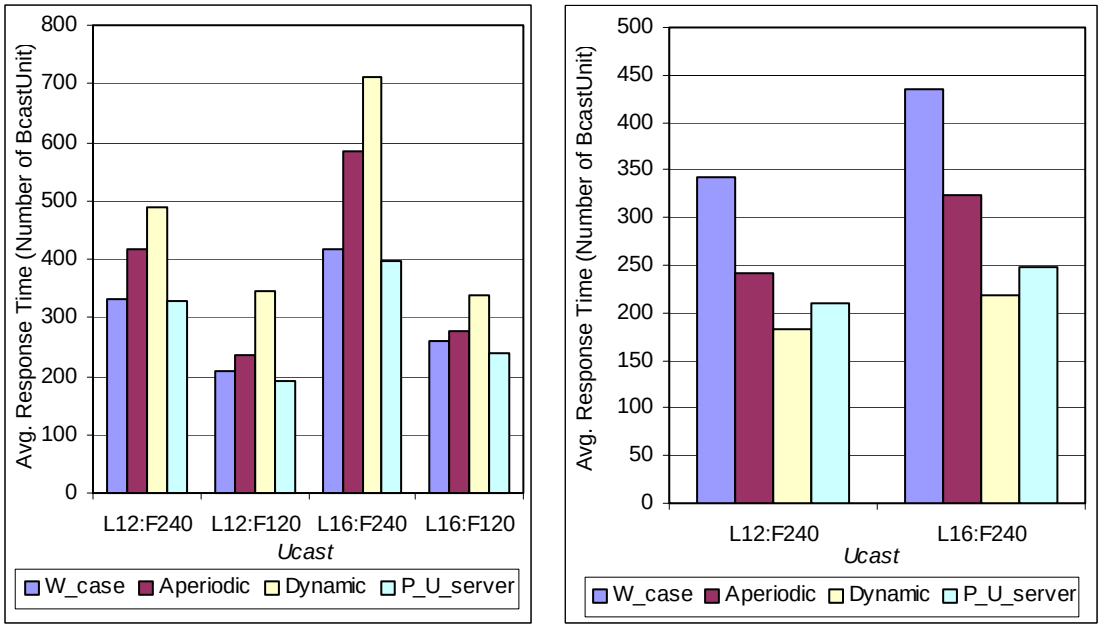


Figure 4-11 Multi-disk scheduling with periodic *Ucast* server used in the simulation



(a) (b)

Figure 4-12 Sample performance results of periodic *Ucast* server and other scheduling methods, with Model 1, D256, NON_UNIFORM and MULTIDISK. a) UPDATE_ACCESS_MODE Cold; b) UPDATE_ACCESS_MODE Hot

4.2.1.7 Section conclusion

It is necessary to adapt *Ucast* to an evenly spaced primary scheduling algorithm to reduce the possible performance degradation by inserting *Ucast* randomly. This section

designed a periodic *Ucast* server method and found the optimal mapping schemes for two *Ucast* frequency and length models in order to also minimize *Ucast* response time. Optimal frequency and spacing assignment for normal data was also analyzed in order to apply periodic *Ucast* server in multi-disk and fair queuing scheduling. Finally, we adopted all the results above in simulations and compared P_U_server with other methods, including another Dynamic scheduling solution. From the results we see it is only necessary to use even spacing adapted *Ucast* scheduling when *Ucast* occupies a very large portion of broadcasting, and P_U_server usually performs the best in all given methods. However, Dynamic scheduling can perform the best when each *Ucast* includes the hottest data and they only occupy a limited amount of broadcasting bandwidth.

4.2.2 Reschedule the Verification Sequence of Update Transactions by Worst-case Fair Weighted Fair Queuing (WF^2Q)

In STUBcast, the nature of sequential verification of SVP can influence the performance when FCFS verification is used. This section solves a server fairness problem by rescheduling the verification sequence from the FCFS scheme.

4.2.2.1 Problem

- The resource needed by an update transaction with L data updates is:
  Verification time (ignored) + the time to broadcast its Ucast = L BcastUnit
- The server's total resource is the *full timeline* to broadcast data.
- With N existing entities, an entity's *share* (non-weighted) of resource is:
  $\frac{1}{N}$ of the total server resource.
- An entity is well-behaving if it requires no more than its share of resource, otherwise it is misbehaving.
- By definition, an entity receives its fair service if it will not be queued whenever it is well-behaving.

Figure 4-13 Definition of server fairness in STUBcast

In the application environment of STUBcast, a server should provide fair [DEM89] service among update transactions from different clients (or sessions, or any entity that fairness is based on, so we use the word entity hereafter). Server fairness in STUBcast is defined in Figure 4-13.

The current FCFS scheme, however, may fail to provide fair service to all well-behaving entities if there exists any misbehaving one that sends burst transactions to the server. The unfairness of FCFS is discussed as following:

- Under FCFS, update transactions are sequenced for verification and commit by their arrival time without considering the source of transactions. When some misbehaving entities send burst transactions to the server such that transactions are queued for the lack of server resource, transactions submitted later and sent from well-behaving entities will be queued and the delay will be unpredictable. A sever should provide a guaranteed fair service for transactions from well-behaving entities no matter how other entities misbehave.
- The queuing caused by transactions from misbehaving entities and FCFS verification results in a higher probability for transactions from well-behaving entities to be aborted because a read ahead status is more likely to happen. Therefore, misbehaving entities and FCFS can unpredictably reduce the chance of successful commit for transactions from well-behaving entities. A server should avoid the occurrence of this unfairness.

The unfairness caused by FCFS verification is very similar to the one caused by FIFO queuing in communication gateways, where bandwidth, buffer, and promptness fairness among all sessions are required. Since in the communication field we solve this problem by FQ, WFQ, and WF^2Q [BEN96a, BEN96b, BEN96c, DEM89] algorithms, we look into these algorithms and design our solution based on them.

4.2.2.2 Solution's theoretical background

An ideal way to provide fair service to different communication sessions on a gateway is to use PS or GPS [BEN96b, BEN96c, KLE76], as defined in Table 4-6. However, it is not feasible to implement these methods since they send bit by bit from

each session. Correspondingly, packet-based version of PS and GPS, FQ and WFQ, have been designed, as shown in Table 4-6. These methods can guarantee a performance no worse than the ideal PS and GPS by the way shown in Table 4-7. However, it is pointed out by [BEN96b, BEN96c] that WFQ can actually be far ahead of GPS, which can influence the judgment of flow control algorithms at a communication source. Therefore, a WF^2Q algorithm (Table 4-6) is designed to solve this problem.

Table 4-6 Algorithm for fairness among communication sessions in network gateways

Name	Solution
Bit-wise solutions	
PS	Processor Sharing: There is a separate FIFO queue for each session sharing the same line. During any time interval when there are exactly N non-empty queues, the server services the N packets at the head of the queues simultaneously, and each at a rate of one Nth of the line speed.
GPS	General Processing Sharing: Same as PS, but it allows different sessions to have different service shares and serves the non-empty queues in proportion to the service shares of their corresponding sessions.
Packet-based solutions	
FQ	Fair queuing: The non-weighted version of WFQ.
WFQ	Weighted Fair Queuing: When the server chooses the next packet for transmission at time T, it selects, among all the packets that are backlogged at T, the first packet that would complete service in the corresponding GPS system.
WF^2Q	Worst-case Fair Weighted Fair Queuing: When the next packet is chosen for the service at time T, rather than selecting it from among all the packets at the server, as in WFQ, the server only considers the set of packets that have started (and possibly finished) receiving service in the corresponding GPS system at time T, and selects the packet among them that would complete service first in the corresponding GPS system.

Table 4-7 Performance of FQ and WFQ

Condition	When traffic is leaky bucket constrained at the source
Delay	A packet will finish service in a WFQ (FQ) later than in the corresponding GPS (PS) by no more than the transmission time of one maximum-size packet [BEN96b]
Bits	Total number of bits served for each session in a WFQ (FQ) does not fall behind a corresponding GPS (PS) by more than one maximum-size packet

4.2.2.3 Algorithm

Notation:

N: Number of entities

R_i: Service share for each entity, $\sum_{i=1}^{N} R_i = 1$

L_U: The number of data updates of an update transaction U

T_U: Service time for an update transaction U, $T_U = \frac{1}{R_i} \times L_U$ if U is from entity i

S_U: Earliest service starting time in corresponding GPS system for an update transaction U, S_U = Max (F', T'), F' is the service finish time in corresponding GPS system of last update transaction from the same entity that U is from, and T' is the arrival time of U.

F_U: Service finish time in corresponding GPS system for an update transaction U, $F_U = S_U + T_U$.

L_i, T_i, S_i, F_i: The value of L_U, T_U, S_U and F_U for the head update transaction in the waiting queue of entity i, $1 \le i \le N$. If there is no transaction in the queue yet, these values are set as $+\infty$

Algorithm:

Step 1: Set $S_i = +\infty$ for all entities at the beginning.

Step 2: Whenever there is an update transaction U from entity i submitted, calculate S_U, T_U, and F_U correspondingly. Also update S_i, T_i, and F_i if U is the current head transaction in the waiting queue of entity i.

Step 3: When the server needs to choose a transaction to verify, it chooses from all head update transactions on all entities' waiting queues whose S_i is larger than current time T and F_i is minimal to verify.

Step 4: After the verification, no matter whether the commit is successful or not, update $T = T + T_i$, and set S_i, T_i, F_i, L_i as the value of the new head transaction in queue i or set them as $+\infty$ when no more transactions are waiting in queue i. Repeat Step 2 to 4.

Note: Here we count the time to broadcast its *Ucast* even if an update transaction fails. This can control the fairness in submitted number of transactions from all entities. It can also work as a method to give penalty to misbehaving entities, like in a FQ system that still counts the transmission time of a rejected packet from a misbehaving session. However, an application can choose not to count the service time of a failed transaction.

Figure 4-14 Rescheduling the verification of update transactions using WF^2Q

Considering the similarity of the two problems discussed above, we can map an entity in STUBcast to a session in the network communication and map each update transaction to a packet. The number of updates in an update transaction is counted as a

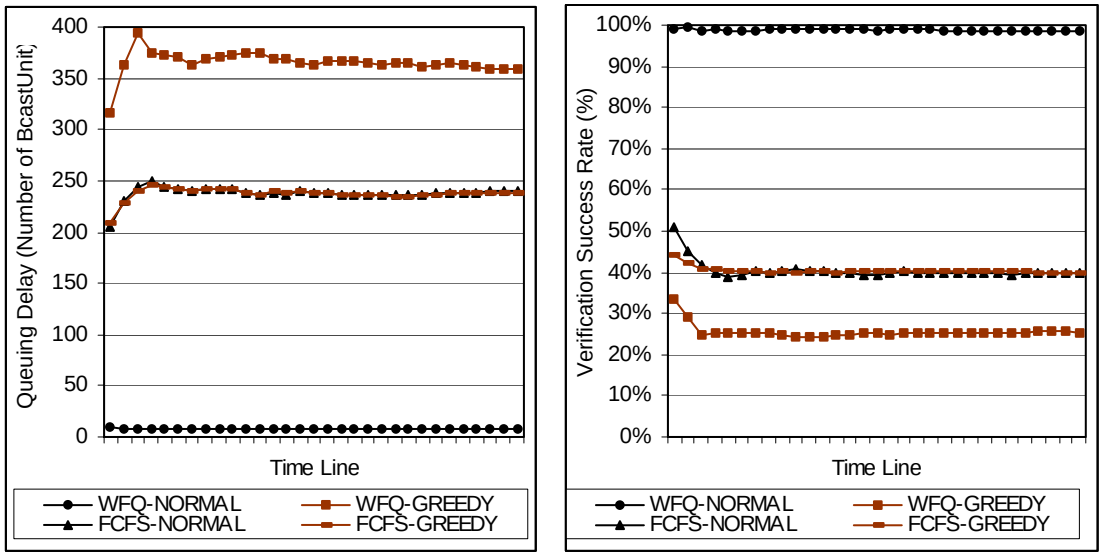
packet's size because the total time for the verification and to broadcast a *Ucast* can map to the time to transmit a packet. Consequently, the server can provide fair verification service among all client entities and provide an equal chance of successful commits among well-behaving entities. Figure 4-14 gives the detailed algorithm that was designed based on WF^2Q (to support weighted verification service among entities and future flow control at client end).

4.2.2.4 Simulation results

We have simulated the verification rescheduling using the algorithm in Figure 4-14 and compared it with FCFS. To simplify the implementation, we developed a WFQ version of the above WF^2Q algorithm. We simulated N = 2 clients, named GREEDY and NORMAL. All update transactions submitted by both clients update four data units with a UNIFORM (D256) access mode. Each update transaction also has one read operation with UNIFORM access mode. To observe the influence of queuing on transaction abort probabilities, we set all read operations' timestamp to be the one attached to the required data unit in the database at the time this update transaction is submitted to the server. By the earlier definition, each client can be assigned a fair share of 1/N = 1/2 (non-weighted) of the server resources. Therefore, a well-behaving client should submit an update transaction no faster than every $4 \times N = 8$ BcastUnits. In the simulation, client GREEDY is a misbehaving client and it submits transactions with a mean inter-arrival time of four BcastUnits (Poisson distribution). Client NORMAL is a well-behaved client and submits transactions with a mean inter-arrival time of 8 BcastUnits (Poisson distribution).

If WFQ can provide a fair verification, we expect client NORMAL transactions should never be queued longer than the service time of one update transaction

($\frac{1}{1/N} \times 4 = 8$ BcastUnits), based on the ideal fairness and performance of WFQ (Table 4-7). We can also expect all transactions from client NORMAL to achieve a nearly 100% success rate of verification. Because the timestamp of the read data is obtained from the database at the time a transaction is submitted, the UNIFORM access mode and queuing time of shorter than sending one *Ucast* makes it almost impossible to have a read-ahead status. On the other hand, client GREEDY should face a long queuing delay and low commit success rate among its transactions, which can be used as reference for a misbehaving clients to adjust its sending rate.



(a) (b)

Figure 4-15 Simulation results of WFQ verification rescheduling compared with FCFS
a) Queuing delay; b) Verification success rate

The above predictions are proved by the simulation results in Figure 4-15, which shows a queuing delay of client NORMAL transactions less than 8 BcastUnits (Figure 4-15a), and their verification success rate of nearly 100% (Figure 4-15b). However, client GREEDY receives an average queuing delay of 360 BcastUnits and verification success

rate of 25%, as expected. These prove WFQ (WF^2Q) is an efficient solution to our problem. Moreover, Figure 4-15 also shows the results using FCFS verification, where, although the two clients receive similar queuing delay and verification success rate, it is not fair because client GREEDY sends twice the number of transactions as client NORMAL, who is supposed to have no queuing delay and no verification failure as a well-behaving client.

4.3 STUBcast Index

4.3.1 STUBindex--Power Saving Solution for STUBcast-based Wireless Mobile Applications

Section 2.1.3. introduced some indexing strategies for broadcasting environments [CHE97, IMI94b]. As discussed in [IMI94b], indexing is an essential part of broadcast if it is used in a wireless mobile computing environment because of the necessity to save the battery power of portable equipment. Therefore, a STUBcast-based wireless application should also adopt efficient index schemes.

However, simply adopting existing indexing schemes into STUBcast-based server broadcasting is not enough. Because of the insertion of *Ucast*, the content of a server schedule dynamically and unpredictably changes. Consequently, the original location of each data unit pointed by index items can change in the middle of the schedule. Moreover, to guarantee consistency, client transactions essentially depend on the information provided by each *Ucast* in their transaction spanning time to detect non-serializable status. But under existing index schemes, clients only wake up when required data is on-air and cannot guarantee retrieving all *Ucast* information in all wake-up durations. These problems motivate us to modify the design of basic index schemes and

STUBcast to support power-saving in STUBcast-based wireless mobile applications. The solution described in this section is called STUBindex.

4.3.2 Assumptions and Data Index

We first define following assumptions before giving the design of STUBindex.

- We assume we will use the "partial path replication" version of "distributed index" [IMI94b] as the basic data index scheme for all *Pcast* data under STUBcast. By assuming a distributed index, the data broadcasting is composed of sequential "bcast" cycles, each of which is indexed by an index tree [IMI94b]. Let n denote the number of search keys plus pointer that a data unit can hold. And let k denote the number of levels in an index tree ($k = \lceil \log_n (DB\_SIZE) \rceil$ when the index tree is fully balanced). The description of a design and analysis of a partial-path replication distributed index [IMI94b] concluded that the optimal number of replicated levels r, which corresponds to minimizing the data access time, is:
$$\left\lfloor \frac{1}{2} \times \left(\log_n \left(\frac{DB\_SIZE \times (n-1) + n^{k+1}}{n-1} \right) - 1 \right) \right\rfloor + 1 . \quad (4\text{-}5)$$
If the index tree is a complete tree based on the value of n and DB_SIZE, then the total number of index segments M [IMI94b] in a bcast cycle [IMI94b] is:
$$n^r . \quad (4\text{-}6)$$
We choose this index scheme because it is the most efficient when providing minimal single-read operation data-access time, while the energy-saving performance is as good as other solutions [IMI94b]. We omit additional detail of the partial-path replication distributed index [IMI94b].
- When there is no update at the server, the data schedule consisted of *Pcas*t and data index, which is the same as the one in [IMI94b] that uses a partial-path replication distributed index, where each bcast schedule is divided into $M = n^r$ data segments [IMI94b] and an index segment is inserted at the beginning of each data segment.
- Additional assumptions are: each data segment has the same length; each data segment has a pointer to the beginning of the next index segment; the detailed structure of each index segment is ignored; assume using FLAT disk scheduling under UNIFORM data access mode.
- When no server update exists, a read request of data can start from the first index segment it meets, wake up at the beginning of one or two index segments [IMI94b], then traverse the nodes on the index tree path that leads to the pointer to the nearest occurrence of the required data [IMI94b]. STUBindex takes more complicated steps for this procedure because of the existence of server *Ucast.*

4.3.3 *Ucast* index

4.3.3.1 Analysis

We need to solve two problems when designing STUBindex: first, to avoid the unpredictable data location change from the index pointing to it; and second, to allow client transaction wake-up at limited indexed locations but still have knowledge of every committed update transactions. Eliminating the unpredictable insertion of *Ucast* is essential to solving both problems.

Although STUBcast inserts a *Ucast* immediately after an update transaction commits at the server, it is not necessary that it does so. The purpose of immediate *Ucast* is to provide the newest updated data as early as possible and, at the same time, let client transactions obtain the most up-to-date information about committed update transactions within their spanning time. We can sacrifice this immediate *Ucast* by delaying all new *Ucast* until the beginning of a pre-assigned index segment. Consequently, no unpredictable *Ucast* would be inserted between any index segment and the *Pcast* data segment following it. At the same time, since all information of newest committed update transactions is predictably located at the beginning of some pre-selected index segment, a client transaction only needs to tune to the channel at regular intervals to retrieve all *Ucast*.

We name the accumulated *Ucast* (*Ucast* segment) located in front of an index segment as *Ucast* Index (Figure 4-16). This is not a proper name since this accumulating of *Ucast* is actual information instead of an index to it. However, in later sections we will integrate all *Ucast* in front of one index segment into a simplified structure. Although not a structure giving the location of each *Ucast* like a data index, this simplified structure provides time and operation information of all update transactions in a given past time

period, which is like indexes pointing backward. To be described in parallel to the data index, we call this simplified structure or *Ucast* segment a *Ucast* Index. Accordingly, we will use the term Data Index (Figure 4-16) to refer to each index segment pointing to upcoming data.

4.3.3.2 Design

Any combined segment of *Ucast* index and data index is called an Integrated Index (Figure 4-16). Therefore, the indexes within a bcast are divided into two types: Single Index (the one that only includes a data index) (Figure 4-16) and Integrated Index. All integrated indexes are evenly spaced. So as to not lose generality, we define the number of Integrated Indexes in each bcast as F ($1 \leq F \leq M$) and F is divided by M. We call the time period from the end of each integrated index to the start of next integrated index a *Ucast* Period (Figure 4-16). The detailed rules used in all index types are given in Figure 4-17.

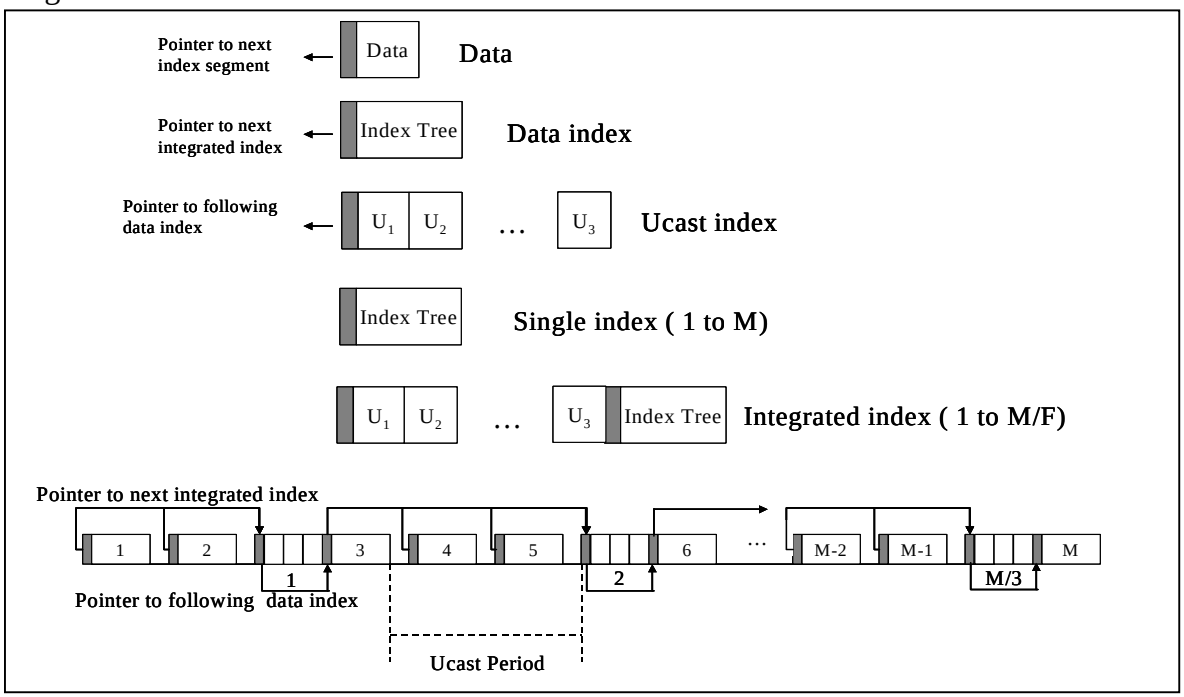


Figure 4-16 STUBindex structures

(1) Data: Each data in *Pcast* gives the offset to the start of next index (either Single or Integrated Index).

(2) Data Index: Each data index stays the same as an index segment in a distributed index, except it includes a pointer to the start of the next integrated index at the beginning.

(3) *Ucast* Index: Each *Ucast* index includes: a pointer to the following data index (at the beginning); and all *Ucast* corresponding to the update transactions committed in the past *Ucast* period, by the sequence of commit time of these transactions.

(4) Index Computation: Server computes data indexes at the beginning of each bcast; based on the value of F and M, server can easily calculate the offset of next integrated index from any data index; within each *Ucast* period, the server also accumulates update transaction information and creates a *Ucast* index for the past *Ucast* period at the beginning of each integrated index; knowing the length of current *Ucast* index, server attaches the length as the pointer to the following data index.

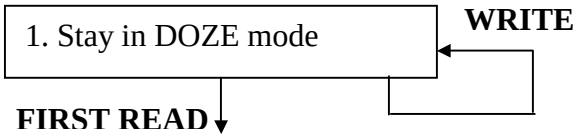
Figure 4-17 Index design in STUBindex

4.3.4 STUBindex Transaction Processing

The transaction processing policies under STUBindex are shown in Figure 4-18.

4.3.5 STUBindex Concurrency Control

We call the concurrency control part of STUBindex STUBindex-CC. Concurrency control is applied to a transaction in transaction-3(c) when a data unit is read, and is applied to transaction-3(a)/transaction-3(e) when a *Ucast* index is met in the spanning time of a transaction. Note that, based on STUBindex-CC, transaction-3(a), and transaction-3(e), even when the last operation of a transaction has finished, the transaction cannot commit/submit until the next earliest *Ucast* index is checked. STUBindex-CC is presented in Figure 4-19. It only supports single serializability.



2. At the first read

(a) The transaction probes on-air for the location of next index, and then goes back to doze mode.

(b) It wakes up at the beginning of the next index. If it is an integrated index, it obtains the pointer to the following data index and goes back to doze.

(c) The transaction tunes to the channel at the beginning of the data index, and,

- Records the pointer to the next integrated index into "NextIntg."
- Based on partial path replication distributed index scheme, in the current data index, the transaction either,
  - Finds a pointer to the intermediate index (the index segment that has the index path to the real pointer to the required data [IMI94b]), records the pointer into "IndexPoint." We call "IndexPoint" alive and "DataPoint" silent in this situation. Or,
  - Directly finds the path to the real pointer to the required data. Then the transaction starts traversing the path and finds the real pointer, records the pointer into "DataPoint." We call "IndexPoint" silent and "DataPoint" alive in this situation.

  ("IndexPoint" and "DataPoint" cannot be alive at the same time).
- Goes back to doze mode.

(d) This transaction wakes up based on "NextIntg," and "IndexPoint" or "DataPoint" (the one that is alive), whichever comes earlier.

- If a "NextIntg" comes earlier, the transaction,
  - Obtains the pointer to the following data index.
  - Adds this pointer to "IndexPoint" or "DataPoint," depending on which is alive.
  - Goes to doze mode.
  - Wakes up at start of data index, updates "NextIntg" to the current pointer to next integrated index.
  - Goes to sleep again.
  - Goes to start of Step 2(d) above and repeats the steps until the data unit is read.
- If a "IndexPoint" is alive and comes earlier, the transaction:
  - Traverses the path in the index segment and find the pointer to required data.
  - Records the pointer to "DataPoint" (alive). Sets "IndexPoint" as silent.
  - Goes to the start of 2(d) above and repeats the steps until the data unit is read.
- If a "DataPoint" is alive and comes earlier, the transaction:
  - Reads the data.
  - Updates timestamp array accordingly.
  - Goes to Step 3.

Figure 4-18 STUBindex's transaction processing diagram

AFTER FIRST READ ↓

3. The transaction goes to doze mode first then wakes up until,

- Next integrated index arrives (based on "NextIntg"). Or
- Next intermediate index arrives (based on "IndexPoint"). Or
- Next requested data arrives (based on "DataPoint"). Or
- Next operation arrives.

(a) If an integrated index comes first based on "NextIntg," the transaction,

- Wakes up at the beginning of the integrated index.
- If there is any ongoing wait for a requested data unit, then obtains the pointer to the following data index and adds it to "IndexPoint" or "DataPoint," depending on which one is alive.
- Conducts concurrency control (**STUBindex-CC**) based on the *Ucast* index and previously conducted operations. Aborts if the transaction did not pass the STUBindex-CC. Otherwise continues.
- Commits or submits the transaction if no operation left.
- If more operations left, at the beginning of the following data index, records the pointer to next integrated index into "NextIntg" and goes to the start of Step 3.

(b) If "IndexPoint" is alive and a "IndexPoint" arrives first, the transaction,

- Traverses the path in the index segment and find the pointer to required data.
- Records the pointer to "DataPoint" (alive). Sets "IndexPoint" as silent.
- Goes to the start of Step 3.

(c) If "DataPoint" is alive and a required data arrives first based on "DataPoint", the transaction

- Reads the data
- Conducts concurrency control (**STUBindex-CC**). Abort if failed, otherwise goes to the start of Step 3.

(d) If a write operation happens first, the transaction writes locally and goes to the start of Step 3.

(e) If a read operation happens first, the transaction,

- Probes on-air, obtains the pointer to the next index, and goes to doze.
- Wakes up at the beginning of next index. If it is an integrated index, conducts concurrency control (**STUBindex-CC**) based on the *Ucast* index and previously conducted operations, and records the pointer to next integrated index into "NextIntg."
- Based on partial path replication distributed index scheme, in the current data index, the transaction either,
  - Finds a pointer to the intermediate index, records the pointer into "IndexPoint." Or
  - Directly finds the path to the real pointer to the required data. Then the transaction starts traversing the path and finds the real pointer, records the pointer into "DataPoint."
- Goes to the start of Step 3.

Figure 4-18 (continued)

Policy 1: STUBindex-CC uses data structures similar to basic STUBcast, except that each timestamp array item changes to have two fields.

- Whenever a data unit is read, the timestamp attached to it is saved in the second field of its timestamp array item.
- Among all reads on this data unit, the timestamp attached to the first read data is recorded in the first timestamp array item field.

Policy 2: (Applied in transaction 3-(c)) Before reading a data unit, it checks whether the transaction's conflict array's update field of this data unit is set. Abort the transaction if it is true.

Policy 3: (Applied in transaction 3-(a) and 3-(e)) Sequentially, for each *Ucast* in a *Ucast* index, STUBindex-CC applies the same protocol used in basic STUBcast to detect read ahead status, set no-commit flag, construct conflict arrays, and detect inconsistent status, except that:

- A read ahead status is detected by using the data unit's first timestamp array field and the current *Ucast*'s data unit's timestamp, i.e., read ahead flag is set when a data unit's first timestamp array field is less than the timestamp attached with the same data unit in a *Ucast*.
- Because of the delay of *Ucast*, an updated data unit might be broadcast earlier than it appears in an *Ucast*. To adjust to this change, at the end of one *Ucast*, STUBcast-CC applies the following policies.
  - When read ahead status is detected within a *Ucast*, then for any data unit in this *Ucast*, if the corresponding second field in the transaction's timestamp array is equal or larger than the timestamp attached with the *Ucast*, abort the transaction.
  - When no read ahead status is detected by a *Ucast* is added to the transaction's conflict array, then for any data unit in this *Ucast*, if the corresponding second field in the transaction's timestamp array is larger than the timestamp attached with the *Ucast*, abort the transaction.

Figure 4-19 STUBindex-CC

4.3.6 Performance of STUBindex

We conducted some simulations of STUBindex under transaction configurations similar to the STUBcast simulations (Figure 3-4); different and new parameters for the STUBindex set-up are given in Table 4-8. In the right column, the text in bold indicate the values used in the simulation. We have collected the data on energy consumption and transaction average-response time under all given configurations.

Table 4-8 Configurations for STUBindex simulation

INDEX_UNIT (n)	The number of search keys plus pointer that a data unit can hold [IMI94b]. [n=4]
OPTIMAL_ REP_LEVEL (r)	The optimal number of replicated levels in a partial-path replication index tree, considering the data access time ([IMI94b] and formula (4-5)). [r=2 for D256; r=3 for D1024]
NUM_OF_ INDEX_SEG (M)	Total number of data index segments in a bcast cycle. $M=n^r$ ([IMI94b] and formula (4-6)). [M=16 for D256 ; M=64 for D1024]
STUB_F (F)	Number of *Ucast* indexes in a bcast cycle. [F=1, 2,4,8,16 for D256; F=1,2,4,8,16,32,64 for D1024]
Parameters in STUBcast, but with different configuration	DB_SIZE: D256 , D1024 TRAN_INTARR_TIME: Tr50 PCAST_SCHEME: FLAT ACCESS_DISTR_MODE: UNIFORM

4.3.6.1 Power-saving of STUBindex

Comparison of STUBindex with STUBcast. Table 4-9 shows the average number of BcastUnits that a set of transactions with MAX_TRAN_LEN 4 needs to be awake within their processing time, under STUBcast and STUBindex with different STUB_F values. These values can represent the average energy consumption of each transaction. Under these configurations, by using STUBindex each transaction can consume as little as 3.3% for D256 or 1.4% for D1024 of the power it would need using a primary STUBcast. Similar conclusions also can be drawn from all other simulation configurations. These show that we can achieve the goal of saving power in STUBcast-based wireless mobile applications by using STUBindex. The life of a battery can be prolonged 30 times more for D256 or 70 times more for D1024 under the configurations given in Table 4-9.

Table 4-9 Power consumption comparison between STUBcast and STUBindex

STUBcast	STUBindex						
	F=1	F=2	F=4	F=8	F=16	F=32	F=64
D256							
215	7.1	9	12.1	11.6	11.8	X	X
D1024							
838	12.1	15.1	19	28	30.3	31.6	33.6

STUBindex with different STUB_F values. We also compare the energy-saving performance of STUBindex under a different set-up of STUB_F values. We believe STUB_F value influences the performance because a transaction cannot commit or submit until the next *Ucast* index is checked, and a transaction needs to wake up at each *Ucast* index for concurrency control or for update pointers to required index or data.

Figure 4-20 gives an example of the results where r = 2, M = 16 for D256. It shows that the least power consumption is achieved when there is only one *Ucast* index (STUB_F = 1) in each bcast cycle. We interpreted this result as follows: when there are the same number of *Ucast* in each bcast cycle, each transaction can wake up the least if all *Ucast* are put in one *Ucast* index because the wake up to update the pointer to the next index or data unit is the least with the same power consumed when checking concurrency with all *Ucast*. Figure 4-20 also shows that the power consumption does not increase with the value of STUB_F for D256 and the greatest power consumption is at STUB_F = 4. Because a transaction cannot commit/submit before checking the next *Ucast* index, dividing all *Ucast* in a bcast cycle into several segments allow some finished transactions to commit/submit earlier. Consequently, such transactions will consume less power by checking less with *Ucast*. A combination of the factors above causes the results shown in Figure 4-20. A similar conclusion can also be drawn for D1024 under some

MAX_TRAN_LEN values (but Table 4-8 shows for D1024 power consumption increases with STUB_F value when MAX_TRAN_LEN = 4).

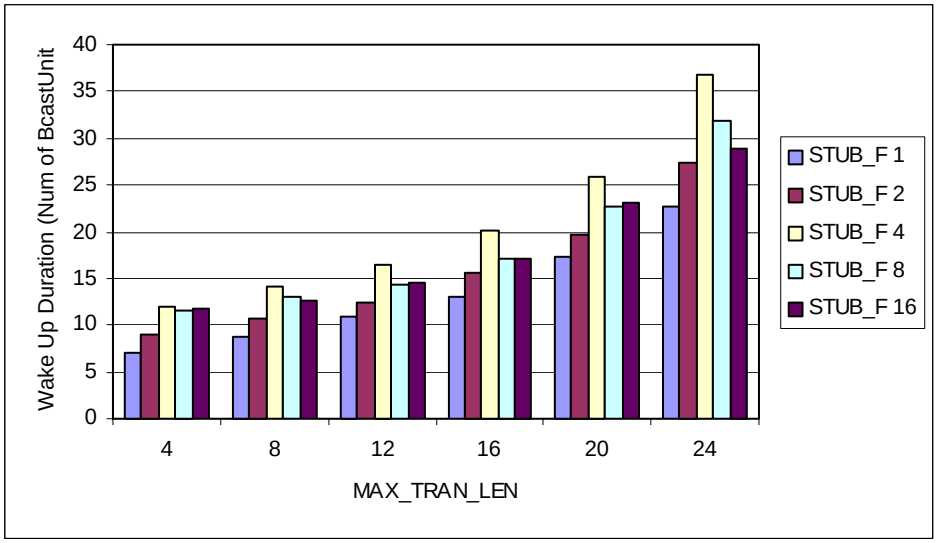


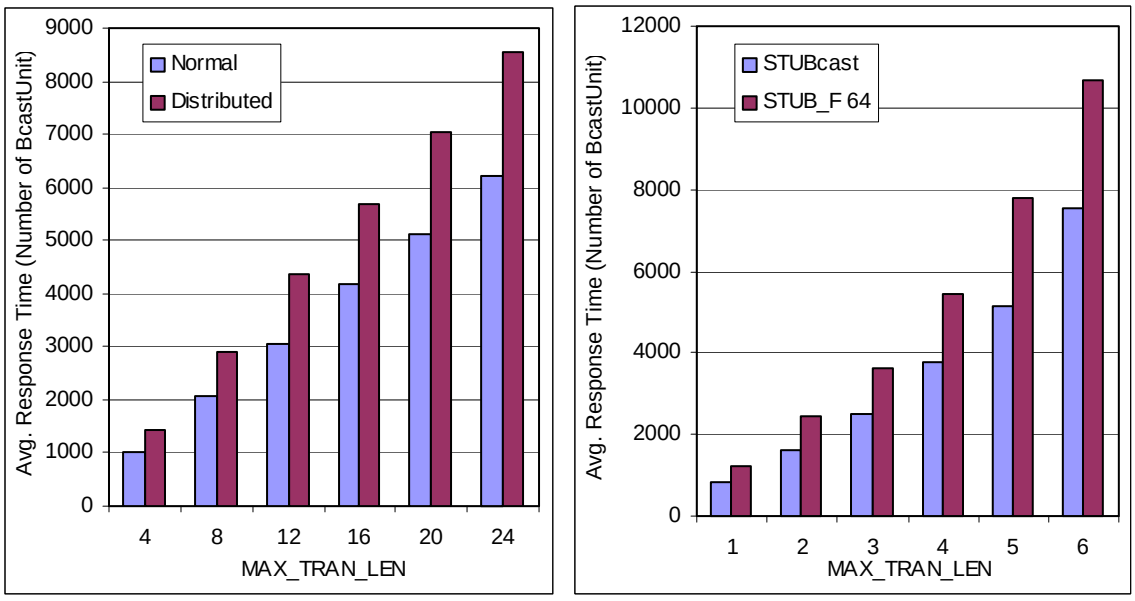
Figure 4-20 Average power consumption of transactions under different STUB_F values

4.3.6.2 Average response-time performance of STUBindex

Comparison of STUBindex with STUBcast. As discussed in [IMI94b], reducing the time a client tunes to the channel by using an index will prolong the average data access time. Therefore, although battery power is saved using STUBindex, transactions can have a longer average response time under it. According to the analysis of [IMI94b], a partial path replication distributed index can achieve the least average data access time among all given solutions [IMI94b]. Optimal value of the number of replicated levels (r) to achieve minimal data access time is given [IMI94b] and Formula (4-5).

To evaluate the average response-time performance of STUBindex, we first simulated a partial path replication distributed index (with optimal r) on a set of transactions with parameters set in Tables 3-4 and 4-8, except that all transactions are read-only. We collected their average transaction response time performance. Then we

ran this set of transactions under a normal schedule without any index and collected the transaction average response time results. Figure 4-21a gives the results under "distributed" and "normal" schedule simulations for D1024. We then simulated the same configurations, but with update transactions using STUBindex and STUBcast. Figure 4-21b gives the transaction average response time results for STUBcast and STUBindex with STUB_F = 64 for D1024.



(a) (b)

Figure 4-21 Compare average response time performance of distributed index over normal schedule and STUBindex over STUBcast. a) Normal schedule to distributed index; b) STUBcast to STUBindex

The simulated results show that the ratio of transaction response delay caused by STUBindex over STUBcast is similar to the one caused by distributed index over a normal schedule. Therefore, we say that STUBindex has very a satisfactory average response time performance because the partial-path replication distributed index with

optimal r value has the best data access time performance in many existing general index schemes [IMI94b].

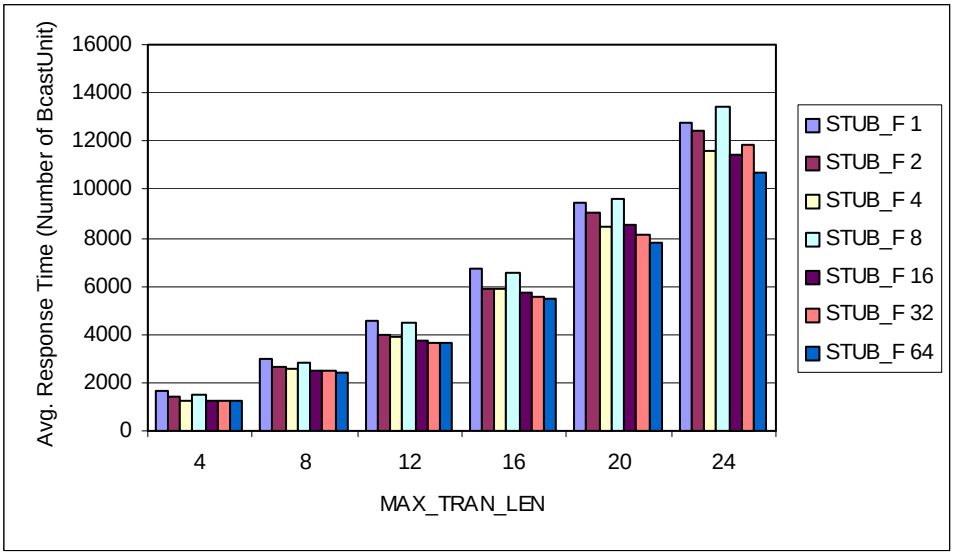


Figure 4-22 Average response time performance of different STUB_F values of STUBindex

STUBindex with different STUB_F values. Similar to power-saving performance, we also compared the influence of different STUB_F values on the transaction average response time. Figure 4-22 for D1024 shows that best performance is achieved when the *Ucast* index is inserted at the beginning of every data index. At the same time, with only one *Ucast* index each bcast the worst performance is not achieved (we expected this because it makes each finished transaction wait the longest to check with the next *Ucast* index before commit/submit), rather, it is when STUB_F = 8. In conclusion, for a real application with the same configurations, the choice of STUB_F should depend on which is more important, faster response time or greater power saving, because better performance of one means worse performance of the other since the optimal STUB_F value for one is the least optimal for the other.

4.3.7 Efficient *Ucast* Index Format

4.3.7.1 Motivation and notations

Just delaying all *Ucast* to the beginning of an integrated index might not be efficient enough to save energy because when the length of these *Ucast* is very long it will consume large quantities of power when client transactions read them from on-air. This section discusses a solution that shrinks each *Ucast* index into an efficient *Ucast* index format.

Instead of listing all the updates (with value) and read items for each *Ucast*, the efficient *Ucast* index format lists each update or read items only once, even if they are accessed by more than one transaction in the *Ucast* period. Moreover, only the ID is broadcasted in *Ucast* index. We first define some notations (Figure 4-23).

N:	Number of *Ucast* in the previous *Ucast* period.
U_i:	The ith *Ucast* in this index ($1 \le i \le N$), based on their time sequence.
U_{iw}:	The write set of U_i
U_{ir}:	The read set of U_i.
T_i:	The commit timestamp of transaction that U_i refers to.
W_M^j **:**	The item list that contains the items commonly updated by any j *Ucast* among M given *Ucast*. Such an item list is divided into C_M^j groups. Each group contains the data item IDs that are commonly updated by the transactions selected for this group. For example, W_5^4 has the items from $C_5^4 = 5$ groups: $U_{1w} \cap U_{2w} \cap U_{3w} \cap U_{4w}$, $U_{1w} \cap U_{2w} \cap U_{3w} \cap U_{5w}$, $U_{1w} \cap U_{2w} \cap U_{4w} \cap U_{5w}$, $U_{1w} \cap U_{3w} \cap U_{4w} \cap U_{5w}$, and $U_{2w} \cap U_{3w} \cap U_{4w} \cap U_{5w}$. We use L_{jWk} to represent the kth group of commonly updated items in a W_M^j, $1 \le k \le C_M^j$. The detailed item lists involved in W_M^j is displayed as $L_{jw1} L_{jw2} ..... L_{jwC_M^j}$.
R_M^j **:**	It is the same as W_M^j, except that it includes item lists of commonly read data by j *Ucast* in given M *Ucast*. L_{jRk} is used to represent the kth group of commonly read items in a R_M^j, $1 \le k \le C_M^j$.
S_N:	The detailed format of an efficient *Ucast* index.
E_N:	The shortened efficient *Ucast* index.

Figure 4-23 Notations of an efficient *Ucast* index format

4.3.7.2 Detailed efficient *Ucast* index S_N

A detailed version of efficient *Ucast* index S_N is defined as:

$$N\ W_N^N R_N^N W_N^{N-1} R_N^{N-1} .....W_N^1 R_N^1 T_1 T_2 ... T_N .$$

In this detailed format, a L_{iWk} or L_{iRk} of any W_N^i or R_N^i is listed whether or not there is any item in it or not. Using these empty items helps STUBindex recognize the owner of a group of data by the sequence number of the group, i.e., what the transactions (the value of k) that commonly read/write a group of data are.

4.3.7.3 Shortened efficient *Ucast* index E_N

The detailed efficient *Ucast* index is not efficient enough because it wastes much space on empty common item groups. This results in the *Ucast* index's length to be at least $1+N+2\times(C_N^N + C_N^{n-1} + ... + C_N^1)$ even when there are no commonly accessed data units among transactions, which easily occupies more space than the listing in the original *Ucast*. A shortened efficient *Ucast* index E_N reduces the space using a variable "Distance" to represent the number of following empty groups after each non-empty commonly accessed data group. Given N *Ucast*, the number of groups in all W_N^i and R_N^i ($1 \le i \le N$) can be decided in advance. Therefore, by using “Distance” and counting the non-empty sets, STUBindex-CC also can always recognize what the transactions are that commonly access a group of data based on the calculated sequence number.

An example E_N index is:

$$3:8:L_{1w1}L_{1w2}L_{1w3}2\ \ L_{1r3}T_1T_2T_3 .$$

This index represent three *Ucast* that have no commonly read and write items. Moreover, the first *Ucast* and the second *Ucast* have no read items.

By using the shortened efficient *Ucast* index, the ID of each data unit updated or read by a group of *Ucast* in a *Ucast* period can only be broadcast once. It can save much broadcasting space when the overlap of data access among transactions is large.

4.3.7.4 Server shortened efficient *Ucast* index-encoding algorithm

This section shows how the server decides the content of a shortened efficient *Ucast* index for a *Ucast* period, which is called server shortened efficient *Ucast* index-encoding algorithm (pseudo-code in Figure 4-24). An overview of the efficient *Ucast* index-encoding and index-decoding process between server and clients is shown in Figure 4-25.

4.3.7.5 Client efficient *Ucast* index-decoding algorithm

When a *Ucast* index in shortened efficient format is encoded and broadcasted by the server, a client transaction uses the decoding algorithm to translate it into a sequence of *Ucast* in the original format. Then STUBindx-CC uses this *Ucast* information to control the concurrency.

The decoding algorithm first constructs a table of transaction names (Figure 4-26) based on the number of *Ucast* N. For example, $Name_{jk}$ represents the sequence number of j transactions that is located at the kth in W_N^j *or* R_n^j of the efficient index (detailed). For example, name of transactions like $U_1U_2U_4$ represents the data items in this group are commonly written or read by transaction U_1, U_2, and U_4. The fields in the transaction name table are shown in Table 4-10.

```
int i=0, Distance=0;
set wgo=∅, wgo[]=∅, rgo=∅, rgo[]=∅;
start:
        wait until:
        an update transaction U commits:
                wgo=wgo[1]=L(i+1)w1=Liw1∩Uw;
                rgo=rgo[1]=L(i+1)r1=Lir1∩Ur;
                for (j=i;j>=1;j--)
                        for (k=1; k<= C_i^j ; k++)
                                Ljwk=Ljwk-wgo[k];
                                Ljrk=Ljrk-rgo[k];
                        for (k=1; k<= C_i^(j-1) ; k++)
                                wgo[k]=Ljw(k+ C_i^j )=L(j-1)wk∩(Uw-wgo);
                                rgo[k]=Ljr(k+ C_i^j )=L(j-1)rk∩(Ur-rgo);
                        wgo=wgo[1]∪wgo[2]…∪wgo[k];
                        rgo=rgo[1]∪rgo[2]…∪rgo[k];
                i++;
                go to start;
        a Ucast period finishes:
                print (i);
                for (j=i;j>=1;j--)
                        for (k=1;k<= C_i^j ;k++)
                                if Ljwk ≠ ∅
                                        if Distance ≠ 0 print (Distance);
                                        print (Ljwk); Distance=0;
                                else Distance++;
                        for (k=1;k<= C_i^j ;k++)
                                if Ljrk ≠ ∅
                                        if Distance ≠ 0 print (Distance);
                                        print (Ljwk); Distance=0;
                                else Distance++;
                for (j=1;j<=i;j++) print (Tj);
                go to start;
end.
```

Figure 4-24 Server shortened efficient *Ucast* index encoding algorithm

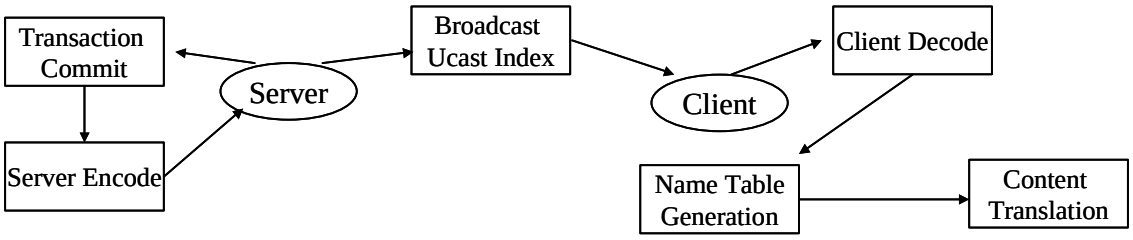


Figure 4-25 The process of efficient *Ucast* index-encoding and index-decoding

Table 4-10 Fields in transaction name table for efficient *Ucast* index

Number of transactions for commonly accessed data (j)	Sequence number in W_N^j *or* R_n^j (k)	Name of transactions ($Name_{jk}$)

```
start:
        set all Name as "".
        read (N);
        for (i=0;i<=N-1;i++)
                Name(i+1)1=Namei•Ui+1;
                for (j=i;j>=1;j--)
                        for (k=1;k<= C_i^{j-1} ;k++)
                                Namej(k+ C_i^j )=Name(j-1)k•Ui+1
end.
```

Figure 4-26 Client-efficient *Ucast* index-decoding algorithm--transaction name table construction

After constructing the name table, the client traverses the efficient *Ucast* index and decides T_i, U_{iw}, and R_{iw} for all N update transactions using the algorithm (pseudo-code) in Figure 4-27.

```
set Data = ∅;
int count=0;

start:
        for (j=N;j>=1;j--)
                for (k=1;k<= C_N^j ; k=k+count;)
                        if (Data == ∅)
                                Read (Data);
                        if (Data contains Distance)
                                if (distance <   C_N^j  - k)
                                        count=distance; Data=∅; continue;
                                else
                                        count= C_N^j  - k; Distance-=count; continue;
                        if (Data contains item list L_jwk)
                                for each U_m in Name_jk, put all data in L_jwk
                                into the update items set of its Ucast

                for (k=1;k<= C_N^j ; k=k+count;)
                        if (Data == ∅)
                                Read (Data);
                        if (Data contains Distance)
                                if (distance <   C_N^j  - k)
                                        count=distance; Data=∅; continue;
                                else
                                        count= C_N^j  - k; Distance-=count; continue;
                        if (Data contains item list L_jrk)
                                for each U_m in Name_jk, put all data in L_jrk
                                into the read items set of its Ucast

        for (j=1;j<=N;j++)
                Read (T_j) and set it as the commit timestamp of the jth Ucast.

end.
```

Figure 4-27 Client efficient *Ucast* index decoding algorithm -- *Ucast* content translations

CHAPTER 5
TRANSACTION PROCESSING IN HIERARCHICAL BROADCASTING MODELS

Most research addressed in a broadcast-based asymmetric communication environment, including what we proposed in the last two chapters, is based on one-level broadcasting, where a single server broadcasts to all possible clients directly and clients can only access this server's broadcast (Figure 5-1). Although the physical-level support of broadcasting (such as satellite, base station) allows this one-level model to fit into most applications, there are some situations where hierarchical broadcasting can perform better or may become necessary. Hierarchical broadcasting means there is more than one level of data communication in the system architecture and at least one of those levels of communication is broadcasting. This chapter addresses some transaction processing issues, focusing on concurrency control, in several hierarchical broadcasting models to be defined later. These studies also give us an opportunity to show how to use STUBcast, STUBcache, STUBindex, and other transaction-processing strategies described in Chapters 3 and 4 in a different application environment.

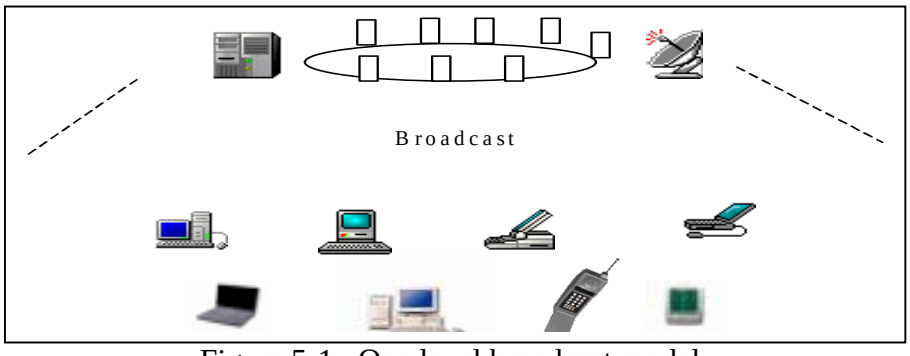


Figure 5-1 One level broadcast model

5.1 Proxy Server Model

5.1.1 Proxy Server

Client caching is very important for achieving fast data-access response in data broadcasting. However, some client equipment might not have the ability to cache broadcasted data or has a very small cache size. In order to provide adequate cache for these clients, proxy servers can be used in a broadcast model. A proxy server [LAW97] can be inserted between a data broadcast server and a group of clients (Figure 5-2). This proxy server listens to the server broadcast like a client and acts as cache for this group of clients based on their data interests. When accessing data, a client can either listen to the broadcast channel or fetch cached data at the proxy server. Unfortunately, some clients might not be able to retrieve data from broadcasting. For example, they might not have satellite receivers. For such types of users to take advantage of broadcasting, a proxy server can be inserted on top of a group of such users to act as receiver for them.

Both request/response and broadcast can be used as the method of communication between the proxy server and a group of clients. When request/response is used, each client sends a request to the proxy server and the response is received individually. When broadcast is used (e.g., the base station of a cell can act as a proxy server and broadcast data to the cell), the proxy server broadcasts a relatively small amount of cached data to the group of clients under it.

The proxy server can be a base station in a wireless environment or a fixed host, such as a router in a wired Internet environment. In an ad-hoc wireless network environment, any mobile unit with caching or satellite signal receiving ability can act as a temporary proxy server for other units in this ad-hoc network. Moreover, Bluetooth technology [HAA98] can be used in small proxy server units with cache or receiver.

These proxy server units with Bluetooth chips can be distributed anywhere. Therefore, mobile units can also detect and connect with these proxy units using Bluetooth technology. We address possible proxy models and their concurrency control in the following sections.

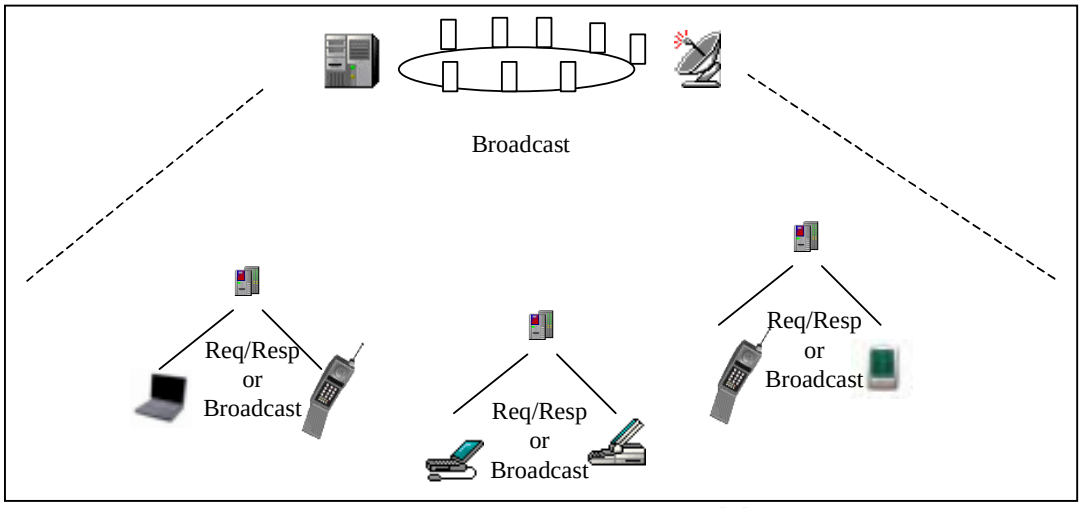


Figure 5-2 Proxy server model

5.1.2 Proxy Models

There are a number of options that can be applied to a broadcast-based proxy server model. For applications where proxies are only used for caching, clients also can access data by server broadcasting. An index scheme can either be chosen or not chosen in server broadcasting based on the energy consumption restraints of the client. In some cases clients will have their own limited cache, so proxies are used as second-level caches. For applications where proxies are used as receivers, clients have to retrieve data from the proxy server. Clients can also cache data retrieved from proxies in this type of applications. Moreover, both request/response and broadcast can be used for communication between proxies and their clients. Therefore, someone using a broadcast-based application but in need of the use of proxy servers must decide which proxy model

is most suitable. Here we define all possible broadcast-based proxy-server models based on whether the proxy is to be used for cache or as a receiver, whether or not the index is used in broadcasting, whether or not clients have local cache, and the communication options among data server, proxy server, and clients (Table 5-1).

When proxies are used as receivers, we assume that they always also provide cache functions. We assume whenever a client request for data is not satisfied immediately (by local cache, server broadcast, proxy broadcast, etc), the client always sends a request to the proxy server. When the proxy can broadcast data to clients, we assume clients can still communicate with the proxy server to let it know about its data requests (such that proxy can cache data efficiently).

Table 5-1 Possible proxy models and their features

Type	Server-to-client communication	Proxy-to-client communication	Client cache	Index	Purpose
1	Broadcast	Broadcast or Req/Resp	No	No	Cache or receiver
2	No	Req/Resp	No	No	Both
3	No	Broadcast	No	No	Both
4	Broadcast or No	Broadcast or Req/Resp	Yes	No	Both
5	Broadcast or No	Broadcast or Req/Resp	No	Yes	Both
6	Broadcast or No	Broadcast or Req/Resp	Yes	Yes	Both

5.1.3 Concurrency Control in Proxy Models

When transaction processing is used in the proxy models above, existing concurrency control strategies cannot be used directly because more complications are added by the insertion of a middle-level proxy server. Based on the same assumptions for transactions and schemes as used in STUBcast, we describe concurrency control strategies for the proxy models listed in Table 5-1 in Figures 5-3 to 5-8.

Type 1. Communication relation between top server and clients is broadcast; communication relation between proxy server and clients is broadcast or request-response (Figure 5-3).

- Use **STUBcast** protocol on the clients and top server as in one-level model.
- Consider proxy server as cache and apply the schemes of **STUBcache**.
- The only difference is how to handle the **delays caused by fetching data** from the proxy server.

Server:

Use the same protocol as STUBcast's SVP.

Client protocol:

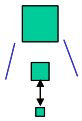
1. Client tries to read data A but not on-air (we call the broadcast channel from server to clients the primary channel in order to differentiate it from the channel from proxy to clients in the case of proxy broadcasting), so client A sends a request to the proxy server.
2. Client continues to wait on the primary channel and applies STUBcast protocol policies, and
   a. If A appears on the primary channel before any response or broadcast of A from proxy server, simply read it and discard any later response from A.
   b. If A is available from proxy server earlier than any appearance of A on the primary channel
      i. It is in *Pcast* stage on the primary channel, then simply read data A.
      ii. It is in *Ucast* stage on the primary channel, wait for this *Ucast* to finish; if A appears in the *Ucast*, take that value and discard the one from the proxy server, otherwise read the A from proxy server.

Proxy server:

1. Receive client requests and sequence them by arrival time.
2. Respond to a request if the corresponding data is in the cache.
3. At the same time, for data not in the cache, try to grab them on-air out of order, and send them to clients once it is available.
4. In the proxy broadcast, primarily broadcast each data unit in the cache sequentially or by other algorithms. Insert a response for a new request or for newly cached data for waiting requests when necessary. The data for response are also served in batch style.
5. For the cache update, update the value and time of cached data whenever a data unit is in the *Ucast*. However, it is not necessary to set the *Ucast* flag for each cached data unit, unlike STUBcache.

Figure 5-3 Concurrency control for proxy model Type 1

Type 2. There is no communication relation between the top server and clients; the communication relation is between a proxy server and its clients (request-response) (Figure 5-4).



- Use **STUBcast** protocol at the top server as in one-level model.
- Also use STUBcast at client end but clients cannot access each *Ucast* directly. Modifications are made subsequently.
- Consider proxy server as cache and apply the schemes of **STUBcache**.
- The major difference is **how to let all clients know that the *Ucast* happened at server.**

Proxy server:

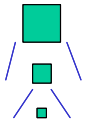
1. Cache data use same schemes as in basic STUBcache.
2. Keep the history of update transactions and their update list and read list based on time sequence.
3. Proxy server receives data requests from all clients under it and can handle these requests out of order based on whether or not the data are in the cache or when they can fetch the data from the air.
4. Proxy server replies to a request when the data unit is in the cache or it appears in the server broadcast, and
   a. If it is in a *Pcast* stage, send the data unit and also attach the part of the update transaction history whose timestamps are after the *LastUcastTime* to the client. When client receives the history of transactions, it updates *LastUcastTime* to the timestamp attached with the last *Ucast* in the history.
   b. If it is in a *Ucast* stage, take the actions in 4a after the *Ucast* finishes.

Client protocol:

1. Client tries to read data A and sends a request to the proxy server.
2. Client also keeps and attaches a timestamp called *LastUcastTime* to the request.
3. When client receives the history of transactions, it updates *LastUcastTime* using the timestamp attached with the last *Ucast* in the history. It also uses the history to update its conflict array.
4. Client can notify proxy server regularly about the earliest started running transactions such that the proxy server only needs to keep the update transaction history after the earliest started transaction among all clients under it.

Figure 5-4 Concurrency control for proxy model Type 2

Type 3. There is no communication relation between the top server and clients; communication relation between a proxy server and its clients is broadcast (Figure 5-5).



- Use **STUBcast** protocol at the top server and client as in one-level model.
- Proxy server will broadcast all *Ucast* to the clients.
- Also consider proxy server as cache and apply the schemes of **STUBcache**.

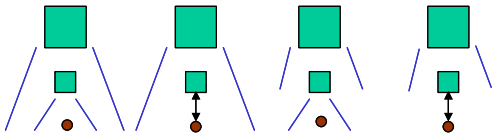
Proxy server:

1. Cache data using the same schemes as in basic STUBcache.
2. Broadcast cached data sequentially, and
   a. Insert a cached data to the cycle if there is a request for it.
   b. Insert a requested data not cached but appearing on-air.
3. Insert a *Ucast* whenever it appears on the air.

Client protocol:

1. Client tries to read data A and sends a request to the proxy server.
2. Then client reads from the air until the data appear. It also monitors the *Ucast* and applies the same protocols used in basic STUBcast.

Figure 5-5 Concurrency control for proxy model Type 3

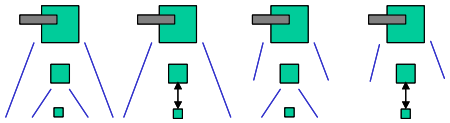


1. Use similar schemes to those used in Type 1-3. However, each client caches some data itself, and only sends requests to the proxy when the data are not in its own cache or not on-air.
2. The scheme for client to cache data is STUBcache + delay handling for cases (a) and (b).
3. For case (c), client caches server response with the timestamp attached to it.
   a.If a request can be satisfied with a cache read, and,
      i. This is the last operation, then commit.
      ii. This is not the last operation, then continue to next operation.
   b.When an operation cannot be satisfied by cache, a request is sent to server and same scheme used in Type 2 is used, and
      i. A cached data unit is discarded from the cache if updated in this *Ucast* history.
      ii. If an earlier cache read's data unit is included in the update history, a read ahead status is set for the corresponding transaction and conflict array is updated.
      iii. Conflict array is also updated based on the history and conflict chain theory.
4. For case (d), use STUBcache to cache data broadcast by proxy server and send request to proxy server whenever a data unit is not available in the cache or on-air.

Figure 5-6 Concurrency control for proxy model Type 4

Type 4. Communication relation between top server and clients is broadcast or there is no communication; the communication relation between a proxy server and its clients is broadcast or request/response; clients also have cache (Figure 5-6).

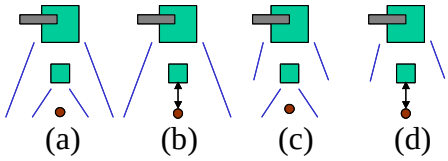
Type 5. Communication relation between top server and clients is broadcast or there is no communication; the communication relation between a proxy server and its clients is broadcast and request/response; top server uses index for broadcasting (the proxy server broadcast is dynamic so there is no need to use index) (Figure 5-7).



1. Use similar schemes as those used in Type 1-3. The only difference is how proxy server and client access the air. So we will use **STUBindex** at server, proxy server, and clients. The proxy server can access data out of order for all client requests.
2. In the cases of (a) and (b), clients use index to wake up at proper time of *Ucast* and data, as in basic STUBindex.
3. In all cases,
   a. Proxy server always wakes up at *Ucast* indexes.
   b. Proxy server uses time-out queue to sequence the distance to all data that it needs to access from the air and wakes up at proper time to cache or read them.
4. Protocol used among proxy and clients stays the same as in Type 1-3.
5. STUBindex protocol for client also applies to clients in cases (a) and (b) and proxy server in all cases.

Figure 5-7 Concurrency control for proxy model Type 5

Type 6. Communication relation between top server and clients is broadcast and there is no communication; the communication relation between a proxy server and its clients is broadcast and request/response; top server uses index for broadcasting (the proxy server broadcast is dynamic so there is no need to use index); clients also use cache (Figure 5-8).



Simply use a combination of the solutions to all types above.

Figure 5-8 Concurrency control for proxy model Type 6

5.2 Two-Level Server Model

This section discusses two other hierarchical broadcasting models, a data distribution server model and a common and local data interest server model. They are also called two-level server models because both involve two levels of major data servers.

5.2.1 Data Distribution Server

In some applications, the broadcast data server has a large database and supplies data to clients over a great area (such as globally). However, instead of all data being provided at the server, the clients under it can have different data interests depending on their location. Moreover, some clients cannot afford the equipment to receive signals broadcast to a large geographic area. Several problems are inherent if a one-level broadcast model is used in such an application environment. First, a client without a powerful receiver cannot use the application at all. Moreover, the server broadcast schedule would be very inefficient because the database size is large and data interest is non-uniformly distributed among different client locations. Therefore, we propose another hierarchical broadcasting model (Figure 5-9), which uses second-level servers, where a level of servers is inserted between the top data server and clients. These second-level servers are responsible for data fetching from the top server and data broadcasting

to a local area where data interests are similar among clients. These servers are distributed at different locations. Therefore, we call them data distribution servers.

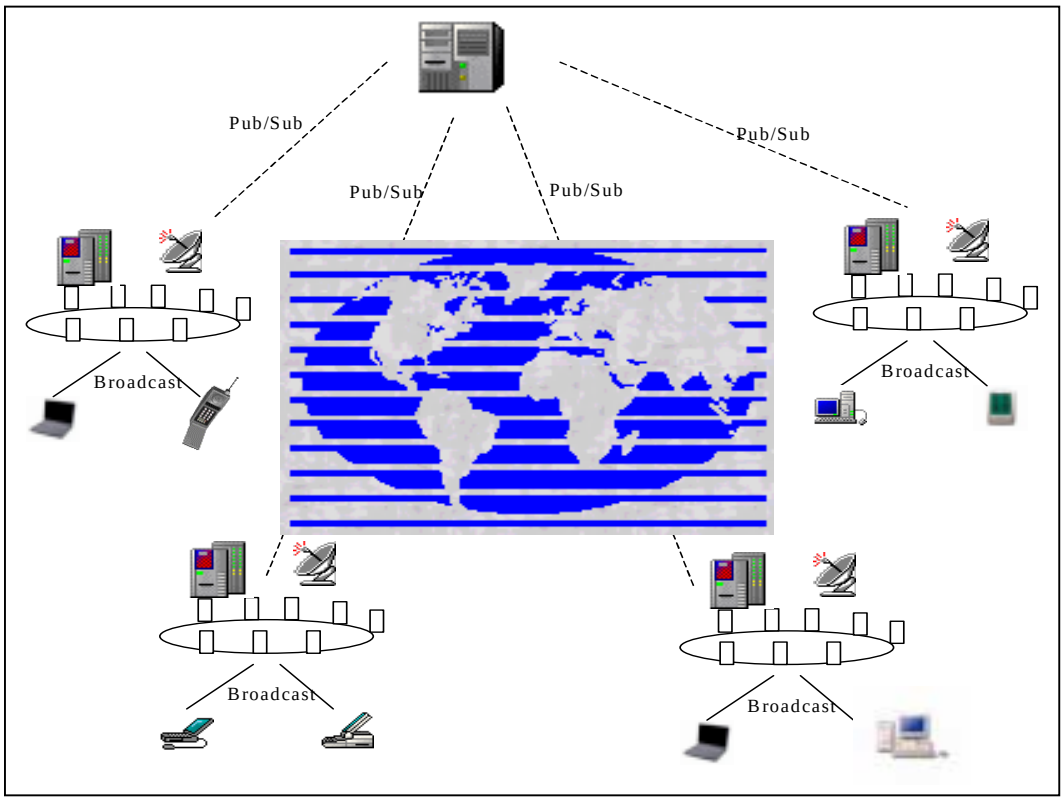


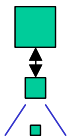
Figure 5-9 Data distribution server model

5.2.1.1 Communication model between top server and data distribution server

The available choices are request/response, publish/subscribe, and broadcast. We propose using publish/subscribe since it best suits this type of applications. Since each data distribution server knows the data interests among local clients, it can send the combined data interest profile to the top server. The top server would publish any new data or updated data to a group of data distribution servers based on the profiles it received from all second-level servers. Each data distribution server then broadcasts all data of local interest published from the top server to local clients based on a specific schedule.

5.2.1.2 Concurrency control

We also designed a basic concurrency control scheme (Figure 5-10) for data distribution server to accommodate new complications.



1. Use STUBcast on data distribution server and clients.
2. Clients submit update to distribution server and it is forwarded to top sever.
3. Use SVP on top server. Then top server publishes a *Ucast* to each distribution server based on interest.
4. Distribution server then broadcasts *Ucast* in its schedule.

Figure 5-10 Concurrency control for data distribution server model

5.2.2 Common and Local Data Interest Server

In some applications, there is a group of data items that commonly interest all clients. At the same time, clients are also interested in different groups of data based on their locations. A one-level broadcasting model, where a top server collects/maintains/broadcasts all common-interest data and different location-based data, is not efficient. For this type of application, we propose a hierarchical broadcasting model (Figure 5-11) where servers are also divided into two levels. The top-level server collects/maintains/broadcasts common-interest data and the second level server at a different location collects/maintains/broadcasts local-interest data. Each client can listen to both levels of servers. They access common-interest data from the top sever broadcasting channel and local-interest data from the second-level server-broadcasting channel. We call these servers common and local data interest servers.

In the common and local data interest server model, both top server and second-level server broadcast data to clients. A complication of this model is client transactions can read or write data from either level of server. When a transaction commits, it needs to verify whether it maintains consistency and read concurrent data on both servers. Therefore, concurrency control is also an important issue that needs to be addressed. To address this problem, a solution scheme is given in Figure 5-12.

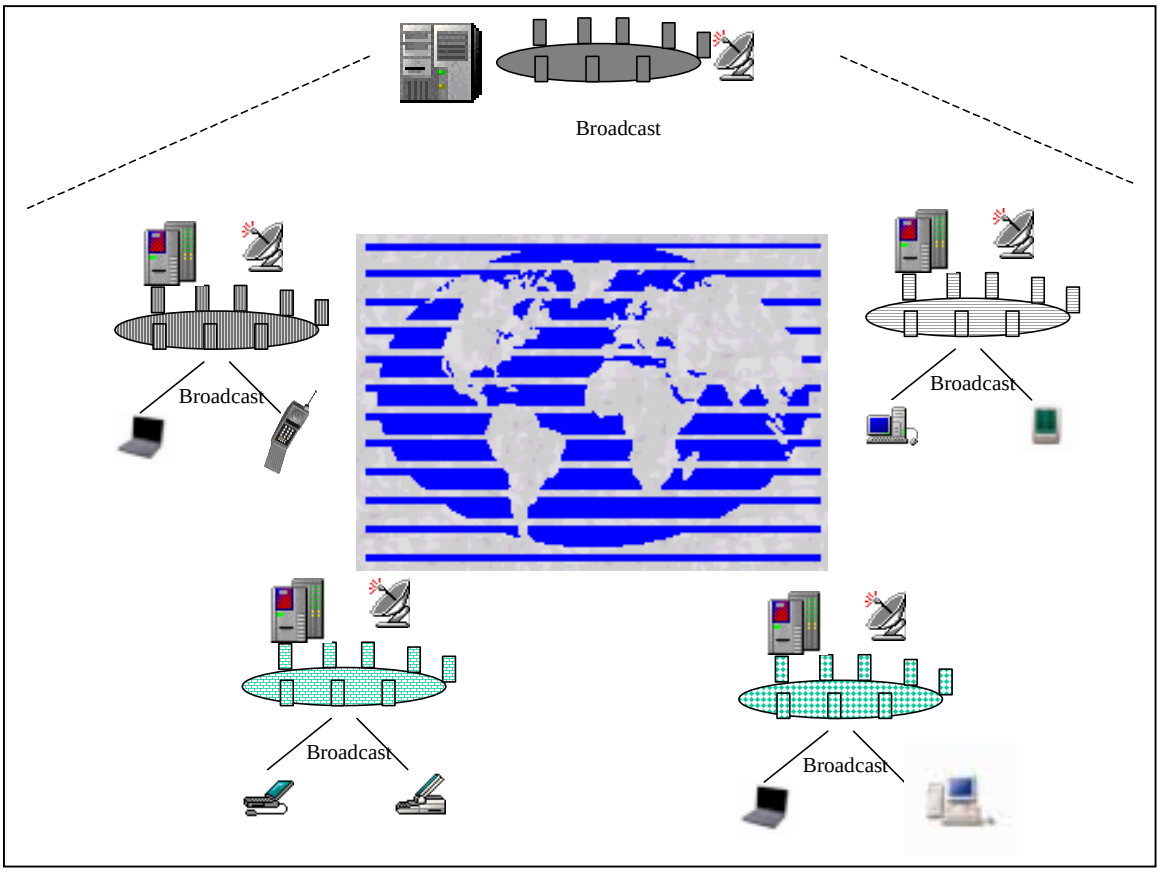
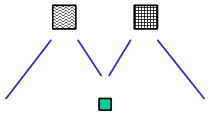


Figure 5-11 Common and local data interest server model



1. Apply STUBcast on all servers and the clients.
2. Each client reads from two channels separately.
3. TSA, CFA, RECarray, RAF, and NCF are maintained separately for each channel.
4. An update transaction at client first sends both RECarrays to the local server then verifies the local server RECarray. If it is accepted, the local server sends the other RECarray to the common server for verification. Common server inserts a *Ucast* into its own broadcast if it is successful and returns a success response to the local server. Otherwise a failure is returned. Based on the result, the local server sends a success or failure reply back to the client. A *Ucast* for the local server is also inserted if it is a success. Note that each update transaction on a local server must wait on the one in front of it to receive a notification from the common server before its own verification starts.
5. Another option is to send two RECarrays separately to corresponding servers and verify them at each server. Then the two servers send the result to the other one. An update transaction can only commit when it is verified as acceptable at either server and receives an accept notice within a given time-out from another server. Each update transaction can only be verified when the one in front of it has been completely accepted or rejected at the current server.
6. The first update verification scheme can reduce overhead at the common server because less clients access it. But the second scheme can make the verification in parallel and reduce response time.

Figure 5-12 Concurrency control for common and local data interest server model

CHAPTER 6
PERFORMANCE ISSUES FOR TRANSACTION PROCESSING

This chapter addresses some miscellaneous transaction performance issues in a broadcasting environment. We study two scheduling and two caching issues related to transaction processing in this chapter and aim at meeting a specific performance requirement in each issue. In scheduling, we assume no cache is used.

6.1 Scheduling

6.1.1 Balance and Minimize Transaction Response Time

6.1.1.1 Transaction response time fairness

The objective of data broadcast scheduling is to achieve a good performance of average access time for each read operation. If we assume we know the overall access possibility of each data unit, the essential idea of most solutions includes broadcasting hot data more frequently than cold; and instances of each data unit should be evenly spaced in the schedule [ACH95a, HAM97, VAI97a]. A frequency assignment, which gives optimal average access time, is given in [HAM97, VAI97a]

However, these studies are based on single operations. They do not evaluate performance in circumstance of different lengths of transactions and different distributions of data interests among transactions in transaction-processing applications. Here we investigate a problem called transaction response time fairness.

When transaction is the major entity of an application, users would expect to achieve short response time for a whole transaction instead of individual operations.

When transactions are of different lengths, a user would expect to have fair response times among these transactions, that is, the response time should be proportional to the length of the transaction. This section defines response time of a transaction as the summation of the response times of all its sequential read operations. Tables 6-1 and 6-2 show an example where transactions with different lengths have unfair average response time using a broadcast schedule.

Table 6-1 Transaction parameters – a

Transaction length	Percentage in all transactions	Data access (1 … 1000)
4	25%	Uniformly distributed access of data 201 … 1000
3	25%	Uniformly distributed access of data 1 … 200
2	25%	
1	25%	

Table 6-2 Transaction parameters – b

Data	Overall access possibility	Access possibility for different transaction lengths			
		4	3	2	1
1…200	0.3%	0%	0.5%	0.5%	0.5%
201…1000	0.05%	0.125%	0%	0%	0%

Let P_i represent the overall access possibility of data i. Let F_i represent the broadcast frequency of data i. Assuming all data have the same length, optimal overall average access time can be achieved [VAI97b] when:

$$\frac{F_i}{F_j} = \sqrt{\frac{P_i}{P_j}} \rightarrow \frac{F_i}{F_j} = \sqrt{\frac{0.3\%}{0.05\%}} \approx 2.45 \left(1 \le i \le 200, 201 \le j \le 1000\right). \qquad (6\text{-}1)$$

Based on Formula (6-1) and [VAI97b], when optimal average data access time is achieved, data items from 1 … 200 are broadcast 2.45 times as fast as any data items

from 201 … 1000. However, from the transaction point of view, the response times of different lengths of transactions are not fair (Figure 6-1). Length 4 transactions' response time is not proportional to transactions of other lengths. The average response time of each length 4 transaction's operation is 2.45 times longer than other, shorter transactions. This example shows that to ignore distribution of data interests in different length transactions could cause unfair response times in transactions. It might not be acceptable when a transaction is the major performance unit in an application.

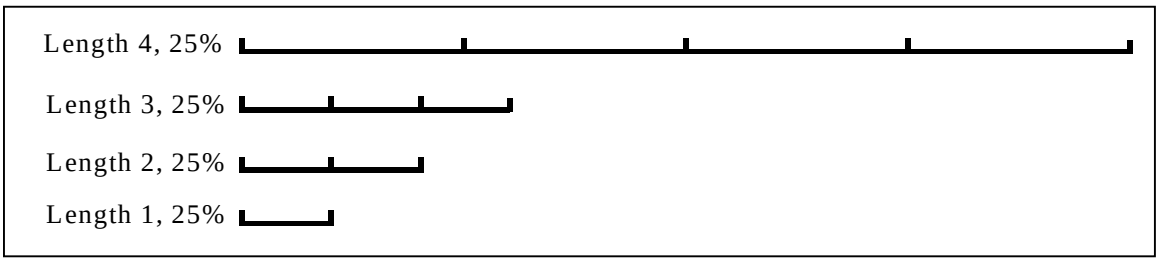


Figure 6-1 Unfair transaction response time

Nevertheless, fair response time in transactions is the not the only concern. Long response time in all transactions is even more unacceptable. Therefore, an application's user should define a threshold value for transactions with specific length to be longer than the shortest ones, and at the same time achieve minimal response time for all transactions. We should also consider the percentage of transactions with a specific length. The fewer transactions with a specific length that exist in an application, the more acceptable it is for them to have longer response time. Therefore, the objective should be to achieve weighted, fair, and minimal transaction response time.

6.1.1.2 Notations and problem definition

Notations and terms. Some notations and terms used to formally define the problem are given in Figure 6-2.

MaxLen: Maximal length (number of reads) of a transaction. Operations are sequential.

TLP_i: Percentage of transactions with length i.$1 \le i \le$ MaxLen. $\sum_{i=1}^{MaxLen} TLP_i = 1$.

P_j: Overall data access possibility. $1 \le j \le$ DB_SIZE. $\sum_{j=1}^{DB\_SIZE} P_j = 1$.

DDP_{ij}: Percentage of accesses to data j that are in length i transactions. $\sum_{i=1}^{MaxLen} DDP_{ji} = 1$.

DAP_{ij}: Percentage of all operations accessing data j in length i transactions.

S_j: Broadcast distance between any two instances of data j.

R_i: Average response time of a read in length i transactions. $R_i = \sum_{j=1}^{DB\_SIZE} DAP_{ij} \times S_j$.

$R_{Shortest}$: Shortest R_i under a given scheduling of $S_1, S_2, \ldots, S_{DB\_SIZE}$.

θ: Maximal acceptable value of $\frac{R_m}{R_n}$ ($1 \le m, n \le$ MaxLen, $m \ne n$) if $R_m > R_n$.

Uniform Distribution of Transactions Length: if $TLP_i = 1 / \text{MaxLen}$ for all i, otherwise it is non-uniform distributed.

Uniform Distribution of Data by Transaction Length: if

$$DDP_{1j} : DDP_{2j} : \ldots : DDP_{MaxLen\ j} = TLP_1 \times 1 : TLP_2 \times 2 \ldots : TLP_{MaxLen} \times MaxLen$$

Figure 6-2 Notations and terms

Table 6-3 Example of DDP_{ij} and DAP_{ij}

DDP_{ij}	i = 4	i = 3	i = 2	i = 1
$1 \le j \le 200$	100%	0%	0%	0%
$201 \le j \le 1000$	0%	50 %	33.33%	16.66%
DAP_{ij}	i = 4	i = 3	i = 2	i = 1
$1 \le j \le 200$	0%	0.5%	0.5%	0.5%
$201 \le j \le 1000$	0.125%	0%	0%	0%

In the earlier example, MaxLen = 4, DB_SIZE = 1000, TLP_i = 25% ($1 \le i \le 4$), P_j = 0.3% for $1 \le j \le 200$, P_j = 0.05% for $201 \le j \le 1000$, and access is non-uniformly distributed by transaction length. Table 6-3 shows DDP_{ij} and DAP_{ij} ($1 \le i \le 4$, $1 \le j \le 1000$) in the earlier example. Furthermore, we assume the distributions of data j within all

length i transactions are uniform, that is, accesses to data j is evenly divided in all length i transactions.

Problem definition. Based on the motivation and notations, a balancing and minimizing transaction response-time problem, which addresses the performance issues of fairness and response time among transactions, is defined as:

Given an application with parameters MaxLen, DB_SIZE, TLP_i, DDP_{ij}, P_j, DAP_{ij}, $(1 \le i \le MaxLen, 1 \le j \le DB\_SIZE)$, and θ $(\theta \ge 1)$, find the broadcast schedule of $S_1, S_2, \ldots S_{DB\_SIZE}$ that has the minimal $R_{Shortest}$ from all schedules which satisfy:

$$\frac{1}{\theta} \le TLP_M \times R_M \Big/ TLP_N \times R_N \le \theta \quad . \tag{6-2}$$

For any (M, N) pair, $1 \le M, N \le MaxLen, M \ne N$. When θ equals 1 and transaction length is uniformly distributed, it implies all R_i are required to have the same length.

6.1.1.3 A solution using branch-and-bound

We have designed a branch-and-bound solution for a subset of the problem defined above. The problem subset is defined by following constraints.

- The database can be grouped into K parts, $K \le \text{Min}\left(\frac{MaxLen \times (MaxLen-1)}{2}, DB\_SIZE\right)$, with size Z_k, $\sum_{k=1}^{K} Z_k = DB\_SIZE$, $1 \le k \le K$, such that all data m in one Part k have the same value of P_m and all DDP_{mj}, and thus all DAP_{mj} $(1 \le j \le MaxLen)$. Therefore, if a solution schedule exists for above problem, the broadcast distance S_m for all data in Part k must be the same. So the problem becomes one of searching for $S_1, S_2, \ldots, S_K$ for K groups of data.
- Each data unit has a maximal acceptable broadcast distance B_j, $1 \le j \le DB\_SIZE$, i.e. $S_j \le B_j$ (each data unit must be repeated at least once for a distance of B_j). Additionally, the value of B_m for all data m in Part k is also the same (then we use B_k for Part k).
- All data are the same size, so all S_j values (based on the number of broadcast unit) must be integers.

- The total bandwidth requirement for all data cannot exceed the bandwidth of broadcast channel. The percentage of bandwidth occupied by each data unit j is $\frac{1}{S_j}$ with minimal percentage $\frac{1}{B_j}$. Therefore, we have $\sum_{k=1}^{K} \frac{Z_k}{B_k} \le 1$ and $\sum_{k=1}^{K} \frac{Z_k}{S_k} \le 1$. A schedule for a given problem should utilize the bandwidth as much as possible. When a solution fully utilizes the bandwidth, we have $\sum_{k=1}^{K} \frac{Z_k}{S_k} = 1$.

The branch-and-bound solution is then given as follows.

Step 1: List Formula (6-2) for all (M, N) pairs, 1 ≤ M, N ≤ MaxLen. Based on the definition of R_i ($R_i = \sum_{j=1}^{DB\_SIZE} DAP_{ij} \times S_j$), each of these formulas will have at most K items from S_1, S_2 … to S_K. The total number of formulas will be

$$N_f = \sum_{i=1}^{MaxLen} (MaxLen - i) = \frac{MaxLen \times (MaxLen - 1)}{2}.$$

Step 2: Sequence all N_f formulas by the number of S_k involved, ordered from smallest to largest. Let $f_1, f_2, ... f_{N_f}$ be these formulas (also the number of S_k involved) by their sequence.

Step 3: Start from the first formula f_1. Assume this formula has $S_{11}', S_{12}', \ldots, S_{1f1}'$ involved. Build a search tree for these items, as shown in Figure 6-3. The nodes on the first level of the search tree represent all possible broadcast distances 1, 2, … B_{11}', based on maximal distance and integer distance constraints noted above. Similarly, the nodes on the kth level represent all possible distances 1, 2, … B_{1k} for S_{1k}, 1 ≤ k ≤ f_1. The tree is built based on the sequence of Depth-First Traversing. Whenever a new child is expanded from its parent node, total bandwidth constraint is verified based on all nodes on the path from the root to this node. Also, when a level f_1 node is expanded, formula f_1 is also verified based on all nodes on the tree path to it. If a new child fails to meet any

constraint, this node will be removed from the tree (colored black in Figure 6-3; otherwise colored green) and the search will return to the parent. The parent will expand the next child. A node will also return to its parent when all its children have been explored.

Step 4: After the full tree of $S_{11}', S_{12}', \ldots, S_{1f\,1}'$ from f_1 is searched, any schedule of $S_{11}', S_{12}', \ldots, S_{1f\,1}'$ conforming to all constraints and f_1 will map to a path from the tree root to a green node on tree level f_1. The search will continue from these green nodes sequentially. Rooting at any of these green nodes, a sub-search tree will be built based on all items in $S_{21}', S_{22}', \ldots, S_{2f\,2}'$ but not involved in f_1. The tree processing of forming, searching, and cutting are the same as in Step 3. However, the total bandwidth constraint is always verified among all nodes on the path from the root of the primary tree (not the sub-tree) to the current node. Formula f_2 is also verified when the leaf level of the sub-tree is explored among all items in $S_{21}', S_{22}', \ldots, S_{2f\,2}'$ on the path from the whole tree's root.

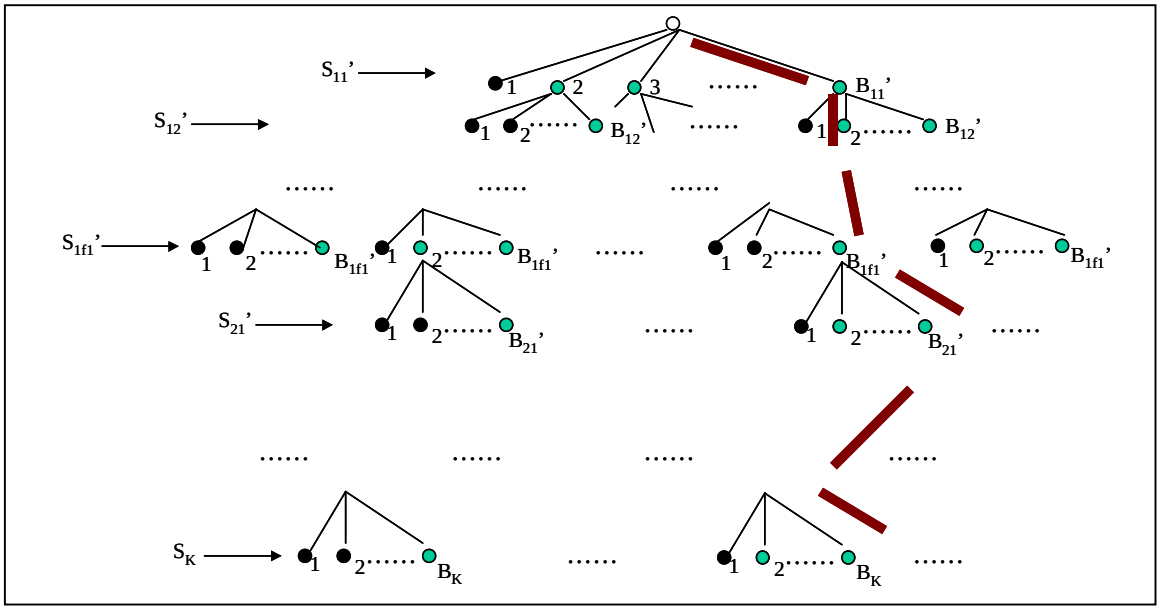


Figure 6-3 The branch-and-bound solution

Step 5. Step 4 is repeated based on all formulas until all tree levels for S_1, S_2, …., S_K (total K levels) have been explored.

Step 6. For all green nodes on the K^{th} level of the search tree, verify all formulas that are not checked yet (for $K \leq \text{Min}\left(\frac{MaxLen \times (MaxLen-1)}{2}\right)$, if there are any. Turn a node to black if the verification fails.

Step 7. The path (thick line in Figure 6-3) from the primary root to a current green node with smallest R_i value represents a solution schedule.

Instead of exhaustively searching for combinations of possible distance values based on the maximal bandwidth constraint of all data, the above solution builds a solution-space search tree. Using the scheme of backtracking and branch-and-bound algorithms, the solution makes branches on the tree by a child node's possible distance values and bounds infeasible search directions by verifying all given constraints and formulas. In this way the searching complexity can be greatly reduced.

6.1.2 Broadcast Scheduling for Real-Time Periodic Transactions

6.1.2.1 Real-time periodic transaction model

Section 2.1.6. introduced some broadcast-based real-time scheduling researches. In this section we propose and solve a different broadcast-based time-critical problem called a real-time periodic transaction model (Figure 6-4).

In a real-time periodic transaction model, there are a total of M (fixed) client read-only transactions. All server data have the same size, so all time measurements are based on the broadcast unit for each data unit. Each transaction T_i ($1 \leq i \leq M$) has sequential data access operations. Consequently, there is time dependency between any two consecutive reads of T_i, and T_i can only proceed to read the next data unit after it finishes

reading the preceding one. There is no inter-arrival time between two adjacent operations. N_i, which is the number of operations of T_i, and the data identification ID_{ij} that the jth operation ($1 \le j \le N_i$) of T_i reads, are known in advance. Each transaction starts a transaction instance periodically with period P_i ($1 \le i \le M$).

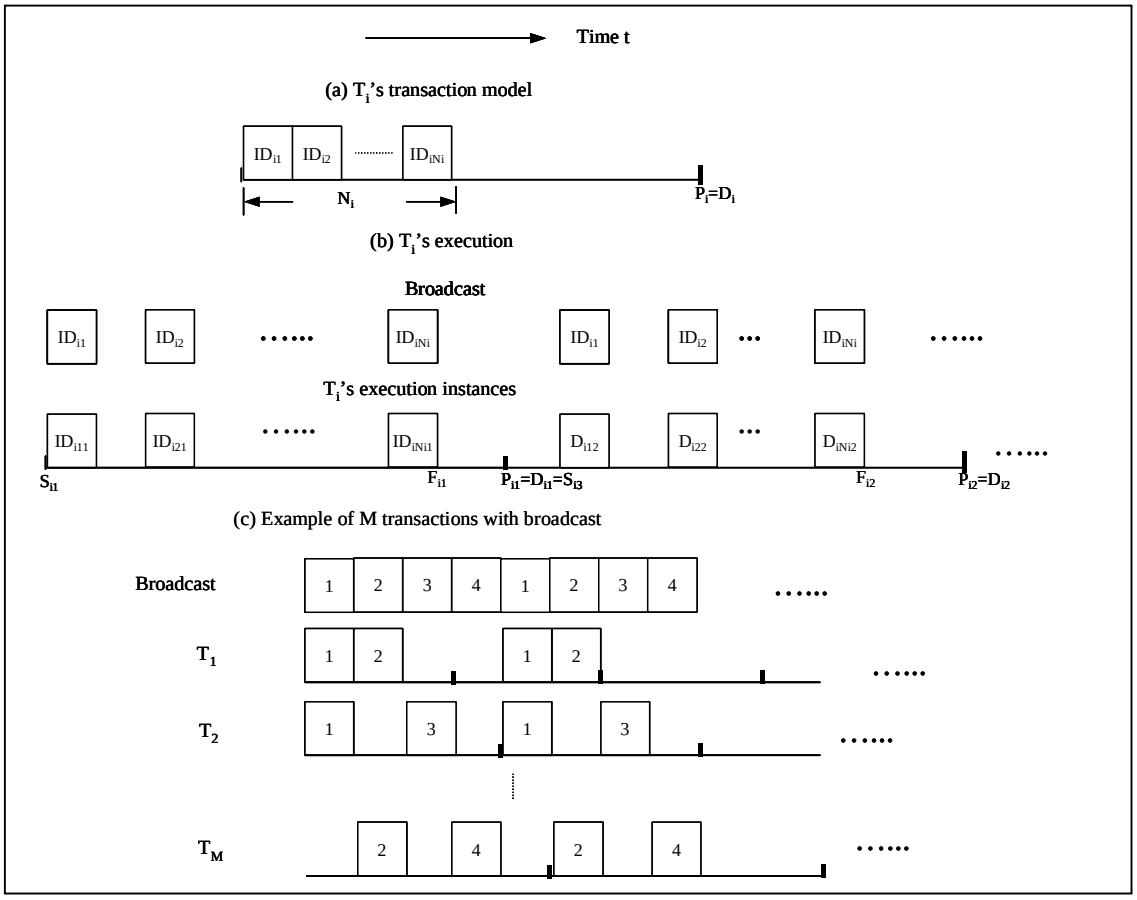


Figure 6-4 Real-time periodic transaction model

Each transaction instance also has a deadline D_i, $D_i = P_i$, which requires that one instance be finished before the next instance starts. Moreover, the first instances of all transactions start running at the same time. A transaction instance executes when it reads a requested data unit from the air. A transaction instance stays idle when it waits on a

requested data unit. Therefore, the worst-case execution time E_i of all T_i's instances equals to N_i. Response time R_{ik} of T_{ik} ($1 \le k \le \infty$), which is the kth instance of transaction T_i, is the time elapsed from S_{ik} (the start time of T_{ik}) to time F_{ik}, at which time T_{ik} finishes reading the last data of T_i. A transaction instance's response time includes its execution time (time spent reading data) plus idle time (time spent waiting for data to appear on the air). Therefore, we have: $S_{jk} = P_i \times (k-1)$ and $F_{ik} = S_{ik} + R_{ik}$. Note that the running time of each transaction instance, other than waiting and reading data from the air, is ignored. A scheduling problem for the real-time periodic transaction model above is:

Given a server database with size DB_SIZE, and a client real-time periodic transaction model with parameters M, T_i, P_i, D_i, N_i, ID_{ij}, ($1 \le i \le M$, $1 \le j \le N_i$), find a server data broadcast schedule such that $R_{ik} \le D_i$ for any instance T_{ik} ($1 \le k \le \infty$) of all transaction T_is

6.1.2.2 Execution schedule and batch execution

Each transaction in the model is a periodic task as defined in traditional real-time systems [KRI97, LIU73, YUA94]. Therefore, each transaction instance is a task instance, or a job. In this problem, scheduling server data is the same as scheduling the execution of transactions. By this analysis, a real-time scheduling algorithm used for periodic tasks (with the deadline same as the period) can be used to find a feasible execution schedule for all transactions based on M, T_i, E_i, P_i, and D_i ($1 \le i \le M$). Note that these periodic tasks are preemptable, but can only be preempted at the end of each time unit. The server can then translate this execution schedule into a broadcast schedule. If, by the execution schedule, task instance T_{ik} runs at a time unit when this is the jth time unit for T_{ik} to run,

the server broadcasts data D_{ij} at the corresponding time slot in the broadcast schedule. In such a broadcast schedule, exactly one data unit is broadcast for every data request in all transaction instances. In conclusion, if an execution schedule can be found for a group of real-time periodic transactions with the parameters given above, a data broadcast schedule can also be found for it.

However, even if there is no feasible execution schedule for a given real-time periodic transaction model (there is not enough bandwidth to broadcast one data unit for each data request for all transaction instances), it is still possible to find a broadcast schedule for it. In an execution schedule, only one task instance can run at any time. However, in a broadcast schedule, broadcast of one data unit might meet the current data requests of multiple transaction instances. This means that multiple transaction instances can execute at the same time. This reduces the total amount of time needed by all transaction instances' executions and moves ahead some transaction instances' executions, which makes it easier for all instances to meet their deadlines.

We call meeting multiple transactions' data requests by broadcasting one data unit “batch execution.” This is the nature and advantage of data broadcasting. A broadcast schedule should take advantage of batch execution in order to meet the time requirements of a given real-time periodic transaction model when bandwidth is limited.

6.1.2.3 Non-deterministic polynomial (NP) problem and time complexity of a brute-force solution

NP problem. Before presenting a heuristic solution based on batch execution, we show above scheduling problem is an NP problem. The following non-deterministic algorithm can be used to find a schedule:

First, decide the length of a primary cycle C of all transactions, where:

$$C = LCM(P_1, P_2, ..., P_M). \quad (6\text{-}3)$$

Then the server can calculate the number of instances of each transaction within time C. The number of instances I_i of transaction T_i ($1 \le i \le M$) within primary cycle C is:

$$I_i = \frac{C}{T_i}. \quad (6\text{-}4)$$

Based on I_i, the data request list by all T_i's instances in C can be derived as:

$${}_0(ID_{i1}ID_{i2}...ID_{iN_i})_1|_{P_i}(ID_{i1}ID_{i2}...ID_{iN_i})_2|_{P_i\times 2}(ID_{i1}ID_{i2}...ID_{iN_i})_3\ |.....|_{P_i\times(I_i-1)}(ID_{i1}ID_{i2}...ID_{iN_i})_{I_i}$$

. (6-5)

In each ${}_x(ID_{i1}ID_{i2}...ID_{iN_i})_y$ segment, "y" represents data requests in the yth ($1 \le y \le Ii$) instance of transaction T_i within C, and "x" represents the start time of the first request in the current segment. From now on we use ID_{ijk} to represent the jth ($1 \le j \le N_i$) data request of the kth ($1 \le k \le I_i$) instance of transaction T_i within C.

A nondeterministic algorithm will randomly pick up a data unit to broadcast for each slot in a primary cycle sequentially. This will take time O(C). After choosing this schedule, a schedulability test will check each slot sequentially to see whether or not any transaction's next started read request can be satisfied based on the scheduled data, Formula (6-5), and the earlier schedule for each transaction. Each step will take time O(M) and it has a total of C steps. If all transaction instances' data requests are met within their deadline in the primary cycle, then the problem is schedulable and this schedule is a solution. Otherwise it is not schedulable. We can see this non-deterministic algorithm takes O(MC) time, which is polynomial. Therefore, this scheduling problem is NP.

Brute-force solution. A brute-force solution based on batch execution can guarantee finding a solution for the problem above if the given model is schedulable. It takes the same steps as the non-deterministic algorithm, except it tries all possible items in the database to broadcast at each time lot instead of randomly picking just one. This way, all DB_SIZE^C possible schedules can be tested for feasibility. The total time complexity of the brute-force solution will be $O((DB\_SIZE \times M)^C)$, which is exponential.

6.1.2.4 Heuristic solution based on time-window model

Motivation. We have not found an optimal polynomial solution that guarantees finding a schedule if the model is schedulable. Although we also have not proved the problem to be NP-hard, we assume this is true by its being an NP problem and the fact that the brute-force solution having exponential time complexity. Therefore, we designed some heuristic solutions for the given problem. These heuristic algorithms, however, do not guarantee finding a schedule even if the model is schedulable.

Time-window solution model. The heuristic algorithms are based on the following observations. To utilize batch execution, the server should delay broadcasting some of the data units in order to wait until multiple (preferably many) transactions are waiting for them simultaneously (and at least one data unit request should be served by a broadcast if there is any request present). However, time dependency among transaction data accesses introduces complexity into using batch execution. The choice of delay or broadcast of any data unit has a transitive influence on later schedules. Choosing different data units to delay or broadcast at any point can block or move ahead different transactions, which can cause a completely different number and combination of simultaneous data requests among transactions at a later stage. Moreover, transaction

instances' time constraints also influence using batch execution. A different selection of delay or broadcast can cause or avoid transaction instances to meet or miss their deadlines. In conclusion, under limited bandwidth, a feasible broadcast schedule should be found using batch execution while considering data access time dependency and transaction time constraints.

The method described in 6.2.1.2. for translating a feasible execution schedule to a broadcast one should be used first in a given transaction model to see whether or not a broadcast schedule can be found directly. Here we propose a solution model to the defined problem to find a feasible broadcast schedule when no feasible execution schedule is found. The solution model is called the time-window model.

Based on Formula 6-5, for each data request ID_{ijk}, the time-window model derives an initial executable time window $TW_{ijk}= [EST_{ijk}, LCT_{ijk}]$ for it; EST_{ijk} is the earliest start time of request $ID_{ijk.}$ LCT_{ijk} is the latest completion time of request $D_{ijk.}$ The earliest start time of a request is the earliest time this request can be invoked by a transaction instance, or, if it is already invoked, the earliest time it can start being served. The latest completion time of a request is the latest time this request should be served in order for the current transaction instance to meet its deadline.

Before scheduling, EST_{ijk} should be the beginning time of instance T_{ik} for the first data request D_{i1k} of T_{ik}, or the earliest possible time for its preceding data requests to be served for any other non-first-data request of T_{ik}. Initially, we have:

$$\begin{aligned} & EST_{ijk} = P_i \times (k-1) \ \ when \ \ j=1 \ \ and \ \ EST_{ijk} = EST_{i(j-1)k} + 1 \ \ when \ \ j>1 \\ & \Rightarrow \\ & EST_{ijk} = P_i \times (k-1) + (j-1) \quad for \ all \ \ j. \end{aligned} \tag{6-6}$$

LCT should be $P_i \times k$ for the last data request D_{iN_ik} of T_{ik}, or the latest possible time for its succeeding data request to start being served for any other non-last-data request of T_{ik} in order to meet the current instance's deadline. We have:

$$LCT_{ijk} = P_i \times k \ \ when \ \ j = N_i \ \ and \ \ LCT_{ijk} = LCT_{i(j+1)k} - 1 \ \ when \ \ j < N_i$$
$$\Rightarrow \tag{6-7}$$
$$LCT_{ijk} = P_i \times k - (N_i - j) \ \ \ for \ all \ \ j.$$

Using the solution model. We call the time windows for all data requests, which are derived based on the given model and before any scheduling, initial time windows.

The server can start deriving a schedule based on the initial time windows of all possible requests for all instances in a primary cycle by using a specific algorithm. Once a feasible schedule is found for one primary cycle, the server simply repeats the schedule in later cycles.

Overlapped request time windows of same data requests among different transaction instances give a picture of how to decide when to use batch execution using a scheduling algorithm. A broadcast is schedulable under a given transaction model if and only if all data requests are served by the broadcast within their initial time windows.

However, simply broadcasting the data within a data request's initial time window does not guarantee the request will be served within it because of the time dependency among sequential accesses within an instance. Any data request being served after its initial EST will shrink the time window of all data accesses in the same instance later than it by delaying their EST. Therefore, a scheduling algorithm is needed to modify the time window continuously based on the scheduled data and the time dependencies among requests. We call these windows updated time windows.

In conclusion, any server-scheduling algorithm works using the following policy.

- It starts with initial time windows.
- It schedules data to use batch execution as much as possible based on overlapped initial/updated time windows. If a data unit is broadcast at time T, only data requests which related to this data and have an EST less than or equal to T are served.
- It modifies time windows of data requests according to the finished schedule, served requests, and time dependencies. At the beginning of each time unit (each data broadcast), if an un-served data request D_{imk}'s EST_{imk} is less than or equal to the current time, the EST_{ink} ($N_i \geq n \geq m$) of all data requests in the same instance are increased by one. That is: set $EST_{ink} = EST_{ink} + 1 \ (N_i \geq n \geq m)$ when $EST_{imk} \leq T$ at time T, where T is the beginning of a broadcast unit.
- The objective is to serve every data request within its current time window.

Heuristic algorithms and time complexity. Based on the policies above, some heuristic algorithms can use this time window model to schedule broadcasting based on some criteria. We propose most overlapped first (MOF), least completion time first (LCT), and the combination of these two criteria to decide which data to broadcast first.

These algorithms will decide which data to broadcast for each slot in a primary cycle sequentially. At each time slot, there will be at most C×M time slots to check based on at most DB_SIZE of data. So the maximal total time complexity of these heuristic algorithms is $O(DB\_SIZE \times M \times C^2)$. Additionally, binary searching can always be used when searching for data with MOF of LCT, which reduces the total time complexity to $O(DB\_SIZE \times \text{Log}(M \times C) \times C)$. We can see these heuristic algorithms based on time window model and batch execution have polynomial time complexity.

6.2 Caching

Caching is an important strategy to improve the performance of data broadcasting. Because of the sequential nature of broadcasting, client data requests are always delayed because they must wait for the data to be on the air. Caching reduces the delay by saving some broadcasted data locally for future reading. However, most existing cache strategies

are based on the performance of individual operations; that is, no transaction behavior is considered. This section discusses the design of some cache strategies that can improve the performance of transaction processing.

6.2.1 Long Transaction-Favored Caching

6.2.1.1 Influence of concurrency control on long broadcast-based transactions

We have discussed several broadcast-based concurrency control protocols in Section 2.1.4. and Chapter 3. This section discusses how concurrency control influences the response time and restart rate of long transactions in broadcast-based transaction processing and then propose some caching schemes to solve the problem.

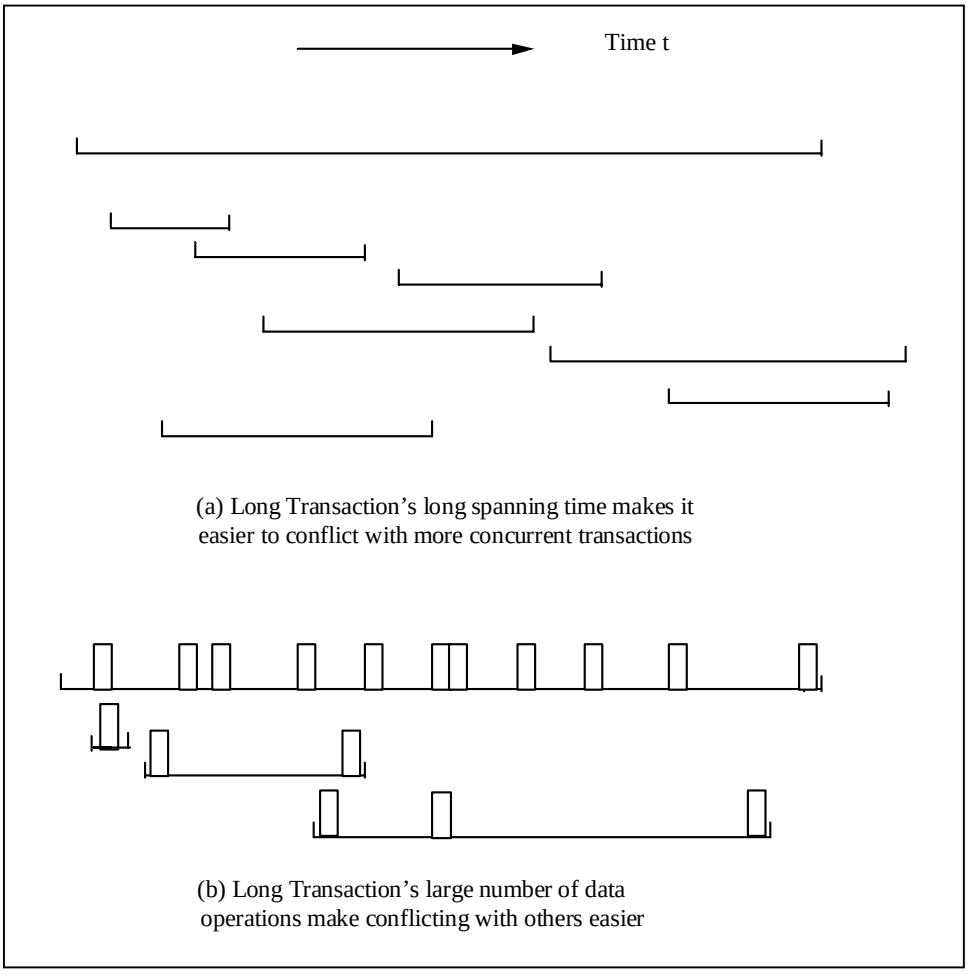


Figure 6-5 Long transaction is easy to abort

In broadcast-based applications, concurrency control has different influences on different lengths (number of data operations) of transactions. An important observation is that, under concurrency control, the longer a transaction is, the more restarts it will suffer. Consequently, a long transaction might have a greatly prolonged response time compared with the one it should have according to its length and the response time of other short transactions. The reasons for this situation are shown in Figure 6-5.

- A long transaction has a long spanning time (the time it reads/writes the first data to the time that it finishes all operations in one run). The longer the spanning time, the more other concurrent transactions exist simultaneous to this transaction. This can cause a higher probability for a long transaction to have more data conflicts with other concurrent transactions for any data this transaction reads or writes.
- A long transaction has more operations, which itself increases the possibility for this long transaction to have more data conflicts with other transactions.
- By the nature of broadcast-based applications, client transactions usually do not know the existence and behavior of other transactions. Therefore, some concurrency control strategies [BAR97a, LEE99] designed for these applications abort a transaction whenever a data conflict is detected, even it does not cause any inconsistency or if aborting one party of the conflict is enough to resolve the conflict. By the same reasoning, some other strategies [PIT99a, PIT99b, SHA99], though they do not abort transactions for any data conflict, are not able to give preference to long transactions. STUBcast aborts a transaction based on read ahead status and conflict array, without considering transaction length or how long it has executed. Consequently, a long transaction is much easier to abort than a short transaction under broadcast-based concurrency control because it can have more data conflicts with other transactions. This increases the number of restarts and may prolong a long transaction's response time greatly.

6.2.1.2 Cache favoring long transactions

A prolonged response time and large number of restarts of long transactions caused by concurrency control can also degrade the performance of short transactions because of the large number of data conflicts caused by a long transaction. This could further influence the overall performance of transaction processing. Therefore, a solution is necessary to improve the performance of long transactions and reduce the number of data conflicts with other transactions.

A large number of data conflicts is caused by both long spanning time and a large number of operations. Because reducing the number of operations is not possible, a solution should focus on reducing the transaction spanning time of long transactions. A transaction's spanning time includes the time spent waiting for the required data and the time spent reading the data from the time it starts its first data operation. Note that no conflict would happen if a transaction had not started an operation.

Since cache saves some broadcasted data locally for future use, we can use some strategies that favor keeping data needed by long transactions to help reduce the waiting time for long transactions, which consequently reduces long transaction spanning time. Compared with a cache that favors providing hot data to individual operations, this type of cache strategy favors long transactions. So we call it a long transaction-favored cache.

Moreover, we can also shorten a transaction's spanning time by using cache to delay its first data operation. A cache can fetch and keep as many data units (assume these data units can be consistently updated in the cache by concurrency control) as possible for a long transaction (only for read requests) by the sequence of its requests. When at some point a transaction has to start its operations, it can finish reading these cached data without any waiting. In the best case, a cache can have all the data a transaction needs before the transaction starts the first operation, though the best case might not always be realized because of the existence of concurrent transactions for one client and the size of cache.

The performance issue addressed by long transaction-favored cache is the overall response time and restart rate for a transaction processing application.

6.2.1.3 Replacement policies

Here we propose three replacement polices for long transaction-favored caching. These policies assume having advance knowledge of any client transaction's data requests. Whenever a transaction is started, the number of data requests for either read or write, the sequence of the requests, and the data ID of all requests are known. A transaction's operations cannot be out of order. The execution time for write operations is ignored.

Policy based on length score (*L*). By this *L* policy, each client maintains a length score table for all data. Length score of a data unit for one transaction can be defined as the number of read operations of that transaction that will read this data.

Table 6-4 shows the fields of a length score table. In the table, field "Data" represents the ID of a data unit. "Total length score" (T) is the total length score (total number of read operations) of a data unit for all current transactions that are waiting to read this data unit. "Transaction number" (N) is the number of current transactions that are waiting to read the data. "Average length score" (L) of a data unit is the average length score (average number of read operations) for current transactions that are waiting for this data. We can see that L can reflect a data unit's involvement in long transactions. The larger the L, the greater the probability that a data unit to is being waited on by a long transaction.

Table 6-4 Fields of length score table

Data	Total Length Score (T)	Transaction Number (N)	Average Length Score (L)

Under replacement policy *L*, whenever a transaction starts, the cache manager adds the number of read operations to the T field of each data unit this transaction reads.

At the same time, it increases the N field of each such data unit by one. Then the client manager updates the L field of each data unit that this transaction will read to a new value, T/N.

Whenever a transaction tries to read a data unit, it can read it from cache directly if it is already in the cache. Otherwise, the transaction has to wait for this data unit to appear on-air. After a transaction reads a data unit, no matter from cache or air, the cache manager will update this data unit's T, N, and L field in the length score table accordingly. In the update, T subtracts by the transaction's read length, N subtracts by one, and L is updated to the new T/N.

When a data unit goes by on-air and any transaction is waiting for it, the client will allow the transactions to read the data and update the length score table first. After these operations, the cache manager will judge whether the cache needs to be updated according to the current length score of the data. The *L* replacement policy is adopted at this point, where, if the data unit with least L value in the cache has an L value less than the data unit currently on-air (just read by any transaction), that data unit in the cache would be replaced by the new one on-air. Note that this replacement only happens when the cache is already full. If the cache is not full when a data unit is read from the air, the data will be inserted into the cache directly. When there is a tie between two data units for L value, policy *L* keeps the data with a smaller frequency value in the cache.

In conclusion, policy *L* is a least average length score first cache replacement policy. This policy replaces a data unit that has a higher probability of being involved in short transactions with another one that is more likely to be involved in longer ones.

Using this policy, the cache will keep more data units that are needed by long transactions.

Policy based on length score inverse frequency (*LIF*). Policy *LIF* is a modification of policy *L*. In policy *LIF*, a new field, "Average length score inverses frequency" (LIF), is added to the length score table (Table 6-5), which changes the table to a Length Score Inverse Frequency Table. In the table, F is the broadcast frequency of a data. LIF is the value of L/F.

Table 6-5 Fields of length score inverses frequency table

Data	Total Length Score (T)	Transaction Number (N)	Average Length Score (L)	Broadcast Frequency (F)	Average Length Score Inverse Frequency (LIF)

In policy *L*, a data unit's broadcast frequency is only considered when there is a tie between two data units' L values. In policy *LIF*, data's broadcast frequency is used as part of the replacement criteria directly. Although a data unit is involved in a long transaction, it might be broadcast more frequently than another one involved in a shorter transaction. In this case, the short transaction might have a much longer spanning time. Therefore, it is more logical to use the combined effect of L and F to decide which data unit to replace.

Policy *LIF* works similar to policy *L*, except that the LIF field is also updated when T, N, and L are updated as described by policy *L*. Furthermore, policy *LIF* uses a length score inverse frequency value instead of L when it decides whether any data in the cache should be replaced.

Policy based on length score inverse early start score (*LIS*). As analyzed in Section 6.2.1.2, delaying the start of a transaction as much as possible can also shorten a transaction's spanning time. Policy *LIS* uses this basic idea to hold as many data items as possible for any one transaction until it is necessary for this transaction to start reading these data. There are some further assumptions that the cache and transactions follow under policy *LIS*.

- Altogether operations

Under policy *LIS*, a transaction does not make progress unless the cache invokes it to. Every time the cache invokes a transaction, the transaction can finish many operations sequentially together. For example, transaction T has data access operations as following:

Read a, Read c, Write k, Read d, Write m, Read f, Read n, Read i, Write j, Commit.

The cache holds data a, c, d, and f and plans to replace f. The cache invokes T. T can proceed to

Read a, Read c, Write k, Read d, Write m, Read f

sequentially together until it is stopped by the cache.

- Delay operations

Under policy *LIS*, we delay any transaction's operations as long as possible. Using the example above, if cache holds data a, c, d, and f and it needs to replace c with other data, at this point, cache would invoke T to start reading a and c together right before c is replaced. However, using the delay operations assumption, T will not continue to read d and f, even though they are already in the cache. T will start reading them at a later stage

before any later data unit is replaced or all data needed by T are in the cache. This assumption is based on the observation that earlier operations have a higher probability of causing inconsistency when interacting with other transactions and later operations.

- Invoke policy

The cache invokes a transaction only under two conditions. The first one is when all data needed by a transaction are in the cache. This is the best case because all operations of a transaction can finish together. The other condition is replacing a data unit. When a data unit is selected to be replaced, the cache would invoke all transactions that need it and let these transactions finish their data accesses from the current point to the point where it finishes reading this data unit. By the delay operation assumption, a transaction will pause after reading this data unit even if later data units are available in the cache.

Policy *LIS* maintains a length score inverses early start score table. Table 6-6 shows the fields in such a table.

Table 6-6 Fields of length score inverses early start score table

Data	Total Length Score (T)	Transaction Number (N)	Average Length Score (L)	Early Start Score (S)	Average Length Score Inverses Early Start Score (LIS)

"Start score" of a data unit for one transaction is defined as the percentage of read operations that have been finished for that transaction when the data unit is read by that transaction. If a data unit is the i^{th} operation in a transaction and there are n read operations, this data unit's start score is: i/n.

"Early start score" (S) is the smallest (earliest) start score of a data unit for all current transactions that read it. We can see that S can reflect the smallest percentage of read operations finished in all transactions that a data unit is involved in when the same data unit is read by those transactions. A data unit having a small S value implies that, if the cache invokes all current transactions that need it to operate until reading this data unit before the same data is replaced, at least one such transaction will not be delayed sufficiently before starting (which failed to have a short spanning time by delaying the start). On the other hand, a data unit with a large S value implies all current transactions reading this data can have a large start delay if invoked for the first time because of the replacement of this data. Therefore, S can be used as a criterion to delay a transaction start as much as possible in cache. The larger a data unit's S is, the more start delaying is achieved in the current transaction for this data unit to be replaced from the cache.

In the table, LIS is the value of L/S. Whenever a transaction is started, each data unit that the transaction reads will have its S and L/S values updated. After a data unit is read by any transaction, its S and L/S values will also be changed accordingly. Note that, when no transaction needs a data unit, its L value is 0 and S value is 1.

Policy *LIS* uses the least LIS replaced-first policy. Whenever a data unit is read from the air, policy *LIS* judges whether or not the data unit in the cache with smallest LIS has a smaller LIS value than the one on-air. A data unit having a smaller LIS value means it is involved in a shorter transaction and/or it is in the later stages of all current transactions. Therefore, replacing this data unit can result in shortening the spanning time of longer transactions by keeping data needed by long transaction and/or allowing long transactions to start later.

When the cache replaces any data unit, it invokes all current transactions that need to read this data unit to run from their current execution point to the finish point of reading this data unit. The cache also invokes a transaction when all data it needs to read are present in the cache.

A replacement policy *S* can also be used which replaces data with larger S values.

6.2.1.4 Performance results

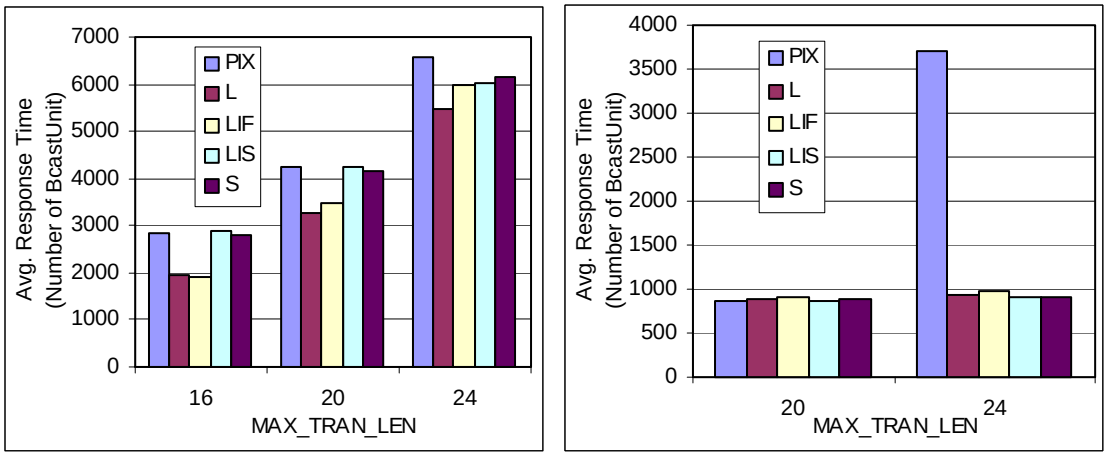
Simulation configuration. We have simulated long transaction-favored caching under the proposed replacement policies *L*, *LIF*, *LIS*, and *S* on some sets of transactions (Table 6-7) configured using the same framework and parameters as in Tables 3-4 and 4-1, and by applying STUBcast as the concurrency control protocol. We also simulated the same sets of transactions using PIX [ACH95a, ZDO94] cache replacement policy in order to compare the performance of long transaction-favored caching with a general cache.

Table 6-7 Values of parameters used in long transaction favored cache simulations

NUM_OF_TRANS	5000	PCAST_TYPE	FLAT, MULTIDISK
DB_SIZE	D100, D1000	ACCESS_ DISTR_MODE	UNIFORM, NON_UNIFORM with n=4
TR_INTARR_TIME	Tr50	MAX_TRAN_LEN	L4, L8, L12, L16, L20, L24
MACHINE	M1	PERCENTAGE	Per.03, Per.05, Per.10

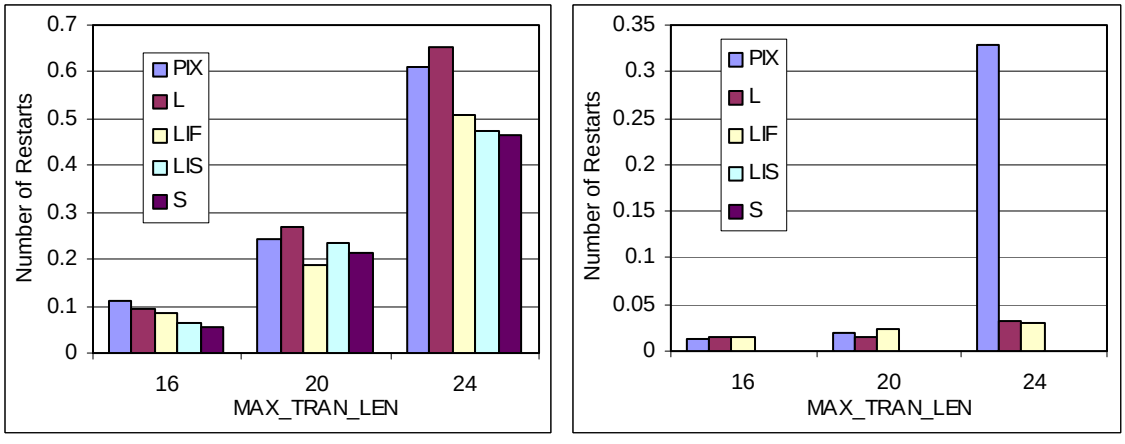
Results. Some example simulation results are shown in Figures 6-6, 6-7, and 6-8. Figure 6-6 includes the average response time results under the configuration of FLAT, UNIFORM, Tr50, D1000, and Per.03 (or Per.10). We only show the data for MAX_TRAN_LEN ≥ 16 with Per.03 and MAX_TRAN_LEN ≥ 20 with Per.10 because the collected data show negligible performance differences among all policies when MAX_TRAN_LEN is shorter. This is logical because we only expect long transaction-

favored cache policies to have an advantage when longer transactions exist. Figure 6-6a shows that, when cache size is very small (Per.03), the best-performing policy favoring long transactions is policy *L*, with about 15% improvement over PIX. However, when the cache size is very large (Figure 6-6b), all proposed policies perform equally well and much better than PIX (around 80% improvements) when the transactions have a MAX_TRAN_LEN of 24.



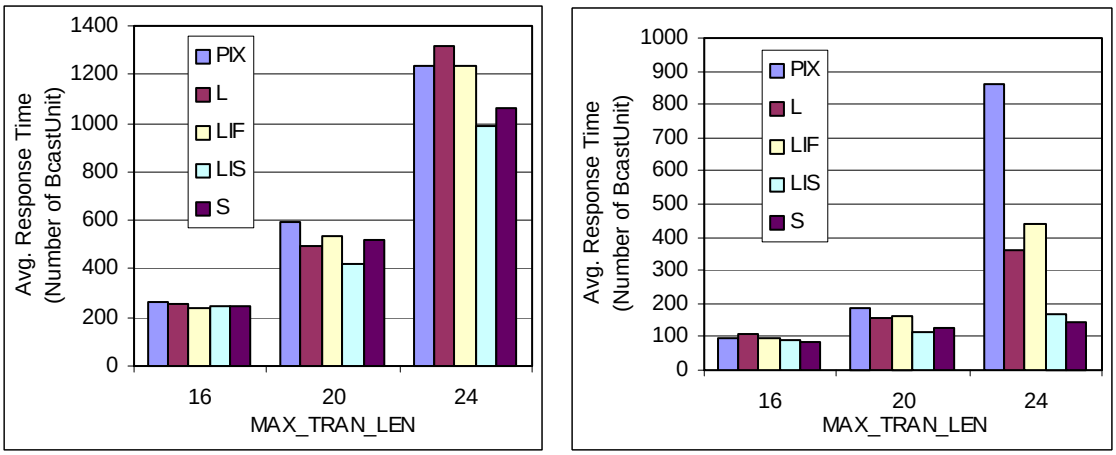
(a) (b)

Figure 6-6 Average response time under FLAT scheduling and UNIFORM data access for Tr50 and D1000. a) Per.03; b) Per.10



(a) (b)

Figure 6-7 Average restarts under FLAT scheduling and UNIFORM data access for Tr50 and D1000. a) Per.03; b) Per.10



(a) (b)

Figure 6-8 Average response time under MULTIDISK scheduling and NON_UNIFORM data access for Tr50 and D100. a) Per.03; b) Per.10

Figure 6-7 shows results for the same configurations as in Figure 6-6, but the performance measurements as the number of restarts among transactions. We can see that policies *LIS* and *S* usually have smallest restart times (zero when cache size is large, even when MAX_TRAN_LEN is 24) because they try to delay the start of transactions as much as possible to reduce the transaction spanning time. Policies *LIS* and *S* are better choices than other policies when transaction restarts are very expensive.

Finally, Figure 6-8 shows average response time performance for MULTIDISK, NON_UNIFORM, D100, and some other configurations as in Figures 6-6 and 6-7. It shows policies *LIS* and *S* become the best performing polices (at MAX_TRAN_LEN ≥ 16) when data requests are non-uniform and hotter data units are broadcast more frequently. Hotter data can stay in cache longer than others under policies *LIS* or *S* in this environment because they most often have small S values because they are accessed more frequently and they easily pass the cache again by being broadcast more frequently. This results in most transactions being able to start late and reduce spanning time. Figure 6-8 b) shows that, especially when cache size is large (Per.10) and the transactions have

MAX_TRAN_LEN of 24, policy *LIS* (or *S*) shows improvement over PIX by 85% and are approximately 60% better than other long transaction-favored cache policies.

6.2.2 Soft Real-Time Caching

6.2.2.1 Cache for soft real-time transactions

A data broadcast can also be adopted to applications whose clients run transactions with real-time constraints. Real-time client transactions may depend on an efficient data broadcast schedule that collects the clients' requests and time constraints and attempts to meet their deadlines [BES96, BUR97, FER99, LAM98]. However, these types of solutions do not assume the existence of cache, which can greatly reduce the response time of each transaction. On the other hand, no existing work on cache strategies has considered using cache to satisfy the real-time requirements of individual data accesses or transactions.

We observe it is efficient [HUA01a] to cache data based on transactions' time requirements and data requests. If the cache keeps data in a proper way such that a transaction with a tighter deadline finds desired data in it more easily, a larger percentage of transactions can meet their deadlines. Based on this observation, we now propose to design cache strategies to assist client transactions to meet their time constraints in a broadcast environment. Because of the unpredictability of cache, this scheme can only be used for real-time transactions with soft deadlines that simply abort or accept the late results without any catastrophic outcome when a deadline is missed. Therefore, we call the strategies addressed in this section soft real-time caching and the performance issue addressed is the percentage of transactions meeting deadlines.

This section also assumes advance knowledge of a client transaction's data requests. We assume that whenever a transaction is started, the number of data requests

for either read or write, the sequence of the requests, and the data ID of all requests are known. Because there is time dependency among a transaction's data accesses, they cannot be out of order. Moreover, we assume each transaction has a soft deadline that is also known in advance of when the transaction is started. No concurrency control is used and the server broadcasts each data unit evenly spaced by a pre-decided frequency.

6.2.2.2 Largest first access deadline replaced (*LDF*) policy

We propose a largest first access deadline replaced (*LDF*) cache policy as the solution to soft real-time caching. The access deadline of a data unit for one transaction that reads this data is defined as an estimated latest time for this data to be read by this transaction in order to allow this transaction to meet its deadline. The first access deadline of a data unit is the smallest access deadline of this data unit on all current transactions that access it.

Table 6-8 Fields in access deadlines table

Data	Access Deadlines (A)	First Access Deadline (D)

Under policy *LDF* the cache maintains an access deadlines table for each data unit based on all current transactions in which it is involved. Table 6-8 shows the fields of such a table. Whenever a transaction starts, the cache estimates the access deadline of each data unit in this transaction based on the sequence of data requests, the transaction's deadline, and an access deadline estimation policy we will define shortly. This estimated access deadline is added to the data unit's “access deadlines” (A) field. The “first access deadline” (D) field contains the smallest value among all access deadlines of this data unit in all current transactions reading it. We can see that D can reflect what is the earliest time that a data unit is needed to meet the deadline of all current transactions that read it.

Whenever any transaction reads a data unit, its A and D values should be modified correspondingly (by removing the A or D value obtained from that transaction and updating to new ones based on other transactions).

Policy *LDF* is then used when a data unit is read from the air. After the reading and update of the read unit's A and D values, the cache evaluates whether or not the data unit in it with the largest D value has a larger value of D than the data unit on air. Such a data unit is replaced with the one on-air. Since this policy replaces the data unit with the largest first access deadline value, it keeps data needed more urgently by existing transactions to meet their deadlines.

6.2.2.3 Access deadline estimation policies

Given the deadline of a transaction and the sequence of data it accesses, we need to decide which policy to use in order to estimate the access deadline of each data unit. The access deadline estimated here is used for A or D in policy *LDF*.

Given a transaction T with a sequence of read operations on data R_1, R_2, … R_n (write operations are ignored here), n the number of read operations in T, and t the deadline of T, an access deadline of each data R_i ($1 \le i \le n$) on transaction T, $A(R_i)_T$, can be estimated using following policies.

- One unit access time-based estimation (ONEUNIT)

$$A(R_i)_T = t - (n - i)\,. \tag{6-8}$$

  This policy is based on each data read needing at least one broadcast unit time.

- Evenly distributed access time-based estimation (EVENUNIT)

$$\begin{aligned} S &= t/n \\ A(R_i)_T &= t - (t/n) \times (n - i) \end{aligned}\,. \tag{6-9}$$

This policy distributes a transaction's deadline evenly among all its read operations and leaves at least this much time for each read operation.

- Average access time-based estimation (AVGACSUNIT)

$$A(R_i)_T = t - \sum_{k=i+1}^{n} \frac{S_k}{2}. \quad (6\text{-}10)$$

S_i is the space between each instance of data R_i in the broadcast. This policy leaves average access time of the data that a read operation accesses for each operation.

- Largest access time-based estimation (LARGEACSUNIT)

$$A(R_i)_T = t - \sum_{k=i+1}^{n} S_k \ . \quad (6\text{-}11)$$

This policy is similar to last one, but leaves the largest possible access time of the data that a read operation accesses for each operation.

- Worst-case deadline estimation (WCSACSUNIT)

$$A(R_i)_T = t - \sum_{j=i+1}^{n} S_{j,j+1} \ . \quad (6\text{-}12)$$

In this policy, the cache manager decides the worst-case broadcast space between each data R_j and R_{j+1} ($S_{j, j+1}$) in advance. Then this policy leaves time $S_{j, j+1}$ for accessing data R_{j+1} after the read of R_j.

6.2.2.4 Simulation results

Configuration. We have simulated policy *LDF* under the above access deadline estimation policies (except worst case deadline estimation) on some sets of soft real-time transactions. The simulations (Table 6-9) are set up using the same framework and configurations described in Table 3-4 for STUBcast and Table 4-1 for STUBcache. The only difference is that all transactions are read-only and they have soft deadlines.

Table 6-9 Values of configuration parameters used in soft real-time cache simulations

NUM_OF_TRANS	5000	PCAST_TYPE	FLAT, MULTIDISK
DB_SIZE	D100, D1000	ACCESS_ DISTR_MODE	UNIFORM, NON_UNIFORM with n=5
TR_INTARR_TIME	Tr50	MAX_TRAN_LEN	L4, L8, L12, L16, L20, L24
MACHINE	M1	PERCENTAGE	Per.03, Per.05, Per.10

For each simulation, an overall average response time for each operation can be decided based on the scheduling algorithm and the size of the database [ACH95a, HAM97, VAI97b]. This average response time is used to set the real-time deadline for each simulated transaction. If AVG_RP is the average response time under the current simulation configuration, then a transaction with length L has a relative deadline of L×AVG_RP.

In addition to simulating policy *LDF* with different access deadline estimation policies on each set of transactions, we also simulated PIX [ACH95a, ZDO94] on the same sets of transactions in order to evaluate whether policy *LDF* caching provides better real-time performance than general caching strategies.

Results. The simulation results are shown in Figures 6-9, 6-10, and 6-11. Figure 6-9 includes the results under the configuration of FLAT, UNIFORM, D100 (or D1000), and Per.03 (or Per.05, Per.10). We can see that policy *LDF* under any given deadline estimation policy allow a higher percentage of transactions to meet their deadline than PIX in these simulations, especially when the transaction length becomes larger. It also shows that when the cache size is larger, the performance of PIX improves greatly. However, the performance of policy *LDF* is similar under different sizes of cache configurations.

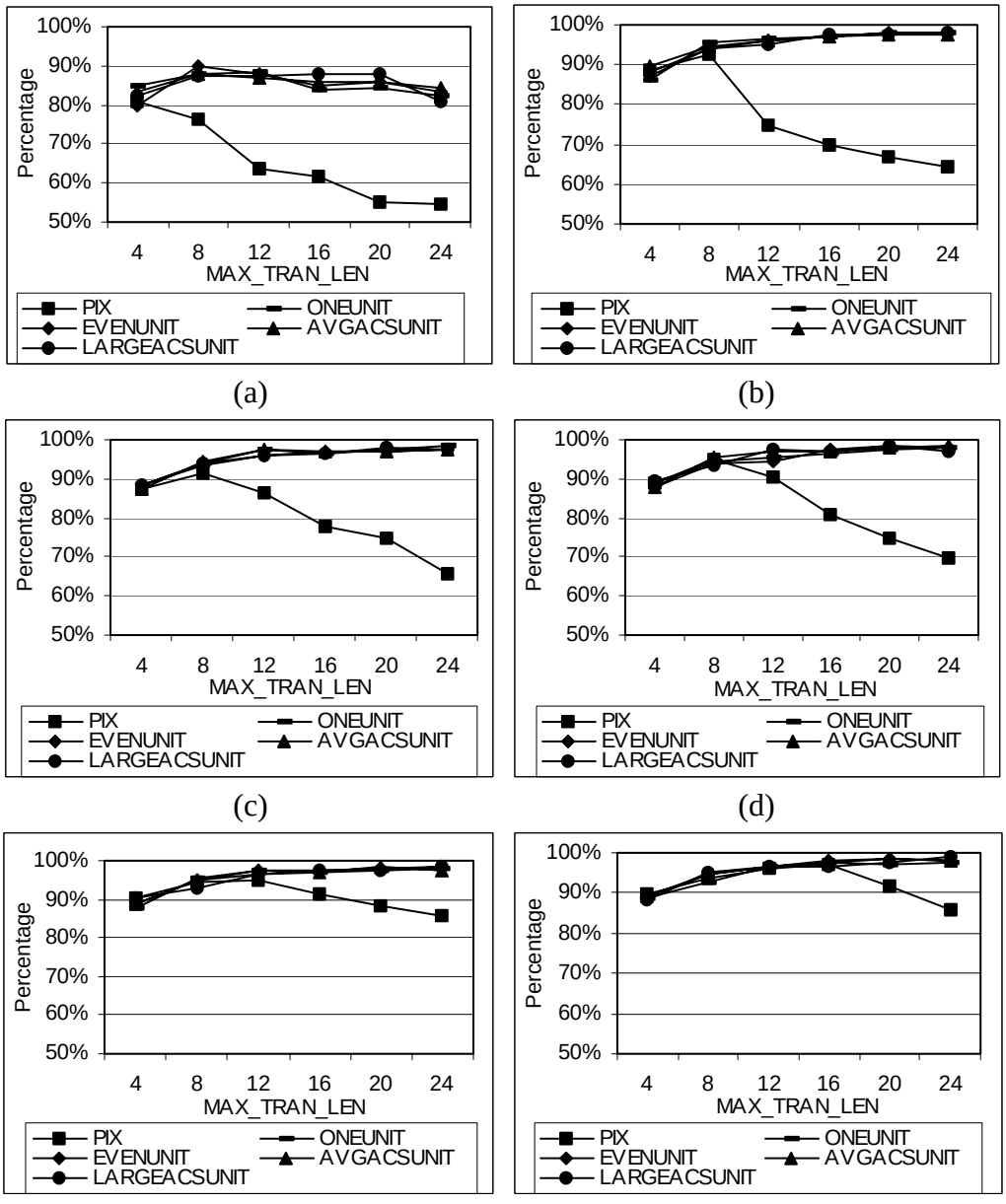


Figure 6-9 Percentage of transactions meeting deadline under FLAT scheduling and UNIFORM data access. a) D100, Per.03; b) D1000, Per.03; c) D100, Per.05; d) D1000, Per.05; e) D100, Per.10; f) D1000, Per.10

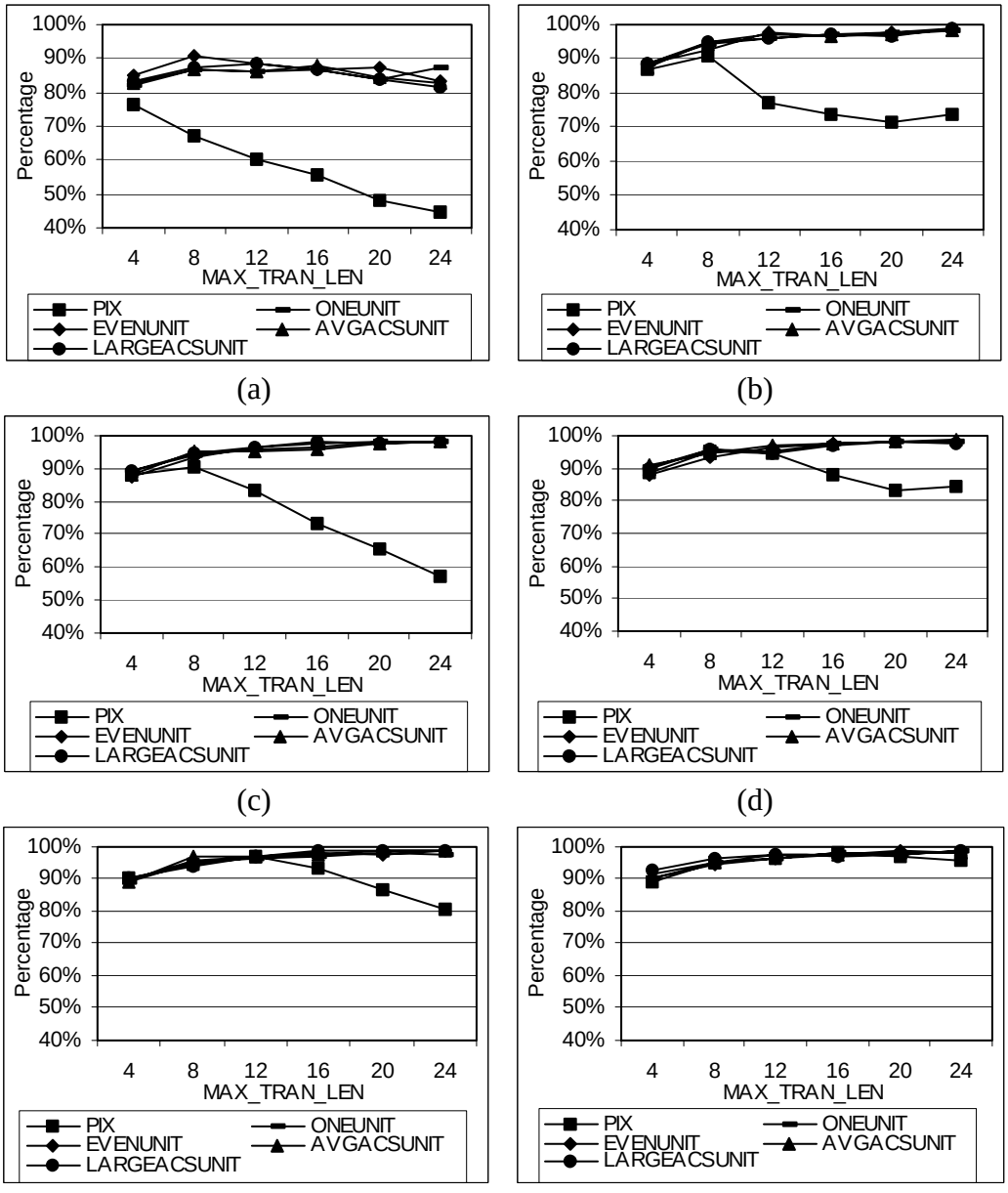


Figure 6-10 Percentage of transactions meeting deadline under FLAT scheduling and NON_UNIFORM data access. a) D100, Per.03; b) D1000, Per.03; c) D100, Per.05; d) D1000, Per.05; e) D100, Per.10; f) D1000, Per.10

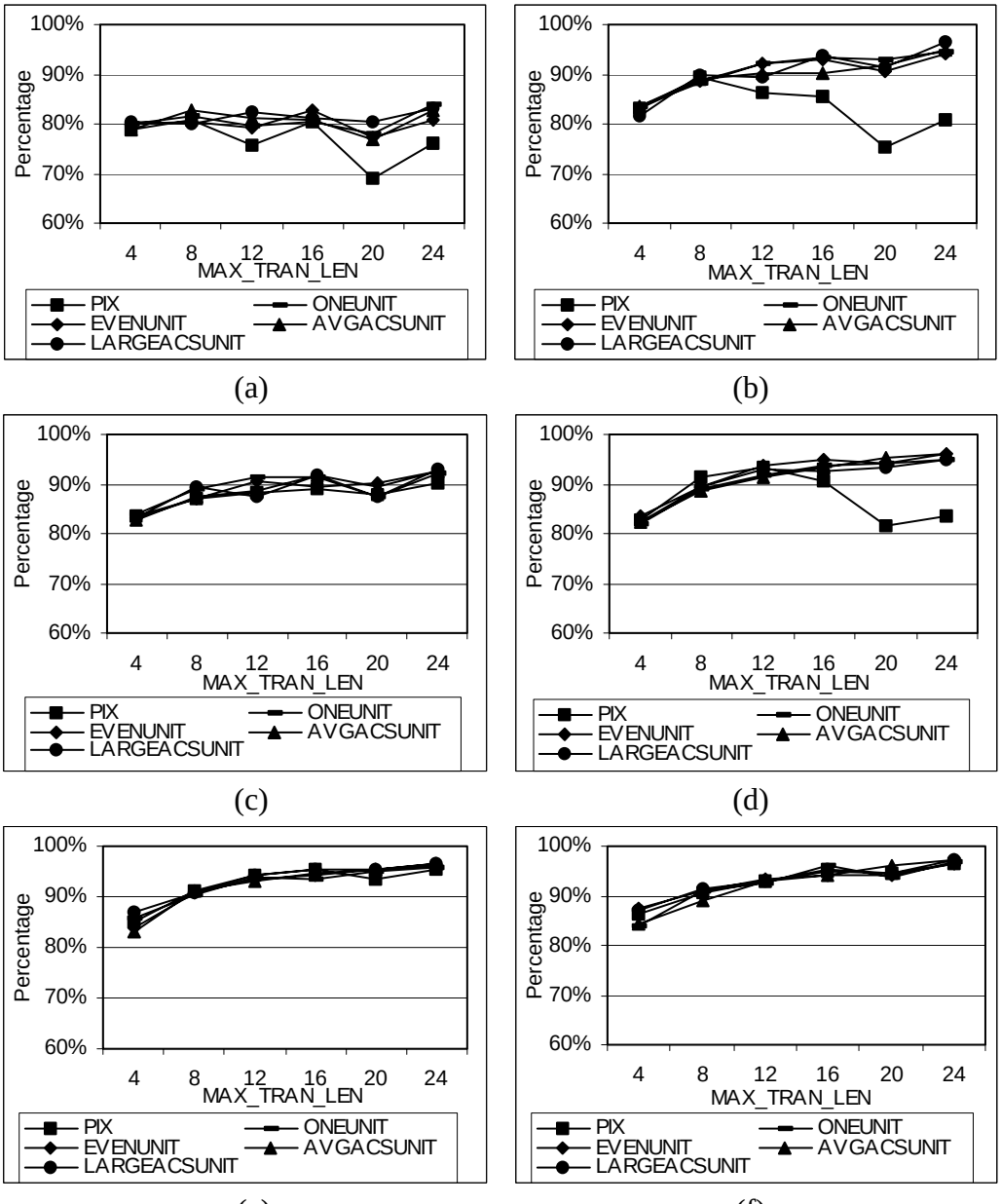


Figure 6-11 Percentage of transactions meeting deadline under MULTIDISK scheduling and NON_UNIFORM data access. a) D100, Per.03; b) D1000, Per.03; c) D100, Per.05; d) D1000, Per.05; e) D100, Per.10; f) D1000, Per.10

Figure 6-10 includes the results of FLAT, NON_UNIFORM, D100 (or D1000), and Per.03 (or Per.05, Per.10). Again, policy *LDF* under any given deadline estimation policy allows a higher percentage of transactions to meet their deadline than PIX in these simulations. The improvement of policy *LDF* from PIX is very small, however, when the database size is 1000 and the cache size is 10% of the database. Because data access is not uniformly distributed in the database in these simulations, while all data is broadcast using the same frequency, PIX starts to show its advantage of keeping hotter data when the cache size is large enough. However, these results still show that policy *LDF* has better real-time performance than PIX.

Figure 6-11 includes the results of MULTIDISK, NO_UNIFORM, D100 (or D1000), and Per.03 (or Per.05, Per.10). Although the difference becomes less, policy *LDF* under any given deadline estimation policy still allows a higher percentage of transactions to meet their deadlines than PIX. The major reason that PIX performs better than in other configurations is that multi-disk scheduling can greatly reduce the response time of hot data.

In conclusion, policy *LDF* with all deadline estimation policies proposed in this section provides better real-time performance than PIX, especially when FLAT scheduling is used, no matter what the data access mode is. Moreover, we predict that if tighter deadlines are used than in current simulations, the performance improvement of policy *LDF* over PIX should be even greater.

Another observation is that policy *LDF* schemes perform very well in all circumstances, such as under both FLAT and MULTIDISK scheduling or in both UNIFORM and NON_UNIFORM access mode or for both smaller and larger databases

Compared with it, PIX only performs better under MULTIDISK scheduling (because of its advantage in scheduling) or under NON_UNIFORM access mode and large cache size. Based on these observations, if using policy *LDF* as caching strategies in soft real-time transaction applications, FLAT scheduling and smaller cache size should always be chosen. Since multi-disk scheduling is much more difficult to implement in large database systems and hard to adjust to new changes to a database, providing similar performance in FLAT scheduling is another very important advantage of policy *LDF* caching strategies in soft real-time applications.

CHAPTER 7
IMPLEMENTATION OF THE SIMULATION FRAMEWORK

7.1 Simulated Works

Chapters 3 to 6 have presented the requirements and solutions of several broadcast-based transaction processing problems. These studies target providing efficient transaction processing in broadcast environment. Therefore, for many of these problems, we have shown some simulation results to prove the efficiency of the solutions. We have shown the performance of the solutions to the following problems.

- RSP_{SS}, UTVP, and SVP of STUBcast, which supports single-serializability in broadcast-based transaction processing.
- STUBcache, which supports caching using RxU or CiR replacement policies in applications using STUBcast. A PIX policy was also simulated for comparison.
- STUBcast-adapted scheduling, including the periodic *Ucast* server solution and a verification-rescheduling algorithm using WFQ. To evaluate the performance of periodic *Ucast* server, aperiodic scheduling, worst-case scheduling, and dynamic broadcast distance adjustment scheduling were also simulated. A FCFS server-verification scheduling was simulated to compare the performance to WFQ. Additionally, we also simulated different ways to map *Ucast* to normal data units for Model 1 and Model 2 in order to find the best mapping schemes for a minimal *Ucast* response time when *Ucast* queuing is considered.
- STUBindex, which supports energy saving in applications using STUBcast. A general partial path replication distributed index was also simulated for comparison.
- Long transaction favored caching, which improve overall performance of transaction processing by reducing the disadvantageous influence of concurrency control on long transactions. It includes the simulation of policies *L*, *LIF*, *LIS*, *S*, and policy PIX in order to find the best-performing solution.
- Soft real-time caching, which supports a higher percentage of transactions that meet deadliness. Different access deadline estimation schemes, ONEUNIT, EVENUNIT, AVGACSUNIT, LARGEACSUNIT for *LDF* replacement policy were simulated and compared with the simulation results of PIX.

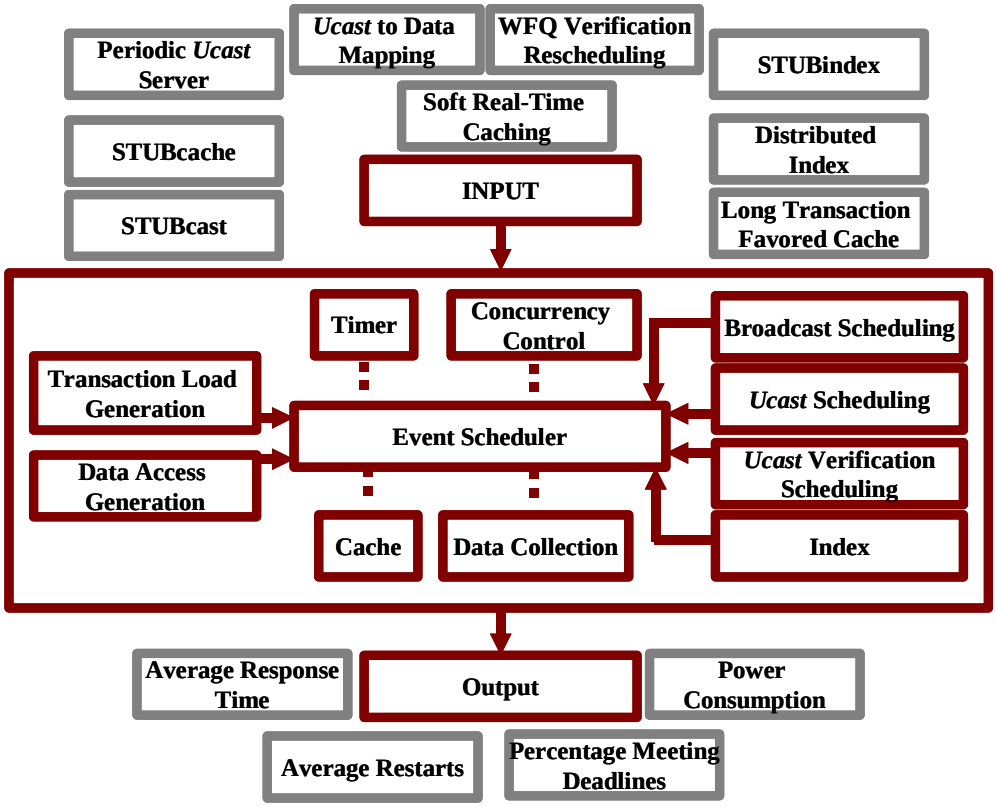


Figure 7-1 The simulation framework

7.2 Simulation Framework

Figure 7-1 presents the basic framework of the simulation implementation by showing all implemented modules and their relations. Here we discuss the details of each module.

7.2.1 Input module

We set up each simulation's configuration by using an input module, which reads the parameters from a text file. All configuration parameters have been explained in each chapter when discussing the simulation results. We summarize these parameters again by

giving a sample input file in Figure 7-2. This input file is for a soft real-time caching simulation. An input file can be configured based on the type of simulation. As shown in Figure 7-1, major supported simulation types include STUBcast, STUBcache, periodic *Ucast* sever, WFQ *Ucast* verification rescheduling, Ucast to data mapping, STUBindex, distributed index, long transaction favored caching, and soft real-time caching.

Figure 7-2 Sample input file

7.2.2 Transaction Load Generation and Data Access Generation Modules

A transaction load generation module is responsible for generating simulated transactions based on the configurations. Related set-up information includes NUM_OF_TRAN, TRAN_INTARR_TIME, OP_INTARR_TIME, MAX_TRAN_LEN, READ_TO_UPDATE_RATIO, READ_TO_WRITE_RATIO, and LOCAL_TO_NONLOC_RATIO. Based on this information, a set of transactions is generated with proper number, transaction inter-arrival time, operation inter-arrival time, maximal transaction length, the percentage of read-only (update) transactions and read

(write) operations, and percentage of local reads in STUBcast update transactions. These transactions are then input to the simulation run, where an event scheduler (introduced shortly) will control their executions.

The data access generation model will depend on such parameters as ACCESS_DISTR_MODE with value UNIFORM and NON_UNIFORM, NON_UNIPART (number of parts of the database under NON_UNIFORM), and DB_SIZE to decide which data unit an operation should read or write in all transactions.

7.2.3 Broadcast Modules

The broadcast modules first include the primary-data broadcasting. Based on information provided by PCAST_SCHEME, NON_UNIPART (at MULTIDISK), and DB_SIZE, this module generates a FLAT or MULTIDISK basic broadcast schedule.

The next broadcast module, the *Ucast* scheduling module, is used in all simulations related to STUBcast. It inserts aperiodic *Ucast* into a primary broadcast schedule in primary STUBcast and STUBcache. It is also the module that controls periodic *Ucast* scheduling, worst-case *Ucast* scheduling, and dynamic distance adjusted scheduling for STUBcast adapted scheduling.

Another broadcast module is the *Ucast* verification scheduling. It implements the FCFS verification sequence and the WFQ version of the rescheduling.

The last broadcast module is the index scheduling. It first implements the partial-path replication distributed index for primary FLAT scheduling. Then it implements STUBindex, which inserts *Ucast* segments into distributed index-based data index segments

7.2.4 Cache Module

The cache module is implemented to provide cache function in STUBcache, long transaction-favored cache, and soft real-time caching. It is set up based on PERCENT, DB_SIZE, and MACHINE. This module also considers a SLACK_FACTOR for setting up the deadlines in real-time caching. The cache module implements all necessary cache tables and cache replacement policies and cooperates with data reading from the air. It also implements the concurrency control policies in STUBcache.

7.2.5 Concurrency Control Module

This module implements STUBcast concurrency control in all simulations related to it (almost all simulations, except soft real-time caching). It applies the protocol policies of STUBcast or a variant of it in other simulations to all transactions, operations, cache, server verification, and data broadcasting. It is essentially embedded in all other modules.

7.2.6 Data Collection Module

The data collection module records desired simulation results in the data structure during all simulation runs. It also records some data in some intermediate result files or trace files. Data collected include detailed information of all generated transactions, such as type, operation number, operation index, and transaction or operation inter-arrival time; detailed access distribution among the database; the trace of transaction behaviors under each simulation; each transaction’s response time; each transaction’s restart number; each transaction’s power consumption; and whether or not a transaction met its deadline. These data collection behaviors can also be turned on or off based on the requirements.

7.2.7 Timer and Event Scheduler Module

After all other modules provide the transaction load, scheduling sequence, cache behaviors, concurrency control handling, and index directing, this module is essential to scheduling the sequence of all events activated based on a timeline and generated by other modules. This represents the basic method used in this simulation framework, which is based on timer and event scheduling and handling. Details of this method are given in Section 7.3.

7.2.8 Output Module

Based on the data collected, the output module accumulates all results and calculates required values for final performance evaluation. From the results shown in Chapters 3 to 6, we can see the major values are average response time, average number of restarts, percent of transactions meeting deadlines, and average power consumption.

7.3 Simulation Method

The simulation method used in this framework is based on event scheduling and handling on a timeline. Based on this method, each simulation starts with some basic events like a transaction start and a data unit broadcast. A transaction start event will trigger its own operations and a later transaction's start based on the inter-arrival time given in the transaction load. Each operation of a transaction will also trigger the start of the next one or the finish of this transaction. Each data unit broadcast will trigger the next based on the generated schedule. All events are also handled at a proper time based on the timeline. For example, whenever a data read operation is handled, the event scheduler can also check whether at the current time the required data unit is available on-air based on any broadcast event. This operation can finish if the data unit is available. The event

scheduler can also check the cache tables created from the cache module at any operation. Moreover, an operation event handler will always involve concurrency-control policy applied under STUBcast related simulations. Data collection is carried out at any events so that required results data are available. In addition to a basic data broadcast event, *Ucast* data, *Ucast* index, data index, and other information to be broadcast can also be triggered by an update transaction commit or generated index schedules. Many important data structures are maintained in the implementation in order to collect, queue, and trigger all the events.

All simulations start by generating some basic events of transactions and broadcasting, such as TR_START and PCAST. All events are inserted into an event queue based on the event time. The time-line moves ahead based on the progress of all events. Whenever a new event is handled, it will trigger another event based on the solution simulated. We enumerate some of the most basic types of events used in our simulation and their triggering behavior as follows.

- TR_START. It can trigger OP_RUN and the next TR_START.
- OP_RUN. It can trigger OP_RUN or TR_END. Many protocol policies are applied at an OP_RUN event.
- TR_END. It can trigger READ_ONLY_COMMIT or UPDATE_SUBMIT.
- UPDATE_SUBMIT. It can trigger VERIFICATION.
- VERICATION. It can trigger UBB broadcast, if successful.
- UBB. It can trigger UCAST broadcast.
- UCAST. It can trigger the next UCAST or UBE.
- UBE. It can trigger the next PCAST.
- PCAST. It can trigger next PCAST or INDEX. It can also trigger cache update behaviors.
- INDEX. It can trigger the next INDEX or PCAST.

7.4 Implementation Environment and Results Analysis

The simulation framework is implemented using C++ in the Microsoft Visual C++ development environment. This environment provides convenient ways to code, manage modules, and manage projects.

All generated results are output to text files in a format that is easy to import to Microsoft Excel files. Then Microsoft Excel can easily draw graphs, such as figures shown in Chapters 3 to 6, which show the performance more intuitively.

In conclusion, we implemented a simulation framework using the event scheduling and handling method, to run simulations and collect performance results of our solutions to various broadcast-based transaction-processing problems. This chapter introduced the details of the simulation framework. Simulation results given by this tool have proved the efficiency of our solutions from different angles, levels and circumstances.

CHAPTER 8
SUMMARY AND FUTURE WORK

8.1 Summary

Broadcast-based asymmetric communication is an appropriate model for Internet and wireless computing. Research problems in such an environment have drawn much attention in the past few years. In this dissertation, we first surveyed existing research work in such an environment. Then, based on the observation that most of that work only considers single-data operations, we explored some issues of supporting efficient transaction processing in a broadcast-based asymmetric communication environment. These issues included both new problems as well as solved transaction-based problems that need better solutions. Solving these problems and evaluating the performance of solutions were the focus of this work.

The studied topics were divided into four categories, including an efficient concurrency control protocol called STUBCast, problems introduced by STUBcast, transaction processing in hierarchical broadcasting models, and some performance issues for broadcast-based transaction processing.

All problems studied have the characteristic of supporting efficient transaction processing. STUBcast, which guarantees single and local serializability, were introduced as a new efficient concurrency control protocol. Moreover, STUBcache, STUBcast adapted scheduling, and STUBindex were designed, respectively, as cache, scheduling, and index strategies integrated with the STUBcast solution model. Transaction

processing in hierarchical broadcasting models mainly addresses transaction concurrency control problems in a proxy server model and two-level server model. Finally, some transaction processing performance issues were studied, including balancing and minimizing transaction response time, broadcast-based real-time periodic transaction scheduling, long transaction-favored caching, and soft real-time caching.

While motivations, designs, solutions, and results of these topics were the main focus of this dissertation, we also presented the design and implementation of the simulation framework. We used this framework to simulate our designs and solutions. Performance results for different problems are then collected and evaluated from it. Simulation results presented in each chapter have shown the efficiency of our solutions to these studied problems from different angles, levels and circumstances.

The contribution of this work is it can provide many efficient solutions to various broadcast-based transaction processing applications problems. Note that the problems studied are not directly related to one single type of application. Many problems were considered under different assumptions and system models, though they are all under broadcast-based asymmetric environments with transaction processing. However, the efficient transaction processing supporting schemes designed in this dissertation for these problems can be adapted to different kinds of applications when their models match the problems' system assumptions. Some future work from this dissertation research could be to develop a solutions package, which implements all solutions found for these studied transaction-based problems. Then, real applications can use this package directly as their implementation components when any part of their application matches any of these

problems. Examples of possible applications mapping to the topics discussed in this dissertation are shown in Table 8-1.

Table 8-1 Possible applications for each studied topic

Problem	Applications
An efficient concurrency control protocol, and problems introduced by it	
STUBcast	Any transaction processing applications under broadcast data distribution, such as stock buyer, ticket agent, intra-company database, etc.
STUBcache	
STUBcast adapted scheduling	
STUBcast indexing	
Transaction processing in hierarchical broadcasting models	
Proxy Server Model	Intra-hotel management, hospital patient situation information system.
Two-level Server Model	Traffic information system, traveler information system (with user providing update and new information source)
Performance issues for transaction processing	
Balance and minimize transaction response time	Insurance agent, intra-company database
Broadcast scheduling for real-time periodic transactions	Hard real-time control systems using broadcasting as data accessing method
Long transaction-favored caching	Insurance agent, stock buying and selling
Soft real-time caching	Stock buying, auction

8.2 Future Work

We should consider the following major directions for future work based on this dissertation.

- Simulate and evaluate the performance of RSP_{LS} that supports local serializability in STUBcast environment. In current work, we have only simulated RSP_{SS} with UTVP and SVP for the STUBcast protocol.
- For all the cache strategies studied in this dissertation, such as STUBcache, long transaction-favored cache, and soft real-time caching, an ideal cache implementation has been assumed. For example, although policy *LDF* supports better real-time performance, its scheme can introduce large time and space complexity in real cache implementations because it keeps every data unit's access deadline among all current transactions, and these transactions change dynamically. In [ACH95a, ZDO94], PIX is implemented using a substitute LIX, since LIX is more easily to implement in a real cache. Therefore, for future work,

we should find and evaluate a substitute version of policy *LDF* such that it can be easily implemented in real applications. The same also applies to STUBcache and long transaction-favored cache.

- We should also conduct simulation work on the concurrency control strategies designed for the hierarchical broadcasting models to evaluate their performance.
- We should either prove that the scheduling problem of real-time periodic transaction model is NP-hard or construct an optimal scheduling algorithm that can find a feasible schedule once it exists.
- As mentioned earlier, we could develop a solutions package that implements all solutions found for the transaction-based problems studied. Then, real applications can use this package directly as their implementation components when any part of their application matches any of these problems.

LIST OF REFERENCES

[ACH95a] Acharya, S., Alonso, R., Franklin, M., Zdonik, S., “Broadcast Disks: Data Management for Asymmetric Communication Environments,” Proceedings of the ACM SIGMOD Conference, San Jose, California, pp.199-210, June 1995.

[ACH95b] Acharya, S., Franklin, M., Zdonik, S., "Dissemination-based Data Delivery Using Broadcast Disks,” IEEE Personal Communications, V2, N6, pp.50-60, December 1995.

[ACH96a] Acharya, S., Franklin, M., Zdonik, S., “Disseminating Updates on Broadcast Disks,” Proceedings of the 22nd VLDB Conference, Bombay, India, pp.354-365, September 1996.

[ACH96b] Acharya, S., Franklin, M., Zdonik, S., “Prefetching from a Broadcast Disk,” Proceedings of the 12th IEEE ICDE Conference, New Orleans, Louisiana, pp.276-285, February 1996.

[ACH97] Acharya, S., Franklin, M., Zdonik, S., “Balancing Push and Pull for Data Broadcast,” Proceedings of the ACM SIGMOD Conference, Tuscon, Arizona, pp.183-194, May1997.

[ACH98] Acharya, S., Muthukrishnan, S., “Scheduling On-demand Broadcasts: New Metrics and Algorithms,” Proceedings of the ACM MobiCom Conference, Dallas, Texas, pp.43-54, October 1998.

[AKS97] Aksoy, D., Franklin, M., “On-Demand Broadcast Scheduling,” Technical Report, CS-TR-3854, University of Maryland, 1997.

[AKS98] Aksoy, D., Franklin, M., “Scheduling for Large-Scale On-Demand Data Broadcasting,” Proceedings of the IEEE INFOCOM Conference, San Francisco, California, pp.651-659, March1998.

[ALM00] Almeroth, K. C., “The Evolution of Multicast: From the MBone to Interdomain Multicast to Internet2 Deployment,” IEEE Network, pp.10-20, January 2000.

[ALT99] Altinetl, M., Aksoy, D., Baby, T., Franklin, M., Shapiro, W., Zdonik, S., “DBIS-toolkit: Adaptable Middleware for Large Scale Data Delivery,”

Proceedings of the ACM SIGMOD Conference, Philadelphia, Pennsylvania, pp.544-546, June 1999.

[BAR94] Barbara, D., Imielinski, T., "Sleepers and Workaholics: Caching Strategies in Mobile Environments," Proceedings of ACM SIGMOD Conference, Minneapolis, Minnesota, pp.1-12, May 1994.

[BAR97a] Barbara, D., "Certification Reports: Supporting Transactions in Wireless Systems," Proceedings of the IEEE ICDCS Conference, Baltimore, Maryland, May 1997.

[BAR97b] Baruah, S., Bestavros, A., "Pinwheel Scheduling for Fault-tolerant Broadcast Disks in Real-time Database Systems," Proceedings of the 13th IEEE ICDE Conference, Birmingham, United Kingdom, pp.543-551, April1997.

[BEN96a] Bennett, J. C. R. Zhang, H., "Hierarchical Packet Fair Queuing Algorithms," Proceedings of the ACM SIGCOMM Conference, Stanford, California, pp.143-156, August 1996.

[BEN96b] Bennett, J. C. R. Zhang, H., "WF^2Q: Worst-case Fair Weighted Fair Queuing," Proceedings of the ACM INFOCOM Conference, San Francisco, California, pp.120-128, March 1996.

[BEN96c] Bennett, J. C. R. Zhang, H., "Worst-cast Fair Weighted Fair Queuing," Technical Report, Computer Science Department, Carnegie Mellon University, 1996.

[BER87] Bernstein, P. A., Hadzilacos, V., Goodman, N., "Concurrency Control and Recovery In Database Systems," Addison-Wesley Publishing Company, 1987.

[BES94] Bestavros, A., "An Adaptive Information Dispersal Algorithm for Time-Critical Reliable Communication," Network Management and Control, V2, Plenum Publishing Corporation, New York, New York, pp.423-438, 1994.

[BES96] Bestavros, A., "AIDA-based Real-Time Fault-Tolerant Broadcast Disks," Proceedings of the 2nd IEEE RTAS Conference, Boston, Massachusetts, June 1996.

[BOW92] Bowen, T.F., Gopal, G., Herman, G., Hickey, T., Lee, K.C., Mansfield, W.H., Raitz, J., Weinrib, A., "The Datacycle Architecture," Communications of the ACM 35, V35, N12, pp.71-81, December 1992.

[CHE97] Chen, M. S., Yu, P. S., Wu, K. L., "Indexed Sequential Data Broadcasting in Wireless Mobile Computing," Proceedings of the 17th IEEE ICDCS Conference, Baltimore, Maryland, pp.124-131, May 1997.

[DAO96] Dao, S., Perry, B., "Information Dissemination in Hybrid Satellite/Terrestrial Networks," Bulletin of the IEEE Technical Committee of Data Engineering, V19, N3, pp.12-18, September 1996.

[DAT97] Datta, A., Celik, A. Kim, J. G., VanderMeer, D. E., Kumar, V., "Adaptive Broadcast Protocols to Support Efficient and Energy Conserving Retrieval from Databases in Mobile Computing Environment," Proceedings of the 13th IEEE ICDE Conference, Birmingham, United Kingdom, pp124-133, April 1997.

[DEM89] Demers, A., Keshav, S., Shenker, S., "Analysis and Simulation of a Fair Queuing Algorithm," Proceedings of the ACM SIGCOMM Conference, Austin, Texas, pp.3-12, September 1989.

[FER99] Fernandez-Conde, J., Ramamritham, K., "Adaptive Dissemination of Data in Time-Critical Asymmetric Communication Environments," Proceedings of the 11th IEEE Euromicro-RTS Conference, York, United Kingdom, June 1999.

[FRA96] Franklin, M., Zdonik, S., "Dissemination-Based Information Systems," Bulletin of the IEEE Technical Committee on Data Engineering, V19, N3, pp.20-30, September 1996.

[FRA97] Franklin, M., Zdonik, S., "A Framework for Scalable Dissemination-Based Systems," Proceedings of the ACM OOPSLA Conference, Atlanta Georgia, pp.94-105, October 1997.

[HAA98] Haartsen, J., Allen, W., Jon, I., "Bluetooth: Vision, Goals and Architecture," ACM Mobile Computing and Communication Review, V2, N4, pp.38-45, October1998.

[HAM97] Hammeed, S., Vaidya, N. H., "Efficient Algorithms for Scheduling Single and Multiple Channel Data Broadcast," Technical Report 97-002, Department of Computer Science, Texas A&M University, February 1997.

[HER87] Herman, G., Gopel, G., Lee, K. C., Weinrib, A., "The Datacycle Architecture for Very High Throughput Database Systems," Proceedings of the ACM SIGMOD Conference, San Francisco, California, pp. 97-103, May 1987.

[HU99] Hu, Q. L., Lee D. L., Lee. W. C., “Performance Evaluation of a Wireless Hierarchical Data Dissemination System,” Proceedings of the ACM MobiCom, Seattle, Washington, pp.163-173, August 1999.

[HU01] Hu, Q. L., Lee, W. C., Lee, D. L., “A Hybrid Index Technique for Power Efficient Data Broadcast,” Distributed and Parallel Databases Journal, V9, N2, pp.151-177, March 2001.

[HUA99] Huang, A. C., Ling, B. C., Ponnekanti, S., “Pervasive Computing: What is it Good For,” Proceedings of the ACM MobiDE Workshop, Seattle, Washington, September 1999.

[HUA00] Huang, Y., “Analysis results of optimal mapping of *Ucast* to data unit, without considering queuing delay of *Ucast* requests”, Report, October 2001.

[HUA01a] Huang, Y., Lee, Y. H., “Caching Broadcasted Data for Soft Real-Time Transactions,” Proceedings of the IIIS SCI/ISAS Conference, Orlando, Florida, V15, pp.37-42, August 2001.

[HUA01b] Huang, Y., Lee, Y. H., "Concurrency Control for Broadcast-based Transaction Processing and Correctness Proof," Proceedings of the 14th ISCA PDCS Conference, Dallas, Texas, pp.130-135, August 2001.

[HUA01c] Huang, Y., Lee, Y. H., “STUBcast - Efficient Support for Concurrency Control in Broadcast-based Asymmetric Communication Environment,” Proceedings of the 10th IEEE ICCCN Conference, Scottsdale, Arizona, pp.262-267 October 2001.

[IMI94a] Imielinski, T., Badrimath, B., “Mobile Wireless Computing: Challenges in Data Management,” Communications of the ACM, V37, N10, pp.18-28, October1994.

[IMI94b] Imielinski, T., Viswanathan, S., Badrimath, B., “Energy Efficient Indexing on Air,” Proceedings of the ACM SIGMOD Conference, Minneapolis, Minnesota, pp.25-36, May 1994.

[IMI94c] Imielinski, T., Viswanathan, S., Badrimath, B., “Power Efficient Filtering of Data on Air,” Proceedings of the 4th EDBT Conference, Cambridge, United Kingdom, pp.245-258, March 1994.

[JAI95] Jain, R., Werth, J., “Airdisks and AirRAID: Modeling and Scheduling Periodic Wireless Data Broadcast,” Technical Report 95-11, Computer Science Department, Rutger University, May 1995.

[JIA99] Jiang, S. Vaidya, N. H., “Scheduling Data Broadcast to 'Impatient' Users,” Proceedings of the ACM MobiDE Conference, Seattle, Washington, pp.52-59, August 1999.

[JIN97] Jing, J., Elmargarmid, A. K., Helal, S., Alonso, R., “Bit-Sequences: An Adaptive Cache Invalidation Method in Mobile Client/Server Environments,” ACM/Baltzer Mobile Networks and Applications, V2, N2, pp.115-127, April 1997.

[KHA98] Khanna, S., Zhou, S., “On Indexed Data Broadcast,” Proceeding of the 30th STOC Conference, New York, New York, V30, pp.463-472, May 1998.

[KLE76] Kleinrock, L. “Queuing Systems,” V2, Computer Applications, Wiley, 1976.

[KRI97] Krishna, C. M., Kang, G. S., “Real-Time Systems,” McGRAW-HILL International Editions, 1997.

[LAM98] Lam, K. Y., Chan, E., Yue C. H., “Data Broadcast for Time-Constrained Read-Only Transactions in Mobile Computing Systems,” Proceedings of IEEE WECWIS Workshop, Santa Clara, California, pp.11-19, April 1998.

[LAM99] Lam, K. Y., Au, M. W., Chan, E., “Broadcast of Consistent Data to Read-Only Transactions from Mobile Clients,” Proceedings of the 2nd IEEE WMCSA Workshop, New Orleans, Louisiana, February 1999.

[LAW97] Law, K. L. E., Nandy, B., Chapman, A., “A Scalable and Distributed WWW Proxy System,” Nortel Limited Research Report, 1997.

[LEE99] Lee, V. C. S., Lam, K. W., Son, S. H., “Maintaining Data Consistency Using Timestamp Ordering in Real-Time Broadcast Environments,” Proceeding of the 6th IEEE RCSA Conference, Hong Kong, China, pp.29-36, December 1999.

[LEO98] Leonard, G. S., “Satellite Communications Systems Move Into the Twenty-first Century,” ACM Wireless Networks, V4, N2, pp.101-107, 1998.

[LEO97] Leong, H. V., Si, A., “Database Caching over the Air-storage,” The Computer Journal, V40, N7, pp.401-415, 1997.

[LIU73] Liu, C. L., Layland, J. W., “Scheduling Algorithms for Multiprogramming in Hard-Real-Time Environment,” Journal of the ACM, V20, N1, pp. 46-61, 1973.

[OZS91] Ozsu, M. T., Valduriez, P., “Principles of Distributed Database Systems,” Prentice Hall, 1991.

[PEN98] Peng, C. S., Pulido, J. M., Lin, K. J., Glough, D. M., “The Design of an Internet-based Real-time Auction System,” Proceedings of the DARE98 Workshop, Denver, Colorado, pp.70-78, June 1998.

[PIT98] Pitoura, E., “Supporting Read-Only Transactions in Wireless Broadcasting,” Proceedings of the DEXA98 Workshop, Vienna, Austria, pp.428-422, August 1998.

[PIT99a] Pitoura, E., Chrysanthis, P. K., “Exploiting Versions for Handling Updates in Broadcast Disks,” Tech. Report TR:99-02, Univ. of Ioannina, Computer Science Dept., 1999.

[PIT99b] Pitoura, E., Chrysanthis, P. K., “Scalable Processing of Read-Only Transactions in Broadcast Push,” Proceedings of the 19th IEEE ICDCS Conference, Austin Texas, May 1999.

[SHA99] Shanmugasundaram, J., Nithrakashyap, A., Sivasankaran, R., Ramamritham, K., “Efficient Concurrency Control for Broadcast Environments,” Proceedings of the ACM SIGMOD Conference, Philadelphia, Pennsylvania, pp.85-96, June 1999.

[STA96] Stathatos, K., Roussopoulos, N., Baras, J. S., “Adaptive Data Broadcasting using Air-Cache,” Technical Research Report, CSHCN T.R. 96-12, University of Maryland, 1997.

[STA97] Stathatos, K., Roussopoulos, N., Baras, J. S., “Adaptive Data Broadcast in Hybrid Networks,” Proceedings of the 23rd VLDB Conference, Athens, Greece, V30, N2, pp.326-335, September 1997.

[SUB99a] Subramaninian, S., Singhal, M. “A Real-time Protocol for Stock Market Transactions,” Proceedings of the WECWIS Workshop, Santa Clara, April 1999.

[SUB99b] Subramaninian, S., Singhal, M., “Real-time Aware Protocols for General E-commerce and Electronic Auction Transactions,” Proceedings of the IEEE ICDCS Conference Workshop, Austin, Texas, May 1999.

[VAI96] Vaidya, N. H., Hameed, S., “Data Broadcast Scheduling: On-Line and Off-Line Algorithms,” Technical Report 96-017, Computer Science, Texas A&M University, July 1996.

[VAI97a] Vaidya, N. H., Hameed, S., “Scheduling Data Broadcast in Asymmetric Communication Environment,” Technical Report, Texas T&M University, 1997.

[VAI97b] Vaidya, N. H., Hameed, S., “Log Time Algorithms for Scheduling Single and Multiple Channel Data Broadcast,” Proceedings of the 3rd ACM/IEEE MobiCom Conference, Budapest, Hungary, pp.90-99, September 1997.

[XUA97] Xuan, P., Sen S., Gonzalez, O., Fernandex, J., Ramamaritham, K., “Broadcast on Demand: Efficient and Timely Dissemination of Data in Mobile Environment,” Proceedings of the IEEE RTAS Conference, Montreal, Canada, June 1997.

[YUA94] Yuan, X. P., Saksena, M. C., Agrawala, A. K., “A Decomposition Approach to Non-Preemptive Real-Time Scheduling,” Real-Time Systems, V6, N1, pp.7-35, January 1994.

[ZDO94] Zdonik, S., Franklin, M., Alonso, R., Acharya, S., “Are ‘Disks in the Air’ Just Pie In the Sky?” Proceedings of the IEEE WMCSA Workshop, Santa Cruz, California, pp.12-19, December1994.

BIOGRAPHICAL SKETCH

Yan Huang was born on May 26, 1972, in Jiangsu, P.R. China. She received her Bachelor of Engineering degree in Computer Science from the University of Electronic Science and Technology of China, Chengdu, P.R. China, in July 1994. She then received her Master of Engineering degree in Computer Science from the University of Electronic Science and Technology of China, Chengdu, P.R. China, in March 1997. She worked as a computer engineer in the Network Department of the Sichuan Branch of China Construction Bank for half a year. In Fall 1997 she joined the Department of Computer and Information Science and Engineering at the University of Florida, Gainesville, Florida to pursue her Doctor of Philosophy degree. Since then she has worked as a research assistant in the Real-Time Systems Research Lab led by Professor Yann-Hang Lee and as a teaching assistant in the CISE Department. Her research experience and interests include real-time systems, distributed computing, system simulation and performance evaluation, broadcast-based database transaction-processing, mobile and wireless computing, and sensor networks. She also has strong development experience with GUI, network programming, concurrent programming, and embedded systems.