

**FILE ALLOCATION IN MOBILE
DISTRIBUTED SYSTEMS**

**ARTHUR LAKES LIBRARY
COLORADO SCHOOL OF MINES
GOLDEN, CO 80401**

by
Lin Zhu

ProQuest Number: 10795498

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 10795498

Published by ProQuest LLC (2018). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Copyright by Lin Zhu 2001

All Rights Reserved

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematical and Computer Sciences).

Golden, Colorado

Date 03/26/2001

Signed: Zhu Lin
Lin Zhu

Approved: DP Mehta
Dr. Dinesh Mehta
Associate Professor

Golden, Colorado

Date 4/4/01

Graeme Fairweather
Dr. Graeme Fairweather
Professor and Head
Department of Mathematical and
Computer Sciences

ABSTRACT

A critical problem in the design of computer systems is that of assigning files to possibly different nodes in a computer network for query/update/execution purposes; this is commonly known as the “file allocation problem” (FAP). In a mobile distributed system, mobility makes the file allocation problem even more complex.

In this thesis, we employ a directed acyclic graph(DAG) to describe the dependencies among the files to be accessed in the FAP. We formulate a new DAG-based version of the FAP. This problem is shown to be NP-complete. Solutions based on integer linear programming and list-scheduling are proposed and compared.

TABLE OF CONTENTS

ABSTRACT	iii
LIST OF FIGURES	vi
LIST OF TABLES	vii
ACKNOWLEDGMENTS	viii
Chapter 1 INTRODUCTION	1
1.1 File Allocation Problem	1
1.2 Previous Work	2
1.3 Mobile Information Access	4
Chapter 2 FILE ALLOCATION IN MOBILE DISTRIBUTED SYSTEMS	7
2.1 Model	7
2.2 An Example of FAP Problem	9
2.3 NP-completeness	10
2.3.1 Proof	11
2.4 Integer Linear Programming Solution	13
2.4.1 Introduction	14
2.4.2 Integer Constants	14
2.4.3 Integer Variables	15
2.4.4 Primary Constraints	17
2.4.5 Objective	19
2.4.6 Code in Lingo	20
2.4.7 Solutions Construction	20
2.5 Heuristic Algorithm	22
2.5.1 Definitions	22
2.5.2 Algorithm	25
2.6 Feasibility Lemma	26
2.6.1 Notation	26
2.6.2 A Small FAP With No Solution	27
2.6.3 Lemma	27

Chapter 3	TESTS AND RESULTS	31
3.1	ILP Solutions	31
3.2	Lower Bounds	33
3.2.1	Lower Bound 1	33
3.2.2	Lower Bound 2	34
3.3	Results	35
3.3.1	Parameters	35
3.3.2	Effect of Density	36
3.3.3	Effect of the Number of Servers	38
3.3.4	Effect of the Number of Files	40
3.3.5	Effect of the Number of Nodes	42
3.3.6	Random Assignment	43
3.4	Conclusion	43
Chapter 4	SUMMARY AND FUTURE WORK	45
References		46
Appendix A		48
A.1	Code in Lingo	48

LIST OF FIGURES

2.1	Mobile Networks	8
2.2	File Access DAG	9
2.3	The ILP Solution	21
2.4	Level Calculation	23
3.1	Time Comparison for Different Value of Density	37
3.2	Time Comparison for Different Number of Servers	39
3.3	Time Comparison for Different Number of Files	41
3.4	Time Comparison for Different Number of Nodes	43

LIST OF TABLES

3.1	Time Comparison for Different Value of Density	37
3.2	Time Comparison for Different Number of Servers	39
3.3	Time Comparison for Different Number of Files	41
3.4	Time Comparison for Different Number of Nodes	42
3.5	Time Comparison with Random Assignment	44

ACKNOWLEDGMENTS

I could not imagine undertaking a task this substantial without the help, support, and guidance of many people. I am extremely fortunate to have had support of the best such community: the faculty, students, and staff of the Colorado School of Mines.

I would particularly like to acknowledge my advisor, Dinesh Mehta, for his patience and support. He provided useful and insightful feedback for all my research ideas, and gave me the confidence to overcome difficult research problems. I consider it an honor to have worked with and known him.

I would like to thank Dr. Krishna Kavi at the University of Alabama in Huntsville for suggesting the problem.

I would also like to thank Dr. Xindong Wu and Dr. Tracy Camp for serving as my committee.

Standing in CSM, I would also like to thank Dr. Bruce Whitehead and all my friends in the University of Tennessee Space Institute. The memory of those wonderful days in UTSI will stay in my heart forever.

I would like to thank my parents. It is them who teach me how to be optimistic about the life. No matter where I am, no matter rain or snow, my heart is full of the sunshine of their love.

Finally, my husband Jing has done more for me than I could ever possibly acknowledge or repay. I could not have reached this point without him, nor can I imagine a better soul with whom to face future challenges. Finishing this degree and getting on with our life is the best I can offer.

Chapter 1

INTRODUCTION

1.1 File Allocation Problem

The optimal distribution of files is a major problem in computer system optimization. The potential gain obtained by solving a file allocation problem is significant. In some cases (Lawrence W. Dowdy, Derrell V. Foster, June 1982), a 40 percent reduction in cost was obtained by moving from an arbitrary file assignment to an optimal one.

The file allocation problem is complex, and its solution is nontrivial. For example, in a network of six servers and ten files, the number of possible file allocations, where each node may have a copy of each file, is 2^{60} (i.e., 2^6 possible allocations for one file; $(2^6)^{10}$ for 10 files). Even when multiple file copies are disallowed (i.e., each file is placed at one and only one server), there are 6^{10} possible assignments. It is unfortunate but possibly true that all file allocation problems are complex to the point of being theoretically impossible to efficiently solve. If so, this would imply that heuristic solution techniques may be the more practical approach.

The selection of suitable heuristics is a nontrivial task, since optimal file allocations may seem counterintuitive. For example, consider a FAP that seeks to maximize the sum of nodal throughput of a system. Allocating a file to the server where its usage requirement is highest may be a poor heuristic; allocating a file to a server that rarely uses it may be optimal because of traffic congestion generated by the remaining servers.

An acceptable FAP solution assigns files to the servers in some “optimal” fashion. FAP solution techniques vary depending upon the definition of “optimal.” One measure of an “optimal” file allocation is performance. Minimum access time and maximum system throughput are common performance objectives. Once the system is designed, and the hardware is obtained, performance becomes a primary issue. The emphasis changes to performance in order to tune the system to realize its performance potential. The performance potential is with respect to a workload, which, at production time, is more precisely known.

The design of distributed processing systems has increasingly drawn the attention of systems designers as more and more organizations recognize the advantages of decentralizing operation. File allocation problem is one of the most important problems in distributed systems dealing with the assignment of files to servers in view of costs and delays. In these systems, it is obvious that there is a trade-off between costs and delays. While the partitioning of a file determines the degree of parallelism in servicing a single request to the file, the allocation of all the files (partitions) onto the servers is an equally important parameter that affects the overall performance of a system. In order to fully benefit from the performance capabilities of a distributed system, it has been widely recognized that the load must be uniformly distributed among all servers. Otherwise, the creation of performance bottlenecks on some of the servers may severely limit the response time of requests, as well as the overall system throughput.

1.2 Previous Work

The FAP was originally studied by (Chu, W., Oct. 1969). In his model, multiple file copies are allocated in the distributed computing systems with the objective of

minimizing operating costs.

(Lawrence W. Dowdy, Derrell V. Foster, June 1982) is a classical survey paper that attempts to classify, notationally unify, and summarize the FAP literature. It compares all the models and gives suggestions for new directions for FAP research. A significant amount of research on FAP has been undertaken subsequent to the survey article. We briefly review some of the literature.

Algorithms for allocating files to disks in parallel or distributed systems have been extensively studied in the literature (S. March, S.Rho, Mar./Apr. 1995), (Deb Ghosh, Ishwar Murthy, 1991). Typically, these algorithms assign data to the disks of a parallel or distributed system in such a way that a particular cost function is minimized. In the most general case, the cost function may involve communication costs, storage costs, update costs, and queuing costs. However, finding the optimal solution, even for very simple cost functions, is a NP-complete problem. Consequently, viable solutions must be based on heuristics.

A large amount of research has been done on allocating files into shared disks in order to minimize the delays, such as (Lin-Wen Lee, 2000) and (P.Scheuermann, G. Weikum, P.Zabback, 1998). They modeled the servers using queuing theory, such as M/M/1 or M/D/1. They assumed that the same type of file has the same mean access rate and the same service time. They concentrate on accessing files from the shared disk.

Recently, various information and communication services have been provided to homes and individuals as well as businesses, by the developments of networks technology and the wide usage of desktop computers and notebooks. So, the file allocation problem has also been studied in conjunction with some networks problems.

(Wentong Cai, Bu-Sung Lee, 1998) reports on the results of a heuristic algorithm

that is used to distribute the data files among the multiple heterogeneous servers interconnected by a fast network. Its objective is to minimize the differences between response times of the servers. It is used to upgrade the old Televue System to a Multiple Machine Televue System.

(Rahul Simha, B. Narahari, Hyeong-Ah Chio, Li-Chuan Chen, 1996) presents two fast heuristic algorithms to the problem of allocating files in a document tree among multiple processors in a parallel webserver. It is assumed that access patterns are characterized by branching probabilities for an access that starts at a node and progresses down the tree.

(Jorg Jensch, Reinhard Luling, Norbert Sensen, 1998) presents a new mechanism for mapping data items onto the storage divides of a parallel web server. The method is based on careful observation of the effects that limit the performance of parallel web servers and by studying the access patterns for these servers.

1.3 Mobile Information Access

The ability to access information on demand at any location confers competitive advantage on individuals in increasingly mobile networks. As users become more dependent on this ability, the span of access of data repositories will have to grow. The increasing social acceptance of the home or any other location as a place of work is a further impetus to the development of mechanisms for mobile information access.

Mobile file access has the same difficulties that traditional file access has. It is characterized by some other fundamental challenges as well (M. Satyanarayanan, 1998). Mobile elements are resource-poor relative to static elements. They rely on a finite energy source. Outdoors, a mobile client may have to rely on low-bandwidth unreliable wireless networks. These constraints are not artifacts of current technol-

ogy, but are intrinsic to mobility. Together, they complicate the design of mobile distributed systems and require us to rethink traditional approaches to mobile file access.

Due to the weak link between the mobile unit and the proxy servers around it, mobile file access is plagued by the high latencies inherent in remote access. A widely used solution is to prefetch information to the proxy servers before it is requested by the user in order to hide latency. Wherever the mobile element is, there must be some proxy servers available to it. Through prefetching, files on different servers may be accessed in parallel, and the fetching of some files may overlap the processing of others. Second, prefetching may result in better network and server utilization, as many requests may be queued at a time. Third, the system can sometimes reorder the prefetching of files. But this raises the problem of knowing what to prefetch, since prefetching data that will not be used can actually hurt performance.

Fortunately, the nature of file access provides an opportunity to take advantage of prefetching. Suppose that while traveling, you would like to use your portable computer and cellular phone to look for a local hotel. You might make a reservation after you find a satisfactory hotel. You won't look at the details of every hotel. Once you are interested in one hotel, you may want to have more details about it. Once you decide to make a reservation, you have to follow the steps as you are required to do. If the details about a hotel are prefetched when the system finds that you are interested in it, if the forms you need to fill are prefetched at the beginning of making your reservation, the latency would be hidden. In this example, prefetching doesn't make a big difference, since what you can prefetch is only a few pages. But, what if you are searching a large amount of data for a complicated computation? What if you are querying data from a massive data base?

This active prefetching idea is used in mobile distributed systems, (Hui Lei, Dan Duchamp, Jan. 1997), (Mahadev Satyanarayanan, February 1996) . In Coda and Odyssey systems that were developed at Carnegie Mellon University, the cache manager Venus combines implicit and explicit sources of information into a priority-based cache management algorithm. The implicit mode uses the basic caching algorithm. The explicit mode takes the form of a per-client hoard database(HDB), which identifies objects of interest to the users at that client. They also have a simple front-end program that allows the user to update the HDB.

Taking advantage of the pre-defined dependencies in prefetching is important for a file allocation problem in order to improve the overall performance of the mobile distributed system.

In our research, we emphasize the relationship among the files. There are many applications in which file accesses are not random. The user does know the sequence of file accesses. The current file access operation gives some hints or has some relationship with the next file access operation. In other words, our research is different from traditional FAP because file access dependencies are modeled by a DAG whereas pervious FAP results do not take these dependencies into account. Also, in a mobile system, everything keeps changing. Due to mobility, a fast solution is necessary in solving such a FAP in mobile distributed systems. In our research, we propose a heuristic algorithm to deal with large scale FAP problems fast and efficiently.

In Chapter 2, we will model the DAG-based File Allocation Problem(FAP), prove its NP-completeness, propose an Integer Linear Program(ILP) formulation and a heuristic algorithm. In Chapter 3, we will compare all the results of ILP and the heuristic algorithms.

Chapter 2

FILE ALLOCATION IN MOBILE DISTRIBUTED SYSTEMS

In a mobile distributed system, the link between the servers and the mobile unit might be unreliable and low-bandwidth, and everything keeps changing all the time. So the algorithm used for file allocation has to be fast. Based on the intrinsic constraints of mobile systems, we provide a new model for the file allocation problem used in mobile distributed systems.

2.1 Model

The file allocation model for a mobile distributed system can be shown as a mobile unit and a set of proxy servers connected to the mobile unit (see Figure 2.1). Our task is to access a set of files from these proxy servers. Since computation times are usually negligible in comparison with access times, they are omitted.

The problem is formally defined below.

Given:

1. A set of proxy servers: S_1, S_2, \dots, S_{ns} ;
2. Each server has a capacity: $Capacity_{S_1}, Capacity_{S_2}, \dots, Capacity_{S_{ns}}$.
3. A set of files: F_1, F_2, \dots, F_{nf} ;
4. Each file has a size: $Size_{F_1}, Size_{F_2}, \dots, Size_{F_{nf}}$.
5. A set of nodes: N_1, N_2, \dots, N_{nn} ;

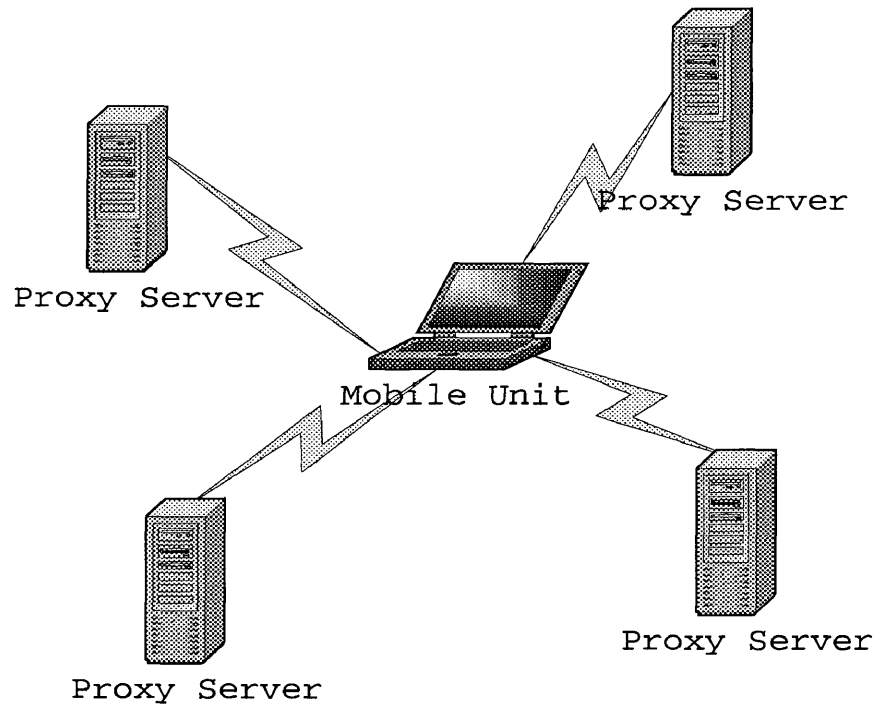


FIG. 2.1. Mobile Networks

Each node represents a “file access” operation. If the same file needs to be accessed several times, we use a different node for each access. So, each “file access” could access a file or a portion of a file. If node N_n is or is part of file F_f , we say $N_n \in F_f$.

6. A mobile unit has a directed acyclic graph(DAG) associated with it.

Each node in the DAG is the node mentioned above. Each edge in the DAG represents an ordering of file accesses. If there is an edge from node N_i to node N_j , we say node N_i is the predecessor of node N_j , i.e, node N_i has to be finished before node N_j begins its access.

7. A set of file access times, $\text{AccessTime}(N_n, S_s)$;

A file access time is a function of the file being accessed and the server that

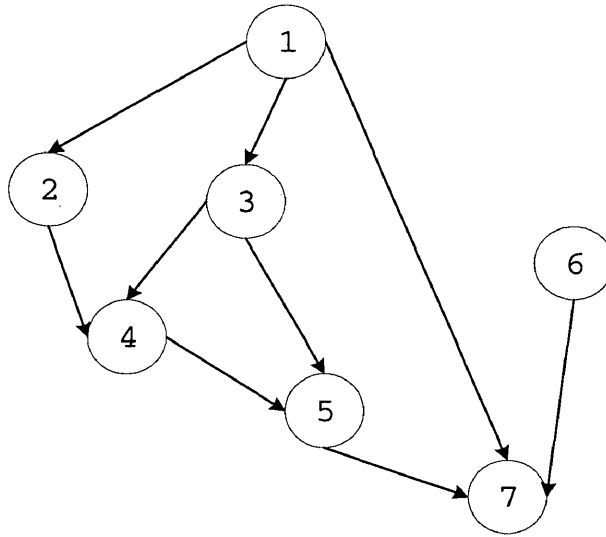


FIG. 2.2. File Access DAG

contains the file.

Objectives:

1. Assign files to proxy servers such that the sum of files sizes on any proxy server is less than its capacity. The same file can be assigned to several proxies, if space permits.
2. Schedule file accesses according to the graph dependencies.
3. Under the above constraints, we want to minimize total file access time.

2.2 An Example of FAP Problem

Given:

1. Three proxy servers: S_1, S_2, S_3 ;
2. Three capacities: $Capacity_{S_1} = 100, Capacity_{S_2} = 50, Capacity_{S_3} = 80$;

3. Six files: F_1, F_2, \dots, F_6 ;
4. Each file has a size: $Size_{F_1} = 20, Size_{F_2} = 50, Size_{F_3} = 5, Size_{F_4} = 30, Size_{F_5} = 10, Size_{F_6} = 25$;
5. A set of graph nodes: N_1, N_2, \dots, N_7 ;
 $N_1 \in F_1, N_2 \in F_2, N_3 \in F_3, N_4 \in F_4, N_5 \in F_5, N_6 \in F_4, N_7 \in F_3$;
6. A mobile unit with a directed acyclic graph(DAG) (see Figure 2.2).
7. A set of file access times;

	N_1	N_2	N_3	N_4	N_5	N_6	N_7
S_1	1	1	2	1	1	2	2
S_2	1	2	3	2	2	1	1
S_3	2	1	1	2	2	2	2

Given the above conditions, we want to assign the 6 files to the 3 proxy servers, schedule file accesses according to the graph dependencies, and minimize the total access time.

2.3 NP-completeness

Algorithms for assigning files to disks in parallel or distributed systems have been extensively studied in literature. However, finding the optimal solution, even for very simple models, is an NP-complete problem. A file access problem based on DAG is definitely a NP-completeness problem.

2.3.1 Proof

Proof by restriction is the simplest, and perhaps the most frequently applicable technique used for proving NP-completeness. (A detailed guide to the theory of NP-completeness can be found in (Johnson, 1979).) Using this technique, the process of devising a NP-completeness proof for a decision problem Π will consist of the following three steps:

1. show that Π is in NP,
2. select a known NP-complete problem Π' ,
3. construct a transformation f from Π' to Π , and
4. prove that f is a polynomial transformation.

The heart of such a proof lies in the specification of the additional restrictions to be placed on the instances of Π so that the resulting restricted problem will be identical to Π . It is not required that the restricted problem and the known NP-complete problem be *exact* duplicates of one another, but rather that there be an “obvious” one-to-one correspondence between their instances that preserves “yes” and “no” answers. There are several well-known NP-complete problems that have been used frequently, such as HAMILTONIAN CIRCUIT and CLIQUE. PARTITION is one of them which can be used as Π' in our proof.

PARTITION

Instance: A finite set A and a “size” $s(a) \in Z^+$ for each element $a \in A$.

Question: Is there a subset $A' \subseteq A$ such that

$$\sum_{a \in A} s(a) = \sum_{a \in A - A'} s(a)?$$

Our problem can be restricted to a PARTITION problem as follows:

- Each element a is represented by a node.
- Each node is assumed to belong to a different file.
- The size of a file equals to the size of the element.
- No dependency in the DAG.
- Two servers of capacity $\frac{1}{2} \sum_{a \in A} s(a)$
- Two servers of equal speed.
- For each file, its access time equals to 1

Question: Is it possible to finish file accesses in n units of time?

MULTIPROCESSOR SCHEDULING is another NP-complete problem which is close to the problem we are solving. A MULTIPROCESSOR SCHEDULING problem is also proved from the PARTITION problem. The strict proof of it can be found in (Johnson, 1979).

MULTIPROCESSOR SCHEDULING

Instance: A finite set A of “tasks”, a “length” $l(a) \in Z^+$ for each $a \in A$, a number $m \in Z^+$ of “processors,” and a “deadline” $D \in Z^+$.

Question: Is there a partition $A = A_1 \cup A_2 \cup \dots \cup A_m$ of A into m disjoint sets such that

$$\max\left\{\sum_{a \in A_i} l(a) : 1 \leq i \leq m\right\} \leq D \quad ?$$

We can prove the NP-completeness of our problem by restricting it to a MULTIPROCESSOR SCHEDULING problem as follows:

- Each element a is represented by a node.

- Each node is assumed to belong to a different file.
- No dependency in the DAG.
- There are m servers of equal speed.
- Capacity of each server equals to number of the nodes.
- Size of each file is 1 unit.
- For each file, its access time equals to 1.

Question: Is there a partition $A = A_1 \cup A_2 \cup \dots \cup A_m$ of A , where $A_i \subseteq$ server S_i , such that

$$\max\left\{\sum_{a \in A_i} l(a) : 1 \leq i \leq m\right\} \leq D \quad ?$$

Either a PARTITION problem or a MULTIPROCESSOR SCHEDULING problem could be used as Π' in order to prove the NP-completeness of our problem. In another word, our problem is kind of a combination of a PARTITION problem and a MULTIPROCESSOR SCHEDULING problem with some other strong constraints represented by a DAG.

After the analysis above, it is quite straightforward to draw the conclusion that the file assignment problem based on DAG is a NP-complete problem.

2.4 Integer Linear Programming Solution

As proved in last section, our file access problem is a NP-complete problem. This means that it is unlikely that there is a fast algorithm that obtains the optimal solution. In this section, we will solve this problem by linear programming.

2.4.1 Introduction

Mathematical programming, or constrained optimization, or math programming for short, is a mathematical procedure for determining optimal allocation of scarce resources. Math programming, and its most popular special form, Linear Programming (LP), has been found to be a practical solution strategy for several optimization problems. The optimization helps us find the answer that yields the BEST result—the one that attains the highest profit, output, or happiness, or the one that achieves the lowest cost, waste, or discomfort. Often these problems involve making the most efficient use of the resources—including money, time, machinery, staff, inventory, and more. Optimization problems are often classified as linear or nonlinear, depending on whether the relationships in the problem are linear with respect to the variables.

For most optimization problems, one can think of there being two important classes of objects. The first of these is limited resources, such as hard disk capacity, speed, human resource, and sales force size. The second is objectives, such as “release a version smaller than size A,” “release a version faster than B,” and “release a version cheaper than C”. Each objective consumes or possibly contributes additional amounts of the resources. The problem is to determine the best combination of objective levels that does not use more resources than are actually available.

Here, we formulate the problem to minimize the total access time, subject to each node’s access time function and the directed acyclic graph.

2.4.2 Integer Constants

In order to model the problem, we need some integer constants.

1. For each server S_s

$$Capacity(S_s) = \text{capacity of server } S_s$$

2. For each file F_f

$$Size(F_f) = \text{size of file } F_f$$

3. Node N_n is or is part of file F_f , that is

$$Belong(N_n) = F_f$$

4. Time steps

$$Step(T_t) = T_t$$

5. Edges in the DAG between all pairs of nodes node N_i and node N_j , that is

$$Edge(N_i, N_j) = \begin{cases} 1 & \text{if node } N_i \text{ is the predecessor of node } N_j \\ 0 & \text{otherwise} \end{cases}$$

6. Access time for all pairs of node N_n and server S_s ,

$$AccessTime(N_n, S_s) = T_t \text{ (if access time of node } N_n \text{ from server } S_s \text{ is } T_t)$$

2.4.3 Integer Variables

To formulate the optimal file allocation problem, we define three kinds of major 0-1 variables. We formulate this optimization problem as 0-1 integer linear programming by using the following 0-1 variables.

1. The variables on the allocation of files $Include(F_f, S_s)$.

$$Include(F_f, S_s) = \begin{cases} 1 & \text{if file } F_f \text{ is stored in the server } S_s \\ 0 & \text{otherwise} \end{cases}$$

Number of variables is: $\#File * \#Server$

2. The variables used to make sure two nodes corresponding to two files which are in the same server will not be accessed simultaneously, $X(N_i, N_j, S_s)$

$$X(N_i, N_j, S_s) = \begin{cases} 0 & \text{if node } N_i \text{ is accessed before node } N_j \text{ from server } S_s \\ 1 & \text{if node } N_j \text{ is accessed before node } N_i \text{ from server } S_s \end{cases}$$

Number of variables is: $\#Node * \#Node * \#Server$

3. The variables on operation $Begin(N_n, S_s, T_t)$ and $End(N_n, S_s, T_t)$.

$$Begin(N_n, S_s, T_t) = \begin{cases} 1 & \text{if node } N_n \text{ is accessed from server } S_s \text{ starts at time } T_t \\ 0 & \text{otherwise} \end{cases}$$

$$End(N_n, S_s, T_t) = \begin{cases} 1 & \text{if node } N_n \text{ is accessed from server } S_s \text{ ends at time } T_t \\ 0 & \text{otherwise} \end{cases}$$

Number of variables is: $2 * \#Node * \#Server * \#Step$

Total integer variables is:

$$\#File * \#Server + \#Node * \#Node * \#Server + 2 * \#Node * \#Server * \#Step$$

2.4.4 Primary Constraints

1. The number of copies of each file in the whole system is greater than or equal to 1, that is,

$$\forall F_f : \sum_{S_s} Include(F_f, S_s) \geq 1$$

Number of constraints is: $\#File$

2. The sum of the size of files in each server must be equal to or less than server's capacity, that is,

$$\forall S_s : \sum_{F_f} Include(F_f, S_s) * Size(F_f) \leq Capacity(S_s)$$

Number of constraints is: $\#Server$

3. Each node should be accessed exactly once, that is,

$$\forall N_n : \sum_{S_s, T_t} Begin(N_n, S_s, T_t) = 1$$

$$\forall N_n : \sum_{S_s, T_t} End(N_n, S_s, T_t) = 1$$

Number of constraints is: $2 * \#Node$

4. If node N_n is in file F_f , file F_f is not stored in server S_s , node N_n can not be accessed from server S_s .

If F_f is not in server S_s , i.e., $Included(F_f, S_s) = 0$, node N_n can not be accessed from that server, i.e., $\sum_{T_t} Begin(N_n, S_s, T_t) = 0$ and $\sum_{T_t} End(N_n, S_s, T_t) = 0$.

So, both (2.1) and (2.2) hold.

If F_f is in server S_s , i.e., $Included(F_f, S_s) = 1$, node N_n might, but does not have to, be accessed from this server, i.e., $\sum_{T_t} Begin(N_n, S_s, T_t) \leq 1$ and $\sum_{T_t} End(N_n, S_s, T_t) \leq 1$. So, both (2.1) and (2.2) hold as well.

$\forall(N_n, S_s)$, where $Belong(N_n) = F_f$:

$$\sum_{T_t} Begin(N_n, S_s, T_t) \leq Include(F_f, S_s) \quad (2.1)$$

$$\sum_{T_t} End(N_n, S_s, T_t) \leq Include(F_f, S_s) \quad (2.2)$$

Number of constraints is: $2 * \#Node * \#Server$

5. The access time of each node is a function of the size of the node and the speed parameter of the server. It takes $AccessTime(N_n, S_s)$ for node N_n to be accessed from server S_s , that is,

$$\forall(N_n, S_s) : \sum_{T_{t1}} (End(N_n, S_s, T_{t1}) * T_{t1}) =$$

$$\sum_{T_{t2}} (Begin(N_n, S_s, T_{t2}) * T_{t2}) + \sum_{T_{t3}} (Begin(N_n, S_s, T_{t3}) * AccessTime(N_n, S_s))$$

Number of constraints is: $\#Node * \#Server$

6. The dependencies among those nodes are represented by the DAG. The edge from node N_i to node N_j means that node N_i is the predecessor of node N_j , i.e. node N_j must be accessed after N_i is finished, that is

$$\forall Edge(N_i, N_j) : \sum_{S_{sj}, T_{tj}} Begin(N_j, S_{sj}, T_{tj}) \geq \sum_{S_{si}, T_{ti}} End(N_i, S_{si}, T_{ti})$$

Number of constraints is: $\#Edge$

7. If two files corresponding to two DAG nodes (node N_i and node N_j) are on the same server, they can not be accessed simultaneously.

Situation 1: Node N_j is accessed before node N_i . If $X(N_i, N_j, S_s) = 1$, (2.3) holds. (2.4) holds if and only if node N_j ends its access before node N_i starts its access.

Situation 2: Node N_j is accessed after node N_i . If $X(N_i, N_j, S_s) = 0$, (2.4) holds. (2.3) holds if and only if node N_i ends its access before node N_j starts its access.

$$\forall(S_s, N_i, N_j), \quad \text{where } N_i \neq N_j :$$

$$\sum_{T_{ti}} \text{End}(N_i, S_s, T_{ti}) * T_{ti} \leq \sum_{T_{tj}} \text{Begin}(N_j, S_s, T_{tj}) * T_{tj} + \text{Max_step} * X(N_i, N_j, S_s) \quad (2.3)$$

$$\sum_{T_{ti}} \text{End}(N_j, S_s, T_{tj}) * T_{tj} \leq \sum_{T_{ti}} \text{Begin}(N_i, S_s, T_{ti}) * T_{ti} + \text{Max_step} * (1 - X(N_i, N_j, S_s)) \quad (2.4)$$

Number of constraints is: $2 * \#Node * (\#Node - 1) * \#Server$

Total constraints except for the objective is

$$(\#Node * \#Server) * (1 + 2 * \#Node) + \#File + \#Server + 2 * \#Node + \#Edge$$

2.4.5 Objective

To simplify the integer linear program, it is assumed that in the DAG there is a virtual sink node (This is not necessary in our heuristic algorithms). To minimize

total access time, find the earliest start time for the sink node N_{sink} . So, the objective is,

$$\min[\sum_{S_s, T_t} Begin(N_{sink}, S_s, T_t)]$$

2.4.6 Code in Lingo

LINGO is a comprehensive tool designed by LINDO Systems Inc. to make building and solving linear, nonlinear and integer optimization models faster, easier and more efficient. LINGO provides an integrated package that includes a language for expressing optimization models, and a full featured environment for building and editing problems. That is why we used LINGO to solve our problem. More information on optimization could be found in (Linus Schrage, 1998).

The basic linear program is made of a set of equations which express the constraints in the problem to be solved. Linear program in software packages provide methods compactly represent equations by expressions in a programming language form. Although LINGO is one of the leading software tools for building and solving optimization models, there were some limitations of the language. Because of this, we had to make our problem formulation much more complicated than it should have been.

The essential parts of our code in LINGO are included in appendix A.

2.4.7 Solutions Construction

Using such an integer linear program, we can get the optimal solution for our file allocation problem. Given the problem described in Chapter 2.2, we got the ILP solution in the following values of variables:

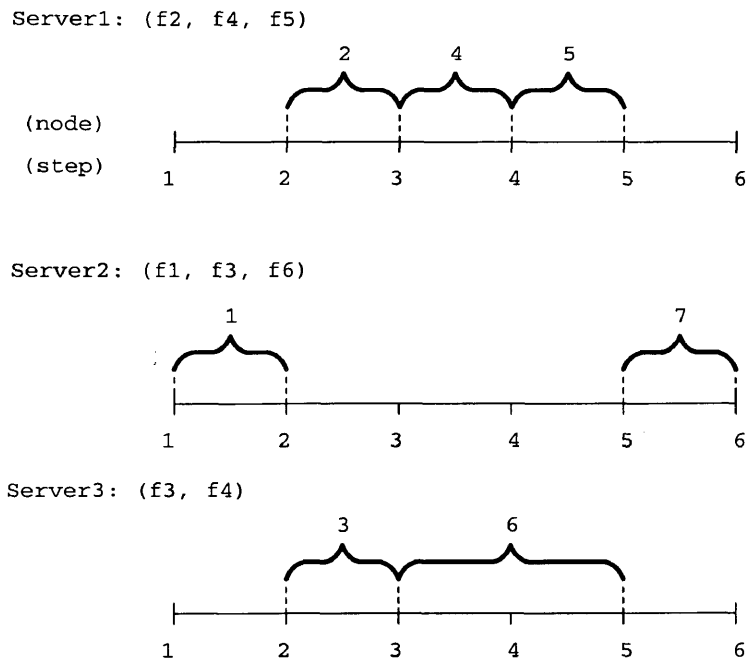


FIG. 2.3. The ILP Solution

```

INCLUDE( 1, 2)      1.000000
INCLUDE( 1, 4)      1.000000
INCLUDE( 1, 5)      1.000000
INCLUDE( 1, 7)      1.000000
INCLUDE( 2, 1)      1.000000
INCLUDE( 2, 3)      1.000000
INCLUDE( 2, 6)      1.000000
INCLUDE( 3, 3)      1.000000
INCLUDE( 3, 4)      1.000000

BEGIN( 1, 1, 2) 1.000000      END( 1, 2, 2) 1.000000
BEGIN( 2, 2, 1) 1.000000      END( 2, 3, 1) 1.000000
BEGIN( 3, 2, 3) 1.000000      END( 3, 3, 3) 1.000000

```

```

BEGIN( 4, 3, 1)  1.000000      END( 4, 4, 1)  1.000000
BEGIN( 5, 4, 1)  1.000000      END( 5, 5, 1)  1.000000
BEGIN( 6, 3, 3)  1.000000      END( 6, 5, 3)  1.000000
BEGIN( 7, 5, 2)  1.000000      END( 7, 6, 2)  1.000000

```

The solution is displayed in Figure 2.3.

2.5 Heuristic Algorithm

In the previous section, we introduced an ILP formulation for the optimal file allocation problem. ILP is the best way to find the optimal solution. Unfortunately, it is incredibly slow for a linear program to solve integer linear problems. So, using ILP, we may solve only unrealistic small models. Because of the intrinsic mobility of a mobile distributed system, everything keeps changing, and there is hardly any stability. Since it isn't practical to use ILP in a mobile system, an efficient algorithm is necessary. Here, we use a heuristic algorithm to deal with large-scale systems.

2.5.1 Definitions

Firstly, we need some definitions to describe our heuristic algorithm.

Candidates

A node N_n is a candidate node if its predecessors have all been finished.

A file F_f is a candidate file when $N_n \in F_f$.

A server S_s is a candidate server when no file is being accessed from it.

Level

In the heuristic algorithm, we calculate the "level" of each node as shown in Figure 2.4. Each node cannot begin its access until all of his predecessors have been finished. It is quite straightforward to see that the higher the node is, the higher

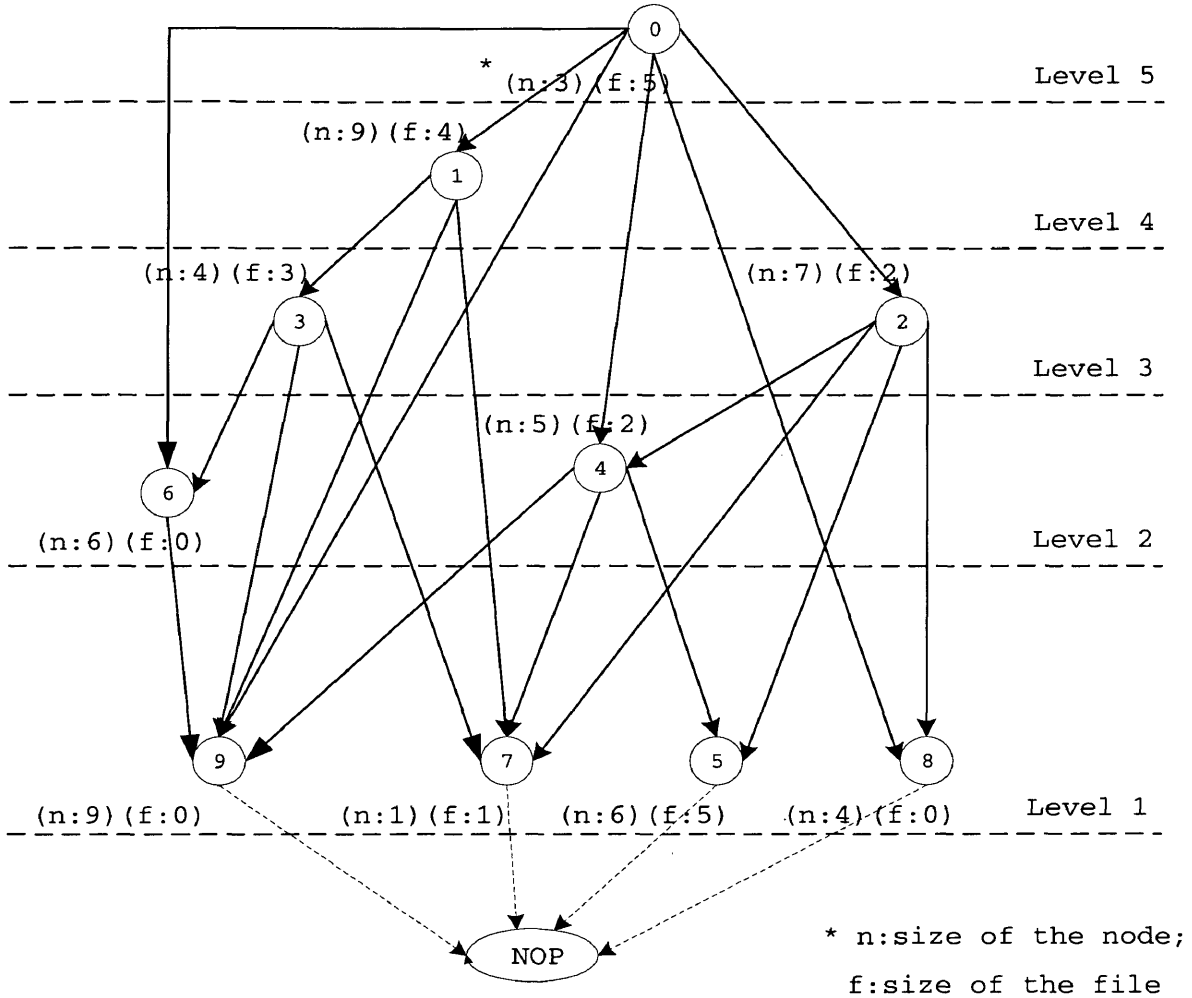


FIG. 2.4. Level Calculation

priority it should have. The calculation of the level is based on ALAP algorithm described in (Giovanni De Micheli, 1994).

We traverse the dependence graph in order to get the level of each node, that is

$$Level(N_n) = \begin{cases} 1 & \text{(if it is a leaf node)} \\ \max(N_n\text{-successor's_level}) + 1 & \text{(otherwise)} \end{cases} \quad (2.5)$$

Load

“Load” is another important value associated with each node. For two nodes at the same level, their priorities should be determined by the subsequent workload following them. In other words, one node which is at a higher level followed by a node of small size should have lower priority than another node which is at a lower level followed by a node of high size. We defined the equation of calculating the load of each node as

$$Load(N_n) = \begin{cases} Size(N_n) & \text{(if it is a leaf node)} \\ \sum(N_n\text{-successor}'s_load) + Size(N_n) & \text{(otherwise)} \end{cases} \quad (2.6)$$

Weight

In the heuristic algorithm, we sort the candidate list of nodes according to their weight. The node with higher weight has higher priority, i.e. the node with higher weight will be accessed earlier. The weight could be calculated in many methods based on different criteria. We used two equations, corresponding to two heuristic algorithms, to calculate the weight of each node.

The first equation is based on level of each node and the size of each file. The higher the node’s level, the higher its priority. The larger the node, the higher its priority. So, the equation we use is

$$Weight(N_n) = \alpha * Level(N_n) + (1 - \alpha) * Size(N_n); \quad 0 \leq \alpha \leq 1 \quad (2.7)$$

The second equation is based on the load of each node. Since the minimum load is 1, but the maximum load could be larger than the maximum integer of the system,

we use *scale* to adjust it. The scale might be 0.01, 0.001 or 0.0001...

$$Weight = Load(N_n) * scale \quad (2.8)$$

2.5.2 Algorithm

The pseudo-code of the heuristic algorithm is as follows:

BEGIN

Step = 0;

WHILE (there exist unscheduled nodes) **DO**

A = set of candidate nodes at this *step* sorted in decreasing order by *weight*;

B = set of candidate servers at this *step* sorted in decreasing order by *speed*;

WHILE (*A* and *B* are not empty) **DO**

Delete *N* = first node from *A*;

Let *F* = file that contains *N*;

Let *F* = first server from *B*;

IF ((*F* has already been assigned to some server $S_F \in B$) and

(there will be a capacity violation if *F* is assigned to another server))

access(*N*, *Step*, S_F);

delete S_F from *B*;

ELSE

WHILE (*F* is not in *S* and *S* doesn't have room for *F*) **DO**

S = next server in *B*;

ENDWHILE

IF (*F* is not in *S*)

put *F* into *S*;

```

    ENDIF
    access( $N$ ,  $Step$ ,  $S$ );
    delete  $S$  from  $B$ ;
  ENDIF
ENDWHILE
  Step ++;
ENDWHILE
END

```

The two different formulas for weight result in different priority orders for the nodes. This could result in different file access schedules. In Chapter 3, we will make a comparison between these two equations that we use in the heuristic algorithm.

2.6 Feasibility Lemma

2.6.1 Notation

The following notations are used in the equations below:

- ns : Number of proxy-servers;
- S_i : The i^{th} proxy-server;
- C_{S_i} : Capacity of proxy-server S_i
- nf : Number of files;
- F_i : The i^{th} file;
- $F_{S_i j}$: The j^{th} file in proxy-server S_i ;
- $Size_{F_i}$: Size of the i^{th} File;

2.6.2 A Small FAP With No Solution

It is not guaranteed that our heuristic algorithm gets the optimal solution for the problem. Furthermore, for our problem, given a set of tight constraints of the proxy server capacities, there might be no solution for a certain problem. The following FAP is an example.

Suppose there are five files, i.e, $nf = 5$; Their sizes are: $Size_{F_1} = 7$, $Size_{F_2} = 2$, $Size_{F_3} = 5$, $Size_{F_4} = 6$, $Size_{F_5} = 10$.

There are two servers, i.e, $ns = 2$. The capacities are: $C_{S_1} = 22$, $C_{S_2} = 16$.

$$\sum_{i=1}^{ns=2} C_{S_i} = 22 + 16 = 38$$

$$Size_{total} = \sum_{i=1}^{nf=5} Size_{F_i} = 7 + 2 + 5 + 6 + 10 = 30;$$

The total capacities are enough for all the nodes. But, there exists one case that F_1 and F_4 are in S_1 ; F_2 and F_3 are in S_2 . In this situation, there will be no space for F_5 .

Next, we propose a condition under which the problem has at least one solution, no matter how files are allocated to servers in a greedy strategy.

2.6.3 Lemma

Lemma

An algorithm that assign files to servers in any order will always result in all files being successfully assigned, if and only if

$$\sum_{i=1}^{ns} C_{S_i} > Size_{total} + (ns - 1) \cdot Size_{F_{max}} - ns, \text{ where } Size_{total} = \sum_{i=1}^{nf} Size_{F_i}; \quad (2.9)$$

Proof” \Rightarrow ”

Suppose at time T, there are X_1 files in server S_1 , X_2 files in server S_2 , and so on.

$F_{S_{11}}, F_{S_{12}}, \dots, F_{S_{1X_1}}$ are in server S_1 .

$F_{S_{21}}, F_{S_{22}}, \dots, F_{S_{2X_2}}$ are in server S_2 .

...

$F_{P_{ns1}}, F_{P_{ns2}}, \dots, F_{P_{nsX_{ns}}}$ are in server S_{ns} .

If there isn't enough space for F_f in any server, then:

$$C_{S_1} - \sum_{i=1}^{X_1} Size_{F_{S_{1i}}} \leq Size_{F_f} - 1$$

$$C_{S_2} - \sum_{i=1}^{X_2} Size_{F_{S_{2i}}} \leq Size_{F_f} - 1$$

\vdots

$$C_{S_n} - \sum_{i=1}^{X_{sn}} Size_{F_{S_{ni}}} \leq Size_{F_f} - 1$$

Suppose F_f is the last file, then:

$$\Rightarrow \sum_{i=1}^{ns} C_{S_i} - (Size_{total} - Size_{F_f}) \leq ns \cdot Size_{S_f} - ns, \text{ where } Size_{total} = \sum_{i=1}^{nf} Size_{F_i} \quad (2.10)$$

$$\Rightarrow \sum_{i=1}^{ns} C_{S_i} \leq (Size_{total} - Size_{F_f}) + ns \cdot Size_{F_f} - ns \quad (2.11)$$

$$\Rightarrow \sum_{i=1}^{ns} C_{S_i} \leq Size_{total} + (ns - 1) \cdot Size_{F_f} - ns \quad (2.12)$$

If $\forall F_f$, there must be enough space for it in some sever, there must be

$$\sum_{i=1}^{ns} C_{S_i} > Size_{total} + (ns - 1) \cdot Size_{F_f} - ns \quad (2.13)$$

The worst situation is that F_f is the last and the largest file, i.e.

$$Size_{F_f} = Size_{F_{max}}$$

The worst situation for the equation (2.13) is

$$\sum_{i=1}^{ns} C_{S_i} > Size_{total} + (ns - 1) \cdot Size_{F_{max}} - ns \quad (2.14)$$

” \Leftarrow ”

If equation (2.9) is not satisfied, then there might be no solution for this problem.

Suppose:

$$\sum_{i=1}^{ns} C_{S_i} > Size_{total} + (ns - 1) \cdot Size_{F_{max}} - ns \quad (2.15)$$

$$\Rightarrow \sum_{i=1}^{ns} C_{S_j} - (Size_{total} - Size_{F_{max}}) = ns \cdot Size_{F_{max}} - ns \quad (2.16)$$

\exists one case:

$$\begin{aligned} C_{S_1} - \sum_{i=1}^{X_1} C_{F_{S_1i}} &= Size_{F_{max}} - 1 \\ C_{S_2} - \sum_{i=1}^{X_2} C_{F_{S_2i}} &= Size_{F_{max}} - 1 \\ &\vdots \\ C_{S_{ns}} - \sum_{i=1}^{X_{ns}} C_{F_{S_{ns}i}} &= Size_{F_{max}} - 1 \end{aligned}$$

where:

$$F_{S_{11}}, F_{S_{12}}, \dots, F_{S_{1X_1}} \quad \text{are in server } S_1.$$

$$F_{S_{21}}, F_{S_{22}}, \dots, F_{S_{2X_2}} \quad \text{are in server } S_2.$$

...

$$F_{S_{ns1}}, F_{S_{ns2}}, \dots, F_{S_{nsX_{ns}}} \quad \text{are in server } S_{ns}.$$

Then there is no space for F_{max} in any server.

So, the lemma holds.

For the small FAP in Chapter 2.6.2, where

$$\sum_{i=1}^{ns=2} C_{S_i} = 22 + 16 = 38$$

$$Size_{total} = \sum_{i=1}^{nf=5} Size_{F_i} = 7 + 2 + 5 + 6 + 10 = 30;$$

$$Size_{total} + (ns - 1) \cdot Size_{F_{max}} - ns = 30 + (2 - 1) \cdot 10 - 2 = 38$$

$$\Rightarrow \sum_{i=1}^n C_{S_i} = Size_{total} + (ns - 1) \cdot Size_{F_{max}} - n$$

Under such conditions, it is not guaranteed that our heuristic algorithm has a solution for this small FAP. If the total capacity is one unit larger than current value, there must be at least one solution for it.

Chapter 3

TESTS AND RESULTS

In this chapter, we show the optimization results of ILP and of our heuristic algorithms for the FAP described in the last chapter and the comparison among those results.

3.1 ILP Solutions

ILP gives the optimal solution for our file allocation problem. but it takes an incredible long time to solve one small problem. So, we only used it for some small scale tests. Some results are as follows:

1. #Edge=12, #Server=3, #File=7, #Node=7, #Step=10
Time = 00:02:45; Iteration = 16495;
2. #Edge=16, #Server=3, #File=7, #Node=8, #Step=10
Time = 00:14:29; Iteration = 80596;
3. #Edge=16, #Server=4, #File=7, #Node=8, #Step=10
Time = 00:30:39; Iteration = 133164;
4. #Edge=12, #Server=4, #File=7, #Node=8, #Step=20
Time = 03:11:01; Iteration = 726652;

Time and *Iteration* are the output of our linear program. Notice that the time increases dramatically even when the number of the nodes or the number of servers increases by only one. When the number of steps increases, the time increases even

faster. An analysis of our ILP program resulted in the following equations for the number of variables and constraints:

$$\#Variable = \#Server * \#File + \#Node * \#Node * \#Server +$$

$$\#Node * \#Server * \#Step * 2$$

$$\#Constraints = (\#Node * \#Server) * (1 + 2 * \#Node) + \#File +$$

$$\#Server + 2 * \#Node + \#Edge + 1$$

The number of variables and constraints in the ILP program for the above tests respectively are as follow:

1. $\#Edge=12, \#Server=3, \#File=7, \#Node=7, \#Step=10$
 $\#Variable = 588; \quad \#Constraint = 345 ;$
2. $\#Edge=16, \#Server=3, \#File=7, \#Node=8, \#Step=10$
 $\#Variable = 693; \quad \#Constraint = 443 ;$
3. $\#Edge=16, \#Server=4, \#File=7, \#Node=8, \#Step=10$
 $\#Variable = 924; \quad \#Constraint = 580 ;$
4. $\#Edge=16, \#Server=4, \#File=7, \#Node=8, \#Step=20$
 $\#Variable = 1564; \quad \#Constraint = 580 ;$

When the number of Nodes increases by one, the number of variables and constraints increase by almost 100 each. When the number of servers, edges, and steps are larger, the increase would be extraordinarily large.

If Density of the DAG=0.3, $\#Server=30, \#File=50, \#Node=200, \#Step=500$
 Then $\#Variable = 7200150, \#Constraints = 2412481$

The increase in the number of variables and constraints is very rapid relative to the increase in the number of servers, files, nodes and steps. In another words, this linear programming method is not feasible for large scale problems.

3.2 Lower Bounds

We know that the heuristic algorithm doesn't guarantee the optimal solution, but we have to use some heuristic algorithms to deal with the large-scale problem quickly. We have to make sure that their results are close enough to the optimal solutions as well. The best way to assess a heuristic algorithm is to compare its results with the optimal solutions. However, finding out the optimal solution for a given problem is non-trivial. The practical way to do it is to find out the lower bound of a given problem and compare the results of the heuristic algorithms with the lower bound.

3.2.1 Lower Bound 1

The workload is accessing each node in the DAG, that is

$$W = \sum_{i=1}^{nn} Size_{N_i} \quad (3.1)$$

(where $Size_{N_i}$ is size of node N_i , nn is the number of nodes)

The resources available are proxy servers of different speeds. We normalize the proxy servers. We think of a server which is 5 times faster than the standard server as 5 standard servers. We think of a server which is 5 times slower than the standard server as 1/5 standard server. The sum of all these normalized server is the available

server resource, that is

$$R = \sum_{i=1}^{ns} \frac{Speed_i}{Speed_{standard}} \quad (3.2)$$

(where $Speed_i$ is speed of server S_i , $Speed_{standard}$ is the standard speed)

Even if there was no dependency among the nodes, i.e. all the file access operations could be done concurrently, this task could not be finished in the time less than workload divided by resources, that is

$$LB_1 = \frac{W}{R} \quad (3.3)$$

3.2.2 Lower Bound 2

The DAG is the most difficult point in our problem. The start time of each node must satisfy the original dependencies of the DAG, which limit the amount of parallelism of the file access operations, because any pair of operations related by a dependency (or by a chain of dependencies) may not execute concurrently. If node N_i depends on node N_j , node N_i can not be accessed until node N_j is finished. Furthermore, node N_i cannot be accessed until all its predecessors are finished. We can find out a longest chain of dependencies based on the size of each node. Along this longest chain, each node has to wait until its adjacent predecessor has been finished, therefore, the whole task can not be done until the last node in the longest chain has been finished, that is

$$LB_2 = \frac{\sum_{i=1}^k Size_{N_i}}{Speed_{standard}} \quad (3.4)$$

(where N_i are nodes in the longest chain, $Speed_{standard}$ is the standard speed, k is the number of nodes in the longest chain).

In Lower Bound 1, we don't take the DAG into account. It is the dependency

graph that makes this file allocation problem complicated. Dependencies determine the concurrency of the resulting implementation, and therefore it affects the performance of the whole system. Because of this, the second lower bound is always closer to the optimal solution than the first lower bound. On the other hand, it shows the effect of the DAG in our file allocation problem. We will see this point in the next section.

3.3 Results

3.3.1 Parameters

We have identified the following parameters that affect the performance of the heuristic algorithms:

- MS: Max Size. The size of node is randomly selected from 1 to Max Size.
- MT: Max Type. The type of servers is randomly set from 1 to Max Type.
- UC: Unit Capacity. The capacity of the server is $UC * ServerType$.
- NS: Number of Servers.
- NF: Number of Files.
- NN: Number of Nodes.
- DS: Density. The number of edges in the DAG is

$$Density * \frac{NN * (NN - 1)}{2}; \quad (3.5)$$

MS, MT and UC are fixed in the following simulations. We will see the effect of DS, NS, NF and NN. We try to use those parameters which make the whole system relatively stable. We will give detailed explanations in each of the following sections.

In algorithm 1, we use equation (2.7). In algorithm 2, we use equation (2.8). The first lower bound is calculated with equation (3.3). The second lower bound is calculated with equation (3.4).

3.3.2 Effect of Density

The density of the dependency graph is one of the key parameters which affect the results the file allocation problem. Figure 3.1 plots the results with different density from 0.1 to 1 and compares them with the two lower bounds. Other parameters are as follows.

- Max Size: 10 units
- Max Type: 5 types
- Unit Capacity: 50 units
- Number of Servers: 30 servers
- Number of Files: 50 files
- Number of Nodes: 200 nodes

The higher the density, the more dependencies there are. When the density of the graph increases, the number of predecessors of each node increases. A node has to wait for all of its predecessors to finish their accesses. So, the time it takes increases with the density. when the density is 1, there is almost no concurrency for

density	first lower bound	second lower bound	result_1	result_2
0.1	72	219	284	350
0.2	72	293	429	480
0.3	72	425	581	628
0.4	72	495	680	713
0.5	72	627	767	791
0.6	72	719	814	839
0.7	72	789	890	868
0.8	72	854	919	932
0.9	72	959	988	989
1	72	1058	1059	1060

Table 3.1. Time Comparison for Different Value of Density

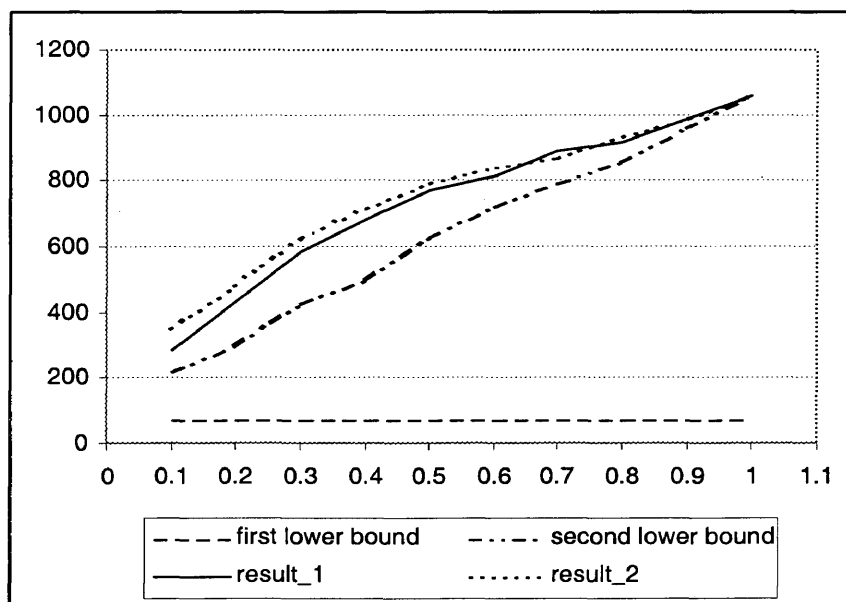


FIG. 3.1. Time Comparison for Different Value of Density

the file access operations, therefore, there is not any difference between the heuristic algorithm and the second lower bound which is calculated based on the longest chain of the DAG. Since the DAG and the size of each node are not changed, the first lower bound is not changed either. We see that, the density and the units of time have a linear relationship, we will use $\text{density} = 0.3$ in the remaining experiment.

3.3.3 Effect of the Number of Servers

Figure 3.2 plots the results with different number of servers from 20 to 100. Other parameters are as follows.

- Max Size: 10 units
- Max Type: 5 types
- Unit Capacity: 50 units
- Number of Files: 50 files
- Number of Nodes: 200 nodes
- Density: 0.3

Since the capacity of the server is created randomly in our heuristic algorithms, an increase of the number of servers result in an increase of the total available capacity. The more servers we have, the more concurrency we would get. That is why the units of time needed decrease with an increase in the number of servers. When the total capacities of server are large enough, the units of time spent in finishing the file accesses should converge. In the plot, we can find a point after which the result doesn't change. The reason is that when the capacities of the server are not large

Server	first lower bound	second lower bound	algorithm 1	algorithm 2
20	108	415	599	658
25	93	381	663	651
30	72	425	581	628
35	62	439	502	519
40	54	403	483	491
45	51	374	462	462
50	45	375	415	419
60	38	401	459	459
70	30	398	450	450
80	25	383	438	438
90	22	404	454	454
100	20	401	448	448

Table 3.2. Time Comparison for Different Number of Servers

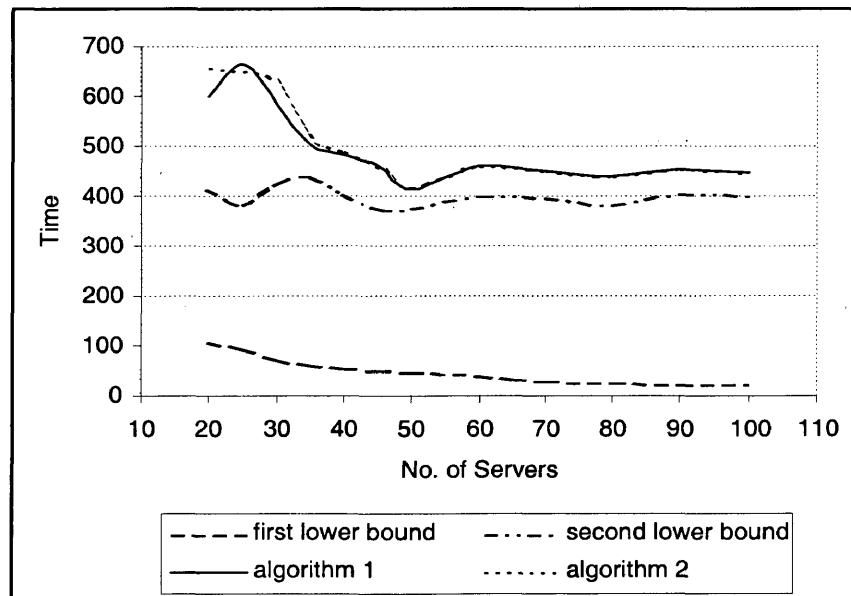


FIG. 3.2. Time Comparison for Different Number of Servers

enough for the files to have enough copies in the whole system, there might be a situation where a candidate node has to wait because the server in which the file is stored is being accessed by another node. It shows that our algorithms work better with more servers. The results and the lower bound become very closer to each other between 30 servers and 40 servers, so we use 30 servers for other tests and comparisons.

3.3.4 Effect of the Number of Files

Figure 3.3 plots the results with different number of files from 40 files to 100 files. Other parameters are as follows.

- Max Size: 10 units
- Max Type: 5 types
- Unit Capacity: 50 units
- Number of Servers: 30 servers
- Number of Nodes: 200 nodes
- Density: 0.3

Since the workload is related to nodes and the available capacity is related to servers, changing the number of files doesn't have any effect on the first lower bound. The second lower bound does have some oscillation, but the trend of it is increasing. The more files in the DAG, the larger the total capacity needed to ensure that the problem has a possible solution. With the same number of servers, more files implies that there are fewer copies of each file resulting in less concurrency. We choose to use 50 files in the remaining simulations.

File	first lower bound	second lower bound	result_1	result_2
40	68	381	472	538
45	68	339	487	521
50	72	425	581	628
55	68	371	522	557
60	69	439	532	555
65	71	461	591	608
70	68	403	610	579
75	68	407	561	596
80	72	374	551	581
85	68	476	610	619
90	68	375	600	659
95	72	407	609	622
100	68	452	599	645

Table 3.3. Time Comparison for Different Number of Files

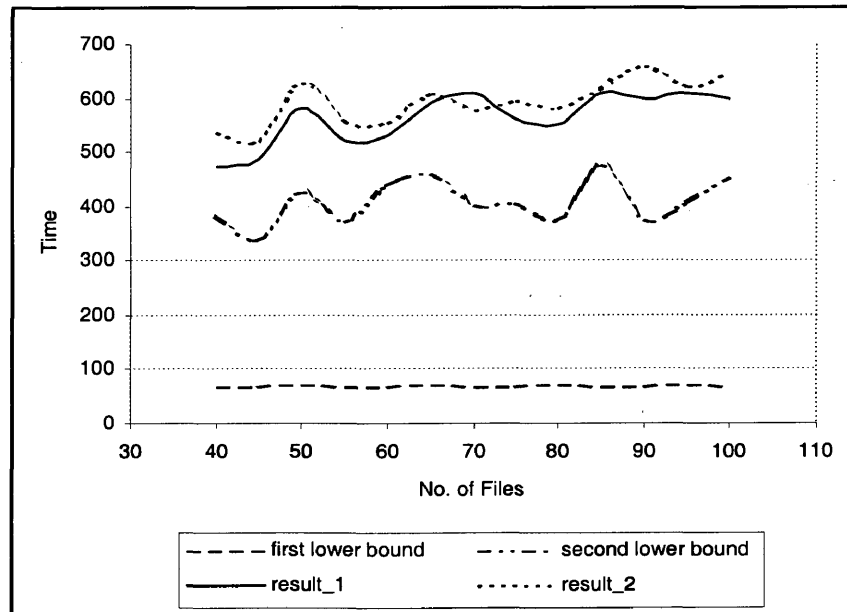


FIG. 3.3. Time Comparison for Different Number of Files

Node	first lower bound	second lower bound	result_1	result_2
50	17	120	130	130
70	24	165	188	188
80	27	128	166	174
90	30	196	242	241
100	34	178	224	216
110	38	188	265	256
120	42	253	302	304
150	53	349	452	463
200	72	425	581	628
220	79	454	644	668
250	89	535	757	756

Table 3.4. Time Comparison for Different Number of Nodes

3.3.5 Effect of the Number of Nodes

Figure 3.4 plots the results with different number of nodes from 50 files to 250 nodes. Other parameters are as follows.

- Max Size: 10 units
- Max Type: 5 types
- Unit Capacity: 50 units
- Number of Servers: 30 servers
- Number of Files: 50 files
- Density: 0.3

The size of a node is created randomly. Increasing the number of node does increase the total workload. That is why the first lower bound increases with the number of nodes. We noticed that the second lower bound increased even faster. The more nodes in the DAG, the more dependencies they would have, the less concurrency

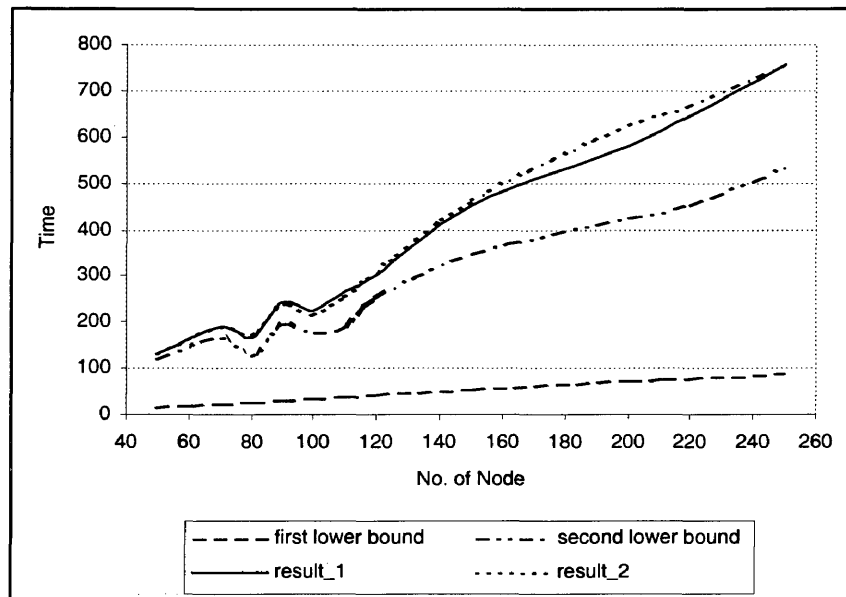


FIG. 3.4. Time Comparison for Different Number of Nodes

we would get, therefore, the more units of time would be taken to finish all the access operations. When the number of nodes is more than 200, results of our algorithms are almost parallel to the second lower bound, so we use 200 nodes in other simulations.

3.3.6 Random Assignment

If we just randomly assign files to servers instead of using our heuristic algorithm, the results are shown in Table 3.5.

3.4 Conclusion

For the problems as small as the one shown in Chapter 2.2, the solutions given by our heuristic algorithms are almost as good as the optimal solutions given by integer linear programming. But for large scale problems, we can not compare the results of our algorithm with the optimal solutions, because it takes incredibly large amount

parameters		result_1	result_2	ramdom
NS: 30	DS: 0.3	581	629	726
NF: 50	DS: 0.6	719	814	852
NN: 200	DS: 0.9	959	988	1013
DS: 0.3	NS: 40	483	491	724
NF: 50	NS: 60	459	459	730
NN: 200	NS: 80	438	438	679
DS: 0.3	NF: 40	473	539	721
NS: 30	NF: 70	611	580	632
NN: 200	NF: 90	601	660	509
DS: 0.3	NN:70	189	189	249
NS: 30	NN:100	225	217	318
NF: 50	NN:130	317	336	454

Table 3.5. Time Comparison with Random Assignment

of time for linear program to get the optimal solutions. In the sections above, we compared the results of our heuristic algorithms and the theoretical lower bounds. Our algorithms don't give the optimal solutions for the large scale problems, but our results are very close to the theoretical lower bound. The largest difference between our algorithms and the second lower bound is less than 30 percent. And our heuristic algorithms can solve large scale FAP problems fast and efficiently. So far, it is safe for us to draw the conclusion that these heuristic algorithms are eligible to be used for file allocation in mobile distributed systems.

Chapter 4

SUMMARY AND FUTURE WORK

In our research, we have optimized a DAG-based File Allocation Problem(FAP), such that the total access time is minimized in mobile distributed system. We have formulated this problem as an integer linear program. We also use two heuristic algorithms for the optimization. As a result of comparison with the theoretical lower bounds, the heuristic algorithms are found to be very useful. They may deal with more practical large scale systems fast and efficiently.

In the future, the following issues should be considered:

Firstly, in our current FAP model, we don't take into account the communication ports of the mobile unit. The parallelism is only limited by the dependencies in the DAG. In future work, the model should think about communication ports of the mobile unit. The number of simultaneous accesses does not exceed the number of communication ports of mobile unit.

Secondly, in our current FAP model, we assume that the source of the mobile unit is unlimited. Actually, the mobile unit has a memory capacity limitation. The number of simultaneous accesses do not transfer more data than the mobile unit can store. Furthermore, queuing delay has to be considered.

The current research is at a theoretical level. It could be incorporated with the existing operating systems, file systems and protocols used in mobile distributed systems.

References

- Chu, W. Oct. 1969. Optimal File Allocation in a Computer Network. *IEEE Trans. on Computer*, **C-18**(10), 865–889.
- Deb Ghosh, Ishwar Murthy. 1991. A Solution Procedure for the File Allocation Problem with File Availability and Response Time. *Computers Operations Research*, **18**(6), 557–568.
- Giovanni De Micheli. 1994. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill Inc.
- Hui Lei, Dan Duchamp. Jan. 1997. *Hui Lei, Dan Duchamp*. Tech. rept. USENIX Annual Technical Conference, Anaheim CA.
- Johnson, Michael R. Garey / David S. 1979. *Computers and Intractability – A Guide to the theory of NP-completeness*. W & H Freeman & Company.
- Jorg Jensch, Reinhard Luling, Norbert Sensen. 1998. Optimising the File Allocation of Parallel Web Server Online Using Access Patterns. *IEEE Trans. on Computer*, **X**, 66–74.
- Lawrence W. Dowdy, Derrell V. Foster. June 1982. Comparative Models of the File Assignment Problem. *Computing Surveys*, **14**(2), 95–120.
- Lin-Wen Lee. 2000. File Assignment in Parallel I/O Systems with Minimal Variance of Service Time. *IEEE Transactions on Computers*, **49**(2), 127–140.
- Linus Schrage. 1998. *Optimization Modeling with LINGO (Second Edition)*. Lindo System Inc.

- M. Satyanarayanan. 1998. *Fundamental Challenges in Mobile Computing*. Tech. rept. Carnegie Mellon University.
- Mahadev Satyanarayanan. February 1996. Mobile Information Access. *IEEE Personal Communications*, 26–33.
- P.Scheuermann, G. Weikum, P.Zabback. 1998. Data Partitioning and Load Balancing in Parallel disk Systems. *VLDB J.*, **7**(1), 48–66.
- Rahul Simha, B. Narahari, Hyeong-Ah Chio, Li-Chuan Chen. 1996. File Allocation for a Parallel Webserver. *IEEE Trans. on Computer*, **8**, 16–21.
- S. March, S.Rho. Mar./Apr. 1995. Allocation Data and Operations to Nodes in Distributed Database Design. *IEEE Trans. Knowledge and Data Eng.*, **7**(2), 305–317.
- Wentong Cai, Bu-Sung Lee. 1998. File Allocation with Balanced Response Time in a Distributed Multi-server Information System. *Information and Software Technology*, **40**, 27–35.

Appendix A

A.1 Code in Lingo

MODEL:

Title File Allocation/Access;

SETS:

Server /1..Number_of_Server/: Capacity;

File /1..Number_of_File/: Size;

Save (File, Server) : Include;

Node /1..Number_of_Node/ : Belong;

Graph (Node, Node) : Edge;

Overlap (Node, Node, Server) : X;

Access(Node, Server) : AccessTime;

Step /1..Max_Step/ ;

Operation(Node, Server, Step) : Begin, End;

ENDSETS

DATA:

...

ENDDATA

MIN = @SUM(Operation(N_sink,S_s,T_t) | N_sinc = the sink node:
 Begin(N_sinc, S_s, T_t) * T_t);

```
@FOR(Save: @BIN(Include));
```

```
@For(Operation: @Bin(Begin));
```

```
@For(Operation: @Bin(End));
```

```
@For(Overlap: @BIN(X));
```

```
@For(File(F_f) :
```

```
  @SUM(Server(S_s): Include(F_f,S_s)) >= 1);
```

```
@For(Server(S_s) :
```

```
  @SUM(File(F_f): Include(F_f,S_s) * Size(F_f)) <= Capacity(S_s));
```

```
@For(Node(N_n) :
```

```
  @SUM(Server(S_s): @SUM(Step(T_t): Begin(N_n, S_s, T_t))) = 1 );
```

```
@For(Node(N_n) :
```

```
  @For(File(F_f) | Belong(N_n) #EQ# F_f:
```

```
    @For(Server(S_s) :
```

```
      @SUM(Step(T_t): Begin(N_n, S_s, T_t)) <= Include(F_f, S_s) )));
```

```
@For(Node(N_n) :
```

```
  @For(File(F_f) | Belong(N_n) #EQ# F_f :
```

```
    @For(Server(S_s) :
```

```
      @SUM(Step(T_t): End(N_n,S_s,T_t)) <= Include(F_f,S_s) )));
```

```
@For(Node(N_n) :
```

```

@For(Server(S_s):
  @SUM(Step(T_t1): End(N_n, S_s, T_t1) * T_t1) =
  @SUM(Step(T_t2): Begin(N_n,S_s,T_t2) * T_t2) +
  @SUM(Step(T_t3): Begin(N_n,S_s,T_t3) * AccessTime(N_n,S_s) ));

```

```

@For(Node(N_ni):
  @For(Node(N_nj) | Edge(N_ni,N_nj) #EQ# 1 :
    @SUM(Server(S_sj):@SUM(Step(T_tj):Begin(N_nj,S_sj,T_tj) * T_tj))>=
    @SUM(Server(S_si):@SUM(Step(T_ti):End(N_ni,S_si,T_ti) * T_ti))));

```

```

@For(Server(S_s):
  @For(Node(N_ni):
    @For(Node(N_nj) | N_ni #NE# N_nj:
      @SUM(Step(T_ti): End(N_ni,S_s,T_ti) * T_ti) <=
      @SUM(Step(T_tj): Begin(N_nj,S_s,T_tj) * T_tj) +
      Max_step * X(N_ni,N_nj,S_s) ));

```

```

@For(Server(S_s):
  @For(Node(N_ni):
    @For(Node(N_nj) | N_ni #NE# N_nj:
      @SUM(Step(T_tj): End(N_nj,S_s,T_tj) * T_tj) <=
      @SUM(Step(T_ti): Begin(N_ni,S_s,T_ti) * T_ti) +
      Max_step * (1 - X(N_ni,N_nj,S_s)) ));

```

END