

Appointed File Prefetching for Distributed File Systems

Gwan-Hwan Hwang[†], Hsin-Fu Lin[†], Chun-Chin Sy[‡], and Chiu-Yang Chang[‡]

[†]Department of Information and Computer Education, National Taiwan Normal University, Taipei, Taiwan

[‡]Department of Electronic Engineering, National United University, Miao-Li, Taiwan

Abstract

One of key issues in the design of distributed file systems is how to reduce the latency when accessing remote files, with the solutions including cache replacement and file-prefetching technologies. In this paper, we propose a novel method called appointed file prefetching, in which the main idea is to enable the user or system administrator to specify how to perform file prefetching. We define the appointed file-prefetching language (AFPL) that the user and system administrator can use to instruct the system to perform desired prefetching at appropriate times. The prefetching instructions in the AFPL can be divided into two categories: (1) selecting the required files and (2) specifying when to perform prefetching. According to the information in the file system and the previous file access records, a user can select the needed files as candidates for prefetching and then choose between time prefetching and event prefetching to retrieve them. The experimental results show that the waiting time of remote file fetching is reduced by 30% to 90% and the hit ratio is increased by 6% to 18% in most cases.

Keywords: Distributed File System, File Prefetching, Thin-Client/Server Computing, Appointed File Prefetching

1 Introduction

The thin-client/server (TC/S) computing model is rapidly becoming more widespread due to its low cost and the rapid deployment of applications running at the server side, or so-called server-based computing [1]. Server-based computing gives corporations more control over applications by managing them in the server infrastructure instead of at the desktops. The multiuser TC/S computing model takes this one stage further, by mandating that applications run solely on a server: client devices merely monitor inputs from mice and keyboards, pass them to the server, and wait for the displays returned by the server. The TC/S computing model consists of three key components: (1) thin-client hardware devices, (2) the application server, and (3) a display protocol. All the applications and data are deployed, managed, and supported at the application server. In addition, applications are executed only on the server. Thin-client devices gather inputs from users in the form of mouse clicks and

keystrokes, send them to the application server for processing, and collect screen updates as the response from the application server.

The TC/S computing model also benefits end users by providing them with a transparent working environment irrespective of the types of client devices they use and where they use them. The traditional way of implementing a transparent working environment is using a single-application-server topology in which each client device always connects to the same application server, and that all the data and application software of the client are stored on this server. However, such a topology restricts the user to roaming within a restricted area (the local area network, or LAN) close to the application server within which the efficient transfer of mouse clicks, keyboard inputs, and screen updates via a display protocol is possible. The display updates require far more bandwidth than the user mouse and keyboard inputs, and so it is the technology used for the display updates that restricts the area of the network.

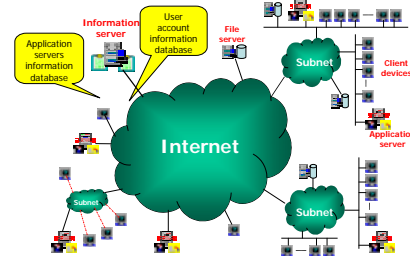


Figure 1: The MAS TC/S model

We have previously [8] proposed a multiple-application-server (MAS) architecture model for the TC/S computing model (see Figure 1). In this model, multiple application servers are installed over a wide area network (WAN), and each client device can freely connect to any application server that is close to it. A user can log into any application server via a thin-client device, and thus an application server needs to prepare the data for each user so as to provide a fast service. A trivial solution is to replicate the data and application software for each user on all the application servers, but this is usually too expensive. In addition to the huge disk-space wastage, synchronizing a user's data between the application servers may consume much of the available network bandwidth. Note that we assume that the application software is fully duplicated on all the application

* G. H. Hwang's work were supported in part by the ROC NSC under grant 94-2213-E-003-006-.

servers, since the application software is usually shared among users and must be installed and set up properly beforehand. However, the data of each user are stored only in a subset of servers. Before providing services to a user, an application server must fetch any absent files that are required by the services. We have achieved this by designing a distributed file system for the MAS TC/S model that includes the functionality of traditional distributed file systems plus some enhancements, including an automatic prefetching mechanism and an appointed file-prefetching mechanism.

Automatic prefetching is achieved by the application server predicting which files are going to be accessed by a user, and then prefetching those files in parallel with the user's work. Automatic prefetching uses the past file access records to predict the future file system requests. The objective is to provide data in advance of the request for it, effectively removing access latencies [3-8]. According to previous experimental results [3-8], the performance of automatic prefetching can be slightly better than that of traditional demand prefetching as employed in distributed file systems such as the Sun network file system (NFS) [9] and the Andrew file system (AFS) [10]. However, because these schemes can only derive an approximation prediction, some of the files prefetched will not be accessed by the user, which increases the network traffic without improving the performance. In this paper, we propose the use of an appointed file-prefetching mechanism for distributed file systems, in which the main idea is to enable the user or system administrator to specify how the system performs the prefetching. We define the appointed file-prefetching language (AFPL), which is based on Java [11]. A user or system administrator can write AFPL programs to instruct the system to prefetch the required files from the file server. Two prefetching methods can be specified in the AFPL: time prefetching and event prefetching. In time prefetching, the system prefetches the selected files at specific times, whereas in event prefetching the prefetching is performed when specific events occur. Both methods can access the past file access records as well as the schedule of the user to specify the prefetching.

The remainder of the paper is organized as follows. Section 2 discusses the operational model of appointed file prefetching, Section 3 presents the usage syntax for the AFPL, Section 4 presents the implementation of the method and experimental results therefrom, and Section 5 presents the conclusions that can be drawn from this paper.

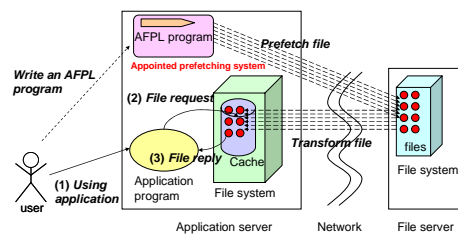


Figure 2: The operational model of appointed file prefetching

2 The Operational Model of Appointed File Prefetching

The goal of appointed file prefetching is to provide a mechanism for the user to specify how to prefetch files in specific situations. First of all, we present the demand fetching mechanism employed in many distributed file systems such as NFS and AFS. The file access includes the following steps:

- Step 1. The application program is executed by the user.
- Step 2. The application program issues a file operation via system calls.
- Step 3. The file system in the application server checks if the requested file is stored in its local file cache. If the file is not present, it sends the file request to the file server.
- Step 4. The file server transmits the requested file to the local file cache of the application server.
- Step 5. Finally, the application program can access the requested file.

It is obvious that performing Steps 3 and 4 may be time consuming as they usually involve data transmission in the network, which usually results in a noticeable delay in file access. As we mentioned above, some researchers have proposed the use of automatic prefetching to solve this problem [3-8]. Here we propose a new approach for solving this problem, called appointed file prefetching (see Figure 2). The basic file access with the present implementation of appointed file prefetching is similar to that of demand fetching. However, the user uses the AFPL program to instruct the appointed file-prefetching daemon in advance so that the needed files could be fetched into the cache of the application server prior to the actual file access.

To perform time and event prefetching, the system requires some information (not including the AFPL program) provided by the user. shows the data flow during the operation of the appointed file prefetching. The following information has to be sent to the appointed file-prefetching activator:

- The past file access records of the user: The operating system of the application server must

to an application server. He wants the system to prefetch some conference files to that application server during the conference.

Task 7. When he executes the application software “kword”, he wants the system to prefetch all the files in his directory “~/doc”. Note that the “KWord” is an application software of Linux [13].

Task 8. When he logs into the application server of the coffee shop in his campus, he wants the system to prefetch all the files in the directory “~/music”. Then, he can enjoy his favorite music without waiting for the network transmission.

Task 9. He wants to set up appointed file prefetching according to his timetable.

We now present the AFPL. For portability and convenience of implementation, the AFPL is based on Java and has several extended classes (Figure 5).

The “FilePro” class is an extension from Java class “File” that records the information of an accessed file, the “TimePro” class is used to store time and date, and the “FileVector” class is an extension of “Vector” class of Java that has the following methods added:

- `addFromDir(String sourceDir)`: This adds the files in the directory “*sourceDir*” to this vector, where “*sourceDir*” is the pathname of the directory.
- `addFromPFAR(String sourcePFAR)`: This adds the accessed files recorded in the past file access records, where “*sourcePFAR*” is the pathname of the past file access records.
- `Prefetch()`: This instructs the system to immediately prefetch all the files stored in this vector.
- `timePrefetch(String serverIP, TimePro startTime, int type, int priority)`: This specifies when to perform a prefetch. Note that “*type=0*” means that prefetching should be performed once at the time “*startTime*”, whereas “*type=1*” and “*type=2*” indicate that prefetching is performed every day and every week, respectively. “*Priority*” specifies the priority of the prefetching in the case where two prefetching requests are scheduled to be performed at the same time (a larger number means a higher priority).
- `eventPrefetch(int type, String`

description, int *minutes*, int *priority*): This specifies that the system is to prefetch after an event occurs after “*minutes*” minutes according to the type of events:

- ◆ *type=1*: when the user logs into the system.
- ◆ *type=2*: when the user logs into a specified application server, where “*description*” is the name of the application server.
- ◆ *type=3*: when the user executes application software, where “*description*” is the name of the application software.
- ◆ *type=4*: when the user accesses a file, where “*description*” is the name of the target file.

```
public class FilePro extends File {
    public String getName(); // file name
    public String getPath(); // file path
    public long length(); // file length
    public long lastModified(); //the last modified time of file
    long LastAccessTime; //the last accessed time of file
    long LastLoginTime; //the last login time of file
    String LastApplication; //the last application which use file
    String LastServer; //the last server IP which use file
    int LastSession; // the last session which use file
    int AccessFrequency; // the frequency of accessing file
    :
    .
}

public class TimePro implements Serializable {
    int year;
    int month;
    int date;
    int day;
    int hour;
    int minute;
}

public class FileVector extends Vector {
    public void addFromDir(String sourceDir);
    public void addFromPFAR(String sourcePFAR);
    public void prefetch();
    public void timePrefetch(String serverIP, TimePro startTime, int type, int priority);
    public void timePrefetch(String serverIP, TimePro startTime, TimePro endTime, int timeSlot, int priority);
    public void eventPrefetch(int type, String description, int minutes, int priority);
    :
    .
}
```

Figure 5: The prototypes of the added classes in the AFPL

Figure 8 shows how to write the AFPL to specify the appointed file prefetching of Tasks 1 to 9 listed above.

```
// Task 1
FileVector fs1=new FileVector(), fs2=new FileVector();
fs1.addFromDir("~/paper");
fs2.addFromDir("~/course");
fs1.add(fs2);
TimePro startTime = new TimePro(0, 0, 0, 0, 20, 0);
fs1.timePrefetch("dormitory.XUniv.edu", startTime, 0, 5);

// Task 2
```

<pre>FileVector fs1=new FileVector(),fsTemp = new FileVector(); fsTemp.addFromDir("~/course"); ListIterator it = fsTemp.listIterator(); Object tempObj; FilePro tempFile; TimePro startTime = new TimePro(0, 0, 0, 4, 14, 0); TimePro endTime = new TimePro(0, 0, 0, 4, 16, 0); while (it.hasNext()) { tempObj = it.next(); tempFile = (FilePro)tempObj; if (tempFile.getName.endsWith("ppt")) { fs1.add(tempFile); } } fs1.timePrefetch("classroom.XUniv.edu", startTime, endTime, 20, 5);</pre>	
<pre>// Task 3 FileVector fs1 = new FileVector(), fsTemp = new FileVector(); fsTemp.addFromPFAR("~/file_access_record"); ListIterator it = fsTemp.listIterator(); Object tempObj; FilePro tempFile; Calendar showDate = Calendar.getInstance(); while (it.hasNext()) { tempObj = it.next(); tempFile = (FilePro)tempObj; if ((showDate.getTimeInMillis() - tempFile.getLastAccessTime().getTimeInMillis()) <= (86400000*3)) { fs1.add(tempFile); } } fs1.eventPrefetch(1, "login", 0, 1);</pre>	<pre>// Task 8 FileVector fs1 = new FileVector(); fs1.addFromDir("/afs/abc.afs/usr/dr_williams/music"); fs1.eventPrefetch(2, "coffee", 3, 5);</pre>
<pre>// Task 4 FileVector fs1 = new FileVector(), fsTemp = new FileVector(); fsTemp.addFromPFAR("/afs/abc.afs/usr/dr_williams/file_access_record"); ListIterator it = fsTemp.listIterator(); Object tempObj; FilePro tempFile; Calendar showDate = Calendar.getInstance(); while (it.hasNext()) { tempObj = it.next(); tempFile = (FilePro)tempObj; if (((showDate.getTimeInMillis() - tempFile.getLastModifyTime().getTimeInMillis()) <= (86400000*7)) &&& (tempFile.length() >= (1*1024*1024))) { fs1.add(tempFile); } } fs1.eventPrefetch(1, "login", 0, 1);</pre>	<pre>// Task 9 FileVector fs1 = new FileVector(); fs1.addFromDir("/afs/abc.afs/usr/dr_williams/course"); DomParser tempSchedule = new DomParser("/afs/abc.afs/usr/dr_williams/activities_1.xml")³; ListIterator it = tempSchedule.listIterator(); while (it.hasNext()) { Object tempObj = it.next(); Activity tempAct = (Activity)tempObj; if (tempAct.getType() == 2) { prefetchFS.timePrefetch(tempAct.getLocation(), tempAct.getStartTime(), tempAct.getEndTime(), tempAct.getType(), tempAct.getPriority()); } }</pre>
<pre>// Task 5 FileVector fs1 = new FileVector(), fsTemp = new FileVector(); fsTemp.addFromPFAR("/afs/abc.afs/usr/dr_williams/file_access_record"); ListIterator it = fsTemp.listIterator(); Object tempObj; FilePro tempFile; while (it.hasNext()) { tempObj = it.next(); tempFile = (FilePro)tempObj; if ((tempFile.getLastAccessTime().getTimeInMillis() - tempFile.getLastLoginTime().getTimeInMillis()) <= (60000*5)) { fs1.add(tempFile); } } fs1.eventPrefetch(1, "login", 0, 1);</pre>	<p style="text-align: center;">Figure 6: Examples of the AFPL</p>
<pre>// Task 6 FileVector fs1 = new FileVector(); fs1.addFromDir("/afs/abc.afs/user1/conference"); TimePro startTime = new TimePro(2005, 9, 26, 1, 8, 30); TimePro endTime = new TimePro(2005, 9, 28, 3, 21, 30); fs1.timePrefetch("conferenceroom.XUniv.edu", startTime, endTime, 20, 2); // Task 7</pre>	<h3>4 Implementation and Experimental Results</h3> <p>It is obvious that the appointed file prefetching can improve the efficiency of file access if the user can write an appropriate APFL program to instruct the system to perform the prefetching correctly. We implement the system according to the architecture shown in . The implementation work includes the follows:</p> <ul style="list-style-type: none"> • The Java API shown in Figure 5. • The system-call interceptor of Linux for collecting the past file access records: We modify the Linux kernel, the details of which can be found elsewhere [8]. • Event trigger: This is a C program that filters the real-time file access records from the kernel. • Appointed file-prefetching activator: This is a Java program that executes the AFPL program written by the user. • Appointed file-prefetching daemon: This is a C program that performs file access according to messages from the appointed file-prefetching activator. <p>To demonstrate the feasibility of the system, we performed a simulation that allowed some data to be measured that cannot be obtained in a real system. First of all, we define the network environment of our simulation. The network performance parameters of primary interest for our purposes are those affecting the speed at which individual messages can be transferred between two interconnected computers: the latency and the point-to-point data transfer rate. According to [12], the time required for a network to transfer a message between two computers is</p> <hr/> <p>³ The details of how to instruct the AFPL according to the schedule of the use are available elsewhere [].</p>

approximately

$$\text{Message transmission time} = \text{latency} + \text{message size}/\text{data transfer rate}.$$

We assume that the average latencies in a LAN and a WAN are 100 and 3 ms, respectively. We have a Linux machine running the KOffice program [13]. We performed our simulation using real data obtained from the users' past file access records. The file cache replace policy employs the least-recently-used algorithm [14]. The following items are the measured data:

- Total waiting time for a remote file fetch (in milliseconds): This is the total time required for transmitting files between an application server and a file server during a session. A session starts when the user logs on and ends when the user logs out. If the accessed file is already stored in the local file cache of the application server, the waiting time is zero. The total waiting time is the summation of all the message transmission times for all file accesses in a session.
- Number of prefetched files: This is the total number of prefetched files.
- Size of prefetched files: This is the total size of the prefetched files.
- Hit number: This is the total number of the files that are stored in the cache when they are accessed.
- Hit ratio of the file access.
- Hit file size: This is the total size of the files stored in the local file cache when they are accessed.
- Total file size: This is the total size of accessed files.
- Hit ratio by data size: This is the size of the accessed files relative to the total size of the data files, expressed as a percentage.

In the first part of the experiment, we measured the effect of time prefetching as mentioned in Task 5 of the AFPL example shown in Figure 6. The AFPL program was set up to prefetch those files accessed during the first 5 and 10 minutes of previous sessions, and compared with demand fetching (see Table 1 to Table 4). For Table 1 and Table 2, the network environment was a LAN with an average latency of 3 ms, and present the results for cache sizes of 5 and 25 MB, respectively. Also, we simulated four data transfer rates: 512 kbps, 1 Mbps, 10 Mbps, and 100 Mbps. It is obvious that the total waiting time for remote file fetching is reduced significantly in all four tasks. In particular, the reduction in the total waiting time is greater when the data transfer rate is smaller, which is due to time prefetching increasing the hit ratio. For Table 3 and Table 4, the network

environment was a WAN with an average latency of 100 ms. Compared with the results in Table 1 and Table 2, the decrease in the total waiting time is more significant in the WAN environment, which is obviously due to the average latency increasing.

In the second part of the experiment, we measured the effect of event prefetching as mentioned in Task 7 of the AFPL example shown in Figure 6. The AFPL program was set up to prefetch those files accessed by the application program "kword" more than three times in the previous sessions. These results show that event prefetching can also reduce the total waiting time significantly. Because of space limitation, refer to [15] for details. The prefetching is especially effective when the average latency is higher, the data transfer rate is lower, and the cache is larger.

5 MB cache (LAN)		Demand fetching	Prefetching files accessed during the first 5 minutes after logging in	Prefetching files accessed during the first 10 minutes after logging in
Total waiting time for remote file fetch (ms)	512 kbps	362212	182596	182699
	1 Mbps	181747	91894	91998
	10 Mbps	19342	10284	10389
	100 Mbps	3106	2127	2232
Number of prefetched files		0	105	142
Size of prefetched files (kB)		0	2646	4660
Hit number		341	417	418
Hit ratio		57.1%	69.8%	70.0%
Number of accessed files		597	597	597
Hit file size (kB)		9169	10828	10830
Hit ratio by data size		9.0%	10.6%	10.6%
Total file size (kB)		101446	101446	101446

Table 1: Result 1: Cache size of 5 MB in a LAN

25 MB cache (LAN)		Demand fetching	Prefetching files accessed during the first 5 minutes after logging in	Prefetching files accessed during the first 10 minutes after logging in
Total waiting time for remote file fetch (ms)	512 kbps	48361	44832	130
	1 Mbps	24392	22517	122
	10 Mbps	2824	2436	117
	100 Mbps	670	428	117
Number of prefetched files		0	105	158
Size of prefetched files (kB)		0	2646	24629
Hit number		448	525	558
Hit ratio		75.0%	87.9%	93.4%
Number of accessed files		597	597	597
Hit file size (kB)		77477	79136	101438
Hit ratio by data size		76.3%	78.0%	99.9%
Total file size (kB)		101446	101446	101446

Table 2: Result 2: Cache size of 25 MB in a LAN

5 MB cache (WAN)		Demand fetching	Prefetching files accessed during the first 5 minutes after logging in	Prefetching files accessed during the first 10 minutes after logging in
Total waiting time for remote file fetch (ms)	512 kbps	407511	223530	226928
	1 Mbps	227046	132828	136227
	10 Mbps	64641	51218	54618
	100 Mbps	48405	43061	46461
Number of prefetched files		0	105	142
Size of prefetched files (kB)		0	2646	4660
Hit number		341	417	418
Hit ratio		57.1%	69.8%	70.0%
Hit file size (kB)		9169	10828	10830
Hit ratio by data size		9.0%	10.6%	10.6%
Total file size (kB)		101446	101446	101446

Table 3: Result 3: Cache size of 5 MB in a WAN

25 MB cache (WAN)		Demand fetching	Prefetching files accessed during the first 5 minutes after logging in	Prefetching files accessed during the first 10 minutes after logging in
Total waiting time for remote file fetch (ms)	512 kbps	62814	51816	3913
	1 Mbps	38845	29501	3905
	10 Mbps	17277	9420	3900
	100 Mbps	15123	7412	3900
Number of prefetched files		0	105	158
Size of prefetched files (kB)		0	2646	24629
Hit number		448	525	558
Hit ratio		75.0%	87.9%	93.4%
Number of accessed files		597	597	597
Hit file size (kB)		77477	79136	101438
Hit ratio by data size		76.3%	78.0%	99.9%
Total file size (kB)		101446	101446	101446

Table 4: Result 4: Cache size of 25 MB in a WAN

5 Conclusions and Future Work

This paper presents a proposed method of appointed file prefetching for distributed file systems. We propose the AFPL and a corresponding architecture to support it. We demonstrate the feasibility of the method by implementing a prototype and conducting experiments. The experimental results show that the waiting time for remote file fetching is reduced by 30% to 90% and the hit ratio is increased by 6% to 18% in most cases. Moreover, as network transmission slows, both time and event prefetching deliver a greater improvement compared with demand prefetching.

References

1. *Linux Terminal Server Project*. <http://www.ltsp.org/>.
2. Gwan-Hwan Hwang, Jia-Qing Li. MAS TC/S: Roaming Thin Clients in a Wide Area Network with Transparent Working Environments. Proceedings of Advanced Technologies and Applications for Next Generation Information Communication Networks, HsinChu, Taiwan, 2002.
3. J. Griffioen and R. Appleton. Reducing File System Latency using a Predictive Approach. *In Proc. 1994 USENIX Summer Conference*, pp. 197-207, June 1994.
4. T.M. Kroeger and D.D.E. Long. Predicting Future File-System Actions from Prior Events. *In Proc. 1996 USENIX Annual Technical Conference*, pp. 319-328, Jan 1996.
5. T.M. Kroeger and D.D.E. Long. The case for efficient file access pattern modeling. *In Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, IEEE, March 1999.
6. T.M. Kroeger and D.D.E. Long. Design and Implementation of a Predictive File Prefetching Algorithm. *In Proceedings of the 2001 USENIX Annual Technical Conference*.
7. Hui Lei, Dan Duchamp. An Analytical Approach to File Prefetching. *Proceeding of the USENIX Annual Technical Conference*, 1997.
8. Yun-Sheng Chen. *A Practical Approach for File Prefetching in Distributed File System*. National Taiwan Normal University, Graduate Institute of Computer Science & Information Engineering Master Thesis, 2003.
9. B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel, and D. Hitz. *NFS Version 3 Design and Implementation*. In Proceedings of the Summer 1994 USENIX Technical Conference, Jun 1994.
10. J. H. Howard. *An Overview of the Andrew File System*. In Proceeding of the USENIX Winter Technical Conference, page 23-26, 1998.
11. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, USA, 1996.
12. George Coulouris, Jean Dollimore, and Tim kindberg. *Distributed systems: Concepts and Design*. Third Edition. Addison-Wesley, 2001.
13. *KOffice - Integrated Office Suite*. <http://www.koffice.org/>
14. Abraham Silberschatz, Peter Baer Galvin, Greg Gagne. *Operating System Concepts*. John Wiley & Sons, 7 edition, 2004
15. Hsin-Fu Lin. *Appointed Prefetching for Distributed File System of Thin-Client/Server Computing in WAN*. National Taiwan Normal University, Graduate Institute of Computer Science & Information Engineering Master Thesis, 2004.