The Design and Implementation of Appointed File Prefetching for Distributed File Systems

Gwan-Hwan Hwang

Department of Computer Science and Information Engineering,
National Taiwan Normal University, Taipei, Taiwan

Hsin-Fu Lin

Information Engineering Institute, Institute for Information Industry, Taipei, Taiwan

Chun-Chin Sy and Chiu-Yang Chang

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

Many types of distributed file systems have been in widespread use for more than a decade. One of key issues in their design 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. 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

ACM Classification: D.4.3, D.4.6

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 (LTSP, 2002; K12, 2002). 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, as shown in Figure 1(a). 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

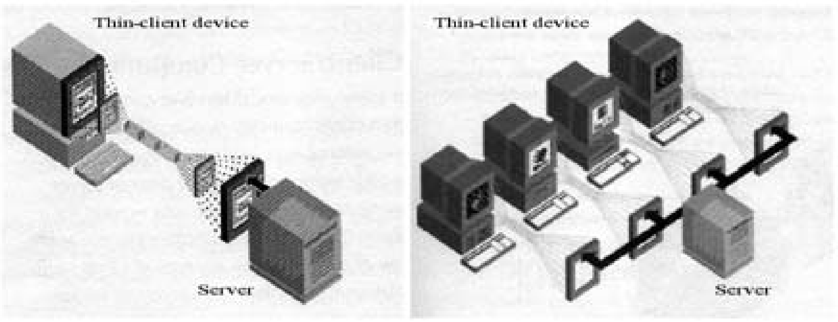
[*] It is the revised version for publication in *Journal of Research and Practice in Information Technology*. The corresponding author is Gwan-Hwan Hwang (e-mail: ghhwang@csie.ntnu.edu.tw, fax: +886-2-29322378).

Manuscript received: 22 May 2007
Communicating Editor: Bill Appelbe

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. All the interactions between thin-client devices and the application server occur via an efficient display protocol, which is highly optimized for specific software APIs in order to reduce their bandwidth requirements; e.g., X (NYE, 1992), independent computing architecture (ICA) (BOCA, 1999; Kanter, 1998), remote desktop protocol (Microsoft, 1999), and stateless low-level interface machine (SLIM) (Schmidt *et al*, 1999). The application server provides a centralized maintenance environment since all the applications are installed on it. The information-systems department of a corporation can deploy and update the applications instantly without ever needing to physically touch the desktops or PCs, thereby dramatically reducing the cost of upgrading and deploying applications. Users can also be provided with access to applications and data over a wider physical area, which increases their productivity; and security is also enhanced because all data are maintained on the application server. In addition, the TC/S computing model improves the utilization of computing and memory resources on the application server. For example, consider Figure 1(b) where several users execute the same application on the application server. These users share a single copy of the code image of the application, and the entire computing power of the server.

Thin-client hardware devices can be realized using low-cost, diskless computers with the display protocol built into their ROMs. They need only comprise the following hardware components: keyboards, monitors, serial or network interfaces, high-speed serial ports, and bidirectional parallel ports. Examples of proprietary thin-client devices include X terminal (NYE, 1992), SLIM console (Schmidt *et al*, 1999), and ICA's windows-based terminal (Kanter, 1998). An ordinary personal computer or workstation can also be used as a thin-client device by installing software that supports the display protocol (Kanter, 1998).

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 (NYE, 1992; BOCA, 1999; Kanter, 1998; Microsoft, 1999; Schmidt *et al*, 1999), 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 (BOCA, 1999; Kanter, 1998; Microsoft, 1999; Schmidt *et*



(a) (b)

Figure 1: TC/S computing models

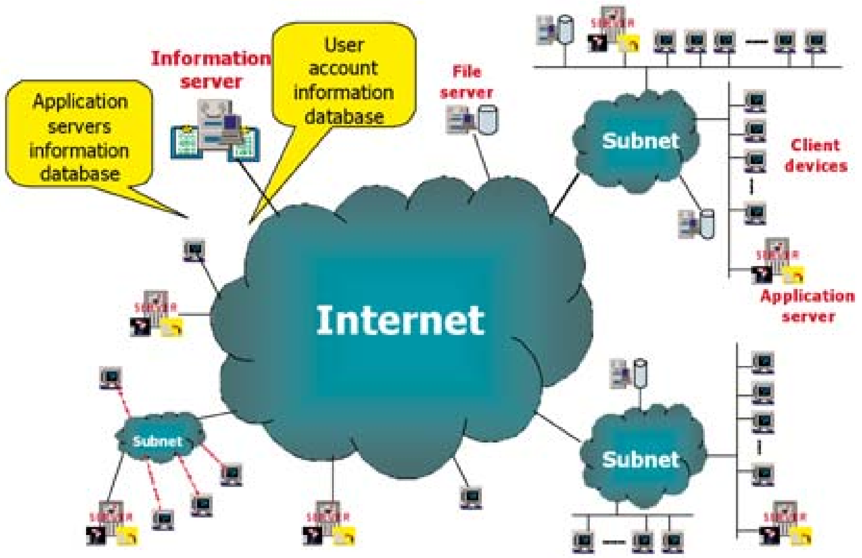


Figure 2: The MAS TC/S model

al, 1999). 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.

We have previously (Hwang *et al*, 2002) proposed a multiple-application-server (MAS) architecture model for the TC/S computing model (see Figure 2). 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: for example, 1,000 GB of disk storage would be required in an enterprise with 10,000 users whose individual disk quota is 100 MB. 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. For example, 100 GB of data would need to be routed among all the application servers in order to update a 10 MB file for each of the 10,000 users (by using the well-known read-one-write-all scheme, see Levy *et al*, (1990)). Note that we assume that the application software is fully duplicated on all the application 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. One method for automatic prefetching was proposed by Griffioen *et al*, (1994). They designed and implemented a simple analyzer that attempts to predict future file accesses based on past access patterns. Driven by trace data, the analyzer dynamically creates a logical graph called a probability graph, in which each node represents a file. Kroeger *et al* (1996) proposed the use of a context model to activate automatic prefetching. A context model uses preceding events to model the next event. For example, in the string "object" the character "t" is said to occur within the context "object". A context model uses a trie (a data structure based on a tree) to efficiently store sequences of symbols. The state space for this model is proportional to the number of nodes in this tree, which is bounded by O(n^m), where m is the highest order tracked and n is the number of unique files. On a typical file system the number of files can range between 10 thousand and 100 million, and hence such space requirements are clearly unreasonable. In response, they developed the PCM (Kroeger *et al*, 1999) and EPCM (Kroeger *et al*, 2001). Lei *et al* (1997) presented a file-prefetching mechanism based on real-time analytic modeling of interesting system events. The mechanism, incorporated into a client's file cache manager, seeks to build semantic structures that capture the intrinsic correlations between file accesses. We previously proposed a scheme that analyzes the system calls issued by the user to build an application tree to perform automatic prefetching (Chen, 2003).

According to previous experimental results, 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) (Pawlowski *et al*, 1994), the Andrew file system (AFS) (Howard, 1998), the distributed computing environment (Kazar, 1990), the UFO global file system (Alexandrov *et al*, 1997), the Coda file system (Mummert *et al*, 1995), and the Samba file system (Vernooij *et al*, 2002). 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 (Gosling *et al*, 1996). 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.

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 (see Figure 3). 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.

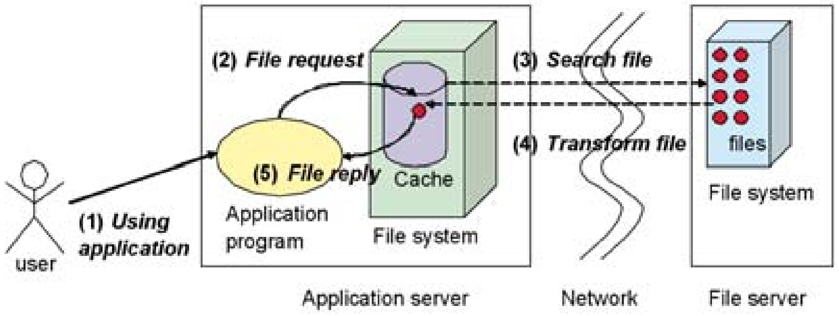


Figure 3: Demand fetching in distributed file systems

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. Here we propose a new approach for solving this problem, called appointed file prefetching (see Figure 4). 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 provided by the user. Figure 5 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 support the collection of information on file operating-system calls that are invoked by the

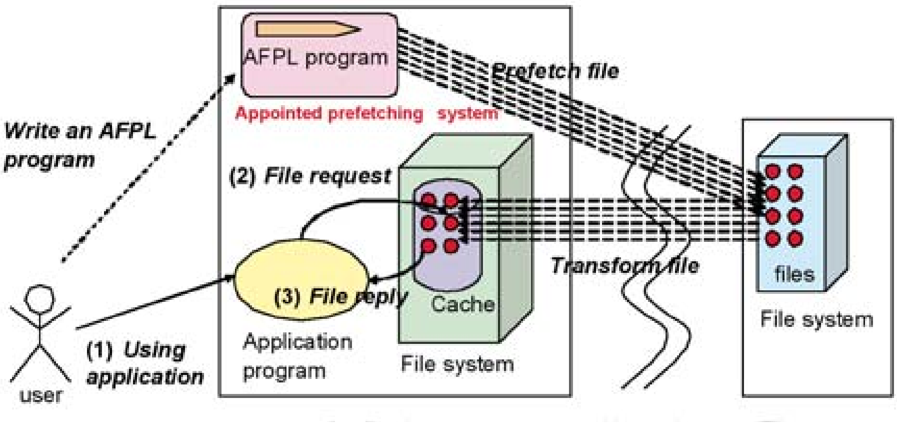


Figure 4: The operational model of appointed file prefetching

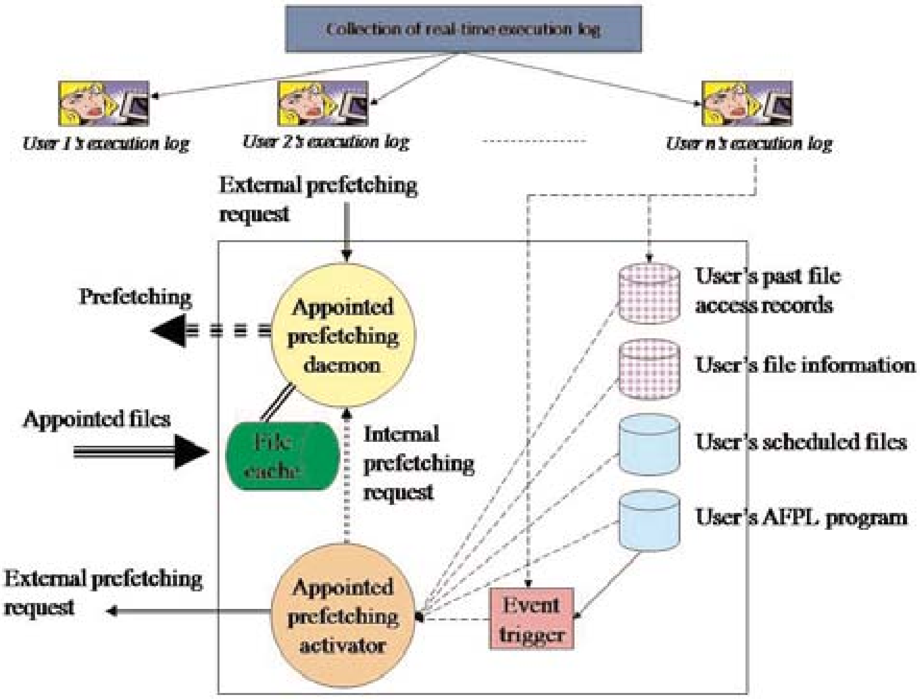
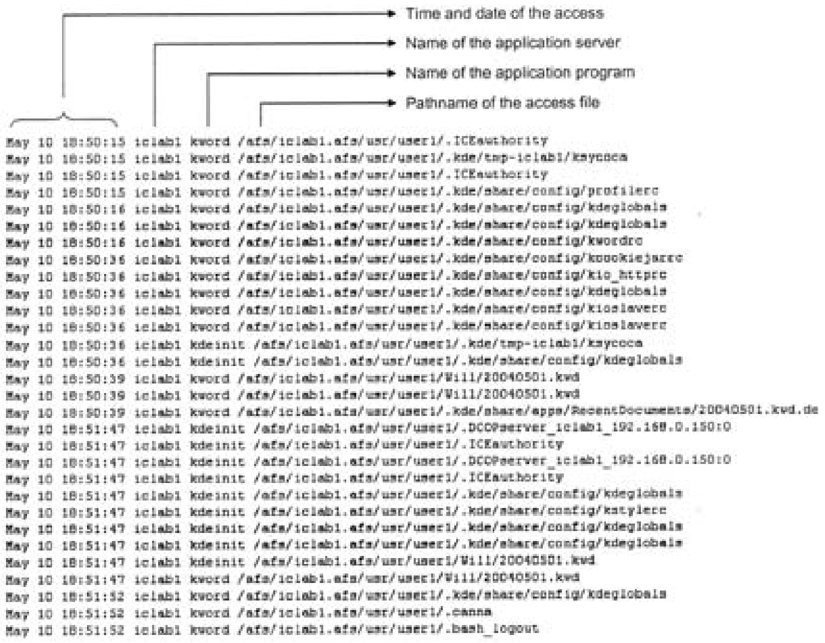

Collection of real-time execution log
User 1's execution log
User 2's execution log
User n's execution log
External prefetching request
Prefetching
Appointed prefetching daemon
Appointed files
File cache
Internal prefetching request
User's past file access records
User's file information
User's scheduled files
User's AFPL program
External prefetching request
Appointed prefetching activator
Event trigger

Figure 5: The architecture of appointed file prefetching


Time and date of the access
Name of the application server
Name of the application program
Pathname of the access file

```
May 10 18:50:15 iclab1 kword /afs/iclab1.afs/usr/user1/.ICEauthority
May 10 18:50:15 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/tmp-iclab1/ksycoca
May 10 18:50:15 iclab1 kword /afs/iclab1.afs/usr/user1/.ICEauthority
May 10 18:50:15 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/share/config/profilerc
May 10 18:50:16 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/share/config/kdeglobals
May 10 18:50:16 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/share/config/kdeglobals
May 10 18:50:16 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/share/config/kwordrc
May 10 18:50:36 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/share/config/kcookiejarrc
May 10 18:50:36 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/share/config/kio_httprc
May 10 18:50:36 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/share/config/kdeglobals
May 10 18:50:36 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/share/config/kioslaverc
May 10 18:50:36 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/share/config/kioslaverc
May 10 18:50:36 iclab1 kdeinit /afs/iclab1.afs/usr/user1/.kde/tmp-iclab1/ksycoca
May 10 18:50:36 iclab1 kdeinit /afs/iclab1.afs/usr/user1/.kde/share/config/kdeglobals
May 10 18:50:39 iclab1 kword /afs/iclab1.afs/usr/user1/Will/20040501.kwd
May 10 18:50:39 iclab1 kword /afs/iclab1.afs/usr/user1/Will/20040501.kwd
May 10 18:50:39 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/share/apps/RecentDocuments/20040501.kwd.de
May 10 18:51:47 iclab1 kdeinit /afs/iclab1.afs/usr/user1/.DCOPserver_iclab1_192.168.0.150:0
May 10 18:51:47 iclab1 kdeinit /afs/iclab1.afs/usr/user1/.ICEauthority
May 10 18:51:47 iclab1 kdeinit /afs/iclab1.afs/usr/user1/.DCOPserver_iclab1_192.168.0.150:0
May 10 18:51:47 iclab1 kdeinit /afs/iclab1.afs/usr/user1/.ICEauthority
May 10 18:51:47 iclab1 kdeinit /afs/iclab1.afs/usr/user1/.kde/share/config/kdeglobals
May 10 18:51:47 iclab1 kdeinit /afs/iclab1.afs/usr/user1/.kde/share/config/kstylerc
May 10 18:51:47 iclab1 kdeinit /afs/iclab1.afs/usr/user1/.kde/share/config/kdeglobals
May 10 18:51:47 iclab1 kdeinit /afs/iclab1.afs/usr/user1/.kde/share/config/kdeglobals
May 10 18:51:47 iclab1 kdeinit /afs/iclab1.afs/usr/user1/Will/20040501.kwd
May 10 18:51:47 iclab1 kword /afs/iclab1.afs/usr/user1/Will/20040501.kwd
May 10 18:51:52 iclab1 kword /afs/iclab1.afs/usr/user1/.kde/share/config/kdeglobals
May 10 18:51:52 iclab1 kword /afs/iclab1.afs/usr/user1/.canna
May 10 18:51:52 iclab1 kword /afs/iclab1.afs/usr/user1/.bash_logout
```

Figure 6: The past file access records obtained from a Linux PC

application programs of the user. In our implementation, we modified the file system calls to collect this information[1]. Figure 6 shows an example of the past file access records obtained in a Linux PC.

- The file information of the user: This includes the name and last access time of all files and directories of the user.
- The schedule information of the user: Appointed file prefetching can be performed according to the schedule of the user.
- The APFL program: This is written by either the system administrator or the user.

The appointed file-prefetching activator receives the AFPL program and then parses the prefetching instructions. The activator generates file-prefetching requests based on the past file access records, the file information, and the schedule information of the user. Internal prefetching requests are sent to the appointed file-prefetching daemon that is responsible for prefetching files from the file server. Note that the prefetched files are stored in the local file cache of the application server. External prefetching requests are sent to the appointed file-prefetching daemons of other application servers. An event trigger is used to handle the event prefetching: it analyses the specification of event prefetching in the AFPL program and then monitors the real-time execution logs from the operating system. As the expected events occur, the event trigger sends the pre-fetching requests to the appointed file-prefetching activator, which forwards the requests to the appropriate prefetching daemon.

3. THE APPOINTED FILE-PREFETCHING LANGUAGE

In this section, we present the design and syntax of AFPL. Consider the following scenario:

> "Professor Williams is teaching at a university. Instead of installing many PCs in the campus, his university adopts the MAS TC/S model. Application servers and the attached client devices are installed in the campus. He lives at the dormitory of his school, and he connects to different application servers while he is in his dormitory, office, classrooms, or the coffee shops on the campus. All his files are stored in the file server that is maintained by the system administrator of the university."

For convenience, Professor Williams would like to set up the following appointed file prefetching:

Task 1. Professor Williams wants to review all the files in his home directory "`~/paper`" and "`~/course`" at 2000 hours every day on the application server of his dormitory.

Task 2. He has a course from 1400 to 1600 hours every Thursday. He wants to prefetch all the files with filename "`* .ppt`" to the application server of the classroom at that time.

Task 3. He wants the system to prefetch all the files that he accessed in the past three days when he logs into each application server.

Task 4. He wants the system to prefetch all the files that are smaller than 1 MB and which he has modified in the last week when he logs into each application server.

Task 5. He wants the system to prefetch the files he accessed in the first five minutes when he logged into each application server in all earlier sessions.

Task 6. From September 26, 2005, to September 28, 2005, Professor Williams is going to attend a conference held at another campus of his university. The conference room in that campus

[1] The details of how to modify the file system calls to record the past file access record are presented elsewhere (Chen, 2003).

has a client device connected 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[2]", he wants the system to prefetch all the files in his directory "`~/doc`".

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 7). An AFPL program is first compiled by the Java compiler. As it is executed, the prefetching requests in it are sent to the event trigger and the appointed prefetching activator as shown in 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).
- `timePrefetch(String` *`serverIP`*`, TimePro` *`startTime`*`, TimePro` *`endTime`*`, int` *`timeSlot`*`, int` *`priority`*`)`: This specifies when to perform a prefetch. The prefetching should start at time "*`startTime`*" and continue to do prefetching every "*`timeSlot`*" minutes until the time "*`endTime`*".
- `eventPrefetch(int` *`type`*`, String` *`description`*`, int minutes, int` *`priority`*`)`: This specifies that the system is to prefetch when 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.

[2] Note that the "KWord" is an application software of Linux (KOffice, 2002) .

```
// Task 1
public class Task1 {
      public static void main(String[] args) {
            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
public class Task2 {
      public static void main(String[] args) {
            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);
      }
}
```

```
// Task 3
public class Task3 {
      public static void main(String[] args) {
            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);
        }
    }
```

```
    // Task 4
    public class Task4 {
        public static void main(String[] args) {
            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);
        }
    }
```

```
    // Task 5
    public class Task5 {
        public static void main(String[] args) {
            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);
        }
    }
```

```
// Task 6
public class Task6 {
    public static void main(String[] args) {
        FileVector fs1 = new FileVector();
        fs1.addFromDir("/afs/abc.afs/usr/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
public class Task7 {
    public static void main(String[] args) {
        FileVector fs1 = new FileVector();
        fs1.addFromDir("/afs/abc.afs/usr/dr_williams/doc");
        fs1.eventPrefetch(3, "KWord", 1, 3);
    }
}
```

```
// Task 8
public class Task8 {
    public static void main(String[] args) {
        FileVector fs1 = new FileVector();
        fs1.addFromDir("/afs/abc.afs/usr/dr_williams/music");
        fs1.eventPrefetch(2, "coffee", 3, 5);
    }
}
```

```
// Task 9
public class Task9 {
    public static void main(String[] args) {
        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");
        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());
            }
        }
    }
}
```

Figure 7: Examples of the AFPL

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

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 Figure 5. The implementation work includes the following:

- The Java API shown in Figure 7.
- 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 (Chen, 2003).
- 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.

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 (Kroeger *et al*, 1996). The latency is the delay that occurs after a send operation is executed before data starts to arrive at the destination computer, which can be measured as the time required for transferring an empty message. The data transfer rate is the speed at which data can be transferred between two computers in the network once transmission has begun, usually quoted in bits per second (bps). Following from these definitions, the time required for a network to transfer a message between two computers is approximately

Message transmission time =
latency + message size/data transfer rate.

In accordance with Coulouris *et al* (2001), we assume the that average latencies in a LAN and a WAN are 3 and 100 ms, respectively. We have a Linux machine running the KOffice program (KOffice, 2002). 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 (Silberschatz *et al*, 2004). The following items are the measured data:

- Total waiting time for fetching remote files (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: This is the percentage of the files that are stored in the cache when they are accessed.
- Number of accessed files: This is the total number of access files.
- 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 7. 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). Since a session starts when the user logs on and ends when the user logs out, the past file access record contains the file access history for a lot of sessions. 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 compared to the demand fetching. For example, the hit ratio is increased 12.7% as shown in Table 1. 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 7. The AFPL program was set up to prefetch those files accessed by the application program “KWord” more than three times in the previous sessions (see Table 5 to Table 8). These results show that event prefetching can also reduce the total waiting time significantly. The prefetching is especially effective when the average latency is higher, the data transfer rate is lower, and the cache is larger.

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

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 fetching remote files (ms)	512 kbps	362212	182596	182699
	1 Mbps	181747	91894	91998
	10 Mbps	19342	10284	10389
	100 Mbps	3106	2127	2232
Number of pre-fetched files		0	105	142
Size of pre-fetched 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 fetching remote files (ms)	512 kbps	48361	44832	130
	1 Mbps	24392	22517	122
	10 Mbps	2824	2436	117
	100 Mbps	670	428	117
Number of pre-fetched files		0	105	158
Size of pre-fetched 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 fetching remote files (ms)	512 kbps	407511	223530	226928
	1 Mbps	227046	132828	136227
	10 Mbps	64641	51218	54618
	100 Mbps	48405	43061	46461
Number of pre-fetched files		0	105	142
Size of pre-fetched 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

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

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 fetching remote files (ms)	512 kbps	62814	51816	3913
	1 Mbps	38845	29501	3905
	10 Mbps	17277	9420	3900
	100 Mbps	15123	7412	3900
Number of pre-fetched files		0	105	158
Size of pre-fetched 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 MB cache (LAN)		Demand fetching		Prefetching files accessed by "KWord" more than three times after logging in	
Total waiting time for fetching remote files (ms)	512 kbps	265431		128631	
	1 Mbps	133262		64887	
	10 Mbps	14337		7541	
	100 Mbps	2451		1811	
Number of pre-fetched files		0		62	
Size of pre-fetched files (kB)		0		4832	
Hit number	Hit ratio	542	70.2%	572	74.0%
Number of accessed files		0		772	
Hit file size (kB)	Hit ratio by data size	4319	5.9%	9142	12.5%
Total file size (kB)		72832		72832	

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

event prefetching deliver a greater improvement compared with demand prefetching.

In addition to the thin-client/server computing model, we consider that the concept can be applied to a variety of fields such as Web prefetching, pervasive computing, and groupware. Future

25 MB cache (LAN)		Demand fetching		Prefetching files accessed by "KWord" more than three times after logging in	
Total waiting time for fetching remote files (ms)	512 kbps	47339		1047	
	1 Mbps	23854		643	
	10 Mbps	2724		282	
	100 Mbps	614		249	
Number of pre-fetched files		0		76	
Size of pre-fetched files (kB)		0		23079	
Hit number	Hit ratio	641	83.0%	690	89.3%
Number of accessed files		772		772	
Hit file size (kB)	Hit ratio by data size	49348	67.7%	72425	99.4%
Total file size (kB)		72832		72832	

Table 6: Result 6: Cache size of 25 MB in a LAN

5 MB cache (WAN)		Demand fetching		Prefetching files accessed by "KWord" more than three times after logging in	
Total waiting time for fetching remote files (ms)	512 kbps	304231		167725	
	1 Mbps	172062		103981	
	10 Mbps	53137		46635	
	100 Mbps	41251		40905	
Number of pre-fetched files		0		63	
Size of pre-fetched files (kB)		0		4833	
Hit number	Hit ratio	542	70.2%	572	74.0%
Number of accessed files		0		772	
Hit file size (kB)	Hit ratio by data size	4319	5.9%	9142	12.5%
Total file size (kB)		72832		72832	

Table 7: Result 7: Cache size of 5 MB in a WAN

work should include developing a friendly graphical user interface to aid the user in specifying the AFPL. Also, we consider it feasible to develop a systematic scheme for analyzing the past file access records to derive an appropriate AFPL program for the user automatically. In addition, we do not consider the possibility for the partial prefetching of a single file. For large files, the user may

25 MB cache (WAN)		Demand fetching		Prefetching files accessed by "KWord" more than three times after logging in	
Total waiting time for fetching remote files (ms)	512 kbps	60046		9001	
	1 Mbps	36561		8597	
	10 Mbps	15431		8236	
	100 Mbps	13321		8203	
Number of pre-fetched files		0		76	
Size of pre-fetched files (kB)		0		23079	
Hit number	Hit ratio	641	83.0%	690	89.3%
Number of accessed files		772		772	
Hit file size (kB)	Hit ratio by data size	49348	67.7%	72425	99.4%
Total file size (kB)		72832		72832	

Table 8: Result 8: Cache size of 25 MB in a WAN

only access a small piece of them. The AFPL presented in the paper cannot specify how to make partial prefetching of files.

REFERENCES

ALEXANDROV, A.D., IBEL, M., SCHAUSER, K.E. and SCHEIMAN, C.J. (1997): Extending the operating system at the user level: the UFO global file system. *Proceedings of the USENIX Annual Technical Conference*.

BOCA RESEARCH, INC. (1999): Citrix ICA technology brief. Technical White Paper, Boca Raton, FL.

CHEN, Y.S. (2003): A practical approach for file prefetching in distributed file system. National Taiwan Normal University Graduate Institute of Computer Science & Information Engineering Master Thesis, Advisor: Gwan-Hwan Hwang.

COULOURIS, G., DOLLIMORE, J. and KINDBERG, T. (2001): Distributed systems: Concepts and design. Third Edition. Addison-Wesley.

GOSLING, J., JOY, B. and STEELE, G. (1996): The Java language specification (First Edition). Addison-Wesley, Reading, Massachusetts, USA.

GRIFFIOEN, J. and APPLETON, R. (1994): Reducing file system latency using a predictive approach. In *Proc. 1994 USENIX Summer Conference*, 197-207.

HOWARD, J.H. (1998): An overview of the Andrew file system. In *Proceeding of the USENIX Winter Technical Conference*, 23-26.

HWANG, G.H., HWANG, S.Y., CHEN, Y.S. and LI, J Q. (2002): 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.

K-12. (2002): K-12Linux project. http://www.k12ltsp.org/. Accessed Nov-2002.

KANTER, J.P. (1998): Understanding thin-client/server computing. Microsoft Press.

KAZAR, M.L., LEVERETT, B.W., ANDERSON, O.T., APOSTOLIDES, V., BOTTOS, B.A., CHUTANI, S., EVERHART, C.F., MASON, W.A., TU, S.-T. and ZAYAS, E.R. (1990): Decorum file system architectural overview. In *Proceedings of the summer 1990 USENIX Technical Conference*.

KOFFICE (2002): KOffice - Integrated office suite. http://www.koffice.org/ Accessed Nov-2002.

KROEGER, T. M. and LONG, D. D. E. (1996): Predicting future file-system actions from prior events. In *Proc. 1996 USENIXAnnual Technical Conference*, 319-328.

KROEGER, T.M. and LONG, D.D.E. (1999): The case for efficient file access pattern modeling. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, IEEE.

KROEGER, T.M. and LONG, D.D.E. (2001): Design and implementation of a predictive file prefetching algorithm. In *Proceedings of the 2001 USENIX Annual Technical Conference*.

LEI, H. and DUCHAMP, D. (1997): An analytical approach to file prefetching. *Proceeding of the USENIX Annual Technical Conference*.

LEVY, E. and SILBERSCHATZ, A. (1990): Distributed file systems: Concepts and examples. *ACM Computing Surveys*, 22(4).

LTSP (2002): Linux terminal server project. http://www.ltsp.org/. Accessed Nov-2002.

MICROSOFT CORPORATION (1999): Comparing MS Windows NT server 4.0, Terminal server edition, and Unix application deployment solutions, *Technical White Paper*, Redmond, WA.

MUMMERT, L.B., EBLING, M.R. and SATYANARAYNAN, M. (1995): Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, USA.

NYE, A. (ed) (1992): X Protocol reference manual. O'Reilly & Associates, Sebastopol, CA.

PAWLOWSKI, B., JUSZCZAK, C., STAUBACH, P., SMITH, C., LEBEL, D. and HITZ, D. (1994): NFS Version 3 design and implementation. In *Proceedings of the Summer 1994 USENIX Technical Conference*.

SCHMIDT, B.K., LAM, M.S. and NORTHCUTT, J.D. (1999): The interactive performance of SLIM: a Stateless, thin-client architecture. *The 17th ACM Symposium on Operating Systems Principles (SOSP'99)*.

SILBERSCHATZ, A., GALVIN, P.B. and GAGNE, G. (2004): Operating system concepts. John Wiley & Sons, 7th edition.

VERNOOIJ, J.R., TERPSTRA, J.H., and CARTER, G. (2002): The official Samba-3 HOWTO and reference guide. http://us3.samba.org/samba/docs/man/ Samba-HOWTO-Collection/. Accessed Nov-2002.

BIOGRAPHICAL NOTES

Gwan-Hwan Hwang is an associate professor in the Department of Computer Science and Information Engineering at National Taiwan Normal University, Taiwan. He received the BS and MS degrees while in the Department of Computer Science and Information Engineering at National Chiao-Tung University, in 1991 and 1993, respectively, and the PhD degree while in the Department of Computer Science at National Tsing-Hua University, HsinChu, Taiwan, in 1998. His research interests include distributed system, Internet security, software testing, and workflow management system.

Gwan-Hwan Hwang

Hsin-Fu Lin is currently an engineer in Information Engineering Institute at Institute for Information Industry, Taiwan. He received the BS and MS degrees while in the Department of Computer Science and Information Engineering at National Taiwan Normal University in 2001 and 2003, respectively. His research interests include distributed system, web service, e-Gov related issues, and system management.

Hsin-Fu Lin

Chun-Chin Sy is an instructor in the Department of Electronic Engineering at National United University, Miao-Li, Taiwan. He received the BS degree while in the Department of Mathematics at Chung Yuan Christian University, Chung-Li, Taiwan in 1975, and the MS degree while in the Department of Computer Science at Mankato State University, Mankato, Minnesota in 1986. His research interests include workflow management, Internet security, data structure, DNA analysis system, and operating systems.

Chun-Chin Sy

Chiu-Yang Chang is currently an instructor in the Department of Electronic Engineering at National United University, Miao-Li, Taiwan. He received his BS degree from the Department of Mathematics at Chung Yuan Christian University, Chung-Li, Taiwan in 1975, and his MS degree from the Department of Computer Science at East Texas State University, Commerce, Texas in 1986. His research interests include systems programming, software engineering, Internet technologies, and Internet-based applications.

Chiu-Yang Chang