2005

Design and Implementation of a Media Uploading System

Mu Zhang
Iowa State University

Johnny S. Wong
Iowa State University, wong@iastate.edu

Wallapak Tavanapong
Iowa State University

Design and Implementation of a Media Uploading System

Abstract
This paper presents the design and performance analysis of an uploading system that automatically uploads multimedia files to a centralized server given client hard deadlines. If not uploaded by the deadlines, existing files may be lost or new files cannot be recorded. The uploading systems with hard deadlines have several important applications in practice. For instance, such systems can be used in hospitals to gather videos generated from medical devices from various operating rooms for post-procedure analysis and in law enforcement to collect video recordings from police cars during routine patrolling. In this paper, we study the uploading system with hard deadlines in detail. We present the software architecture of the uploading system. Two server scheduling algorithms that determine which client uploads its file first are investigated . We introduce two emergency control algorithms to handle situations when a client is about to use up its disk space. We evaluate the proposed algorithms via analysis and simulations. Our performance study additionally reveals the impact of the emergency control algorithms on the server scheduling algorithms.

Keywords
Upload, Hard real-time systems, Emergency control, scheduling algorithm

Disciplines
Databases and Information Systems

Design and Implementation of a Media Uploading System

Mu Zhang Johnny Wong Wallapak Tavanapong

Abstract

This paper presents the design and performance analysis of an uploading system that automatically uploads multimedia files to a centralized server given client hard deadlines. If not uploaded by the deadlines, existing files may be lost or new files cannot be recorded. The uploading systems with hard deadlines have several important applications in practice. For instance, such systems can be used in hospitals to gather videos generated from medical devices from various operating rooms for post-procedure analysis and in law enforcement to collect video recordings from police cars during routine patrolling. In this paper, we study the uploading system with hard deadlines in detail. We present the software architecture of the uploading system. Two server scheduling algorithms that determine which client uploads its file first are investigated . We introduce two emergency control algorithms to handle situations when a client is about to use up its disk space. We evaluate the proposed algorithms via analysis and simulations. Our performance study additionally reveals the impact of the emergency control algorithms on the server scheduling algorithms.

Keywords: Upload, Hard real-time systems, Emergency control, Scheduling algorithm

1 Introduction

Recent years have seen numerous designs and performance evaluations of downloading applications for multimedia files such as video streaming and file sharing applications. Little has been investigated on the system that automatically uploads multimedia files from a large number of clients to a centralized server given clients' hard deadlines. These deadlines are imposed by limited capability of a client machine such as limited storage space.

In such a system, a client machine has a third-party recording application that repeatedly produces multimedia files on a local disk drive. It is required that the files be stored at a central server for future analysis or for secure archiving. When the client disk is full, either a new multimedia file is not recorded or some existing files are overwritten, depending on the application that generates the files. In either case, the results are unacceptable. For instance, the uploading system can be used to collect video files generated from medical devices located in operating rooms for post-procedure analysis. In this case, an operating room has a client machine that captures a video signal from a medical device (say an endoscopy unit) into a file stored on a local disk. In one operating room, several medical procedures are performed one after another (with a short break in between) for several hours per day, which results in several video files being created. These videos are later uploaded to a centralized server for automatic analysis and documentation of abnormality. The uploading system is also useful for gathering video recordings from police cars during routine patrolling [8] and for collecting voice recordings of conversations between customers and sale representatives for quality control purposes, just to name a few.

In this paper, we present the design, implementation, and analysis of our media uploading system that has the following characteristics. First, the system takes into account of hard deadlines imposed by the client storage capacity. The system must ensure that multimedia files are uploaded by deadlines to free up disk space for the recording applications at the clients to repeatedly create more files. Second, the system must be able to detect when some client can no longer store the generated media files in its local disk. In this case, the system must alert the system administrator and attempt to relocate some files out of the client to give the system administrator as much time as possible to correct the problem. Third, the client uploading software must work with different third-party recording applications that may run on different platforms or operating systems. Hence, the client uploading software should not rely on low-level disk operations such as disk scheduling and disk block deallocation.

Our uploading system is unique as follows. The hard deadline requirement is not present in existing systems such as video streaming, distributed backup systems such as Amanda [4], peer-to-peer backup systems (e.g., Samsara [3] or [10]), distributed file systems (e.g., NFS [13], Andrew File System [6], Storage Area Network), and file sharing applications (e.g., KaZaA[9], Gnutella [5], or Napster[12]). Unlike recent advances in video streaming systems summarized in [7], continuous playback is not the requirement of the uploading system.

Our contributions are as follows. First, we design an uploading system that prolongs the *system lifetime* defined as *the time period since the system start-up till the time the system has no other ways to keep any of its clients from exhausting its storage space.* Our design consists of i) a server scheduling algorithm that determines which clients to upload their files first and ii) an emergency control algorithm that determines a migration plan to relocate files from clients that are about to exhaust their disk space. Second, we present an analytical model to estimate the system lifetime given the system configurations and workload characteristics. We validate our analytical results with simulation results. Last, we implement the prototype uploading system, which will later be used to upload video files captured from colonoscopic procedures from operating rooms in a hospital [2]. Note that this paper is a substantial extension of our previous work [15]. The extension includes the two emergency control algorithms, the revision of our analytical model, and the extensive simulation results.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the design and implementation of our uploading system. Simulation environment and results are detailed in Section 4. In Section 5, we present an analytical model to estimate the system lifetime and validation of the analytical results against the simulation results. Our concluding remarks are provided in Section 6.

2 Related Work

Bistro [1] is a file uploading system for Wide Area Network upload applications such as submissions of income tax forms, of projects and homework in distance learning, and of grant proposals to a funding agency [14]. To the best of our knowledge, the Bistro system is the only work that investigates and analyzes the uploading applications in detail.

Bistro aims to provide a scalable and efficient framework for the applications that need to submit files by submission deadlines. These files are later uploaded to a destination server. Bistro submission deadlines are imposed by real-life events (e.g., the income tax deadline), not by clients' limited capacity as in our uploading system. Furthermore, the client application requires the destination server to know that the client has submitted the files by the deadline. The submitted files cannot be modified after the deadline. Actual file transfer of the submitted files to the destination server needs not be done by the deadline.

Bistro uploads submitted files in three steps. First, when the client submits a file, the system timestamps the file. Second, the client software uploads the submitted file to an intermediate node (called a *bistro node*) in a secure way. In the third step, the bistro nodes collaborate to transfer submitted files to the destination server through some other bistro nodes along the network route towards the server. No deadlines are imposed in this step. As an example, federal income tax forms must be submitted before the deadline determined by the government. Once the forms are transferred to a bistro node, the user is informed of a successful submission. The files will be transmitted to the destination server under a coordinated transfer schedule at some time later, possibly after the deadline. The research problem addressed in Bistro is centered around finding an appropriate bistro node to upload the submitted files and the routes to transfer the files to the destination server. These problems differ from the problem investigated in this paper.

Distributed file systems such as the Network File System [13], the Andrew File System [6], Storage Area Network, or networked backup systems (e.g., the Amanda [4]) allow files to be stored on a network of machines in a local area network. Peer-to-peer backup systems (e.g., Samsara [3]) are recently introduced to lower the cost of Internet backup services using coordinated peers to store backup files for one another. Nevertheless, these systems do not have a deadline constraint on the file transmission. The design of a distributed file system typically concentrates on network transparency and high dependability.

Another related study is scheduling policies. Many scheduling policies have been developed for real-time systems such as Earliest Deadline First (EDF) [11]. With EDF, the task with the earliest deadline is assigned the highest priority and will be selected for execution. EDF effectively reduces the number of tasks missing deadlines. Our uploading system also contains tasks with hard deadlines. The idea of EDF is used in our system to determine a good uploading schedule that will prolong the system lifetime.

3 Media Uploading System

3.1 Overview

Our uploading system has a client-server architecture. Several client-server models such as server-pull, client push, or client-push-server-pull can be considered. Our uploading system employs the server-pull paradigm since it allows the server to control critical activities in the system, such as scheduling file uploads and handling emergency situations. An effective uploading scheduling technique ensures that clients obtain adequate server resources and avoid missing the deadlines. The emergency control rescues the clients that are about to exhaust its disk space by uploading files from these clients. Once the file has been uploaded, it is deleted from the client disk.

For simplicity, we assume that the server has very large storage space. This is possible given today's hard-drive hot swap technologies, allowing the replacement of a hard drive on the fly. Since our system is intent to be used in a local, high-bandwidth network, the topology of the underlying network does not have a significant impact on the overall system performance as in the wide area network. Hence, our design does not exploit the knowledge of the network topology.

Our performance metric is the system lifetime defined earlier. Note that this metric is different from those of the traditional distributed systems such as the system throughput, the client waiting time between consecutive uploads, or the number of files missing deadlines. These traditional measures are not suitable for our system. For instance, our system throughput is nearly constant since the bottleneck is the network bandwidth. The number of files missing deadlines is difficult to define because it depends on the behavior of the third party recording applications. To achieve the desired system lifetime, we can estimate the number of clients that can be supported given the system and workload characteristics. We introduce two important design concepts as follows.

- *Critical state of a client* is the event in which the available disk space at the client is below a pre-defined *client critical threshold*. For example, for the client critical threshold of 100 MB, we say that the client has entered its critical state at the time when its available disk space is below 100 MB. A client in its critical state is called a *critical client*. Other clients not in their critical state are termed *non-critical clients*.
- *Critical state of a system* is the event in which at least one client is in its critical state. This state indicates that the system is in the danger of losing files. A good uploading system should prevent the system from entering the critical state and perform emergency control to rescue the critical client out of the critical state.

The server is configured to use certain uploading and emergency control algorithms. The workflow of the system consists of two phases: the initialization phase and the working phase. During the initialization phase, the server makes a connection to every client in its list of supported clients. In the working phase, the server repeats the following operations until the system administrator terminates the server software.

1. The server sends a query request to each client asking for status information of that client. Each client sends back its current status information.
2. Based on the reported client status, the server checks whether any client enters a critical state. If at least one client enters its critical state, the server invokes the pre-configured emergency handling algorithm.
3. The server uses the pre-configured scheduling algorithm to choose the order the clients upload their file.
4. Once the schedule is obtained, the server sends the uploading request to each client in the schedule asking the client to deliver one of its files to the server one client at a time. Note that the unit for one upload is the entire file instead of some fixed amount of data. It is our intent for the client uploading software to be able to upload data generated by any third party software on a variety of platforms. Therefore, we do not try to optimize the system performance by relying on low level features of operating systems or try to exploit a particular behavior of the third party software when it is generating a multimedia file.

3.2 Server Design

The uploading scheduling and emergency control schemes are the two important components of the server. The uploading scheduling technique significantly affects the time period before a client enters its critical state, which determines the system

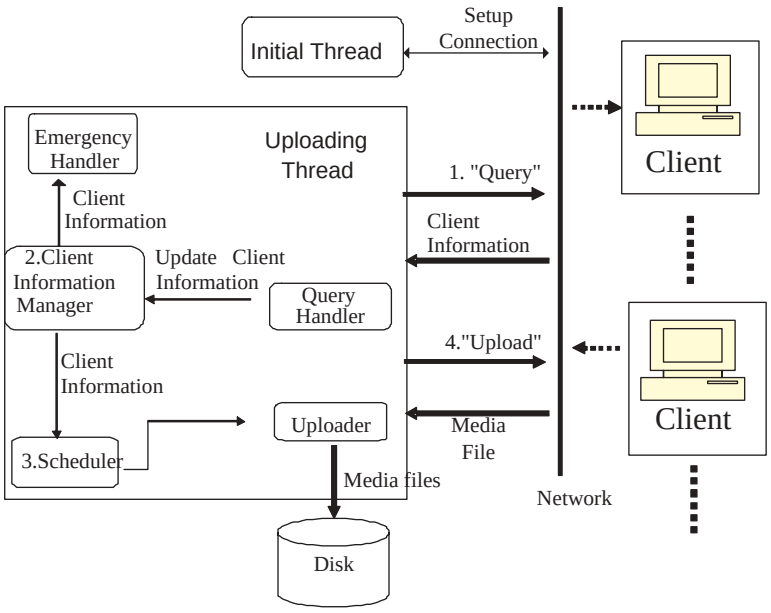


Figure 1. Server software architecture

lifetime. An effective uploading scheduling technique enables some clients that generate high quality multimedia files at a fast rate to obtain enough resources for uploading to avoid missing the deadlines. The emergency control scheme is a backup scheme that rescues the system from a critical state. It uploads files from the clients that are about to run out of disk space to free up some disk space.

Our server is a multi-threaded server and has the software architecture and important modules as shown in Figure 1. The *initial thread* is in charge of establishing connections with clients. The *uploading thread* repeats the following steps:

1. Query: The uploading thread uses its Query Handler to collect client status. Query Handler sends a query request to each client. The client returns its current status that consists of the amount of used disk space and the estimated rate that files are generated at the client. The updated client status is maintained by Client Information Manager.

2. Retrieval: Based on the current system condition derived from the received client information, the uploading thread uses the pre-configured uploading scheduling algorithm described in the next subsection to choose a client from whom a file is retrieved. Uploader sends an uploading request to this client and the client transfers a file to the server in return. After the entire file has been uploaded, the file is removed from the client local disk. If the system is about to enter its critical state (i.e., the received client information indicates that a client has entered its critical state or a client reports that it is about to enter its critical state), the uploading thread executes Emergency Handler that performs the pre-configured emergency handling technique. The techniques supported in our system are described in Section 3.2.2.

3.2.1 Scheduling Algorithms

The scheduling algorithm decides which file from which client should upload at the current scheduling point. In the normal working phase, the scheduling algorithm will be invoked right after a complete transfer of one file from a certain client. The scheduling algorithms must consider the hard deadline derived from client disk size, priority and size of the multimedia file. The scheduler is the main component that prolongs the system lifetime. We investigate Round Robin scheduling and introduce *Vulnerability Based scheduling*.

- With Round Robin scheduling, the server collects files from the clients one after another. The same client is not requested to upload its file again until all the other clients in the system have been visited. The server simply skips the client that has no files to upload. This scheduling algorithm is simple and fair to all the clients, but does not take deadlines into account. We use this algorithm as the baseline algorithm in our performance study.

- Vulnerability Based Scheduling (VBS) employs the concept of the Earliest Deadline First scheduling [11]. The deadline in our system is determined by the vulnerability of the client. Specifically, how soon this client will enter its critical state. Mathematically, Vulnerability of client i at the j^{th} scheduling point ($V_{i,j}$) is defined as

$$V_{i,j} = \frac{\text{Size of available disk space at client i at j scheduling point}}{GR_i},$$

where GR_i is the rate in which the recording application generates multimedia files. The scheduler calculates the vulnerability of every client after uploading one file. The client with the most vulnerability at the time (i.e. it has the shortest time left before entering the critical state) is chosen. Next, the file with the highest priority (determined by the recording application of this client) is chosen to be uploaded. In other words, VBS gives the highest priority to the client with the highest risk of entering its critical state to upload its file first, which will prolong the entire system from entering its critical state.

3.2.2 Emergency Control Algorithms

When the server detects that some clients are in critical states, it alerts the human users and starts up the emergency control. Note that emergency control interrupts the system normal working process and may involve long and complicated processing. The emergency control algorithm is used only as a backup plan and should be invoked only when necessary.

The goal of emergency control is to rescue critical clients out of their critical states by moving some files out of these clients. Two possible destinations to which these files can be moved are either the server or non-critical clients. We choose to use non-critical clients as the moving destination to reduce the server load. Since the server has a global view of the system, the server is most suitable to execute the emergency control algorithm that determines a migration plan. The migration plan is a list of pairs of source and destination clients and the amount of the transfer data for each pair. Once the clients have received the migration plan from the server, the source and destination clients in each pair communicate to transfer files with the total transfer amount up to the amount specified in the migration plan. The server is not involved during the actual file transfer process.

We propose two emergency control algorithms. The *basic emergency control* algorithm aims to bring the system out of the critical state by moving as few files as required out of the critical clients. The *optimized emergency control* algorithm improves the basic control by moving a suitable amount of files out of the critical clients. The algorithm also reduces the number of times the system will re-enter its critical state. Both the emergency control algorithms accept the current client information as the input. The algorithms output a migration plan that indicates which client should transfer how many bytes of data to which client. The set of clients transmitting the files is called *OutSet*. The set of clients receiving the files is called *InSet*. Figure 2 shows an example of the migration plan for the basic emergency control algorithm. We provide the pseudo code for both algorithms in Table 1.

For the basic emergency control algorithm, the most suitable destination client (InSet client) should have both the largest available disk and the longest time left before reaching its critical state (indicated by the large value of the vulnerability). However, if using only the vulnerability as the sorting criterion, some less vulnerable clients with small available disk space may be selected as the destination client, which is not desirable. This case happens when these clients generate files at a slow rate. Hence, our sorting criterion takes into account both vulnerability and available disk space of the clients. To ensure that both values are in the range of $[0, 1]$, the values are normalized over the corresponding maximum value among all the clients, respectively. In Table 1, the values of both weights (w_1 and w_2) are also in the range of $[0, 1]$, and $w_1 + w_2 = 1$. Typically, we choose 0.5 for both weights to indicate that both factors are equally important. The basic emergency control algorithm has the disadvantage that it only rescues the critical client from the critical state but the client may still be close to the critical state. Besides, some non-critical clients may become very close to entering the critical state. Therefore, the system with the basic emergency control algorithm may enter an emergency situation again very quickly, which is not desirable.

The optimized algorithm calculates the desirable free disk space of the clients after the execution of the emergency control algorithm to delay the system from re-entering the critical state as long as possible. The desired system status after migration is that every client has the same vulnerability (i.e., the clients have about the same time left before entering their critical state). Otherwise, some client will enter its critical state quickly. Let AD_i and AD'_i be the available disk space of client i before and after migration, respectively. The total disk space in all clients before migration must be the same as that after migration (i.e., $\sum AD_i = \sum AD'_i$.)

Let GR_i be the file generation rate in Mbps of client i. The vulnerability of each client i after emergency control should

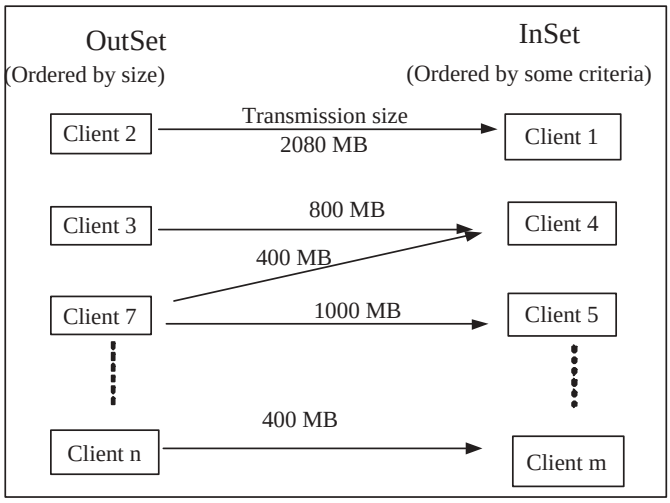


Figure 2. Example migration plan using the basic emergency control algorithm

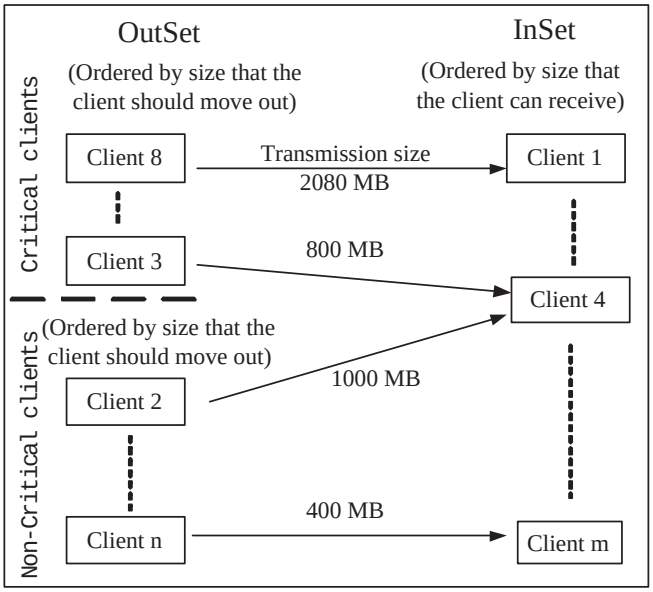


Figure 3. Example migration plan generated using the optimized algorithm

Table 1. Emergency control algorithms

BasicEmergencyControl() {
/* $OutSet[i].Size$: Approximated number of bytes if removed from client i in $OutSet$ can bring it out of the critical state
$InSet[i].Size$: Maximum number of bytes if received by client i does not cause it to enter its critical state
$InSet[i].Sort = w_1 * N\_Vulnerability_i + w_2 * N\_AD_i$, where w_1 and w_2 are weights;
$N\_Vulnerability_i$ is normalized vulnerability of client i;
N_AD_i is normalized available space at client i */

$OutSet$ includes every critical client;
$InSet$ includes every non-critical client;
Sort $InSet$ in the descending order of the values of the $Sort$ field of $InSet$;
Sort $OutSet$ in the descending order of the values of the $Size$ field of $OutSet$;
$MigrationPlan$ = ConstructMigrationPlan($OutSet$, $InSet$);
return $MigrationPlan$;
}

OptimizedEmergencyControl() {
/* AD_i: Available disk space at client i before migration;
AD'_i: Desired available disk space at client i after migration computed using Equation 2;
$OutSet[i].Size$: Approximated number of bytes if removed from client i can bring it out of the critical state;
$InSet[i].Size$: Maximum number of bytes if received by client i does not cause the client to enter its critical state; */

/* Consider moving files from critical clients to non-critical clients with enough desirable available disk space after migration */
$OutSet$ includes every critical client;
$TmpSet$ includes every non critical client i with $AD_i < AD'_i$;
$InSet$ includes every non-critical client i with $AD_i > AD'_i$;

$InSet[i].Sort = min(InSet[i].Size, |AD'_i - AD_i|)$ for every client i in $InSet$;
Sort $InSet$ in the descending order of the values of the $Sort$ field of $InSet$;
Sort $OutSet$ in the descending order of the values of the $Size$ field of $OutSet$;
$MigrationPlan$ = ConstructMigrationPlan($OutSet$, $InSet$);

/* Consider moving files from clients that are currently non-critical but have lower available disk space than the desired value */
$OutSet = TmpSet$;
Update AD_i of corresponding client i as if data are migrated according to $MigrationPlan$;
$InSet$ includes only every non-critical client i with the updated $AD_i > AD'_i$;
$InSet[i].Sort = min(InSet[i].Size, |AD'_i - AD_i|)$ for every client i in $InSet$;
$OutSet[i].Sort = max(OutSet[i].Size, |AD'_i - AD_i|)$ for every client i in $OutSet$;
Sort $InSet$ in the descending order of the values of the $Sort$ field of $InSet$;
Sort $OutSet$ in the descending order of the values of the $Sort$ field of $OutSet$;
$NewMigrationPlan$ = ConstructMigrationPlan($OutSet$, $InSet$);
Append $NewMigrationPlan$ to $MigrationPlan$;
return $MigrationPlan$;
}

Table 2. Algorithm for generating a migration plan

```
ConstructMigrationPlan(OutSet, InSet) {   /* Construct the migration plan */

  Initialize MigrationPlan to be empty;
  i = ID. of the first client in the sorted InSet;
  j = ID. of the first client in the sorted OutSet;
  Do
    tsize = min(InSet[i].Size, OutSet[j].Size);
    Add <src=i, dest=j, transmission_size=tsize> to MigrationPlan;
    InSet[i].Size = InSet[i].Size + tsize;
    OutSet[j].Size = OutSet[j].Size - tsize;
    If client j has no more files to transmit,
      j = ID. of the next client in OutSet;
    If client i has no more space to store the incoming files,
      i = ID. of the next client in InSet;
    If all the clients in InSet have been considered, but some critical clients still exist,
      break; /* the emergency control cannot help all critical clients */
  Until all the clients in OutSet have been considered;
  return MigrationPlan;
}
```

all be close to the optimal vulnerability, V_{op}, to reduce the number of times the emergency control is going to be invoked.

$$\begin{aligned} \sum AD_i &= \sum AD'_i = \sum (V_{op} \times GR_i) \\ \sum AD_i &= V_{op} \times \sum GR_i \end{aligned}$$

$$V_{op} = \frac{\sum AD_i}{\sum GR_i} \quad (1)$$

$$AD'_i = V_{op} \times GR_i. \quad (2)$$

We can compute V_{op} using Equation 1 and compute AD'_i from V_{op} using Equation 2. Figure 3 illustrates a migration plan using the optimized algorithm. The optimized emergency control algorithm is more complicated than the basic emergency control, but can reduce the number of times emergency control is executed.

3.3 Client Design

The client software is configured with the name of the directory to store the generated multimedia files and the maximum amount of storage space allocated for this purpose. Figure 4 shows our multi-threaded client with two important threads.

- **Disk monitoring thread** monitors the availability of the client disk space in the desired file system. If the client enters a critical state while the server is busy uploading a file from another client, the disk monitoring thread contacts the server to handle the emergency situation.

  Once the server is informed of the client emergency situation, the emergency control module is executed. The client waits for the server to distribute the migration plan. If the client belongs to InSet, it waits to receive files from other clients in parallel. If the client belongs to OutSet, it connects to the assigned corresponding client(s) in InSet and transmits the files up to the amount specified in the migration plan.

- **Uploading thread** uses Client Control (see Figure 4) to respond to the server request. In response to the server query request, the thread executes Query Handler (see Figure 4) that sends a message to the server indicating the available disk space, the file generation speed, and the size of the next file to be delivered. When the server requests the client to upload a file, the thread executes Uploader to transmit one media file. Once the entire file has been uploaded, the thread removes the file from the disk. The available space can be used to store a new file. Currently, the uploading thread uses Transport Control Protocol to transmit data. If security is a concern, a secure transport protocol can be used.

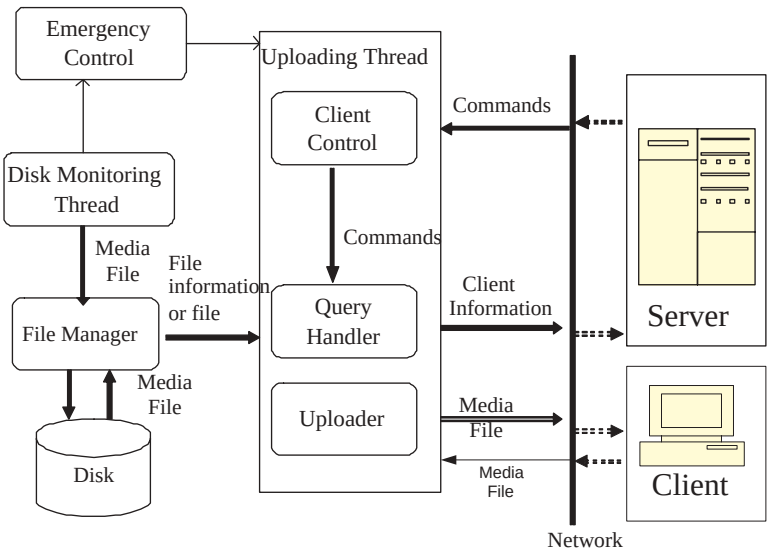


Figure 4. Client software architecture

3.4 Software Implementation

We implemented the server and the client software in Java according to our designs. Both Round-Robin and VBS scheduling algorithms as well as both emergency control algorithms were implemented. Additional implementation includes 1) allowing users to configure the server and client configurations through the graphical user interface of the server software; 2) letting users view statistics of server and clients through the server software interface; and 3) providing some fault tolerance. That is, when the connection between a client and the server is interrupted, the server removes this client from the client information list and logs this information in the server log file. When client comes back up, the uploading process will automatically resume from the last uploading status.

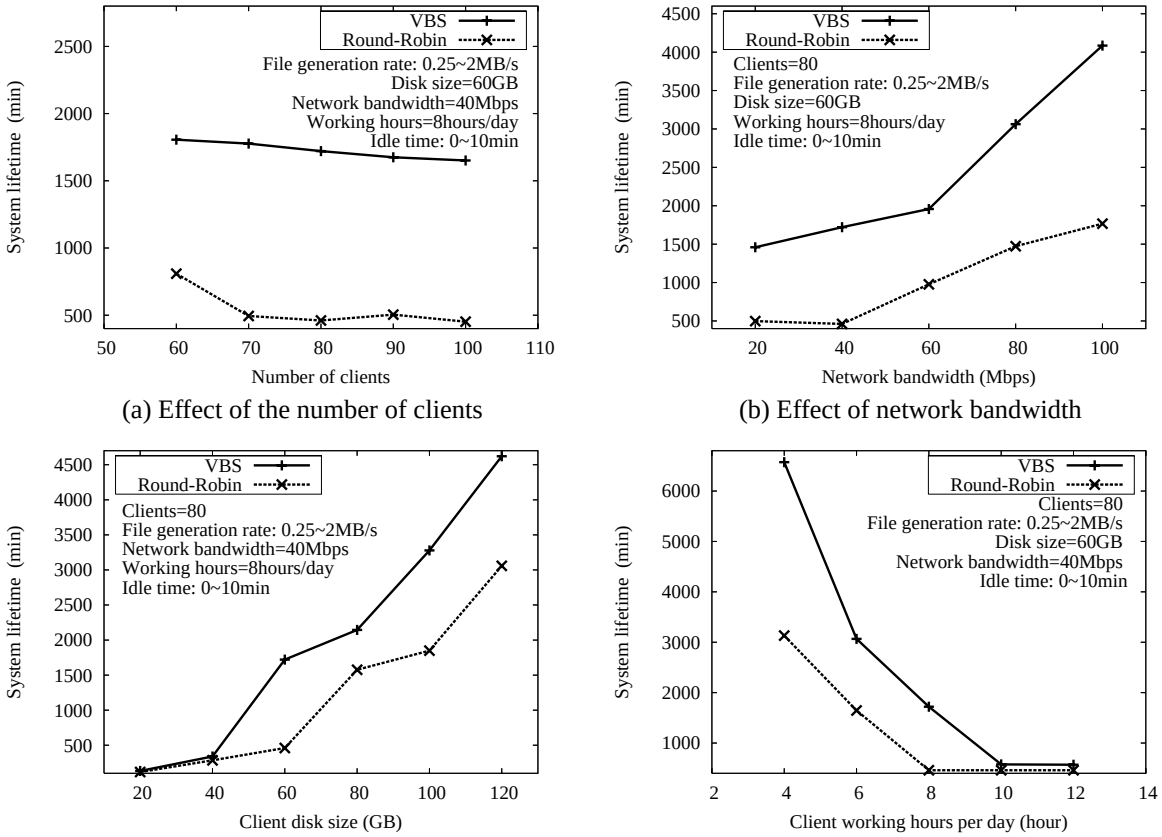
4 Performance Evaluation

In this section, we present our performance study using simulations. The performance metric is the system lifetime. First, we present performance comparison between Round Robin and VBS without emergency control. Then, we demonstrate the effectiveness of the Round Robin and VBS with the basic and the optimized emergency control algorithms.

4.1 Simulation Model

We implemented our simulator in C. The simulator simulated one server with unlimited disk capacity and a number of clients. We investigate the impact of the number of clients, the network bandwidth, the client disk capacity, and the client working hours on the uploading system. To clearly observe the effectiveness of the Round-Robin and Vulnerability-based scheduling algorithms, we measured the performance of the scheduling algorithms with and without the emergency control mechanisms separately. For the simulations without emergency control, we recorded the system lifetime when any of the clients entered its critical state for the first time. For the simulations with emergency control, we recorded the system lifetime when the system can no longer rescue any one of its client from entering its critical state. Each point in the plots is an average of results from 20 simulation runs under the same configurations but with different seeds for a random number generator. Clients have different storage space. We set the storage space to $MaximumDiskSize * (1 - r)$ GB where $MaximumDiskSize$ is the maximum disk space and r is a random number between 0 and 0.25.

As we intend to use our prototype for uploading videos captured from real colonoscopic procedures, we simulated the workload on each client to reflect that of the real scenario. We generated a video file size as a function of a file duration. The distribution of file durations followed a Zipf distribution with the smallest duration of 20 minutes and the largest duration of 45 minutes. The range of the durations reflected that of an actual colonoscopic procedure. Clients have file generation



(a) Effect of the number of clients

(b) Effect of network bandwidth

(c) Effect of client storage space

(d) Effect of client working hours

Figure 5. Round Robin vs VBS

rate of 0.5, 1.0, 1.5, and 2 MB/s. The same number of clients for the different file generation rates is used. All clients have same critical threshold. Recall that clients having available disk space below the critical threshold are in the critical state. For accurate simulation of colonoscopy, we simulate an idle time between two consecutive file generation as a random number between 0 to 10 minutes.

4.2 Simulation Results

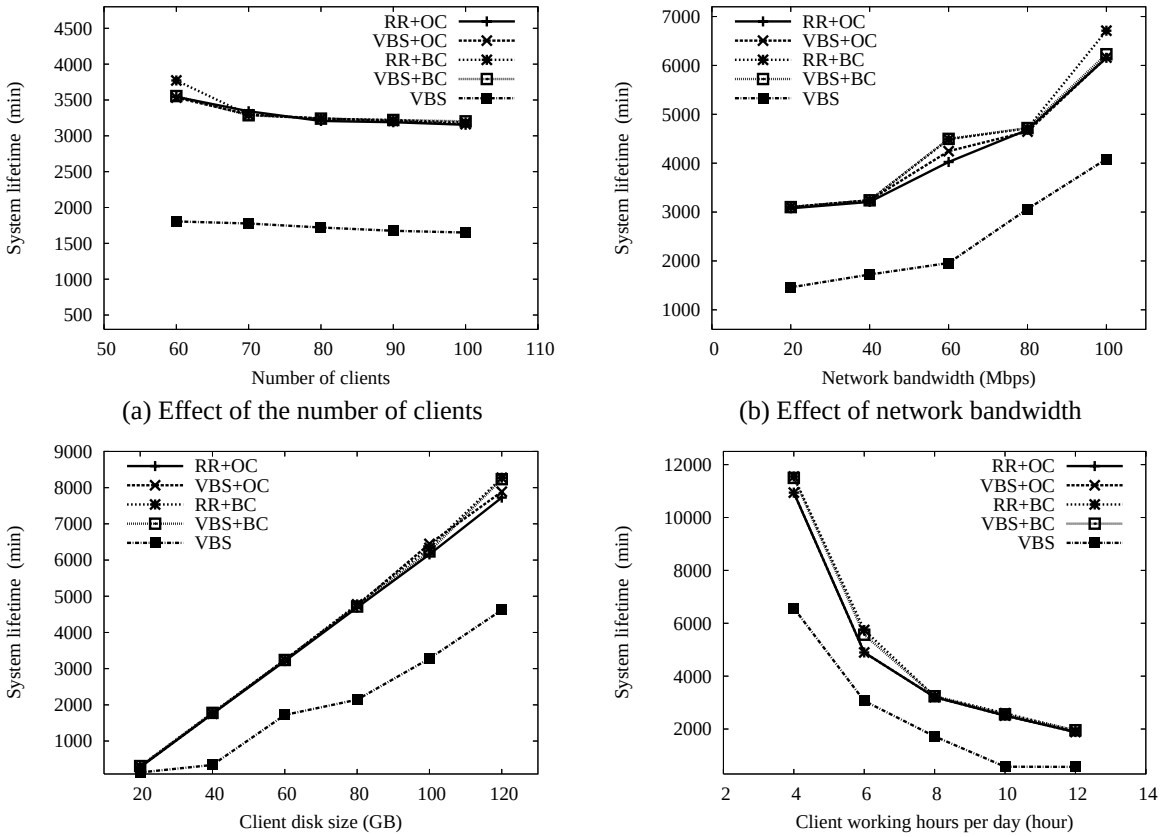
4.2.1 Performance of Scheduling Algorithms without Emergency Control

Figure 5 shows the performance comparison between VBS and Round-Robin. We can see clearly that VBS consistently has higher system lifetime than Round-Robin under various workloads. The system using VBS runs twice longer than that using Round-Robin on average.

The system lifetime when using both scheduling algorithms becomes smaller with more clients as shown in Figure 5(a). The more clients there are, the more data are generated in the system in one unit time, which increases the amount of the data to be uploaded to the server. Figure 5(b) shows that as the network bandwidth increases, the system using either scheduling algorithm enters its critical state later as the time to upload a file is shorter. Similar trend is observed when the client disk space increases as depicted in Figure 5(c). Larger disk space allows the client to store more files, which decreases the possibility that the system will enter the critical state. Figure 5(d) shows that the increase in client working hours results in smaller system lifetime since more data are generated.

4.2.2 Performance of Scheduling Algorithms with Emergency Control

Figure 6 shows that both the basic and the optimized emergency control algorithms consistently increase the system lifetime under various situations compared with VBS without emergency control. The uploading system using either emergency con-



(a) Effect of the number of clients

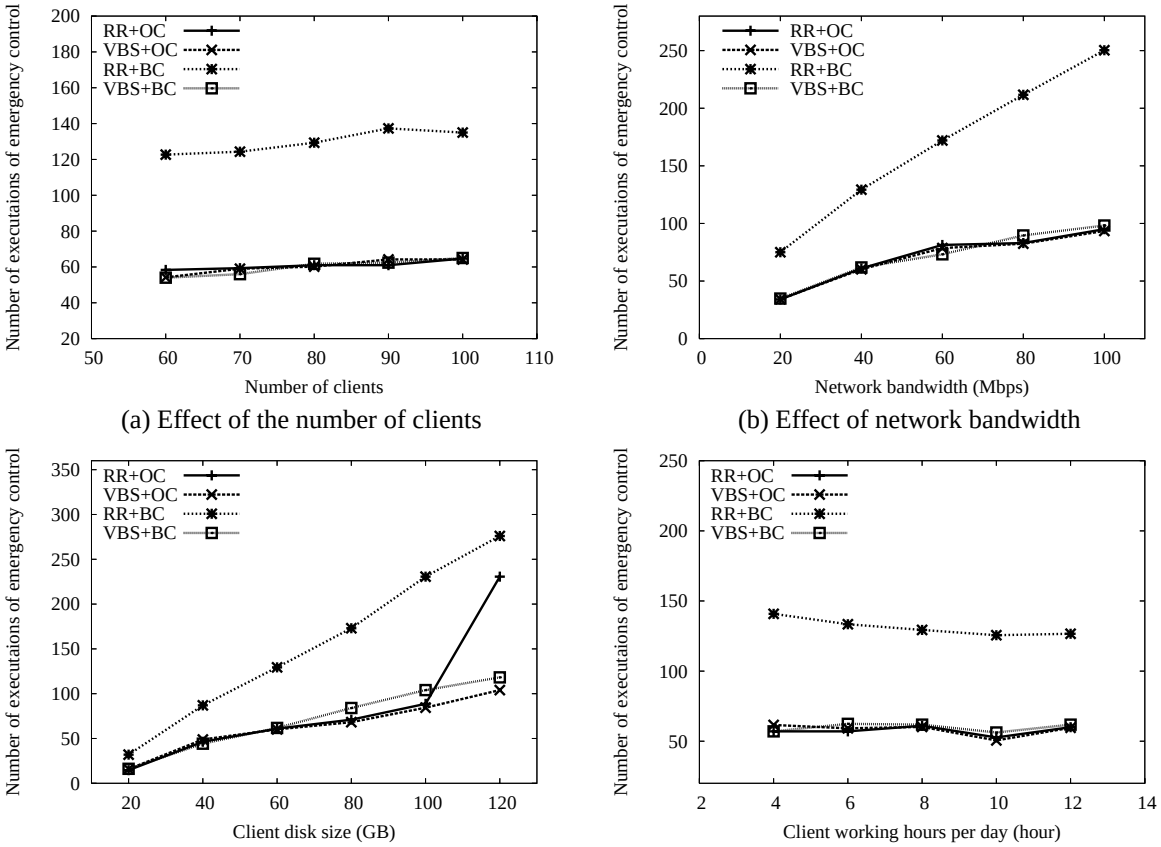
(b) Effect of network bandwidth

(c) Effect of client storage space

(d) Effect of client working hours

Parameters:
(a) 60 GB client disk space; 40 Mbps network bandwidth; 8 working hours per day
(b) 80 clients; 60 GB storage space; 8 working hours per day
(c) 80 clients; 8 working hours per day; 40 Mbps network bandwidth
(d) 80 clients; 60 GB client storage; 40 Mbps network bandwidth

Figure 6. Effect of the emergency control algorithms. RR-Round Robin. VBS-Vulnerability Based Scheduling. BC-Basic Control. OC-Optimized Control.



(a) Effect of the number of clients

(b) Effect of network bandwidth

(c) Effect of client storage space

(d) Effect of client working hours

Parameters:
(a) 60 GB client disk space; 40 Mbps network bandwidth; 8 working hours per day
(b) 80 clients; 60 GB storage space; 8 working hours per day
(c) 80 clients; 8 working hours per day; 40 Mbps network bandwidth
(d) 80 clients; 60 GB client storage; 40 Mbps network bandwidth

Figure 7. Comparison of the number of executions of the emergency control algorithms. RR-Round Robin. VBS-Vulnerability Based Scheduling. BC-Basic Control. OC-Optimized Control.

trol algorithm enters its critical state much later than the system without emergency control. Regardless of which emergency control algorithm and which server scheduling algorithm are used, the uploading system with emergency control performs equally well (i.e., the four top lines (RR+OC, VBS+OC, RR-BC, and VBS-BC) are overlapped in each of the plots in Figure 6). The emergency control algorithms lengthen the system lifetime more when used with Round-Robin than when used with VBS. This is because VBS and the emergency controls actually have the same goal, which is to balance the disk space utilization between clients, though they achieve this goal in different ways. The system using VBS scheduler is more likely to have balanced disk space utilization among clients while the one using Round-Robin does not. Therefore, the emergency control has less effect on VBS than on Round-Robin. The great performance of the Round-Robin with the emergency control indicates that balancing the clients' disk space utilization is a good way to prolong the system lifetime.

Figure 7 shows the number of executions of emergency control under various combinations of the two scheduling algorithms and the two emergency control algorithms. The optimized emergency control algorithm reduces the number of times the algorithm is invoked more than the basic algorithm does, which satisfies our design goal for emergency control. It also confirms the same fact we discussed above. The improvement is dramatic for the Round Robin scheduler but is small for VBS.

In summary, the best combination of scheduling algorithm and the emergency control is VBS with Basic Emergency

Table 3. Notations

Symb.	Explanation	Unit
N	Number of clients the system supports	
gr	File generation rate	Mbps
wt	Client working duration per day	Sec.
bw	Effective network bandwidth	Mbps
fs	Average file size	Mbit
d_{gen}	Average time taken to generate one file	Sec.
d_{rnd}	Average time between two consecutive uploads by the same client	Sec.
d_{xf}	Average time taken to transfer one file	Sec.
$S_i(t)$	Cumulative amount of data generated at a client since the system starts until time t	Mbit
$S_o(t)$	Cumulative amount of data a client has uploaded since the system starts until time t	Mbit
S	Total storage space at a client	Mbit
S_{th}	Client critical threshold	
T	The time when the system enters its critical state	
$f_{work}(t)$	The function that returns the actual working duration given a time period t	Sec.
Δt	Time period a client has to wait before beginning its first upload	Sec.

Control or VBS with Optimized Emergency Control. Because they offer similar good performance in terms of system lifetime and the number of times the emergency control algorithm is executed.

5 Performance Analysis of Media Uploading Systems

Our performance analysis focuses on the performance impact of the two scheduling algorithms on the uploading system excluding emergency control. The dynamic characteristic of emergency control makes the performance analysis mathematically intractable.

We use a *system life time* defined as the total time since the system starts until the system enters the critical state as the performance metric. A good scheduling algorithm should prolong the system from entering its critical state. Note that other performance metrics for downloading applications such as latency and system throughput are not suitable for evaluation of the uploading system.

We present a system performance prediction analysis of Round Robin. It is not possible to quantify the performance difference between Round Robin and VBS via analysis because the order of file uploads changes according to the vulnerability of the clients in VBS. However, we can qualitatively prove that VBS outperforms Round Robin scheduling by showing that any uploading schedule handled by Round-Robin can be handled by VBS, but not vice versa. We give the idea of the proof in section 5.2. In the following analysis, we assume that each client generates data at the same constant rate and has the same storage space.

5.1 Performance Analysis of Round Robin

In this section, we compute the system life time under the Round Robin algorithm given a system configuration. Table 3 summarizes the notations used in this analysis.

To estimate when a client will enter its critical state, we need to model the amount of data stored at the client disk at any time t since the system starts. This amount ($\Delta S(t)$) is the difference between $S_i(t)$ and $S_o(t)$, where $S_i(t)$ is the amount of data created by the file generation application at time t and $S_o(t)$ is the amount of data removed from the client disk once the data has been uploaded at time t. Figure 8 depicts $\Delta S(t)$ over time.

$$\Delta S(t) = S_i(t) - S_o(t) \quad (3)$$

$$S_i(t) = gr \times f_{work}(t), \text{ where} \quad (4)$$

$$f_{work}(t) = \lfloor \frac{t}{60 \times 60 \times 24} \rfloor \times wt + min(t \,\%\, (60 \times 60 \times 24), wt)$$

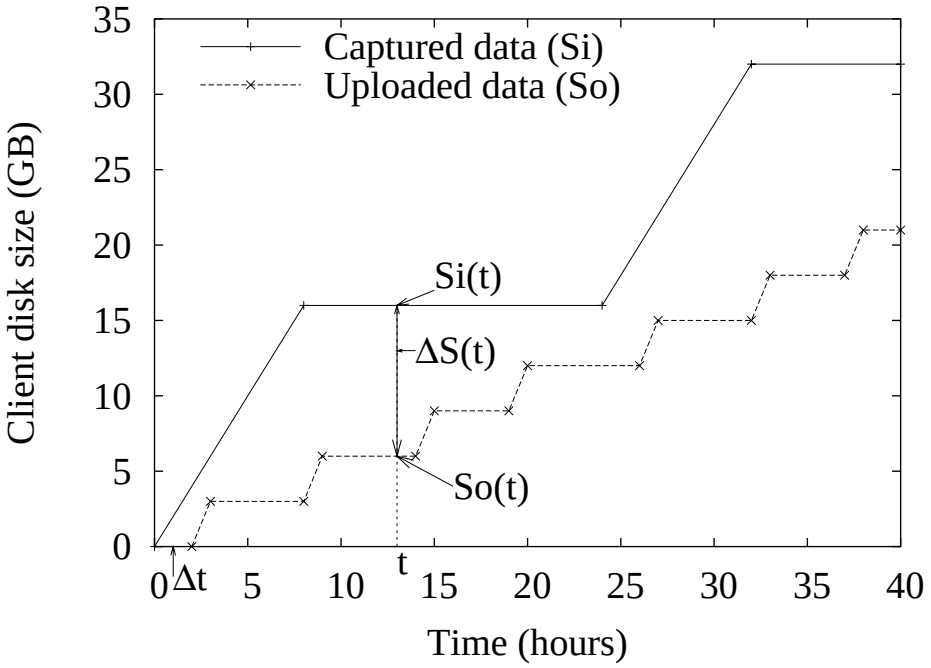


Figure 8. Amount of data in a client disk at time t

In other words, at time t seconds since the system has started, the file generation application has been generating data for $\lfloor \frac{t}{60\times60\times24} \rfloor$ days and $min(t \,\%\, (60 \times 60 \times 24), wt)$ seconds in the last day. For each day, the data are generated for wt seconds at the rate of gr Mbps.

$$S_o(t) = bw \times \left[\lfloor \frac{t - \Delta t}{d_{rnd}} \rfloor \times d_{xf} + min((t - \Delta t)\%d_{rnd}, d_{xf}) \right] \tag{5}$$

The client cannot start its upload until its first file has been generated, which is assumed to be Δt seconds since the system has started. Within $t - \Delta t$, the client has uploaded $\lfloor \frac{t-\Delta t}{d_{rnd}} \rfloor$ files and some $min((t - \Delta t)\%d_{rnd}, d_{xf})$ fraction of a file. On average, the amount of the data in each file is $bw \times d_{xf}$ Mbps.

To gain a conservative estimation of the system life time, we choose the upper bound of $S_i(t)$ and lower bound of $S_o(t)$. Therefore, equations (4) and Equation (5) can be modified as

$$S_i(t) = gr \times (\frac{t \times wt}{60 \times 60 \times 24} + wt) \tag{6}$$

$$S_o(t) = bw \times d_{xf} \times \frac{t - \Delta t}{d_{rnd}} \tag{7}$$

We estimate that $\Delta t \in [d_{gen}, d_{gen} + d_{rnd}]$ since for the first client scheduled for uploading, Δt is as short as the time it takes for the client to finish generating one file (d_{gen}), but for the last client scheduled for uploading, Δt is as long as the time taken to generate one file and the wait time for all the other clients to upload. To restrict the estimation, we estimate Δt as $\Delta t = d_{gen} + d_{rnd}$ We estimate d_{gen} as the expected duration to generate one file.

$$d_{xf} = \frac{fs}{bw} = \frac{d_{gen} \times gr}{bw} \tag{8}$$

$$\begin{aligned} d_{rnd} &= d_{xf} \times (N - 1) + d_{xf} \\ &= N \times d_{xf} \\ &= N \times \frac{d_{gen} \times gr}{bw} \end{aligned} \tag{9}$$

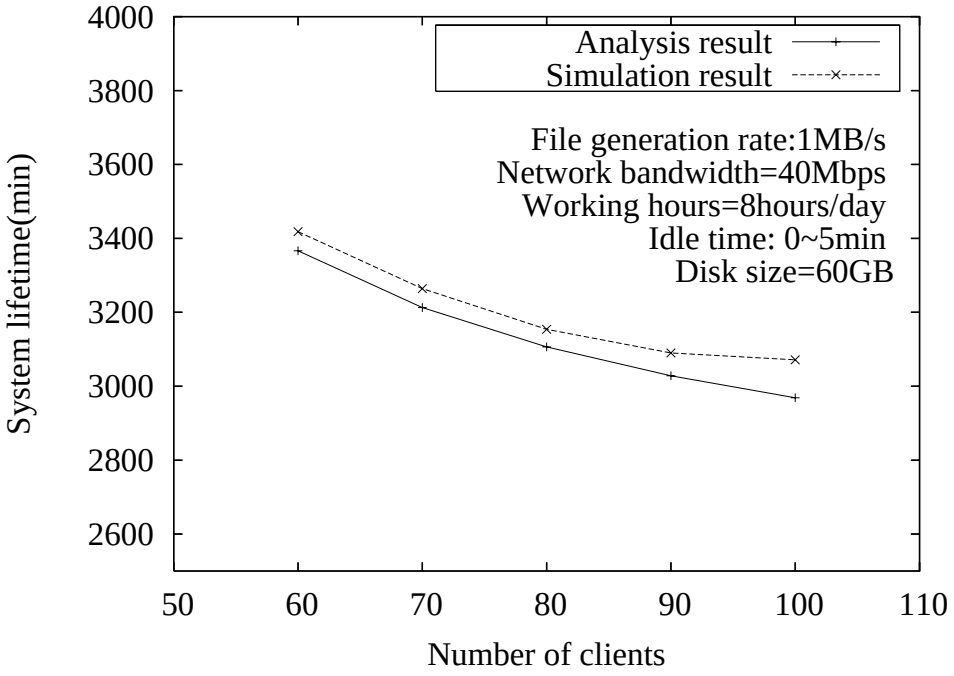


Figure 9. Comparison between analysis result and simulation result

After we substitute (8) and (9) into Equation (7), we have

$$S_o(t) = \left[t - d_{gen} - N \times d_{gen} \times \frac{gr}{bw}\right] \times \frac{bw}{N})$$

Suppose the system life time is T. In other words, the client enters its critical state at time T, we have $\Delta S(T) \geq S - S_{th}$. Now, we have

$$\begin{aligned} \Delta S(T) &= gr \times (\frac{t \times wt}{60 \times 60 \times 24} + wt) - \frac{(T - d_{gen} - \frac{d_{gen} \times gr \times N}{bw}) \times bw}{N} \\ &\geq S - S_{th} \end{aligned} \tag{10}$$

After some arithmetic manipulations, we have

$$T \geq \frac{S - d_{gen} \times \frac{bw}{N} - d_{gen} \times gr - S_{th} - gr \times wt}{\frac{wt \times gr}{60 \times 60 \times 24} - \frac{bw}{N}} \tag{11}$$

Since T is the time when the system enters its critical state, we can use (11) to estimate the system life time when we have a configuration of the system that includes number of clients, estimated net work speed, estimated generating rate and etc. Figure 9 shows the comparison between the analysis result and the simulation result. Analysis result gives a conservative estimation of the system life time.

5.2 Performance Analysis of VBS

The dynamic characteristic of VBS requires the performance analysis to be based on comparison between VBS and Round Robin. The proof that VBS outperforms Round Robin consists of two parts. The first part shows that VBS is as good as Round Robin whereas the second part demonstrates via an example that Round Robin is not as good as VBS. Table 4 summarizes the notations used in the proof.

Proof:

We call a schedule a *feasible schedule* if no clients enter their critical stage when files are uploaded according to the schedule.

Table 4. Notations

Symb.	Explanation	Unit
C_i	Client i	
c	Number of clients	
S_j	Task (one file upload by a client) scheduled at the j^{th} scheduling point	
$AD_{i,j}$	Available disk space at client i at the j^{th} scheduling point	Mbit
GR_i	File generation rate of client i	Mbps
$V_{i,j}$	Vulnerability of client i at the j^{th}scheduling point $V_{i,j} = AD_{i,j}/GR_i$	Sec.
fs	Average file size	Mbit
d_{xf}	Average time taken to transfer one file	Sec.
t_i	Time at the i^{th} scheduling point	Sec.

Table 5. Relationships among scheduling points and the corresponding times, tasks scheduled at the scheduling points, and clients performing the scheduled tasks

Scheduling point	0	...	c-1	c	c+1	...	n	...
Time	t_0	...	t_{c-1}	t_c	t_{c+1}	...	t_n	...
Task scheduled	S_0	...	S_{c-1}	S_c	S_{c+1}	...	S_n	...
Client chosen at this point	C_0	...	C_{c-1}	C_0	C_1	...	$C_{n\%c}$	...
Vulnerability of the client	$V_{0,0}$	...	$V_{c-1,c-1}$	$V_{0,c}$	$V_{1,c+1}$	...	$V_{n\%c,n}$	...

Part I: To demonstrate that VBS is as good as Round-Robin, a feasible schedule arranged by Round-Robin must also be rearranged to a feasible schedule using VBS.

Let $S = S_0, S_1, \ldots, S_n$ be a feasible schedule in Round Robin where S_k is the task of uploading one file at the k^{th} scheduling point. We compute the vulnerability of each client at each scheduling point, in which a file is transmitted to the server. The time between two consecutive scheduling points is the time taken to transmit one file (represented by $d_{xf} = t_{i+1} - t_i$). In VBS, we compute the vulnerability $V_{i,j}$ of client i at scheduling point j as $V_{i,j} = AD_{i,j}/GR_i$. In Round-Robin, the vulnerability is not actually used for scheduling, but it is computed in this analysis to indicate the client status. Table A.3 shows scheduled tasks of a schedule at different scheduling points at different times, vulnerability at the scheduling points, and participating clients. Note that at the k^{th} scheduling point, the corresponding task is S_k performed at time t_k; the corresponding client is $C_{k\%c}$ with vulnerability $V_{k\%c,k}$.

We assume that the file generation process at each client is continuous, which implies that the vulnerability at each scheduling point indicates the actual situation at the client. Let $f_i(k, j)$ be a function that returns the number of times client C_i is selected to upload files between any two scheduling points k and j where $j > k$ excluding the scheduling points k and j. The vulnerability of client i at these two scheduling points, $V_{i,j}$ and $V_{i,k}$, are

$$V_{i,j} = AD_{i,j}/GR_i \tag{12}$$

$$AD_{i,j} = AD_{i,k} - GR_i \times (t_j - t_k) + fs \times f_i(k, j) \tag{13}$$

$$\begin{aligned} V_{i,j} &= \frac{AD_{i,k} - GR_i \times (t_j - t_k) + fs \times f_i(k, j)}{GR_i} \\ &= \frac{AD_{i,k}}{GR_i} - (t_j - t_k) + \frac{fs \times f_i(k, j)}{GR_i} \\ &= V_{i,k} - (j - k) \times d_{xf} + \frac{fs \times f_i(k, j)}{GR_i} \end{aligned} \tag{14}$$

Equation (13) shows that during the period between the two scheduling points of the same client, the available disk space at this client is affected by the generation of new files and the upload and removal of existing files. Once we obtain the relationship between the available disk space at the two scheduling points, we can derive the relationship between the vulnerability at the two points as shown in Equation (14).

Now, we show that a feasible schedule in Round Robin can be rearranged to a feasible schedule in VBS. Assume that

$S = S_0, S_1, \ldots, S_n$ is a feasible schedule using Round-Robin, i.e., $V_{i,j} > 0, \forall i \in [0, c-1], \forall j \in [0, n]$. We modify this schedule to a schedule in VBS by repeatedly rearranging the positions of the tasks at each scheduling point. The rearranging process at each scheduling point is as follows. Without loss of generality, we consider an upload task S_k at the k^{th} scheduling point performed by client $C_{k\%c}$. Two possible situations are as follows.

- Case 1: If the client $C_{k\%c}$ is the most vulnerable one ($V_{k\%c,k}$ is the smallest), S_k should be chosen at this point by VBS. Hence, S_k is already in the correct position in the schedule. No rearrangement is necessary.
- Case 2: If the client $C_{k\%c}$ is not the most vulnerable one, but client C_i is, where $i \neq k\%c$. In other words, $V_{i,k}$ is the smallest at the scheduling point k. The corresponding task S_j where $j\%c = i$ should be selected at this point by VBS. That is, S_j is the upload task with the highest priority at client C_i. We rearrange the schedule by moving S_j in front of S_k. In other words, we change the original Round Robin schedule to a new schedule as follows.

  Original schedule: $S_{k-1}, S_k, S_{k+1}, \ldots, S_{j-1}, S_j, S_{j+1}, \ldots$

  New schedule: $S_{k-1}, S_j, S_k, S_{k+1}, \ldots, S_{j-1}, S_{j+1}, \ldots$

  We need to prove that the new schedule is still feasible. First, we infer the following characteristics about S_j where $j\%c = i$.

  1. The relationship between the two scheduling points j and k is that $j > k$. If $j < k$, S_j should have already been executed; $j \neq k$ is obvious because $C_{j\%c}$ and $C_{k\%c}$ represent different clients.
  2. In the original schedule, the vulnerability of other clients during the scheduling points k and j must be larger than zero since the original schedule is assumed a feasible schedule. Since $V_{i,k}$ is the smallest, at the scheduling point k, the vulnerability of other clients must be larger than the vulnerability of the client C_i.
  3. In the original schedule, C_i is not selected to upload its file until the scheduling point j according to Case 2. Hence, we can infer that $f_i(k, j) = 0$. From the Round Robin scheduling algorithm, this feature indicates the distance between S_j and S_k in the original schedule is less than the total number of clients ($j - k < c$).
     - $j - k \neq c$ is obvious because $j\%c \neq k\%c$.
     - If $j - k > c$, there must be a full round between k and j, which indicates C_i has been selected at least once before scheduling point j. This contradicts to the inference $f_i(k, j) = 0$.

     Hence, we know all the clients between scheduling points k and j (C_l, $l = m\%c$, $k < m < j$,) is selected at most once between scheduling points k and j. Because C_l is not selected until scheduling point m, we have $f_l(k, m) = 0$.

  Now, we prove that the vulnerability of all the clients remain in a safe range after the rearrangement. Using Equation (14) and $f_l(k, m) = 0$ where $l = m\%c$, $k \leq m \leq j$, we have the following conclusions.

  - Obviously, client C_i ($i = j\%c$) becomes less vulnerable than that before the rearrangement since this client is scheduled to upload its file earlier in the new schedule.
  - For the other clients C_l where $l = m\%c,\ k \leq m < j$, their vulnerability is increased by the rearrangement but no more than the vulnerability at the scheduling point j in the original schedule. To demonstrate this fact, we examine the values of $V_{l,x}, \forall x$ where $k \leq x \leq m+1$, $m+1 \leq j$, and $l = x\%c$.

$$
\begin{aligned}
V_{l,x} &= V_{l,k} - (x-k) \times d_{xf} && (from\ A.13 \text{ where } f_l(k,x) = 0) \\
&> V_{i,k} - (x-k) \times d_{xf} && (V_{i,k} \text{ is the smallest}) \\
&= V_{i,k} + ((k-j) + (j-x)) \times d_{xf} \\
&= (V_{i,k} - (j-k) \times d_{xf}) + (j-x) \times d_{xf} \\
&= V_{i,j} + (j-x) \times d_{xf} && (from\ A.13) \\
&\geq V_{i,j} > 0 && (j > x) \qquad (15)
\end{aligned}
$$

From Equation (15), the new schedule is also feasible. Repeating the rearranging process at each scheduling point in the original schedule results in a schedule similar to that determined by the VBS scheduler.

Our proof shows that the new schedule in VBS is also feasible. Hence, we conclude that VBS is as good as Round-Robin scheduling.

Part II: To show that Round Robin is not as good as VBS, we provide an example of upload tasks that Round Robin cannot handle, but VBS can create a feasible schedule.

Suppose the client selected at the k^{th} scheduling point is client C_i, and the most vulnerable client is client C_j, where $j \neq i$ and $j \neq (k+1)\%c$ and $j \neq (k+2)\%c$. That is, Round Robin will not select C_j in the next two scheduling points. Suppose that the vulnerability of C_j at scheduling point $k+1$ can be described as $d_{xf} < V_{j,k+1} < 2 \times d_{xf}$.

VBS will select client C_j to upload at the scheduling point $k+1$. The vulnerability of C_j will increase afterwards. Round-Robin will not select C_j at least until the server finishes the transmission of one file from $C_{(k+1)\%c}$ and another file from $C_{(k+2)\%c}$, which takes $2 \times d_{xf}$. At that time, C_j has already entered its critical stage. Hence, Round Robin cannot handle this situation whereas VBS still can.

In summary, the Vulnerability Based scheduling algorithm outperforms the Round-Robin scheduling algorithm

6 Concluding Remarks

We have presented a new uploading system that collects large media files from a number of clients given deadline constraints. The detailed design and performance analysis of such a system have not been previously studied in the literature. Our design uses the client-server architecture. We propose two scheduling algorithms and two emergency control schemes. Our simulation results show that Vulnerability-based scheduling consistently outperforms Round Robin scheduling. The two emergency controls help prolong the system running time more dramatically.

Our future work investigates solutions that provide security and privacy for multimedia file uploading. We plan to extend the uploading system for applications in other network environments such as uploading surveillance videos from police vehicles in wireless ad hoc networks where the vehicle may move out of the server transmission range.

References

[1] B. Bhattacharjee, W. C. Cheng, C.-F. Chou, L. Golubchik, and S. Khuller. Bistro: a Platform for Building Scalable Wide-Area Upload Applications. *Performance Evaluation Review*, 28(2):29–35, September 2001.

[2] Y. Cao, D. Li, W. Tavanapong, J. Wong, J. Oh, and P. C. de Groen. Parsing and browsing tools for colonoscopy videos. In *Proc. of ACM Multimedia'04*, pages 844–851, New York, USA, 2004.

[3] L. Cox and B. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *Proc. of the ACM Symposium on Operating Systems Principles*, October 2003.

[4] J. da Silva and O. Guomundsson. The amanda network backup manager. In *Proceedings of USENIX Systems Administration (LISA VII) Conference*, pages 171–182, November 1993.

[5] Gnutella. Open Source Community. Gnutella. In *http://gnutella.wego.com/*, 2001.

[6] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[7] K. A. Hua, M. A. Tantaoui, and W. Tavanapong. Video delivery technologies for large-scale deployment of multimedia applications. *Proceedings of the IEEE*, 92(9):1439–1451, 2004.

[8] IBM. IBM Bolsters Police Fleet with "In Car" Digital Video Technology. In *http://www-1.ibm.com/solutions/digitalmedia/doc/content/news/pressrelease/942113122.html*, May 2003.

[9] KaZaA. KaZaA file sharing network. In *http://www.kazaa.com/*, 2002.

[10] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *Proc. of USENIX Annual Technical Conference*, San Antonio, Texas, USA, June 2003.

[11] C. L. Liu and J. W. Layland. Scheduling algorithm for multiprogramming in a hard real time environment. *Journal of ACM*, 20:46–61, January 1973.

[12] Napster. Napster Inc. The napster homepage. In *http://www.napster.com/*, 2001.

[13] Sun Microsystems Inc. NFS: network file system protocol specification. rfc–1094. March 1989.

[14] U.S. National Science Foundation. Fastlane. In *https://www.fastlane.nsf.gov*, 1994.

[15] M. Zhang, J. Wong, W. Tavanapong, J. Oh, and P. C. de Groen. Media uploading systems with hard deadlines. In *Proc. of IASTED Int'l Conf. on Internet and Multimedia Systems and Applications*, pages 305–310, Hawaii, USA, August 2004.