View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Institutional Knowledge at Singapore Management University

Singapore Management University
Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems School of Information Systems

5-2000

Load Sharing in Distributed Multimedia-on-Demand Systems

Y. C. TAY
National University of Singapore

Hwee Hwa PANG
Singapore Management University, hhpang@smu.edu.sg

DOI: https://doi.org/10.1109/69.846293

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Databases and Information Systems Commons, and the Numerical Analysis and Scientific Computing Commons

Citation

TAY, Y. C. and PANG, Hwee Hwa. Load Sharing in Distributed Multimedia-on-Demand Systems. (2000). *IEEE Transactions on Knowledge and Data Engineering*. 12, (3), 410-428. Research Collection School Of Information Systems.
Available at: https://ink.library.smu.edu.sg/sis_research/113

This Journal Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

Load Sharing in Distributed Multimedia-on-Demand Systems

Y.C. Tay and HweeHwa Pang

Abstract—Service providers have begun to offer multimedia-on-demand services to residential estates by installing isolated, small-scale multimedia servers at individual estates. Such an arrangement allows the service providers to operate without relying on a high-speed, large-capacity metropolitan area network, which is still not available in many countries. Unfortunately, installing isolated servers could incur very high server costs, as each server requires spare bandwidth to cope with fluctuations in user demand. In this paper, we explore the feasibility of linking up several small multimedia servers to a (limited-capacity) network, and allowing servers with idle retrieval bandwidth to help out servers that are temporarily overloaded; the goal is to minimize the waiting time for service to begin. We identify four characteristics of load sharing in a distributed multimedia system that differentiate it from load balancing in a conventional distributed system. We then introduce a GWQ load sharing algorithm that fits and exploits these characteristics; it puts all servers' pending requests in a global queue, from which a server with idle capacity obtains additional jobs. The performance of the algorithm is captured by an analytical model, which we validate through simulations. Both the analytical and simulation models show that the algorithm vastly reduces wait times at the servers. The analytical model also provides guidelines for capacity planning. Finally, we propose an enhanced GWQ+L algorithm that allows a server to reclaim active local requests that are being serviced remotely. Simulation experiments indicate that the scheduling decisions of GWQ+L are optimal, in the sense that it enables the distributed servers to approximate the performance of a large centralized server.

Index Terms—Multimedia-on-demand, distributed servers, load sharing, performance modeling.

---

1 Introduction

In recent years, there has been a great deal of interest in multimedia-on-demand (MOD) systems, as evidenced by the flurry of research publications and MOD trials. While the basic technologies are already in place, one important factor that hampers the large scale commercial deployment of MOD is the prohibitive cost of building the underlying high-speed distribution network infrastructure.

Instead of waiting for telephone and cable operators to set up this network infrastructure, service providers have started to install MOD systems at localized communities. The size of such a community usually ranges from 50 to a couple of hundred households, so that a single cable provides sufficient bandwidth for sending programming from the server to all of the households in the community. For example, IPC Interactive Group is offering movie-on-demand and home shopping services to four condominium blocks, each with over 300 units, at a residential estate in Singapore.

While maintaining many small-scale MOD installations eliminates the reliance on an extensive high-speed, large-capacity network infrastructure, there are certain associated disadvantages. Besides administrative overheads, the most important drawback is the higher server costs: It is well known in queuing theory that, for a given number of queue servers[1], having several independent queues produces worse (average) response times than having a common queue (e.g., [47]). Thus, the aggregate capacity of many small, isolated MOD servers must be higher than the capacity required of a single, centralized MOD server in order to avoid performance degradation. The cost escalation due to the increased capacity of the isolated servers could be very substantial, as MOD servers typically scale well up to a certain point, beyond which the servers must be upgraded to a higher range of hardwares, e.g., from SPARC servers to mainframes.

A promising way to contain system cost is to link up several MOD servers to a network as in Fig. 1. This creates a distributed system of loosely coupled servers where (some) objects are replicated, and hence can be retrieved from alternative servers depending on their respective load levels. An MOD server with idle retrieval capacity can then help to service remote requests from another server that is temporarily overloaded.

Admittedly, this distributed configuration would necessitate high-speed networks between every server and its local terminals, and between servers. However, only terminals that are supported remotely generate traffic on the interserver network; most of the terminals are still expected to be serviced by their respective host servers, and hence will only load the local network. Each network thus needs to accommodate only a couple of hundred concurrent streams, which is very much less than the hundreds of

- *Y.C. Tay is with the Department of Mathematics, National University of Singapore, Kent Ridge, Singapore 119260, Republic of Singapore. E-mail: tay@acm.org.*
- *H.-H. Pang is with Kent Ridge Digital Labs, 21 Heng Mui Keng Terrace, Singapore 119613, Republic of Singapore. E-mail: hhpang@acm.org.*

1. The number of queue servers corresponds to server capacity in our context, *not* the number of MOD servers. Henceforth, we shall refer to the former explicitly as queue servers, using the term "servers" for MOD servers.

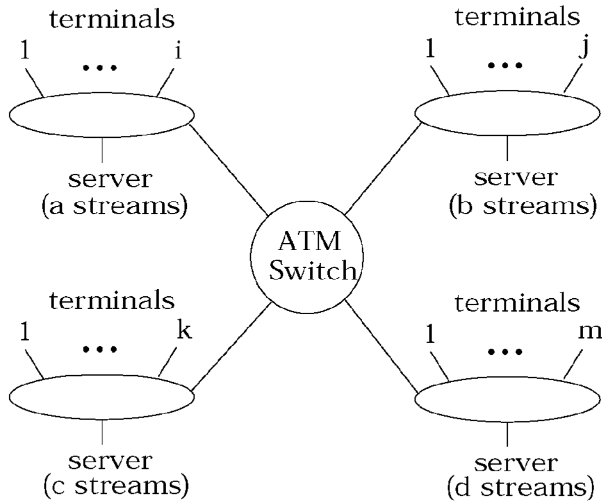

Fig. 1. Distributed multimedia system.

thousand streams that a centralized, gigantic MOD system (Fig. 2) might entail. In other words, the configuration in Fig. 1 allows a service provider like the IPC Interactive Group to reduce costs by using smaller servers and a small switch.

Several networks with enough bandwidth to enable the distributed configuration either are already operational, or are being installed right now. An example is the recently announced SingaporeONE, a national network infrastructure that will be the channel for delivering the multimedia services envisioned in the Singapore IT2000 plan (`www.s-one.gov.sg`). This infrastructure consists of a broadband network, based on ATM switching and optical fibre technologies, that connects up local access networks in businesses, schools, and homes on the island. Applications planned for include distance learning, government services, electronic banking, and digital libraries.

The purpose of this study is to develop a way to overcome the penalty in response time (i.e., waiting time for service to begin) for a multimedia system consisting of small servers linked together. Our contributions are as follows: We first identify the differentiating characteristics for such systems, then design a *Global Wait Queue* (GWQ) algorithm for load sharing that fits and exploits these characteristics. This algorithm maintains all servers' waiting requests in a global queue, from which servers with idle retrieval capacity obtain additional jobs.

An analytical model for the algorithm is also presented. The model serves two functions: 1) It can estimate the performance for a network design in a matter of seconds,

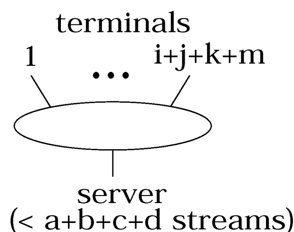

Fig. 2. Centralized multimedia system.

compared to simulation which could take many hours. For example, it enables us to calculate, for a given network bandwidth, what gains load sharing might bring about. 2) It can provide closed-form guidelines on the server capacity, network size and communication bandwidths for a design. For instance, it can estimate the network bandwidth necessary for the distributed servers to approximate the centralized server, and also identify whether link or switch bandwidth is the bottleneck in a network design. The estimations and guidelines from the analytical model are confirmed by a simulator that we developed. Those estimations indicate that the GWQ algorithm utilizes the bandwidth of every server effectively, which in turn produces very substantial reductions in wait time.

Besides GWQ, we also introduce a *Global Wait Queue + Localize* (GWQ+L) version that enables a server to take over its requests that are being serviced remotely. Since the analytical model is not capable of capturing these dynamic changes, the performance gains from the GWQ+L algorithm are quantified by the simulator. Experiments show that the GWQ+L version lowers wait time even further, enabling the distributed servers to approximate the performance of a large centralized server. This confirms that the job assignments generated by GWQ+L are optimal.

The remainder of this paper is organized as follows. Section 2, summarizes the existing work on load sharing, and identifies the unique characteristics of distributed multimedia systems that form the framework for our work. The next section introduces two algorithms to share the workload among multimedia servers. In Section 4, we derive an analytical model to capture the performance of the GWQ algorithm. The following section begins by describing the simulator, then proceeds to present a number of experiments that highlight the gains that load sharing brings about. Section 6 concludes with a review of our contributions.

2 The Load Sharing Problem

Load sharing in distributed systems has to satisfy two often conflicting objectives. On one hand, we would like to maximize *performance* by exploiting all available servers. Usually, the extent to which this can be achieved depends on the amount of information that the servers exchange to keep track of each other's load level. On the other hand, we would like to keep the load sharing algorithms as efficient as possible, which means keeping a cap on the scheduling *overhead* caused by information exchange.

In cases where the total mix of processes in a distributed system is known in advance, a static execution schedule can be generated offline to meet both of the above objectives. Studies that address those cases include [6], [8], [10], [14]. However, in many cases information on the workload is not available a priori, and the scheduling algorithm has to work online and dynamically strike a trade-off between the two conflicting objectives. As a result, most online algorithms employ heuristics and generate suboptimal schedules. Examples include [1], [3], [7], [12], [15], [27], [34], [41], [43], [48]. A few rare instances where optimal schedules are guaranteed are [17], [31], [46]. [5], [42], [48] provide

excellent introductions to the principles underlying those load sharing solutions.

While the insight from and the solutions for conventional distributed systems provide a valuable foundation for our work, distributed multimedia systems possess at least four unique characteristics that warrant a different load sharing algorithm:

C1. Local before Remote. From a load sharing perspective, the most important distinction of a distributed multimedia system is that servicing a job out of a remote server incurs service overheads, i.e., it consumes extra resources (e.g., bandwidth) while the job is active. This is because a multimedia object retrieval involves pulling data frames from a server, and sending them over a network to a display terminal. Hence, serving a job remotely produces traffic over the interserver network (typically a MAN or a WAN), in addition to network traffic on the LAN that hooks up the display terminal. Consequently, a multimedia system needs to be careful about assigning jobs to remote servers. While some conventional distributed system solutions [14], [41] account for communication overhead, it is not as important a consideration there because executing jobs do not interact heavily with the initiating terminals like in multimedia systems.

Moreover, if the servers belong to different organizations, then the latter's primary business clients are their respective communities, and servicing remote requests is only a secondary source of revenue. In other words, local requests again have precedence.

C2. Sharing without Balancing. In a conventional distributed system, it makes sense to balance load by shifting jobs from a heavily-loaded server to a lightly-loaded one because those jobs can finish faster after the shift. Jobs in a multimedia system, however, must be serviced at fixed rates. For example, an MPEG movie should be played back at 30 frames/sec. If the system assigns insufficient resources to a job, it would not achieve the required quality, i.e., play back would be jittery. However, the system cannot service the job faster either—even if additional resources are available—because data frames that have yet to be displayed would accumulate and eventually overflow the buffer. Consequently, a system that is perfectly balanced does not necessarily deliver better quality of service than an unbalanced one where none of the component servers are overloaded. For example, a scheme that freely directs jobs to remote servers for the sake of load balancing is misguided, since this violates characteristic C1. Moreover, once a server begins to serve a remote request, service quality guarantees may mean that it cannot simply preempt the job when a new local request arrives; thus, local requests may be delayed by remote requests. To accommodate characteristics C1 and C2, a multimedia system should only direct a job to a remote server if the local host is already operating at full capacity (regardless of how unbalanced load is at the servers).

In other words, jobs in a conventional distributed system have no predetermined service rate, so are able to complete faster if more resources are available. Hence, a balanced system performs better than an unbalanced one, and that is why existing algorithms for such systems are load *balancing* solutions, e.g., [3], [6], [12], [15], [27], [48]. Those algorithms are not desirable for a multimedia system as they incur unnecessary balancing overheads in trying to even out the workload.

C3. Performance over Overhead. Multimedia object retrievals can last anywhere from several seconds (e.g., commercials) to a couple of hours (e.g., movies). A multimedia system should therefore be able to effectively assign an incoming job to a remote server that has spare bandwidth, as otherwise the job may have to wait for an unacceptably long time until the local server completes one of its existing jobs. Furthermore, the scheduling overhead (in contrast to the service overhead in C1) for load sharing is negligible relative to the huge amounts of system resources that multimedia jobs consume. These two factors mean that performance rather than scheduling overhead is the primary issue: A multimedia system must aim for—and can afford the scheduling overheads necessary to achieve—load sharing that is optimal (i.e., minimizes waiting time).

On the other hand, jobs in a conventional distributed system are usually less resource-intensive, making the scheduling overhead comparable to job duration, so there has to be a trade-off between performance and scheduling overhead. This is why most of the existing algorithms, e.g. [1], [3], [8], [12], [15], [34], are heuristics-based and generate suboptimal schedules.

C4. Relocation is Easy. Systems that accept ad hoc requests can often improve their performance by relocating executing jobs according to changing load condition. Job relocation is relatively straightforward in a multimedia system—it needs only to identify an alternate server that holds the target multimedia object, then transfer from the current server to the new server the display terminal address and the playback location within the object. It does not matter whether the two servers run the same operating system, or whether they have the same hardware and software configuration, as the multimedia object is coded in some standard format like MPEG or MJPEG. The load sharing algorithm of a multimedia system should therefore be designed to exploit the feasibility of job relocation.

In contrast, job relocation in a conventional system is much less practical. Even with the help of process migration facilities proposed in [2], [26], [28], [45], relocating an executing process is still extremely difficult if the servers run different operating systems, or if the process has instantiated variables that are server-specific. For this reason, with a few exceptions like [1], [4], existing load sharing algorithms for conventional systems generally do not exploit job relocation.

This paper focuses on the performance issues arising from the challenge of load sharing in distributed multimedia systems. Where appropriate, we will discuss the implications on other issues like stream synchronization, but we will not attempt to provide detailed solutions to those problems here. Instead, we refer the interested reader

to several previous studies that addressed such issues in implementing a distributed multimedia system. One of these problems is devising suitable specification models for schemes that support the presentation and communication of multimedia objects. This was the subject of the studies by Lin et al. [23]. The runtime integration and synchronization of packets/messages, arriving from various data sources, that constitute a multimedia object is another challenging problem. Works that investigated this problem include [24], [36], [37], [38], [44], [51]. Yet another group of studies examined the role of operating system support for distributed multimedia systems: Leslie et al. [22] proposed the use of a shared address space between different machines, while CPU scheduling, disk scheduling, and flow control were discussed in [16] and [49].

3 Cooperative Algorithms for Load Sharing

In this section, we shall introduce a pair of decentralized load sharing algorithms, GWQ and GWQ+L, that are tailored for the characteristics (C1–C4) of distributed multimedia systems. Instead of attempting to balance load across all servers (C2), the algorithms will assign a job to a remote server only if the local server is fully loaded, in order to avoid unnecessary network traffic flows from remote servers to display terminals (C1). The algorithms will also optimize their scheduling decisions as much as possible, so that the distributed multimedia system can perform like a centralized, gigantic system (C3). Moreover, the GWQ+L algorithm will exploit opportunities to relocate executing jobs back to their local servers, to reduce the demand for network bandwidth (C4).

3.1 Centralized Versus Decentralized Models

The load sharing algorithm of a distributed system can adopt either a centralized or a decentralized model. A centralized model mandates that a single coordinator controls all job assignments. All of the servers have to report their load status to the coordinator. In arranging for a remote job execution, the servers communicate with the coordinator, rather than with each other. In contrast, all servers participate as equals in the load sharing protocol in a decentralized model, and each server can communicate with every other server.

We base our algorithms on a decentralized model for the following reasons: First, a decentralized model is more robust as there is no single point of failure. Second, coordination overhead in a decentralized model is lower since servers need only to exchange messages when there is a job that needs a remote server, rather than to send updates to the coordinator every time they accept or complete a job—this is significant because only a minority of the jobs are expected to be served remotely. Third, servers may have individual admission controls based on nontrivial local information (e.g., job mix), thus making a centralized load sharing model infeasible. Finally, a decentralized model enables a server to choose which other servers to help out with, and what (kind of) jobs to accept, thus preserving site autonomy. Retaining control over the individual servers is especially important in practice if they belong to different vendors.

3.2 GWQ Load Sharing Algorithm

Under the *Global Wait Queue* (GWQ) algorithm, the servers in a distributed multimedia system maintain a global queue of pending requests. When a new request arrives at a server, it will service the request locally if there is idle retrieval capacity; if not, the server immediately notifies the remote servers to insert the request in their wait queues. This effectively puts the request on a global queue — hence, the name GWQ.

The first priority of every server is to service the requests from its host community (C1). A server loans its excess capacity to a remote community's requests in the global queue only if there are no waiting local requests. Each request is serviced by the same server throughout its lifetime; i.e., the algorithm does not review its scheduling decisions once service begins. (We relax this condition later for the GWQ+L algorithm.)

In GWQ, a server that has spare bandwidth may bid for a remote job. In practice, the bidding price is likely to include server usage costs and network charges. For example, if the entire system is owned by a single vendor, the bidding price may be based on the amount of resources required to run the job from the remote server; if the system has multiple vendors, the bidding price may involve some prior charging agreement among them.

3.2.1 Algorithm Definition

The algorithm is depicted in the asynchronous communicating finite state automata (FSA) [13] in Fig. 3. In the FSA, a state transition involves a server reading a nonempty string of messages addressed to it, writing a string of messages, and moving on to the next state. For example, in the FSA on the left, the local server reads a request `REQ1`, outputs `RFS2`, .., `RFSn`, then changes state from `S1` to `W1`. The change from one state to the next state is atomic and instantaneous. Moreover, state transitions at one server are asynchronous with respect to transitions at other servers. The FSA for the local server has three states: a *start* state (`Si`), a *wait* state (`Wi`), and a *commit* state (`Ci`).

The FSA for the remote servers has a *hold* state (`Hi`) and an *abort* state (`Ai`), in addition to the three states for the local server. The difference between `Hi` and `Wi` is that in the former, the server is postponing a decision until some internal information becomes available, whereas in `Wi` the server is waiting for responses from other servers. `Ci` and `Ai` are final states, indicating that the request has been assigned to a server or withdrawn from a server. When that happens, the server will notify other remote servers to remove that request from their wait queues.

3.2.2 Algorithm Implementation

The implementation of GWQ requires each server to maintain two data structures—a `LocalRequest` queue for retrieval requests that are generated locally, and a `RemoteRequest` queue for requests received from other servers. The algorithm consists of a series of actions that each server executes in response to an event:

- Event: Local server in state `S1` receives a multimedia object request `REQ1.` Actions:

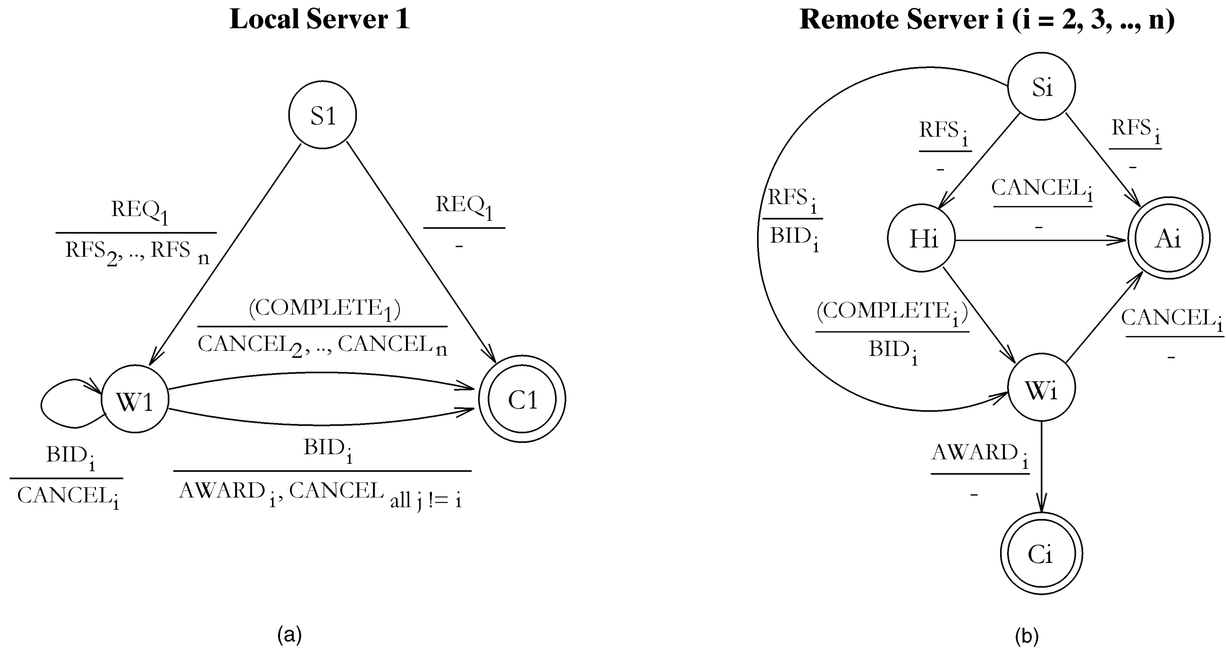

Fig. 3. Finite state automata for load sharing algorithm.

```
if (server has the required bandwidth)
                    /* service request locally */
    reserve the bandwidth;
    change to state C1;
else
              /* seek help from remote servers */
      insert the request in the LocalRequest
      queue; broadcast an RFS message to
      remote servers; change to state W1;
```

Note: `RFS` stands for Request-For-Service. Besides the identity of the object requested and the address of the requester, the `RFS` also indicates the maximum price that is acceptable to the local server.

- Event: Remote server in state `Si` receives an `RFSi` message. Actions:

```
if ((requested object is not available) or
  (maximum acceptable price is too low))
                         /* ignore RFS message */
  change to state Ai;
else
if (server has the required bandwidth)
                         /* bid for the request */
  reserve the bandwidth;
  send a BIDi message to the
    requesting server;
  change to state Wi;
else
               /* wait till there is bandwidth */
  insert the RFS in the RemoteRequest queue;
  change to state Hi;
```

Note: The `BID` includes a bidding price, which may be up to the maximum acceptable price.

- Event: Local server in state `W1` receives a `BIDi` message. Actions:

```
if (bidding price is satisfactory)
                                   /* accept the bid */
  log the award transaction;
  remove the request from the
  LocalRequest queue;
  send back an AWARDi message;
  broadcast a CANCELj message to the other
  remote servers j;
  prepare to receive multimedia
  object stream from server i;
  change to state C1;
else
                                   /* reject the bid */
  send back a CANCELi message;
  change to state W1;
```

Note: The requesting server is not obligated to accept a bid immediately even though the bidding price is within the maximum acceptable price. The server could wait a prespecified duration, then select the lowest bid and reject the rest.

- Event: Remote server in state `Wi` receives an `AWARDi` message. Actions:

```
log the award transaction;
change to state Ci;
start streaming the multimedia object
  to the (remote) terminal;
```

- Event: Remote server receives a `CANCELi` message. Actions:

```
if (server is in state Wi)
        /* has sent in a bid for the request */
   same actions as for the request completion
   event below;
   change to state Ai;
else
if (server is in state Hi)
   remove the request from the RemoteRequest
    queue;
   change to state Ai;
else
  /* server has ignored the RFS previously */
   ignore the CANCEL message;
```

- Event: Server receives a request completion message `COMPLETE` (from itself). Actions:

```
   free the bandwidth reserved for the
   completed request;
if ((there are requests in the
     LocalRequest queue) and
    (server has the required bandwidth))
                              /* start as many */
   reserve the bandwidth;
                          /* local requests as */
   remove request from LocalRequest queue;
                          /* bandwidth allows */
   broadcast a CANCELi message to
     remote servers;
   change state from W1 to C1
     (for this request);
else
if ((there are RFS's in the RemoteRequest
     queue) and
   (server has the required bandwidth))
                            /* bid for as many */
    reserve the bandwidth;
                         /* remote requests as*/
    remove RFS from the RemoteRequest queue;
                          /* bandwidth allows */
    send a BIDi message to the requesting
      server;
    change state from Hi to Wi (for this RFS);
else
    idle;
```

Since the local and remote servers run asynchronously, a server may receive a message that is not expected for the current state. For example, the local server may send out `CANCEL` messages and transition from state `W1` to state `C1` after deciding to service a request locally, even as a remote server is sending in a bid for the same request. While our simulator (described in Section 5.1) includes provisions to deal with these "unexpected" cases, we shall not delve into them so as to avoid obscuring the main logic of the algorithm.

3.3 GWQ+L Load Sharing Algorithm

While the GWQ algorithm attempts to reduce network usage by assigning new requests to local servers whenever possible, there may still be "cross-service" situations where a number of servers are attending to each other's requests concurrently. Here is one such situation: 1) Server X is busy, so it awards a request to server Y. 2) A new request arrives at server Y, which is busy now. 3) Server X finishes a request. Since it has no pending local requests, it bids for and is awarded the waiting request from server Y. 4) Servers X and Y are now serving each other's request over the network.

Ideally, all servers should be attending to their own requests, rather than incurring network charges by engaging in "cross-services" (C1). Since a server cannot foresee what (local) requests it will receive when it is deciding whether to bid for a remote request, "cross-services" have to be removed by revising the remote server assignment decision *dynamically* during the lifetime of a request (C4). This is the motivation for our second algorithm.

The *Global Wait Queue + Localize* (GWQ+L) load sharing algorithm is designed to prevent "cross-services" by having each server take over local requests that have been awarded to remote servers. This entails an enhancement to the algorithm presented earlier: When a server completes a request, it will first attend to waiting local requests as before. However, if there is excess capacity after that, the server will now take over those active local requests that are being serviced remotely. A server will bid for remote requests only if it has no pending local requests, and if none of its active requests are being supported by remote servers.

To take over service for a client, the server first obtains from the client its current object stream position and the identity of the remote server. The two servers then agree on a switch-over time when the remote server will terminate service and the local server will continue the object stream. To ensure that the switch-over is seamless, the clock at the two servers have to be synchronized. Clock synchronization is a well-studied problem and is addressed in [9], [11], [19], [20], [33], [35]. If the client receives its data frames on a UDP socket, the switch-over is transparent as the local server will send data to the same socket. In a TCP-based set-up, the client has to be coded to switch to a new server connection in the midst of playing back a stream. Other implementation issues (quality-of-service, buffering, resource scheduling, etc.) are addressed in a separate paper [32].

While the above modification is straightforward, a system developer may want to ascertain the resulting performance gain, in terms of shorter wait time and reduced network usage, before implementing the algorithm. One of the objectives of our study is to quantify this performance gain. We shall achieve this with the aid of simulation, as our analytical model does not capture the dynamic switches produced by the algorithm.

4 Analytical Model

Having introduced the load sharing algorithms, we now present an analytical model of the GWQ algorithm. Devising an accurate analytical model for GWQ is challenging for three reasons: 1) the diversion of an arriving request to a global queue when it cannot be served locally implies that the queues do not form a separable network [21], so the many results and algorithms for such networks

TABLE 1
Parameters of the Analytical Model

Parameter	Meaning	Default
Network:		
N_{server}	Number of servers in the network	-
B_{switch}	Bandwidth of the ATM switch	2 Gbps
B_{link}	Bandwidth of ATM link in each direction	155 Mbps
Server:		
N_{stream}	Maximum number of streams that a server can retrieve concurrently	50
N_{term}	Number of terminals attached to each server	90
Workload:		
T_{sleep}	Idle time	mean 7200 seconds, exponential dist.
T_{active}	Duration of each object retrieval	uniformly dist. in [3600, 10800] seconds
B_{stream}	Bandwidth of each object stream	15 Mbps
Performance:		
T_{wait}	Wait time for stream to begin after a request is issued	-
ρ	Server utilization	-

are not helpful; 2) load sharing is meaningful only when queue servers have high utilization, but model accuracy is hard to achieve under such conditions; 3) load sharing hinges on *temporary* imbalances in workload, making the application of *steady state* modeling techniques nontrivial.

We will use the model to understand the resource interplays, as well as the performance trade-offs involved in load sharing (C3). The model can also serve as a system configuration tool to size the servers and the network when the algorithm is implemented. We first analyze in Section 4.1 an isolated server and the capacity it must have, then model the load sharing in Section 4.2. Next, Section 4.3 uses the model to draw some conclusions about the appropriate size of a network, including the necessary bandwidths.

In deriving the model, we will assume that each object is replicated at every server. While it is not always possible or desirable in practice, and while it is *not* a prerequisite for the load sharing algorithms, providing for full replication enables us to arrive at the resource requirements needed to fully exploit any load sharing opportunities at run-time. The parameters of the model and performance measures are summarized in Table 1.

4.1 Sizing a Server: Model for an Isolated Server

In standard queuing notation, $A/S/m/B/J$ is a queue with type A arrival process, type S service time distribution, m servers, buffer size B and job population J [18]. Consider now arbitrary distributions for the sleep and active times at an isolated server; then, the server can be modeled as a $G/G/N_{\text{stream}}/N_{\text{term}}/N_{\text{term}}$ queue, i.e., a closed system with N_{term} terminals, and a queue with N_{stream} servers and N_{term} buffer slots for pending requests. Let λ be the throughput (requests per unit time). By Little's Law,

$$N_{\text{term}} = \lambda(T_{\text{sleep}} + T_{\text{wait}} + T_{\text{active}}), \text{ and}$$
$$\rho = \frac{\text{number of requests being served}}{N_{\text{stream}}} = \frac{\lambda\, T_{\text{active}}}{N_{\text{stream}}}.$$

(CPU time makes a negligible contribution in the equation for N_{term} because the terminals, rather than the servers, are responsible for decoding the streams.) It follows that

$$\rho = \frac{N_{\text{term}}}{N_{\text{stream}}}\frac{T_{\text{active}}}{T_{\text{sleep}} + T_{\text{wait}} + T_{\text{active}}}. \quad (1)$$

A multimedia stream like a karaoke song typically has $T_{\text{active}} \approx 240$ seconds; documentaries and movies last even longer. In contrast, users would tolerate only small wait times, say $T_{\text{wait}} < 10$ seconds. We therefore expect $T_{\text{wait}} << T_{\text{active}}$ in practice. From (1), we then get:

Server Utilization.

$$\rho \approx \frac{N_{\text{term}}}{N_{\text{stream}}}\frac{T_{\text{active}}}{T_{\text{sleep}} + T_{\text{active}}}. \quad (2)$$

This gives a good estimate of server utilization for a system designer. The approximation is robust because it is not conditioned upon specific arrival and service distributions. If the RHS exceeds one, then one can expect very high T_{wait}s. Therefore, the RHS should be less than one, giving:

Capacity Requirement.

$$N_{stream} > \frac{T_{active}}{T_{sleep} + T_{active}} N_{term}. \quad (3)$$

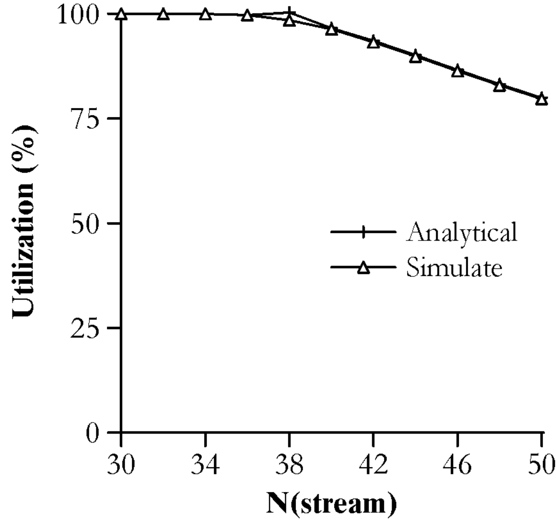

Fig. 4. Server utilization.

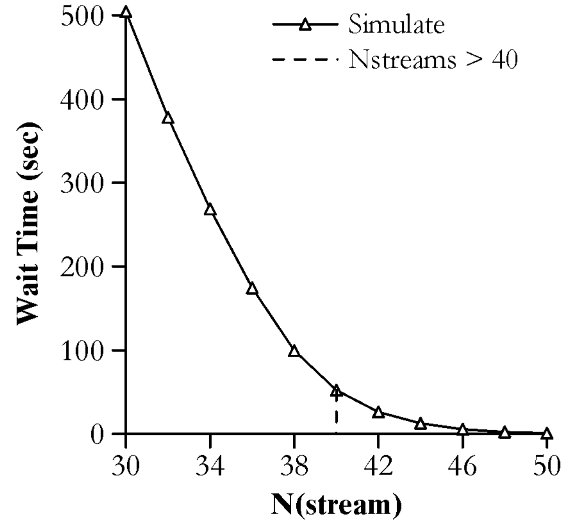

Fig. 5. Wait time.

If this inequality is violated, then one can again expect T_{wait} to be unacceptably high. Thus, from the characteristics of the multimedia objects (T_{active}), customer profile (T_{sleep}) and system configuration (N_{term}), a system designer can determine beforehand the minimum number of streams that the server must support to deliver reasonable performance.

The validity of (2) and (3) are confirmed in a wide range of experiments. Fig. 4 and Fig. 5 plot the server utilization and capacity requirement, respectively, for one of the experiments. The figures are generated by the simulator to be described in the next section, using the workload described in Section 5.3 (N_{term} = 200, T_{sleep} is exponentially distributed with a mean of 1,200 seconds, and T_{active} is exponentially distributed with a mean of 300 seconds). For this workload, (3) yields a requirement of $N_{\text{stream}} > 40$. When this condition is met, the utilization values from (2) and the simulator are almost identical, as Fig. 4 shows. For $N_{\text{stream}} < 40$, the *estimated* ρ is (slightly) higher than 100 percemt, indicating that the wait time is probably unacceptable. Indeed, Fig. 5 confirms that T_{wait} exceeds a minute and rises rapidly as N_{stream} goes below 40. The other experiments that we did used different settings and distributions for N_{term}, T_{sleep}, and T_{active}.

4.2 Model for Load Sharing

We now model the case where several multimedia servers are connected to an ATM switch (Fig. 1) and share their load by using the GWQ algorithm to match idle remote capacity to queued requests. We assume that the servers are homogeneous, i.e., they have the same N_{term}, T_{active}, etc., as in the example of IPC's multimedia servers at the condominium (see Section 1).

We overcome the three modeling difficulties, enumerated at the beginning of Section 4, via two key observations from numerous simulations: 1) load sharing does not significantly affect the server utilization ρ although, of course, it affects the waiting time; and 2) the effect of load sharing on a server's performance is similar to that of adding more capacity to that server.

In 1), it is not surprising that waiting time can change although server utilization is unaffected—a similar effect suggests service counters (in banks, etc.) should use a common queue instead of individual queues. To understand why ρ is approximately constant, recall from (1) that an isolated server has utilization

$$\frac{N_{\text{term}}}{N_{\text{stream}}}\frac{T_{\text{active}}}{T_{\text{sleep}}+T_{\text{wait}}+T_{\text{active}}}.$$

If load sharing is perfect, so that the network behaves like a single server with $N_{\text{server}}N_{\text{term}}$ terminals and capacity of $N_{\text{server}}N_{\text{stream}}$, then the utilization is

$$\frac{N_{\text{server}}N_{\text{term}}}{N_{\text{server}}N_{\text{stream}}}\frac{T_{\text{active}}}{T_{\text{sleep}}+T'_{\text{wait}}+T_{\text{active}}}$$

where T'_{wait} is the waiting time under load sharing. Since $T'_{\text{wait}} < T_{\text{wait}}$, utilization is bounded by

$$\frac{N_{\text{term}}}{N_{\text{stream}}}\frac{T_{\text{active}}}{T_{\text{sleep}}+T_{\text{wait}}+T_{\text{active}}} \le \rho \le \frac{N_{\text{term}}}{N_{\text{stream}}}\frac{T_{\text{active}}}{T_{\text{sleep}}+T'_{\text{wait}}+T_{\text{active}}}.$$

Since $T_{\text{wait}} << T_{\text{active}}$, we have, like (2),

$$\rho \approx \frac{N_{\text{term}}}{N_{\text{stream}}}\frac{T_{\text{active}}}{T_{\text{sleep}}+T_{\text{active}}},$$

i.e., server utilization is approximately constant under any load sharing scheme. Hence, we can estimate the server utilization for any load sharing algorithm simply by calculating ρ for an *isolated* server. This observation is the starting point for our model which, following our other observation 2) captures the load sharing by adding capacity to each server.

We will make several approximations, and we will demonstrate in Section 5 that the resulting accuracy is still

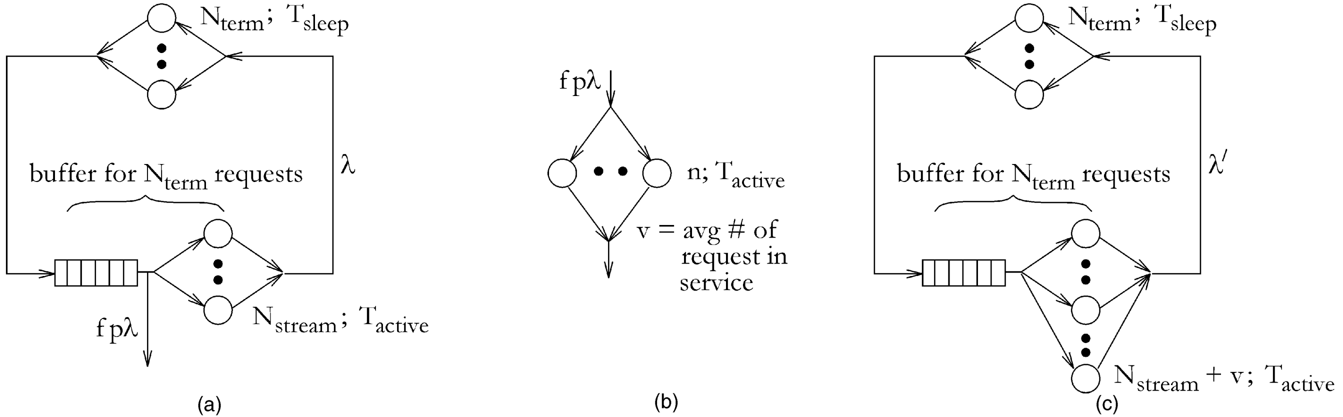

Fig. 6. Three steps in the analytical model. (a) Step 1: M/M/N_{stream}/N_{term}/N_{term} (model for isolated server). (b) Step 2: M/M/n/n (model for remote service). (c) Step 3: M/M/N_{stream}+v/N_{term}/N_{term} (model for load sharing).

acceptable. The model is derived from the perspective of one of the servers, and consists of three steps. While we make use of the memoryless property in these steps, it is not essential; it could be replaced with appropriate alternative distributions for the request arrivals and service times, as long as there is a calculator, or even simulator, to measure the desired quantities. In fact, for our experiments, we use the p (see below) calculated for memoryless distributions even for cases where the active time is uniformly distributed. Despite that, the estimations from the model are not compromised; this illustrates the robustness of the model. The three steps in our model are:

Step 1. $M/M/N_{\text{stream}}/N_{\text{term}}/N_{\text{term}}$ (Fig. 6a).
Measure throughput λ and probability p that the server's capacity is fully allocated (i.e., in queuing terminology, all N_{stream} queue servers are busy).

Step 2. $M/M/n/n$ (Fig. 6b).
Use $fp\lambda$ as the arrival rate for a queue that represents the available remote capacity of n streams, where f is a calibration factor and n is determined by the link and switch bandwidths, and the number and utilization of remote servers. (We will explain how f and n are calculated shortly.) Measure v, the average number of requests in service at this $M/M/n/n$ queue.

Step 3. $M/M/N_{\text{stream}} + v/N_{\text{term}}/N_{\text{term}}$ (Fig. 6c).
Add v streams to the isolated server and measure the new wait time (note the difference between Fig. 6a and Fig. 6c); if this wait time is lower than that calculated from a global queue (Fig. 2), the latter is used.

We now explain the underlying ideas and calculations for these steps.

4.3 Step 1

Among the details in the FSA description of the GWQ load sharing algorithm, the most important is the fact that a request becomes a candidate for remote service only if there is no idle local capacity (C1). The first-cut estimate for the arrival rate of such candidate requests is $p\lambda$, where p is the probability that (in queuing terminology) all N_{stream} queue servers are busy, and λ is the throughput. If there are no formulas for p (say, because the distributions are not memoryless), p can be measured with a simulator for the network in Fig. 6a.

Among the requests that become candidates for remote service, some will in fact be served locally because spare capacity may become available while the requests are waiting for remote service. Thus, the net arrival rate of requests for remote service is less than $p\lambda$ — say $fp\lambda$ for some $f < 1$. Some of this effect is captured by Step 2 (below), but not completely. In any case, there are many other factors that are implementation-dependent and therefore impossible to model a priori.

For example, the application interface affects how quickly a request for service destined for all remote servers can be assembled and passed down to the network protocol that underlies the load-sharing protocol; the implementation of the network protocol and the routing through the switch affect how quickly the requests are delivered to the remote servers; the particular hardware used by the remote servers and the bandwidth reservation protocol affect how quickly they can respond with a bid, etc.

These effects are captured by the fraction f, which is a calibration factor [29] that is to be measured empirically. Once determined, however, the calibration factor for a given implementation will, to a first approximation, remain constant regardless of changes in system configuration (i.e., N_{server}, B_{link}, B_{switch}, etc). Thus, $fp\lambda$ is the estimate for the arrival rate of requests for remote service (see Fig. 6a).

4.4 Step 2

From the perspective of a server, the total idle remote capacity may be viewed as additional local capacity of n streams, and $fp\lambda$ is the offered load for these n streams (see Fig. 6b). If all of the additional capacity is engaged, the request is "lost" from the M/M/n/n queue, which means that the request will (to a first approximation) be serviced locally later.

We now describe how n is calculated. A remote server that operates independently has, on average, idle capacity for $(1-\rho)N_{\text{stream}}$ streams. Since the utilization of the remote server is almost unaffected by load sharing, as observed in the beginning of this section, a server sees (on average) a total idle remote capacity of

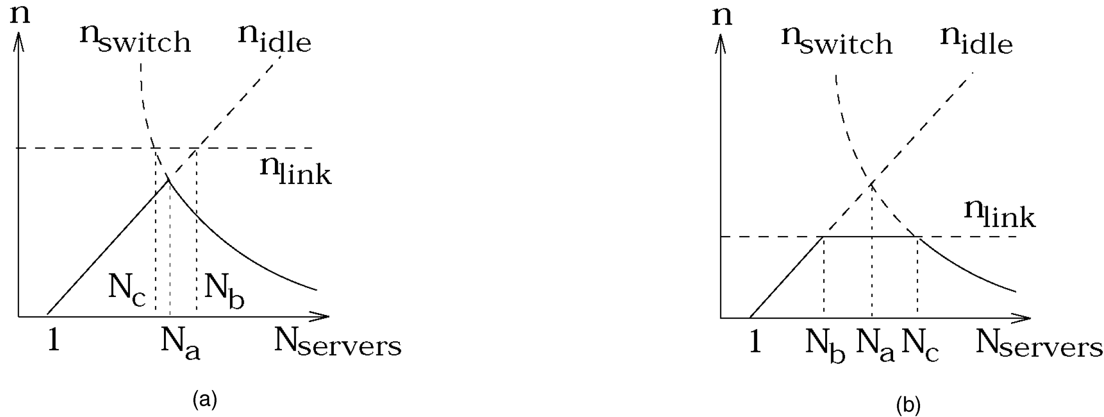

Fig. 7. Available idle remote capacity: $n = min(n_{\text{idle}}, n_{\text{link}}, n_{\text{switch}})$ streams.

$$n_{\text{idle}} = (N_{\text{server}} - 1)(1 - \rho)N_{\text{stream}}. \tag{4}$$

It may seem that this idle capacity should be shared by all servers, and we did experiment with different ways of prorating the idle capacity. However, none works as well as the approximation in (4), perhaps because arrivals at different servers see this idle capacity at different times. How much of this idle capacity is usable by a server depends on the link and switch bandwidths: The number of streams that can flow on a link is at most

$$n_{\text{link}} = \frac{B_{\text{link}}}{B_{\text{stream}}}, \tag{5}$$

and the average number of streams per server flowing through the switch is at most

$$n_{\text{switch}} = \frac{B_{\text{switch}}}{N_{\text{server}} B_{\text{stream}}}. \tag{6}$$

The average number of streams that can be serviced remotely is therefore bounded by

$$n = \min(n_{\text{idle}}, n_{\text{link}}, n_{\text{switch}}). \tag{7}$$

This is illustrated in Fig. 7. (Ignore N_{a}, N_{b}, and N_{c} for the time being.)

Having calculated n, we consider an $M/M/n/n$ queue (using linear interpolation between the solutions for $\lfloor n \rfloor$ and $\lceil n \rceil$) with arrival rate $fp\lambda$ and average service time T_{active}. Let v denote the average number of active streams in this queue. This v is thus our estimate for the average number of requests—at any time—that are serviced remotely.

4.5 Step 3

After looking through data from numerous simulations, we found that, if m is the (measured) average number of requests serviced remotely, then the performance of GWQ can be closely approximated with an $M/M/N_{\text{stream}} + m/N_{\text{term}}/N_{\text{term}}$ queue; i.e., each server behaves as if a (virtual) capacity of m streams are added. We therefore use v from Step 2 as an estimate for m (see Fig. 6c). Then,

$$T_{\text{wait}} = \frac{\text{number of waiting requests}}{\lambda'} \tag{8}$$

where the number of waiting requests and throughput λ' are calculated from the equations for the $M/M/N_{\text{stream}} + v/N_{\text{term}}/N_{\text{term}}$ queue [18]. If this value is lower than the wait time for the case where all terminals and capacity from all servers are pooled in one global queue (Fig. 2), we use the latter instead. (This is similar to replacing a balanced job bound on a queuing network's throughput by an asymptotic bound when the latter is tighter [50].) The link and switch utilization are calculated from

$$\rho_{\text{link}} = \frac{vB_{\text{stream}}}{B_{\text{link}}} \quad \text{and} \quad \rho_{\text{switch}} = \frac{vN_{\text{server}}B_{\text{stream}}}{B_{\text{switch}}}. \tag{9}$$

Note that ρ_{link} measures the link utilization in one direction; the utilization in the other direction is the same since the network is homogeneous.

4.6 Sizing the Network

We now analyze the model to derive three guidelines for sizing the system in Fig. 1. In our model, the wait time is estimated (Step 3) with $M/M/N_{\text{stream}} + v/N_{\text{term}}/N_{\text{term}}$, so T_{wait} is smallest when the virtual capacity v is largest. If the parameters $(N_{\text{stream}}, N_{\text{term}}, T_{\text{active}}, T_{\text{sleep}})$ for each server remain constant, then $fp\lambda$ is fixed (Step 1), so v is largest when n is maximum (Step 2)—this happens when $N_{\text{server}} = N_{\text{a}}$ for the case depicted in Fig. 7a and when $N_{\text{b}} \leq N_{\text{server}} \leq N_{\text{c}}$ for the case in Fig. 7b. For the latter, simulations show that, as N_{server} increases from N_{b} to N_{c}, link utilization actually rises marginally, so m (see Step 3) continues to peak, which in turn causes wait time to decrease. Wait time is therefore minimum at

$$N_{\text{server}}^{\max} = \max(N_{\text{a}}, N_{\text{c}}). \tag{10}$$

In Fig. 7, N_{a} is defined by $n_{\text{idle}} = n_{\text{switch}}$, so from (4) and (6),

$$(N_{\text{a}} - 1)(1 - \rho)N_{\text{stream}} = \frac{B_{\text{switch}}}{N_{\text{a}}B_{\text{stream}}}.$$

Solving this equation and substituting (2), we get

$$N_{\text{a}} = \frac{1}{2} + \sqrt{\frac{1}{4} + \frac{B_{\text{switch}}}{\left(1 - \frac{N_{\text{term}}}{N_{\text{stream}}}\frac{T_{\text{active}}}{T_{\text{sleep}} + T_{\text{active}}}\right)N_{\text{stream}}B_{\text{stream}}}}.$$

Similarly, in Fig. 7, N_{c} is defined by $n_{\text{link}} = n_{\text{switch}}$, so from (5) and (6), $N_{\text{c}} = \frac{B_{\text{switch}}}{B_{\text{link}}}$.

As N_{server} increases, T_{wait} decreases to a minimum at $N_{\text{server}} = N_{\text{server}}^{\max}$, after which T_{wait} increases again. Since this

subsequent increase in T_{wait} is undesirable, the minima offers a natural constraint on network size (see (10)):

Maximum Network Size.

$$N_{\text{server}} \leq N_{\text{server}}^{max} = max\left(\frac{B_{\text{switch}}}{B_{\text{link}}}, \frac{1}{2} + \sqrt{\frac{1}{4} + \frac{B_{\text{switch}}}{\left(1 - \frac{N_{\text{term}}}{N_{\text{stream}}} \frac{T_{\text{active}}}{T_{\text{sleep}} + T_{\text{active}}}\right) N_{\text{stream}} B_{\text{stream}}}}\right). \quad (11)$$

The bound suggests, in particular, that the network should be smaller if the link bandwidth is higher. While this may sound counterintuitive, the right interpretation is that, if link bandwidth is increased without a corresponding increase in switch bandwidth, then the number of servers should be reduced, so that the increased bandwidth can be exploited to reduce the wait time (otherwise, congestion at the switch will negate the increase in link bandwidth).

The inequality (11) gives a closed-form expression for sizing a network. If the bound is violated when the input parameters are substituted into the inequality, that indicates that the wait time can be decreased by reducing the number of servers N_{server}.

From (7), $n_{\text{link}} \geq \min(n_{\text{idle}}, n_{\text{switch}})$ defines the smallest link bandwidth required to achieve minimum wait time:

Minimum Link Bandwidth.

$$B_{\text{link}} \geq B_{\text{server}}^{\min} = \min\left(\frac{B_{\text{switch}}}{N_{\text{server}}}, (N_{\text{server}} - 1) \left(1 - \frac{N_{\text{term}}}{N_{\text{stream}}} \frac{T_{\text{active}}}{T_{\text{sleep}} + T_{\text{active}}}\right) N_{\text{stream}} B_{\text{stream}}\right). \quad (12)$$

If this bound is violated, then the link is the bottleneck in the network design, in the sense that wait time can be reduced further by increasing link bandwidth (without changing the other parameters). Similarly, $n_{\text{switch}} \geq \min(n_{\text{idle}}, n_{\text{link}})$ defines the smallest switch bandwidth necessary to attain minimum wait time:

Minimum Switch Bandwidth.

$$B_{\text{switch}} \geq B_{\text{switch}}^{\min} = \min\left(N_{\text{server}} B_{\text{link}}, N_{\text{server}} (N_{\text{server}} - 1) \left(1 - \frac{N_{\text{term}}}{N_{\text{stream}}} \frac{T_{\text{active}}}{T_{\text{sleep}} + T_{\text{active}}}\right) N_{\text{stream}} B_{\text{stream}}\right). \quad (13)$$

Again, if this bound is violated, then the switch is the bottleneck (even if its utilization is, say, only 60 percent), and wait time can be reduced further by increasing switch bandwidth.

We have thus used the analytical model to estimate the size of the individual servers (N_{stream}) and the network (N_{server}, B_{link}, B_{stream}). The estimates are all in closed-form, which is especially convenient at the capacity planning stage.

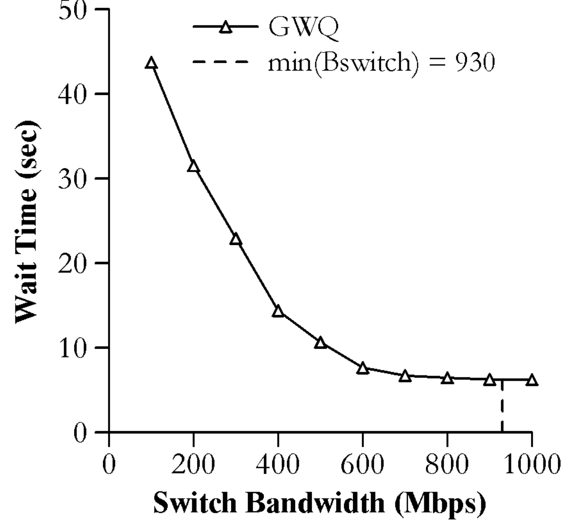

Fig. 8. B_{switch} vs. wait time.

5 Experiments

In this section, we report several experiments to profile the behavior of the load sharing algorithms. In the experiments, we replicate each object at every server in order to demonstrate the efficacy of the algorithms in exploiting load sharing opportunities to bring the performance of the distributed servers to that of a centralized server (C3). Of course, this does *not* mean that the algorithms necessitate full replication; as explained in Section 3, servers that do not host a requested object will simply ignore that particular service request. However, we expect most of the requests to target a small number of popular objects in practice [25]. As most of the servers are likely to host these popular objects anyway, there will still be ample load sharing opportunities without full replication.

Whenever possible, we use both the analytical model, presented in the previous section, and a simulation model for each experiment so as to corroborate the estimation of each model. The experiments should confirm the validity of the models for typical multimedia-on-demand configurations. They should also sample from a broad range of parameter settings, as it is impossible to explicitly delimit the design space in which the models can be used. This is because the behavior of the load sharing algorithms is determined by a large number of parameters, and each variable may follow one of many possible probability distributions.

We begin with a description of the simulator, followed by a series of experiment results. We model workloads with large objects (e.g., movies) in the first experiment, and small objects (e.g., training videos) in the second experiment. Besides highlighting the potential of the load sharing algorithms under these two broad classes of workloads, the experiments also serve to demonstrate the usefulness of the analytical model as a capacity planning tool. The third experiment is intended to show how the algorithms can help a set of operational servers cope with skewed load distributions that could not be foreseen when the servers

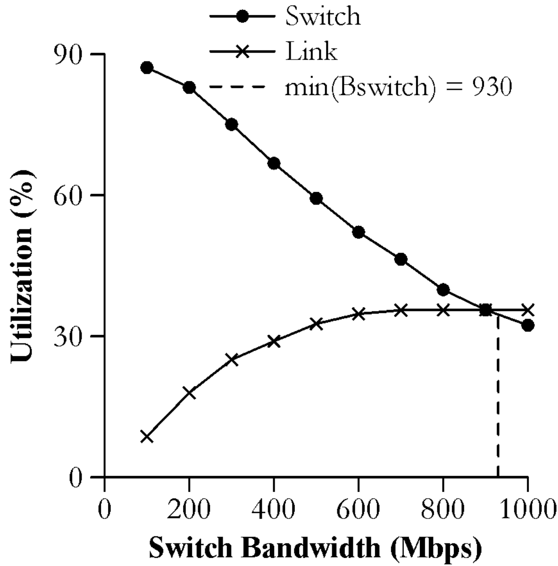

Fig. 9. B_{switch} vs. utilization.

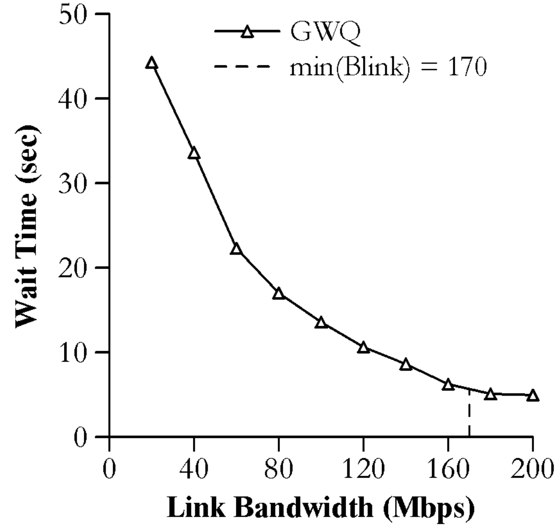

Fig. 10. B_{link} vs. wait time.

were configured. The last reported experiment deals with workload shifts.

We performed numerous other experiments besides the ones described here; however, we omit the others because the observations and qualitative results obtained there are similar.

5.1 Simulation Model

The simulator, written in the CSIM/C++ process-oriented simulation language [40], is constructed after the model in Fig. 1. There are four types of components: a *Terminal* that generates object retrieval requests and collects statistics on completed requests; a *Switch* and a *Link* that model the bandwidth usage of the ATM switch and links, respectively; and a *Server* component that models how the multimedia server allocates its bandwidth to service local and remote object requests. The details of each component are presented below. The parameters of the simulator are the same as those of the analytical model in Table 1.

In the simulation model, there are N_{server} multimedia servers that are connected together by an ATM switch with a bandwidth of B_{switch}, each via a link with a speed of B_{link}.[2] Every server can retrieve up to N_{stream} streams from its own disk storage. In addition, every server hosts a total of N_{term} terminals. A terminal repeatedly goes through a period of idling, followed by an object retrieval. The idle time is exponentially distributed with a mean of T_{sleep} seconds, while the retrieval duration is uniformly distributed (default) with a mean of T_{active} seconds. Unlike the analytical model which makes certain simplifying assumptions, the simulator models the GWQ algorithm in detail, including message exchanges between servers, wait queue maintenance, resource scheduling, race conditions (see end of Section 3.2.2), etc.

2. Of course, in practice the servers may be connected by some other network topology, so that the distance between pairs of servers may differ. This can be taken care of in the bidding prices as explained in Section 3.2.

In subsequent sections, we will use both the analytical model and the simulator to profile the behavior of the GWQ and GWQ+L load sharing algorithms. For comparison purposes, we will also include two boundary cases in our evaluation—M/G/nc and n × M/G/c. (We will instantiate the algorithms with M/U/nc and n × M/U/c, or M/M/nc and n × M/M/c, depending on whether the service time is uniformly or exponentially distributed for the experiment in question.) On one hand, M/G/nc represents a centralized server with the same aggregate number of terminals and stream capacity (Fig. 2) as the distributed system that we are studying, and thus serves as a baseline to measure the optimality of the load sharing algorithms. On the other hand, n × M/G/c denotes the "worst-case" scenario where the servers work independently of each other, and serves to highlight the benefits of the load sharing algorithms.

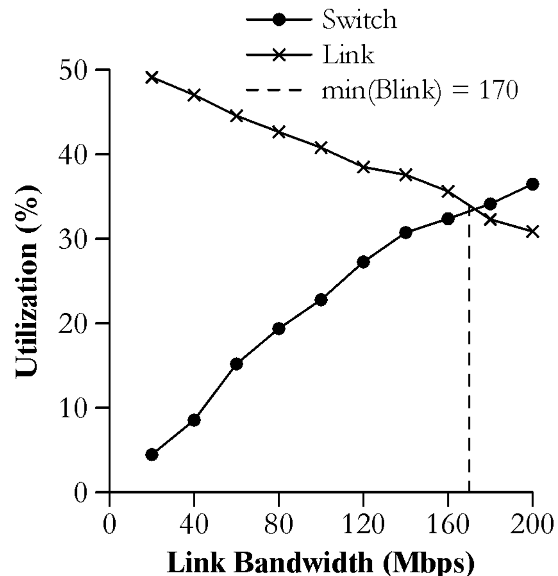

Fig. 11. B_{link} vs. utilization.

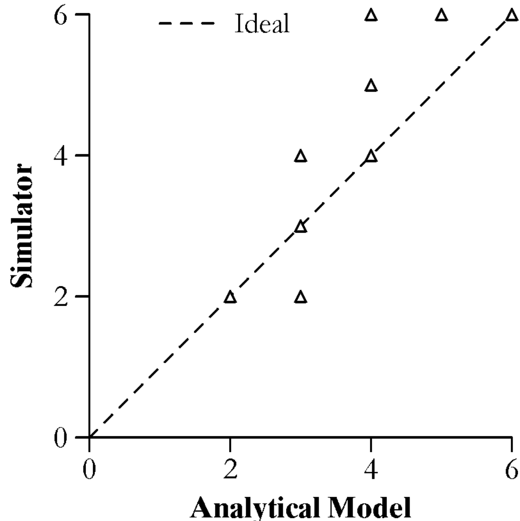

Fig. 12. Optimal # of servers.

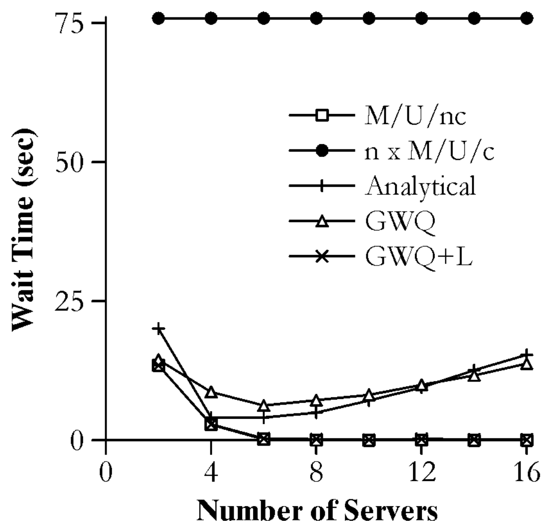

Fig. 14. N_{server} vs. wait time.

The primary performance metric is the wait time T_{wait}. Each simulation experiment was run long enough to allow for a minimum of 20,000 object retrievals at each server, after discarding the initial 1,000 samples. We also verified that the size of the 90 percent confidence intervals for wait time (computed using the batch means approach [39]) was within a few percent of the mean in almost all cases.

5.2 Movie-on-Demand

We begin our investigation with an experiment to profile the performance of the two load sharing algorithms under a movie-on-demand workload, where T_{active} is uniformly distributed in [3,600, 10,800] seconds, T_{sleep} is exponentially distributed with a mean of 7,200 seconds, and B_{stream} = 15 Mbps. As for the resource parameters, we start with a baseline of N_{server} = 6, N_{term} = 90, N_{stream} = 50 (to satisfy the capacity requirement in equation (3)), B_{link} = 155 Mbps, and B_{switch} = 1,000 Mbps. (Although some switches provide 2 Gbps or more, not all of the bandwidth may be available; some bandwidth could be reserved for another set of servers, or for some other applications.)

We calibrated f against the simulator with this workload for some test cases [29] to arrive at a value of $\frac{2}{3}$. As explained in Section 4.2, f is implementation-dependent but immune to changes in workload and system configuration. This f value is therefore specific to our simulator, and we will use the same value for the rest of our experiments.

Let us first examine the accuracy of the analytical model. According to (13), the wait time is minimum when $B_{\text{switch}} \geq$ 930 Mbps if all other parameters remain unchanged. This is confirmed by the simulator's results in Fig. 8. As the figure shows, the wait time produced by the GWQ load sharing algorithm reduces initially with increasing B_{switch}, but levels off after 900 Mbps as the switch is no longer the bottleneck resource (see Fig. 9; there, the two curves are plotted with the simulator, and

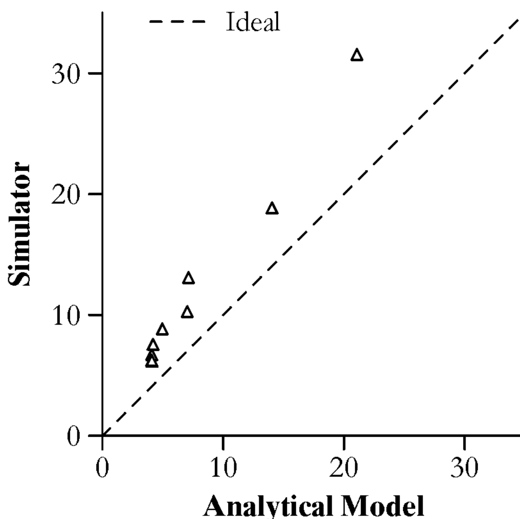

Fig. 13. Minimum wait time.

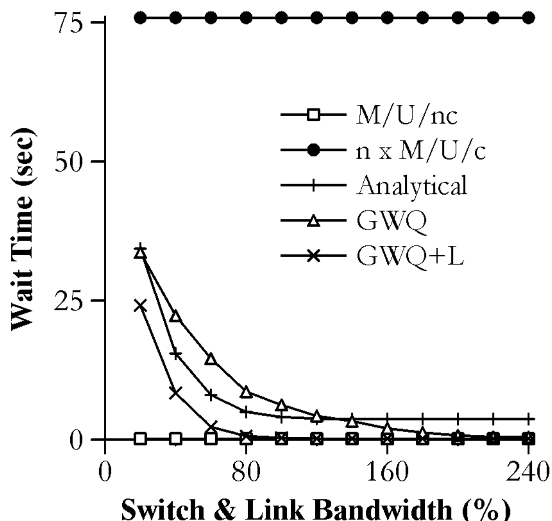

Fig. 15. Network scaling.

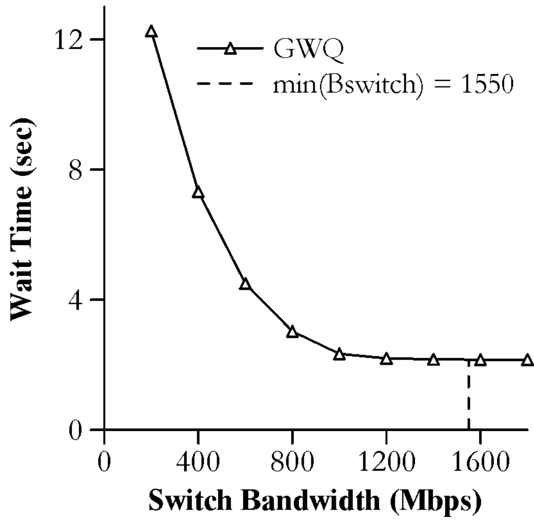

Fig. 16. B_{switch} vs. wait time.

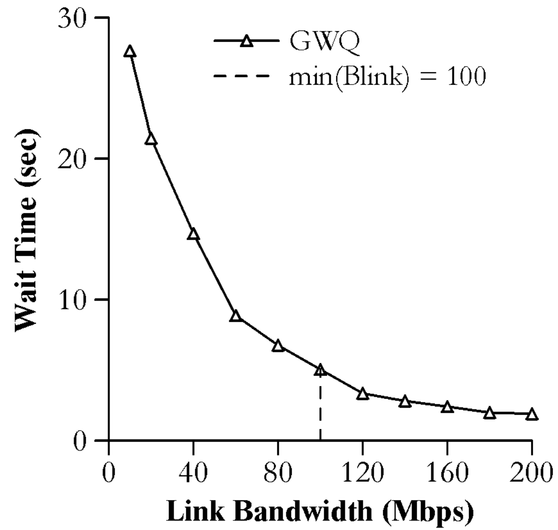

Fig. 18. B_{link} vs. wait time.

their intersection is accurately identified by the analytical model).

Another result from the analytical model (12) is that the ATM link becomes a bottleneck below 170 Mbps, as Fig. 11 confirms. Fig. 10 plots the (simulated) wait time of the GWQ algorithm as a function of B_{link}. The figure shows that, according to the simulator, wait time improves significantly as B_{link} increases from 20 to 180 Mbps, but only very marginally after that as predicted by the analytical model.

The third result (11) is that, for a given workload and network bandwidth, there is an optimal network size at which the average wait time is at its minimum. Below the optimum, there are insufficient load sharing opportunities, while performance suffers beyond the optimum as the network becomes congested by the load sharing activities that GWQ initiates. Fig. 12 and Fig. 13 plot the optimal number of servers and minimum wait times, respectively, for several configurations. In each of the two figures, the x-axis represents the estimates from the analytical model, the y-axis gives the outputs from the simulator, and each data point corresponds to a particular combination of switch and link bandwidths. Taking into account the deviations introduced by rounding the calculated network size, the analytical model and the simulator yield reasonably similar optimal wait times and number of servers.

Incidentally, the results in Fig. 12 show that, with current hardware characteristics, the optimal number of servers should be less than a dozen; this is why we are using small N_{server} settings for all of our experiments.

Next, we consider the benefits of the GWQ load sharing algorithm. Fig. 14 gives the wait times produced by the various algorithms as a function of the number of servers. The figure shows that GWQ results in a vast performance improvement over isolating the servers (n × M/U/c; recall that 'U' denotes uniform distribution). Moreover, the

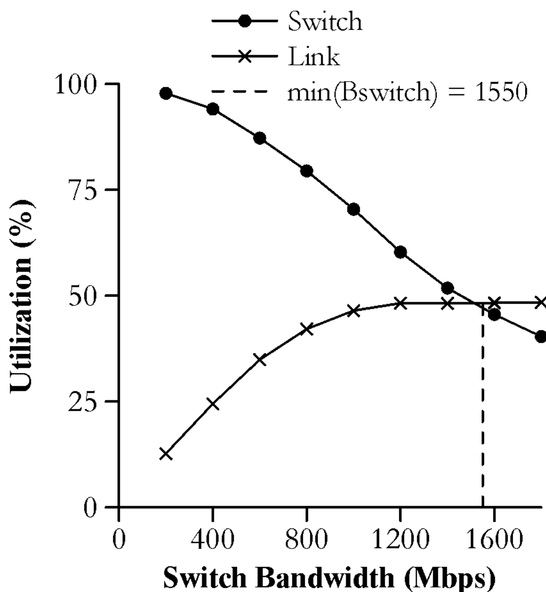

Fig. 17. B_{switch} vs. utilization.

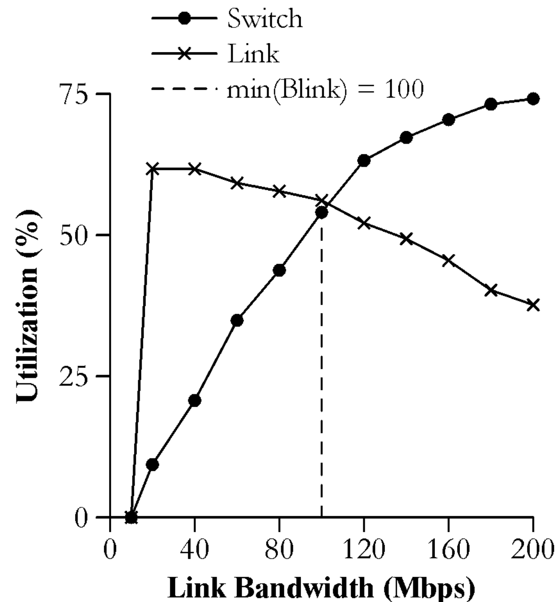

Fig. 19. B_{link} vs. utilization.

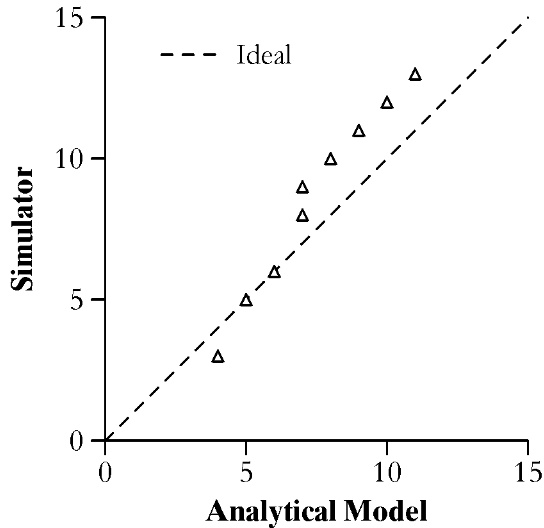

Fig. 20. Optimal # of servers.

GWQ+L version produces a further wait time reduction, especially if there are many multimedia servers on the network. In fact, the GWQ+L algorithm manages to help the distributed system to approximate the performance of a centralized server (M/U/nc). This is because, by allowing servers to take back requests that are being served remotely, the traffic on the ATM switch and links is very much lightened, so the available network bandwidth becomes more than sufficient. These trends are evident in Fig. 15, too, which shows how scaling B_{switch} = 1,000 Mbps and B_{link} = 155 Mbps concurrently by the same percentage (x-axis) affects the wait time (y-axis). Here GWQ+L does not perform as well as M/U/nc when bandwidth is low because the switch and/or link cannot support load sharing sufficiently.

To summarize, we have arrived at a couple of observations in this experiment. First, the GWQ load sharing algorithm produces very substantial reductions in wait times, though they can be improved further to match those

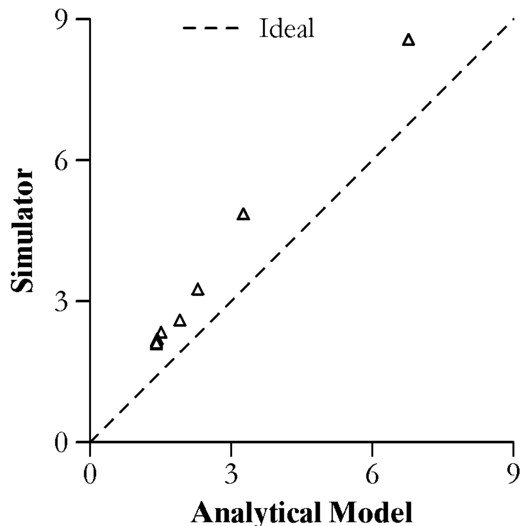

Fig. 21. Mimimum wait time.

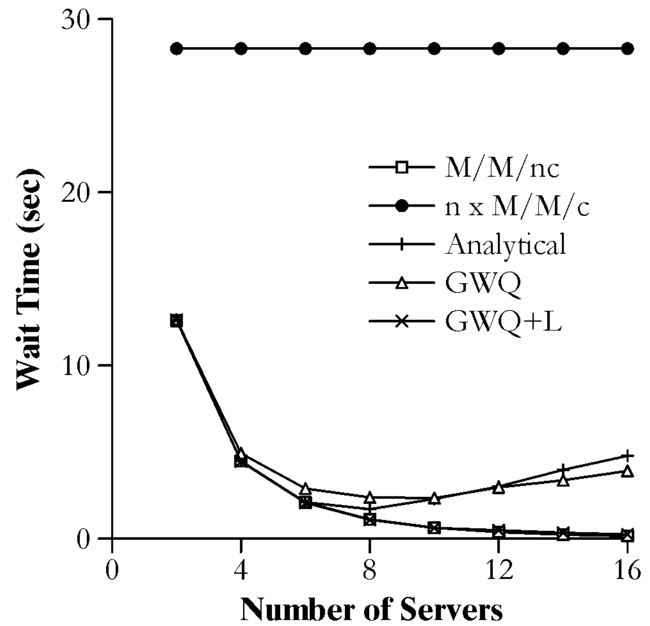

Fig. 22. N_{server} vs. wait time.

in a centralized server by adopting the GWQ+L algorithm (which entails a much more intricate implementation). Second, there is an optimal number of servers that should share loads to achieve minimum average wait times. For realistic workloads and current hardware characteristics, the optimum should be less than a dozen. This means even small service providers can enjoy the benefits that GWQ brings. It also means a large service provider who operates many servers should partition them into smaller clusters, at least where load sharing is concerned. Third, the analytical model is reasonably accurate, both in its numerical estimation of the wait time and in its analytical conclusion about the network parameters.

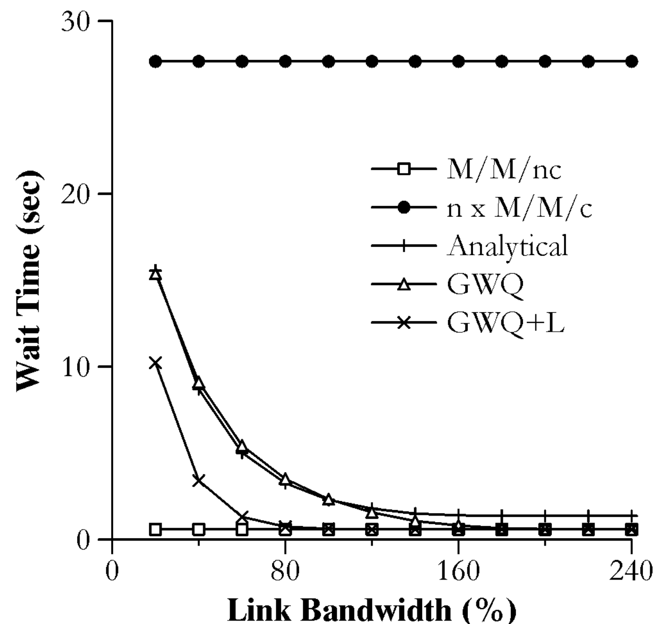

Fig. 23. Network scaling.

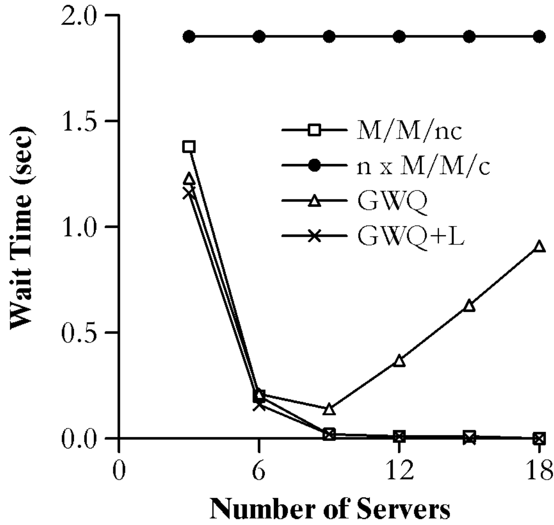

Fig. 24. Lightly loaded server.

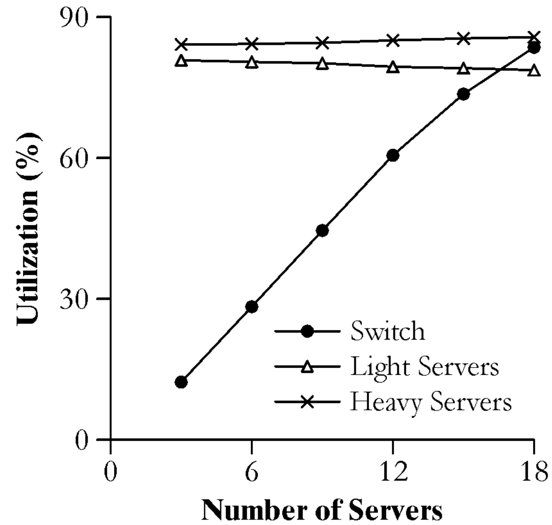

Fig. 26. GWQ utilizations.

5.3 Just-in-Time Training

For the second experiment, we change the workload from long-duration object requests to shorter-duration requests that are more typical in a just-in-time training environment. This is achieved by setting T_{active} = 300 seconds (exponential), T_{sleep} = 1,200 seconds (exponential), B_{stream} = 15 Mbps, N_{server} = 10, N_{term} = 90, and N_{stream} = 20. The rest of the parameters remain as in the last experiment.

Fig. 16, 17, 18, 19, 20, 21, 22, and 23 give the new results. Comparing this set of figures with the corresponding figures for the previous experiment (Fig. 8, 9, 10, 11, 12, 13, 14, and 15) we note that all of our earlier observations still hold, despite significant changes to the workload. Specifically, 1) the analytical model is accurate numerically (Fig. 22 and Fig. 23) and analytically (Fig. 16, 17, 18, 19, 20, and 21), 2) the GWQ algorithm produces a very substantial performance improvement over n $\times$ M/M/c (Fig. 22, and Fig. 23), 3) the GWQ+L algorithm leads to a further improvement to match the performance of a centralized server (Fig. 22 and Fig. 23), and 4) there is an optimal number of servers (Fig. 22) — less than a dozen (Fig. 20) — that should share load if minimum wait times are to be achieved.

5.4 Skewed Load Distributions

In the previous experiments, we have used the same N_{term} and N_{stream} settings for all servers. This reflects the situation at the capacity planning stage, where no details about subscriptions are available. When configured and deployed, the capacity of a multimedia server remains unchanged, but the number of terminals that it hosts is likely to fluctuate as subscriptions are added and terminated. Consequently, some servers may be heavily loaded while others become underutilized. Our next experiment is intended to study how the load sharing algorithms might help in such situations, and thus demonstrate the robustness of the algorithm with respect to deviations from the homogeneity assumption adopted during capacity planning.

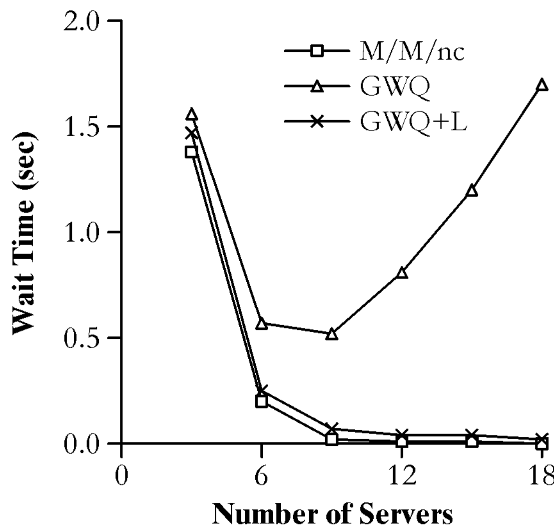

Fig. 25. Heavily loaded server.

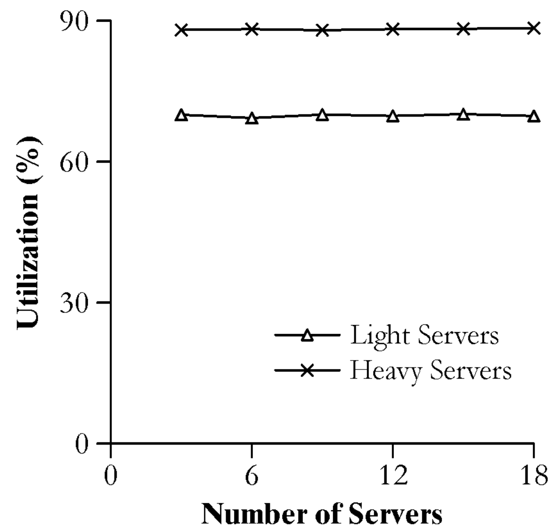

Fig. 27. No load sharing.

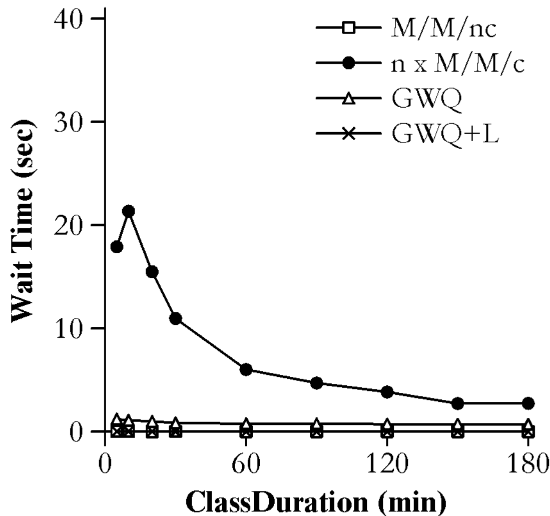

Fig. 28. Light job class.

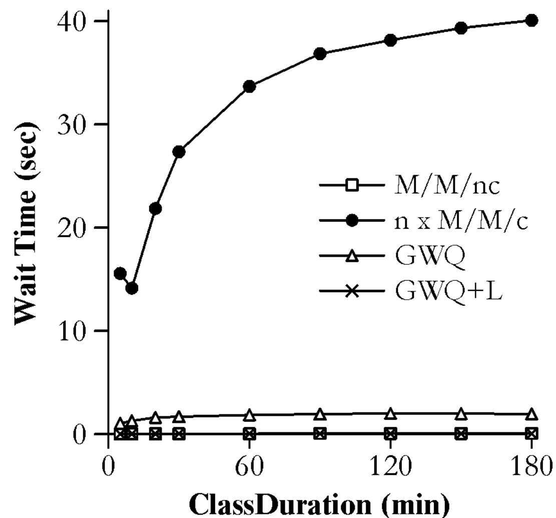

Fig. 29. Heavy job class.

To keep the experiment manageable, we set N_{stream} to 20 for all servers. One-third of the servers are lightly utilized, with N_{term} = 70, while the rest of the servers are heavily loaded at N_{term} = 90. Other parameter settings remain the same as in the previous experiment. As our analytical model does not capture unbalanced configurations that result from fluctuations in user subscriptions, our results are obtained with the simulator.

The average wait time for the lightly loaded servers are plotted in Fig. 24. Whereas these servers could have achieved zero wait time had they shared load only amongst themselves (not shown), they now take over some of the jobs from the heavily loaded servers (see the utilization of the two groups of servers with and without load sharing in Fig. 26 and Fig. 27), and consequently suffer a performance penalty as evident in the positive wait times in the figure. The reason is that, although the servers give priority to local requests when deciding which jobs to start, a new request arriving at a server may be delayed, because of quality of service guarantees, by active jobs from remote servers. This is one of the differences between a distributed multimedia system and a general-purpose distributed computing system (see C1). However, the resulting wait times are still much better than those produced by n × M/M/c.

The qualitative behavior of both GWQ and GWQ+L are the same as those observed in the previous experiment: The wait time of GWQ+L diminishes steadily as the number of servers increases, while GWQ's performance worsens beyond a certain number of servers (Fig. 24) because the ATM switch becomes saturated (Fig. 26), thus hindering the initiation of remote services to share the workload.

Having examined the deterioration on the lightly utilized servers, we now consider the gains enjoyed by the heavily loaded servers. Fig. 25 shows that the performance of these servers improve vastly, relative to the 29 seconds produced by n × M/M/c. (This very high wait time for n × M/M/c is omitted from Fig. 25.) In fact, under both load sharing algorithms, requests from heavily loaded servers experience only marginally worse wait times than their underutilized counterparts (compare Fig. 24 and Fig. 25). From these results, it is apparent that the load sharing algorithms are effective under unbalanced load distributions, with the GWQ+L version again outperforming the GWQ variant.

5.5 Dynamic Workloads

While the previous experiment has workload that is unbalanced geographically, we now consider workload that is unevenly spread out over time: We change the workload at each server periodically by defining two job classes—a *Light* class with T_{active} = 240 seconds (exponential), and a *Heavy* class with T_{active} = 360 seconds (exponential). Each server runs the *Light* workload for *ClassDuration* seconds, switches to the *Heavy* workload for two × *ClassDuration* seconds, before reverting to the *Light* workload. We arrange the starting sequence so that, at any one time, a third of the servers are running the *Light* class and the rest are running the *Heavy* class. Moreover, we set N_{server}, N_{term}, N_{stream}, and T_{sleep} to 15, 90, 20, and 1,400 seconds (exponential), respectively. One scenario that could generate this kind of dynamic workload is where the primary users of the various servers belong to different groups, e.g., home users versus office workers.

Fig. 28 and Fig. 29 plot the average wait times for the two job classes as a function of *ClassDuration*. Let us first investigate the curves for n × M/M/c. At a *ClassDuration* of five minutes, the workload changes so frequently that requests from both job classes are running concurrently at the same server, causing both classes to experience similar wait times (about 20 seconds). Interestingly, at *ClassDuration* = 10 minutes, the *Heavy* class actually experiences shorter wait times than the *Light* class. A close examination reveals that this is because requests from the *Light* class arrive right after the server has become congested under the *Heavy* class, while requests from the *Heavy* class comes after the server has caught up with its load under the *Light* class.

As *ClassDuration* rises further, the interplay between the two job classes reduces, thus explaining the diminishing wait time for the *Light* class and the worsening trend for the *Heavy* class. The behavior of the other algorithms are as expected: The *Light* class fares slightly better than the *Heavy*

class under the GWQ algorithm, while both job classes achieve virtually zero wait time under the GWQ+L algorithm. This experiment shows that the two algorithms work well even with dynamic workloads.

6 Conclusion

In this paper, we propose a solution to contain the cost escalation and wait time penalty that result from installing small, isolated multimedia servers. Our solution entails linking the servers to a (limited-capacity) network, and enabling overloaded servers to tap the idle retrieval bandwidth of other servers. We first identify the unique characteristics that define the framework for load sharing in such distributed multimedia systems. These characteristics are: 1) Servicing a job remotely generates additional network traffic and should be done judiciously, 2) A load distribution on the servers that is unbalanced may perform better than one that is balanced because network bandwidth is required to service remote jobs, 3) The resources needed for job execution are much more than the overheads of enabling load sharing, so the system can afford the scheduling overheads needed to optimize its load distribution, and 4) Relocating an executing job is relatively straightforward, and should be exploited to improve load distribution.

Based on the above framework, we present in detail a decentralized GWQ algorithm to effect this load sharing. Under the algorithm, a user request is supported by the host server whenever possible. If the host server runs out of bandwidth, it will broadcast a service request to the other servers on the network, which will bid for the job if they have idle capacity. Once a job is awarded, it remains with the same server until completion.

To understand the performance trade-offs, we then develop an analytical model to capture the key characteristics of the load sharing algorithm. The model enables us to calculate accurately, for a given network bandwidth, what gains load sharing might bring about, and thus estimate the network bandwidth necessary for the distributed multimedia servers to approximate a centralized server. It also offers closed-form expressions (in terms of workload and network parameters) that specify the server capacity (3), bound the number of multimedia servers (11), and identify whether link or switch bandwidth is the bottleneck in a network design (12) and (13); these expressions can be used to estimate the minimum switch and link bandwidths for the distributed servers to realize the full potential of GWQ. The model has been verified against a simulator and found to yield acceptable accuracy in its numerical predictions and analytical conclusions.

Using the analytical model and the simulator, we carried out a wide range of experiments to profile the behavior of the GWQ algorithm. These experiments unanimously confirm that GWQ vastly reduces wait times at the servers, compared to operating them in isolation. In practice, the amount of improvement depends on the availability of network bandwidths and object replicas at alternative servers.

Another important conclusion is that there is an optimal number of servers that should share loads to achieve minimum average wait times. Below the optimum, there are insufficient load sharing opportunities, while performance suffers beyond the optimum as load sharing activities saturate the network. For realistic workloads and current hardware characteristics, the optimal number should be around a dozen servers or less. This means even small service providers can enjoy the benefits that GWQ brings. It also means a large service provider who operates many servers should partition them into smaller clusters, at least where load sharing is concerned.

Finally, we proposed an enhanced GWQ+L algorithm that enables a multimedia server to dynamically take over local retrieval requests that have been awarded to remote servers, so as to lighten the load on the interconnecting network. Simulation experiments show that the extension could produce some additional wait time improvement, allowing the distributed multimedia servers to approximate the performance of a centralized server. This confirms that the scheduling decisions of GWQ+L are optimal.

References

[1] G.R. Andrews, D.P. Bobkin, and P.J. Downey, "Distributed Allocation with Pools of Servers," *Proc. ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing,* Aug. 1982.

[2] Y. Artsy and R. Finkel, "Designing a Process Migration Facility: The Charlotte Experience," *Computer,* vol 22, no. 9, Sep. 1989.

[3] R.M. Bryant and R.A. Finkel, "A Stable Distributed Scheduling Algorithm," *Proc. Second Int'l Conf. Distributed Computing Systems,* Apr. 1981.

[4] T.L. Casavant and J.G. Kuhl, "Design of a Lossely-Coupled Distributed Multiprocessing Network," *Proc. Int'l Conf. Parallel Processing,* Aug. 1984.

[5] T.L. Casavant and J.G. Kuhl, "A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems," *Trans. Software Eng.,* vol. 14, no. 2, Feb. 1988.

[6] T.C.K. Chou and J.A. Abraham, "Load Balancing in Distributed Systems," *Trans. Software Eng.,* vol. 8, no. 4, July 1982.

[7] S. Chowdhury, "The Greedy Load Sharing Algorithm," *J. Parallel and Distributed Computing,* vol 9, no. 1, May 1990.

[8] W.W. Chu et al., "Task Allocation in Distributed Data Processing," *Computer,* vol. 13, no. 11, Nov. 1980.

[9] F. Cristian and C. Fetzer, "Fault-Tolerant External Clock Synchronization," *Proc. 15th Int'l Conf. Distributed Computing Systems,* May 1995.

[10] K.W. Doty, P.L. McEntire, and J.G. O'Reilly, "Task Allocation in a Distributed Computer System," *Proc. InfoCom,* 1982.

[11] R. Gusella and S. Zatti, "The Accuracy of the Clock Synchronization Achieved by TEMPO in Berkeley UNIX 4. 3BSD," *Trans. Software Eng.,* vol. 15, no. 7, July 1989.

[12] B. Hamidzedeh and D.J. Lilja, "Dynamic Scheduling Strategies for Shared-Memory Multiprocessors," *Proc. 16th Int'l Conf. Distributed Computing Systems,* May 1996.

[13] G.J. Holzmann, *Design and Validation of Computer Protocols.* Prentice Hall Software Series, pp 167-171, 1991.

[14] C.J. Hou and K.G. Shin, "Allocation of Periodic Task Modules with Precedence and Deadline Constraints in Distributed Real-Time Systems," *Trans. Computers,* vol. 46, no. 12, Dec. 1997.

[15] K. Hwang, W.J. Croft, G.H. Goble, B.W. Wah, F.A. Briggs, W.R. Simmons, and C.L. Coates, "A Unix-Based Local Computer Network with Load Balancing," *Computer,* vol. 15, no. 4, Apr. 1982.

[16] P.W. Jardetzky, C.J. Sreenan, and R.M. Needham, "Storage and Synchronization for Distributed Continuous Media," *Proc. ACM Multimedia Systems J.,* vol. 3, no. 4, Sep. 1995.

[17] D. Kappholz and H.C. Park, "Parallelized Process Scheduling for a Tightly-Coupled MIMD Machine," *Proc. Int'l Conf. Parallel Processing,* Aug. 1984.

[18] L. Kleinrock, *Queuing Systems, Vol. I: Theory,* Wiley-Interscience, 1975.

[19] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Comm. ACM,* vol. 21, July 1978.

[20] L. Lamport, "Concurrent Reading and Writing of Clocks," *Proc. ACM Trans. Computer Systems,* vol. 8, no. 4, Nov. 1990.
[21] E.D. Lazowska, J. Zahorjan, G.S. Graham, and K.C. Sevcik, *Quantitative System Performance,* Prentice Hall, pp 64-66, 1984.
[22] I.M. Leslie, D. McAuley, and S.J. Mullender, "Pegasus—Operating System Support for Distributed Multimedia Systems," *Proc. ACM Operating Systems Review,* vol. 27, no. 1, Jan. 1993.
[23] C.C. Lin, J. Xiang, and S.K. Chang, "Transformation and Exchange of Multimedia Objects in Distributed Multimedia Systems," *Proc. ACM Multimedia Systems J.,* vol. 4, no. 1, Feb. 1996.
[24] T.D.C. Little and A. Ghafoor, "Synchronization and Storage Models for Multimedia Objects," *J. Selected Areas in Comm.,* vol. 8, no. 3, Apr. 1990.
[25] T.D.C. Little and D. Venkatesh, "Popularity-Based Assignment of Movies to Storage Devices in a Video-on-Demand System," *Proc. Fourth Int'l Workshop Network and Operating System Support for Digital Audio and Video,* Nov. 1993.
[26] M. Litzkow, M. Livny, and M.W. Mutka, "Condor—A Hunter of Idle Workstations," *Proc. Eighth Int'l Conf. Distributed Computing Systems,* Jun. 1988.
[27] C. Lu and S.M. Lau, "An Adaptive Load Balancing Algorithm for Heterogeneous Distributed Systems with Multiple Task Classes," *Proc. 16th Int'l Conf. Distributed Computing Systems,* May 1996.
[28] F. Matthes and J.W. Schmidt, "System Construction in the Tycoon Environment: Architectures, Interfaces and Gateways," *Proc. Euro-Arch '93 Congress,* Oct. 1993.
[29] D.A. Menascé, V.A.F. Almeida, and L.W. Dowdy, *Capacity Planning and Performance Modeling.* P.T.R. Prentice Hall, 1994.
[30] C. Nicolaou, "An Architecture for Real-Time Multimedia Communication System," *J. Selected Areas in Comm.,* vol. 8, no. 3, Apr. 1990.
[31] J. Ousterhout, D. Scelza, and P. Sindhu, "Medusa: An Experiment in Distributed Operating System Structure," *Comm. ACM,* vol. 23, no. 2, Feb. 1980.
[32] H.-H. Pang, B. Jose, and M.S. Krishnan, "Resource Scheduling in a High Performance Multimedia Server," *Trans. Knowledge and Data Eng.,* vol. 11, no. 2, Mar. 1999.
[33] M.J. Pfluegl and D.M. Blough, "A New and Improved Algorithm for Fault-Tolerant Clock Synchronization," *J. Parallel and Distributed Computing,* vol. 27, no. 1, May 1995.
[34] K. Ramamritham, J.A. Stankovic, and W. Zhao, "Distributed Scheduling of Tasks with Deadlines and Resource Requirements," *Trans. Computers,* vol. 38, no. 8, Aug. 1989.
[35] P. Ramanathan, K.G. Shin, and R.W. Butler, "Fault-Tolerant Clock Synchronization in Distributed Systems," *Computer,* vol. 23, no. 10, Oct. 1990.
[36] S. Ramanathan and P.V. Rangan, "Adaptive Feedback Techniques for Synchronized Multimedia Retrieval over Integrated Networks," *Proc. IEEE/ACM Trans. Networking,* 1992.
[37] P.V. Rangan, H.M. Vin, and S. Ramanathan, "Designing an On-Demand Multimedia Service," *Comm. Magazine,* vol. 30, no. 7, Jul. 1992.
[38] P.V. Rangan, H.M. Vin, and S. Ramanathan, "Communication Architectures and Algorithms for Media Mixing in Multimedia Conferences," *Proc. IEEE/ACM Trans. Networking,* 1993.
[39] R. Sargent, "Statistical Analysis of Simulation Output Data," *Proc. Sump. Simulation of Computer Systems,* 1976.
[40] H. Schwetman, "SIM Users' Guide," MCC Technical Report ACT-126-90, Microelectronics and Computer Technology Corp., Mar. 1990.
[41] K.G. Shin and Y.C. Chang, "A Coordinated Location Policy for Local Sharing in Hypercube-Connected Machines," *Trans. Computers,* vol. 44, no. 5, May 1995.
[42] N.G. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *Computer,* vol. 25, no. 12, Dec. 1992.
[43] J.A. Stankovic and I.S. Sidhu, "An Adaptive Bidding Algorithm for Processes, Clusters and Distributed Groups," *Proc. Fourth Int'l Conf. Distributed Computing Systems,* May 1984.
[44] R. Steinmetz, "Synchronization Properties in Multimedia Systems," *J. Selected Areas in Comm.,* vol. 8, no. 3, Apr. 1990.
[45] C. Steketee, W. Zhu, and P. Moseley, "Implementation of Process Migration in Amoeba," *Proc. 14th Int'l Conf. Distributed Computing Systems,* June 1994.
[46] H.S. Stone, "Critical Load Factors in Two-Processor Distributed Systems," *Trans. Software Eng.,* vol. 4, no. 3, May 1978.
[47] K.S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications,* Prentice Hall, pp. 375-377, 1982.
[48] Y.T. Wang and R.J.T. Morris, "Load Sharing in Distributed Systems," *Trans. Computers,* vol. 34, no. 3, Mar. 1985.
[49] J.L. Wolf, P.S. Yu, and H. Shachnai, "DASD Dancing: A Disk Load Balancing Optimization Scheme for Video-on-Demand Computer Systems," *Proc. ACM SIGMETRICS Conf.,* May 1995.
[50] J. Zahorjan, K.C. Sevcik, D.L. Eager, and B.I. Galler, "Balanced Job Bound Analysis of Queueing Networks," *Comm. ACM,* vol. 25, no. 2, Feb. 1982.
[51] P.N. Zarros, M.J. Lee, and T.N. Saadawi, "A Synchronization Algorithm for Distributed Multimedia Environments," *ACM Multimedia Systems J.,* vol. 4, no. 1, Feb. 1996.

Y.C. Tay received his BSc from the University of Singapore and PhD from Harvard University. He is now a faculty member at the National University of Singapore. His main research interest is in performance modeling. Other recent interests are: correctness in distributed and parallel computing, routing protocols for mobility support and ad hoc wireless networks, and application of data mining to online optimization.

HweeHwa Pang received his BSc (with first class honors) and MS from the National University of Singapore in 1989 and 1991, respectively. He received his PhD from the University of Wisconsin at Madison in 1994, all in computer science. He is now a senior member of the research staff at Kent Ridge Digital Labs in Singapore. He heads a mobile computing project to develop software infrastructure and utilities to facilitate information access and computing from mobile devices. His research interests include database management systems, multimedia servers, and real-time systems.