# DVS: A System for Distribution and Management of Global Video On Demand Services

Alexey Roytman, Israel Ben-Shaul, Israel Cidon*

Department of Electrical Engineering
Technion - Israel Institute of Technology

Technion City, Haifa 32000,ISRAEL
{aroytman, issy, cidon}.ee.technion.ac.il

February 25, 1999

**Abstract**

The Distributed Video Server (DVS) system provides a comprehensive solution for the management and distribution of a major future electronic commodity service, Video on Demand (VOD). Key contributions of DVS include a scalable architecture that combines the global accessibility of the Internet with an acceptable quality of service through an optional private high-speed backbone, novel adaptive algorithms for server selection and movie distribution that are sensitive to load, storage capacity and changing user demands, and a reliable management layer that is resilient to failures of individual nodes. DVS has been implemented (mostly) in Java/RMI, it has a Web interface for end-users, and comes with several utilities, including a runtime monitor that tracks the dynamic aspects of the system and can be used for instrumentation and for manual administration in addition to automatic management.

*Keywords:* Video on the Internet, distributed multimedia.

## 1 Introduction and Motivation

The advent of largely accessible networking infrastructures makes a tremendous impact on the way commercial services are going to evolve in the near future. Major efforts are currently being made by various civic and commercial installments (e.g., banks, retail stores, government offices) in order to evolve toward the competitive world of electronic commerce motivated by the urgency to increase their market share and in particular remain competitive.

A significant part of the electronic commerce based economic establishment is expected to emerge as a direct development of the novel information highway possibilities and abilities. In particular, such businesses that do not only market and sell their merchandise over the network but actually produce, store, maintain and distribute it there have a small presence today but are expected to capture larger and larger share. These new services deal with electronic *content itself* as the commodity to be marketed, distributed and sold.

One major electronic commodity service is Video On Demand (VOD). VOD may replace in the future both video libraries, video stores and TV pay per view businesses. Video clips are also becoming much more common

*also with Sun Microsystems Labs, CA

on the Web, particularly in online news sites. According to [1], approximately 4,000 Web sites offered video clips in 1996, 12,000 in 1997, and the number is expected to triple each year for at least the next three years. It is easy to see the huge potential in having a large-scale motion video base that can be used as a source of information, entertainment, and education.

Extensive research on the ability to transport multi-media through communication networks has been conducted in the past years [2, 3] with some real applications already emerging [4, 5]. In the near future it is expected that the multi-media providers, with VOD being the common application, will attempt to offer high quality *global* services, while minimizing the operational cost of their services.

Large-scale VOD services introduce new challenges in the proper design of frameworks for networked multimedia management and dissemination systems. Two important characteristics that impact global VOD management are:

1. *High bandwidth and high quality of service (QoS)* — Although motion video may be played over networks with different bandwidths and corresponding qualities (e.g., H.263 can be played with ISDN-level bit rate, MPEG-1 requires a T1/Lan bandwidth of 1.5Mbps, and MPEG-2 that requires a bandwidth of up to 100Mbps, with a typical value of 4Mbps), it in general requires continuous and high bit rate.

2. *Very large size and storage requirements* — A typical 90 minutes MPEG-1 movie weights 1GB, but much larger files are not uncommon.

These two characteristics present an inherent conflict from the system design perspective. To address the first requirement, it is desirable to distribute video content as close as possible to end-users by arranging servers in different geographical regions to enable users to access them with high bandwidth and low delay. This is particularly important when considering the global and ubiquitous yet unpredictable (in terms of service quality) Internet for delivering the contents to end-users. Long distance and international communication costs further require "nearby" video sources that retain low cost.

The second requirement means, in contrast, that content distribution and replication is expensive, both in terms of communication as well as storage. This is particularly the case when content changes dynamically (e.g., news clips) or when the content is large (e.g., archival of movies from a virtual video store) and quickly exceeds the storage capacity of a single server. Clearly, storage considerations favor playing video from remote servers over dissemination of content to and playing from a local server, although load-balancing at least among co-located servers must be considered. Thus, the effectiveness of the system is largely determined by a solution that strikes a balance between these two conflicting requirements.

Another important property of the use of VOD is that the demand changes, both temporally (e.g., different movies are watched at different times of the day, in different time zones, and movie popularity changes over time), as well as spatially (e.g., different regions may favor different movies). This implies that *dynamic* management of content distribution, placement and playing is likely to outperform any static approach.

This paper addresses these challenges, presenting a scalable and reliable distributed architecture and several adaptive algorithms for management and distribution of video content among the geographically-dispersed servers. These ideas were embodied in the Distributed Video Server (DVS) system, which was implemented in Java and deployed over an experimental ATM network connecting several universities. DVS serves as our test bed for experimentation and validation of our approach, but its actual construction has raised several additional technical challenges, some of which are presented here.

The rest of this paper is organized as follows. In Section 1.1 we overview existing (partial) solutions to (parts of) the problems mentioned above. In Section 2 we present the system architecture including the server topology and network interfaces, and describe an algorithm for selecting a playing server given a client request. In Section 3 we present a novel sub-optimal but polynomial algorithm for content distribution, which is an

NP-hard problem, and provide experimentation results. Our architecture is hierarchical, implying potential reliability problems due to multiple points of failures (disconnections in the hierarchy). Section 4 addresses this issue and presents a naming and location scheme along with a customized leader election algorithm. The implementation of the DVS system, including its Web-based management component and its initial deployment, is over-viewed in Section 5. Finally, Section 6 summarizes the major contribution of this work and points to future work.

### 1.1 Related Work

There are several theoretical results and practical projects that address scalability and reliability in VOD servers. In [6, 7], the authors suggest to increase the scalability and reliability of a VOD system using an architecture that consists of a cluster of nodes, each node with a local disk array, connected by a high bandwidth switch or network. The user view of the system is of a single large server. The method employs algorithms for load balancing between servers and disks. This solution addresses the load problem but neglects communication cost, which is critical in wide-area transfer. A similar approach is taken in the Microsoft's TIGER video server, which combines a collection of PCs to construct a scalable server [8]. It uses video file striping to distribute segments of a movie across a collection of servers to balance the access load across the servers. In addition, it uses replication at the segment level as a mechanism for fault-tolerance.

[9] presents algorithms for a distributed VOD system in which a collection of video data is located at dispersed sites across a computer network. The servers maintain parts of movies and must get the missing parts of the movie before or during the service. This work is limited to the solution of this specific topic.

The work described in [2] is closely related to ours. It employs an optimization technique for solving the so-called Apportionment problem to determine the optimal number of copies per video in the system, and develops algorithms for the assignment of these copies. However, the distribution of movies is between multiple disks of the same host, while we exploit the distribution of movies between geographically scattered hosts. This distinction changes the nature of the problem (expense of communication, large latency and so on). For example, both works employ a load-balancing algorithm that consists of a static phase for initial movie distribution and a dynamic phase for on-line scheduling of the load between disks ([2]) or servers (in our work). The scheme of [2] permits to change the disks from which the movie is played while the movie is playing. In a large distributed system it seems impractical to switch a server during the duration of the video playing. In general, we do not address here disk-related optimizations and view this topic as complementary to our work.

## 2 DVS Architecture and Server Selection

The high-level architecture of DVS is depicted in Figure 1. The networking infrastructure is bi-level, consisting of a private high-speed intra-server network (e.g., a 155Mbps fiber ATM network) with users connecting to servers through the public Internet. The high-speed network is mainly used for real-time playing of movies to remote users, through local Internet proxy servers to which users are attached. That is, the attachment point and the video source need not be the same node. This design greatly increases the flexibility in server selection and movie placement, but could be expensive if all nodes were required to be attached to the private network. Thus, we only require a small subset to be directly connected to the private network, as will be explained below. This also means that inter-server management and off-line movie transfer also operate over the public network.

The need to provide regional distribution and management of content implies a hierarchical structure of the intra-server network, with the edge nodes providing the actual VOD service and manager nodes performing resource management and allocation. However for reliability purposes, manager programs actually runs an edge nodes that are elected to serve as regional managers but can be replaced by any other node in the level below

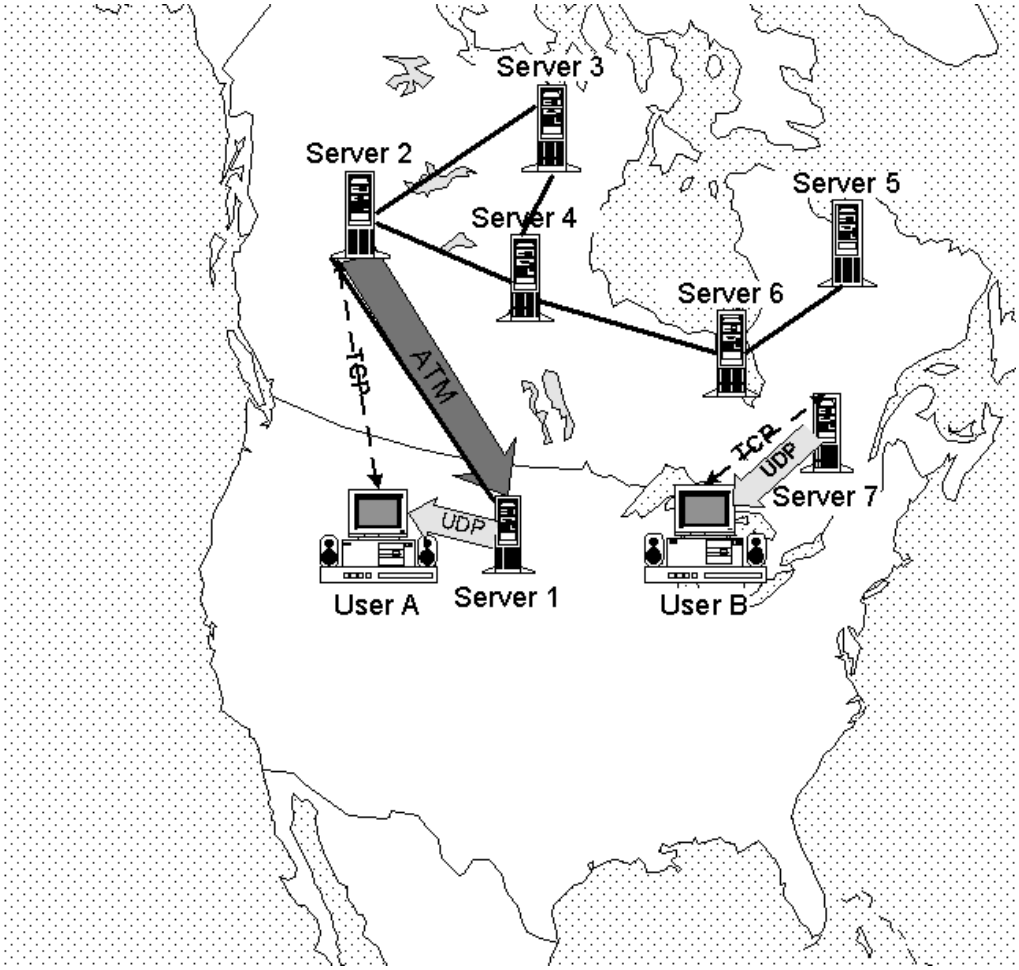


Figure 1: DVS Architecture

it, recursively. The details of the election algorithm will be discussed in Section 4. For the moment it suffices to understand that each manager node acts as the manager of the level below it and maintains information about movie location, load, and demand in its managed level. Each region consists of a flat set of servers, and we make the reasonable assumption that the distance/bandwidth from each server in the same region to all users is roughly the same. Each (elected) manager maintains the following information on each managed server: the current load (i.e., how many movies are being played from it), the server's maximum load as configured by the administrator (most likely to reflect the server's capabilities and the amount of resources that is devoted to DVS activities), the list of local movies (i.e., movies that are currently stored in the server's disk), the demand for each movie (measured in the number of new and currently serviced requests), and a boolean flag indicating whether the server can connect to the private network. This information is used by DVS to determine which server to assign to service a client request, and how to distribute the movies among the clients. We begin with the former problem, and address the latter in Section 3.

The general movie playing service consists of two phases: 1) the user requests from DVS (via one of the servers that are in the user's local region) to watch a specific movie, and DVS selects the proper server and returns its contact address to the user's client; 2) the client contacts the selected server requests movie, and plays it to the user.

The selection process consists of at most three phases. First, if there exist one or more servers in the region

of the client which hold a copy of the requested movie and which are not fully loaded, then the server with the lowest *relative load* is selected for playing, where the relative load is defined as the ratio between the server's current load (plus one) and its maximum load.

If no server can be matched in the local region, the second phase begins. The manager looks for the least-loaded server that can act as a proxy server, i.e., one that is connected to the private network. If a proxy is found, the manager requests from its own manager at the next level up to locate a server that can remotely transfer the requested movie through the private network to the proxy. The upper manager forwards the request to all other children nodes, in parallel. If such a server is found, DVS returns to the client the addresses of the proxy and the remote server. The client then contacts directly the remote server over the public Internet and requests it to deliver the movie through the proxy. Notice that while the movie is transferred to the client through the proxy (in order to improve QoS and reduce costs), the client still communicates with the remote server, not with the proxy. This design enables the user to control the session parameters and issue VCR control operations such as pause, rewind, fast-forward etc. If no server is found by any of the sibling first-level managers, the request is forwarded one level up, recursively, until a server is found or until the root is reached. Thus, each iteration expands the range of the search.

The third and least desirable option is when even an available (not fully loaded) proxy server cannot be found. In this case the request is forwarded to the next-level manager as-is, in the hope to find a remote server that can still play the movie directly to the client through the public Internet with an acceptable quality of service. The server-selection algorithm is summarized in Figure 2, showing the three phases.

```
Input:      M - movie name
Output:     transfer-mode, server, proxy (optional)

(1)
    Find a local server LS
        with a copy of M and with minimal relative load and
        with LS.load + 1 < LS.maxLoad
    if LS found return (DIRECT, LS)

(2)
    Find for a local proxy P
    if P found then
     begin
        request the manager at the next up level to locate a proper remote server RS
        if RS found return  (INDIRECT, RS, P)
    end

(3)
    Search for a remote server RS with a copy of M
    if found return (REMOTE-DIRECT, RS)
```

Figure 2: Server Selection Algorithm

Figure 3 further illustrates the direct and indirect (via proxy) modes through an example. In the upper figure, the request arrives at server 1.1 (server naming scheme is discussed later in Section 4) and gets forwarded to its manager, server 1. Server 1 finds that server 1.3 holds a copy of the movie and has an acceptable load. Server 1 then replies to 1.1, which replies to the user that server 1.3 is capable of playing the movie. Server 1

then informs server 1.3 that it is expected to receive a request from the user to get the movie, and server 1.3 reserves (for a limited period) the necessary resources for the request. Eventually, the user opens a connection with server 1.3 and gets the service directly from it. In the lower figure, the requested movie is not located in the local region, or all local servers that hold a copy are overloaded. However, server 1.3 can still serve as a proxy. The manager (server 1) forwards the request to its parent (in this case the root). The root observes that a copy of the requested movie can be played from server 2. Server 2 finds that server 2.1 is capable of playing the movie, i.e., it has a copy of the movie, it has a connection to the private network and its load is acceptable. Finally, the user contacts server 2.1 and requests it to play the movie through proxy 1.3.

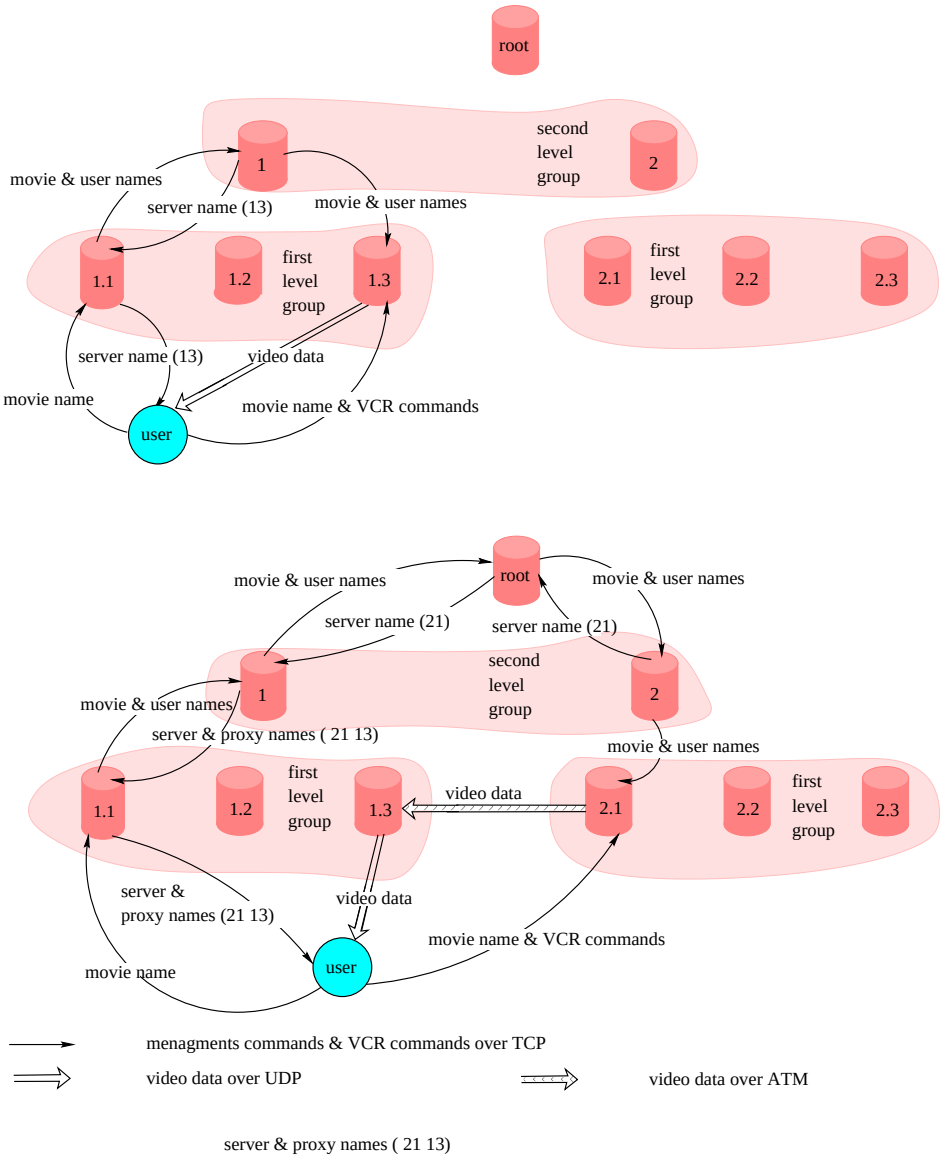


Figure 3: Sample Movie Requests

There are two problems in the above procedures. First, a security problem might arise since there is no guarantee that the user will follow the system instructions (selected servers) or it might change its prior preferences (such as movie selections). The second problem is with potential discrepancy between the actual server loads and the information on their load as known at the regional manager. in particular, the load might

change during the course of the request protocol, due to the possibly long delays between the assignment of servers to clients and their actual playing.

The second problem stems from the fact that the protocol actions may be interleaved with other concurrent requests from other clients. Thus, a standard solution involving locks on resources of the assigned server with a defined timeout is employed. Since typically the VOD service time is large, the wasted resource time of this locking operation is negligible. To address the security problem, the assigned servers may be passed user and session information which is kept with the lock and with other resources allocated for that user request. Upon the reception of the expected request, the server verifies its correctness and validity.

# 3 Content Distribution

So far we were concerned with the selection of a server assuming a proper distribution of the movies among the available servers. But what is the optimal distribution method that minimizes both storage and communication costs ? For one thing, such a method must take into account the changing demands for movies, which leads to a dynamic allocation scheme. Furthermore, given that movies can be moved, added and deleted (to make room for more popular movies) there may be an additional constraint, namely to keep at least one or more copies of each movie.

## 3.1 The distribution process

The distribution process operates as follows. Each server counts the number of requests for each movie, i.e., the *demand* per movie and sends the information to its manager. The sum of the demands is passed by managers to higher level managers. First-level managers (re)distribute the movies according to their demand and possibly the single-copy constraint. To simplify the discussion we first assume that there is no single-copy constraint, all storage is empty and there is no load constraint on the servers. We will later remove these assumptions. The last assumption means that it is sufficient to keep at most one copy of each movie in the group. Thus, the task of a first-level manager is to distribute among its members movies that are in high demand in its region, conforming to the total storage capacity. This task can be mapped to the "multiple knapsack 0/1" problem [10]. There are known branch-and-bound and bound-and-bound optimal algorithms to this problem but their intractability and our expected large input (e.g., thousands of movies in an on-line video store) lead us to seek suboptimal solutions. [10] also showed that dynamic programming techniques cannot be applied to this problem, due to the extremely high memory requirements. [10, 11] describe a greedy algorithm with complexity $O(n^2)$, where $n$ is the the number of movies. It is better than the ordinary greedy algorithm (as described in [10, 12]), in that the deviation from the optimum is bounded.

In practice, the maximal number of users supported by each server is bounded, therefore, it may be useful to keep multiple copies of most popular movies in a single group. Consequently, we have modified the greedy algorithm of [10] to the case of multiple choice constrains knapsack problem. Usually, knapsack problems assume empty knapsacks. In contrast, our servers generally store as many movies as possible, so the leader also needs to decide on the removal of movies from servers. Finally, since transfer of a movie over the network is a costly operation, our algorithm also attempts to minimize the amount of movie transfers.

## 3.2 Formal Problem Formulation

In the following we formalize our optimization problem for the distribution of movies in the servers. We use the following parameters in the movie distribution problem: there are $n$ servers in the system and we need to distribute $m$ movies. Denote by $c_i$ the storage capacity of server $i$ and by $b_i$ the maximal number of concurrent

requests that server $i$ can serve. Let $w_j$ be the storage size of movie $j$ and $p_{ij}$ be the demand for movie $j$ at server $i$ measured by requests per time unit. Finally we denote by a boolean variable $y_{ij}$ ($y_{ij} = 0, 1$) whether movie $j$ is currently located at server $i$. We need to find $x_{ij} \in \{0, 1\}$ and $i \in N$, $j \in M$, where $x_{ij} = 1$ if movie $j$ is to be assigned to server $i$, and 0 otherwise.

Our goal is to select $n$ subsets of movies so that the sum of the demand of the selected movies will be maximal. In addition, each subset must fit (in terms of storage) in a different server and the average load on each server must be less than the maximal load. Formally:

$$max \sum_{i=1}^{n} \sum_{j=1}^{m} x_{ij} * p_{ij}$$

subject to:

$$\sum_{j=1}^{m} w_j * x_{ij} \leq c_i, i \in N = \{1, \ldots, n\}$$

(servers storage size constraint)

$$\sum_{j=1}^{m} x_{ij} * p_{ij}/t \leq b_i, i \in N = \{1, \ldots, n\}$$

(servers load constraint) and

$$\sum_{i=1}^{n} x_{ij} \geq 1, j \in M = \{1, \ldots, n\}$$

(single copy constraint).

We make the following simplifying assumptions. First, the maximum number of clients that each server can serve is proportional to the size of its storage. This assumption relaxes the server load constraint and also makes sense in terms of system configuration. Second, as mentioned in Section 2, the system is constructed as a tree. Each server belongs to a group of servers, and each group contains an elected leader, which belongs to a group of nodes at the next level etc.. We assume that the network latency between a client and each server in the group are very similar. Therefore, any client can be served by any server in the group that holds the requested movie. Consequently, we can combine the requests $p_{ij}$ in a group using $p_j = \sum_{i \in Group} p_{ij}$.

### 3.3 The distribution algorithm

The complete movie distribution algorithm for the first level leader is presented in [13].The algorithm consists of six phases. First, the leader calculates the total numbers of copies of each movie which are located at the group's servers. Second, the leader computes the number of copies of each movie that are required at its group, using the formula

$$k_j = \frac{p_j * C}{w_j * P},$$

where C is the storage size of all servers in the group ($C = \sum_{i \in Group} c_i$), and P is the overall demand for movies in the group ($P = \sum_{j=1}^{m} p_j$). The leader sorts the above numbers, in decreasing order. If the number of copies of the highest demand movie is larger than number of servers in the group, the leader makes this number equal to the number of the servers in the group. Then the leader updates the number of copies of the other movies using the above formula for $k_j$ but C is set to the total size of storage in the group after we have assigned the first movie to all servers ($C = \sum_{i=1}^{n} c_i - n * w_1$) and P is set to the overall demand, except for the first movie.($P = \sum_{j=2}^{m} p_j$). The last procedure is repeated for all movies. Next, the leader subtracts from each $k_j$ the number of already stored copies for movie j. Then, it divides the movies list to two lists. The `addList` is

the list of movies for which we need to add copies and similarly the `removeList` is the list of movies for which we need to remove copies.

In the third phase, the leader attempts to place copies of the movies from the addList. If the mission is completed the algorithm is terminated; otherwise it continues to the fourth phase.

In the fourth phase, the leader decides which movies may be removed and from which server. After that it tries to assign the yet unassigned movies to the cleared space. If all movies are assigned, the leader goes to the sixth phase of the algorithm.

The fifth stage of the algorithm is based on [10]. In this phase the leader tries to improve the solution by testing exchanges between assigned and unassigned movies. We decided not to adopt other improvements presented in [10] because of the difference between the two problems.

In the sixth and last phase, the leader tries to keep movies from the removeList if the free space exist.

If the system is required to keep a single copy of each movie, the algorithm has the following additions: the top level leader starts the algorithm and assigns a single copy of these movies to its children. If possible, the leader assigns the movie to the child server with the maximal demand among servers that haves the ability to connect to the private network and haves enough free storage space. Lower level leaders, except the first level, recursively assign the movies they got to lower levels. When the first level leader calculates the number of desirable copies as described above, it considers its one copy requirement as a given minimum. The complexity of the movie distribution algorithm is $O(nm^2)$. This high algorithmic complexity results from the fifth phase. This phase is not always necessary and can also be stopped at will.

### 3.4 Experimental Results

For the purpose of experimentation we constructed a group of four servers, each with a storage size of 850-950 MBytes and with the ability to serve simultaneous by 5 clients. In addition, we provided another single server group with a storage size greater than the total size of all movies (7GBytes) and the ability to serve 100 clients. The last server enable us to disregard the single copy constraint. In the experiments we used 50 movies of sizes ranging randomly between 50 to 150 MBytes, and user requests that arrive to the system according to a random Poisson process. The demand for each movie changes during the day according to a random Zipf distribution as described in [13, 2]. Since in reality a user can stop viewing at any moment and our system may also contain interactive movies, that makes the viewing time variable, we assume that a movie is viewed for an exponential distributed time, proportional to its storage size. We assume that the average viewing time of a movie of size 1GByte is 90 minutes. In our tests we measure the number of misses in the first server group, i.e., the cases where the system needs to use the private network to serve the clients or cannot serve the requests at all. In the first test the demand for the movies was set in a static way according to average demand during the whole 24 hours. The servers contained no movies (except for the second group server). The system was operated for a time equivalent to 4 days (at which near steady state results were observed). The results are presented in figure 4. From the figure we can see that in the interesting area (between 200-1000 requests per day) the amount of misses increases very slowly with the increase of the demand. In the high demand area the system approach a saturation, i.e., the misses are caused because servers are overloaded. In [13] we also examine the influence of changing the server storage size, the maximal number of clients per server and the movies size. The results are very similar in nature to the one presented above and therefor omitted to save space.

In the second test we fixed the average number of requests per day to 1000 (a point selected from figure 4 in the region of interest). However, the day was divided to 8 3-hours intervals, where in each interval the demand per movie was changed. Per each interval we measured the number of misses and depicted them as a percenting of the number of misses in a system where time intervals are not observed. In other words, in the reference system the demand per movie is averaged over 24 hours intervals and the variation along the day is not taken

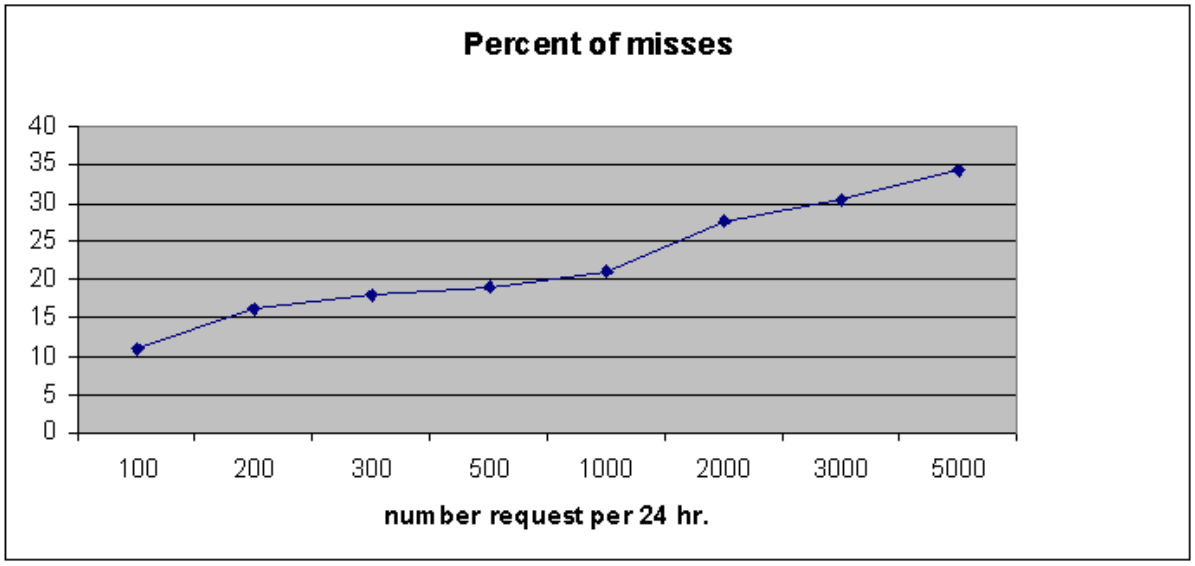


Figure 4: Percent of the misses in a static demand system

into account. (Note, that the system of 4 is exactly such system.) The results of the second test are presented in figure 5.

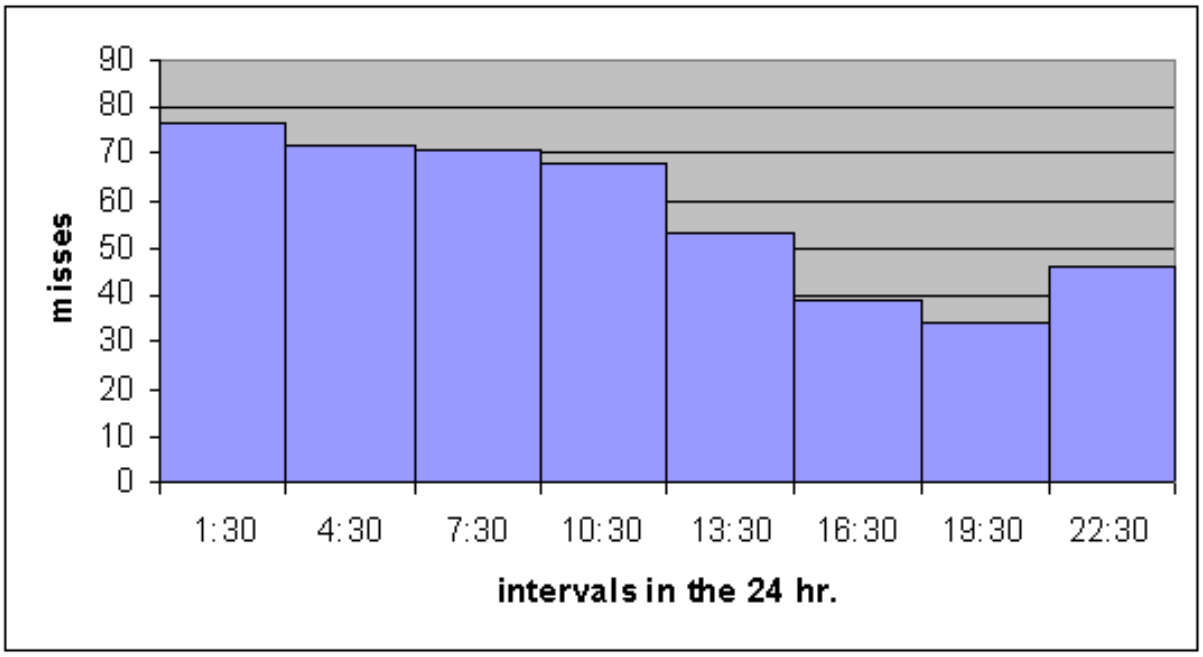


Figure 5: Comparison of a static and adoptive systems

We observe that is the demand per movies is changing even more over the day the adaptation can still improve the miss rate considerably.

# 4 Reliability Support

The hierarchical structure of the intra-server network introduces serious potential reliability problems, since failures of each leader node might bring the system down. To address this issue, we present a solution that is based on the ability to grant leadership to all servers. At system configuration time, each server is assigned a leadership priority number that reflects the system administrator's choices. Every time a server comes up, it joins the leader election process. If it loses in that process it expects to hear from the elected leader periodically and to respond to it. If it does not receive anything within a predefined period, it restarts the election algorithm anew. Each leader maintains an active node list to which it sends the periodic messages. If an active node does not respond, it is removed from the list. In the literature we can find election algorithms with message complexity $O(N \log N)$ [14]. These algorithms can also be adopted to handle leadership priority number at some constant cost. We decided to use an algorithm with higher complexity ( $O(N^2)$ ), since we expect the number of servers in a group to be low (5 - 20), and our algorithm has a much smaller constant factor and is simpler. It is also optimized for the use of Java Remote Method Invocation (RMI) [15].

Before discussing the algorithm, we need to explain how nodes are named and located. A similar hierarchical system of PNNI [16] is using a link state protocol for discovering other nodes ( communication switches). In contrast, the nodes in our system (which are servers) do not have point to point links between them. Therefore, our system needs to supply other means for nodes to locate their neighbors. We present two alternatives for solving this problem.

The first is suitable for relatively small systems. Each server is provided with a list of neighbors from a configuration file. If this list is not complete it can supplement it using information which it gets from the neighbors of its list. This method is simple and easy to implement, but it is not scalable to very large systems.

The second method uses a predefined hierarchical naming structure similar to PNNI [16], and the principle of root nodes, similar to DNS. There are several root nodes which are known to each node in the system whose leadership priority number is higher than all other nodes. In addition to its IP address each node is provided with another identification, which is a concatenation of two parts. The first part is a node number, which is unique in that group, and the second part is the group identifier. When a new leader is elected, the leader represents its group and at that level sets its identifier equal to the group identifier, which in turn can be divided recursively. The leader is a member in the next level group, i.e. with other elected leader. The group with identifier 1 is the top-level group, and its elected leader is the top-level leader, assigned as a root. Figure 6 shows an example of this naming policy.

When a node starts the election algorithm and is not familiar with other nodes in the group, it elects itself as leader, creates a next level node and starts the election algorithm in the next level. When the process arrives at one of the root nodes, the root node checks that in its group there are not multiple nodes with the same identification. If there are two or more nodes with the same identifier, the root node makes them aware of each other. For reduce the load on the root nodes, when a node is elected as leader, all nodes in its group record their name in the configuration files. Recent leaders can be kept for future election rounds. When a node starts the election algorithm, it checks the stored recent leader list before communicating with the root node.

Figure 7 shows two nodes (1.1, 1.2) and an example of their configuration files and single root node. Before the election algorithm starts, each node knows only its IP address, its identification and IP address of the root node. The nodes are not aware of each other. Consequently, each node elects itself as a leader and creates a next level node. The higher level nodes get the same identification name (1). When they apply to the root node, they recognize that they belong to the same group and make it aware of it. Node 1.1 is elected and after that its IP address is written in the configuration files of all the nodes in the group.

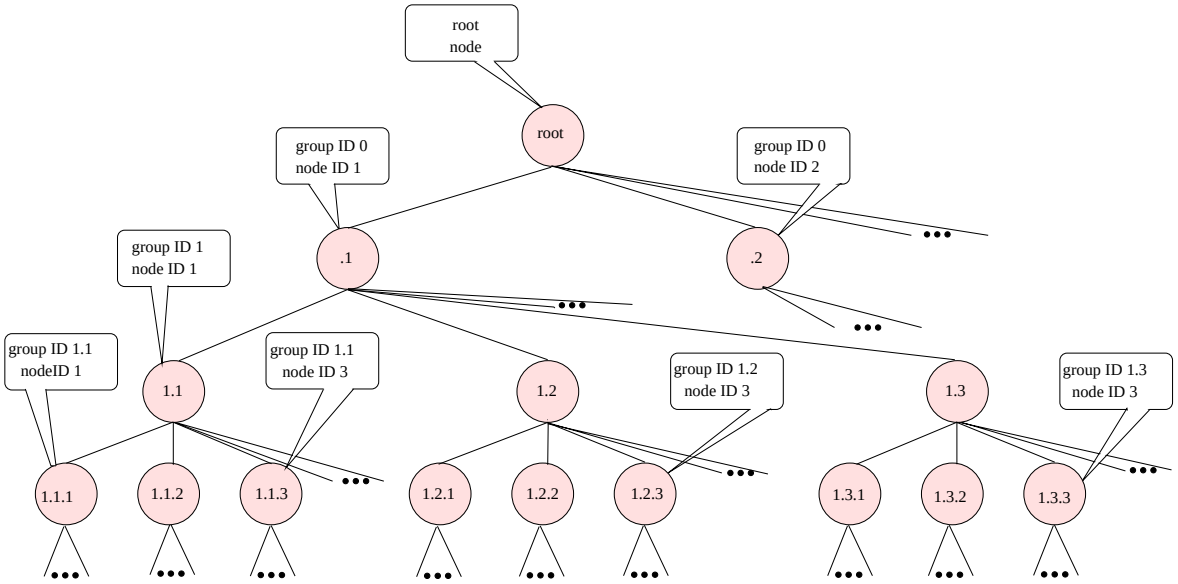


Figure 6: ID address example

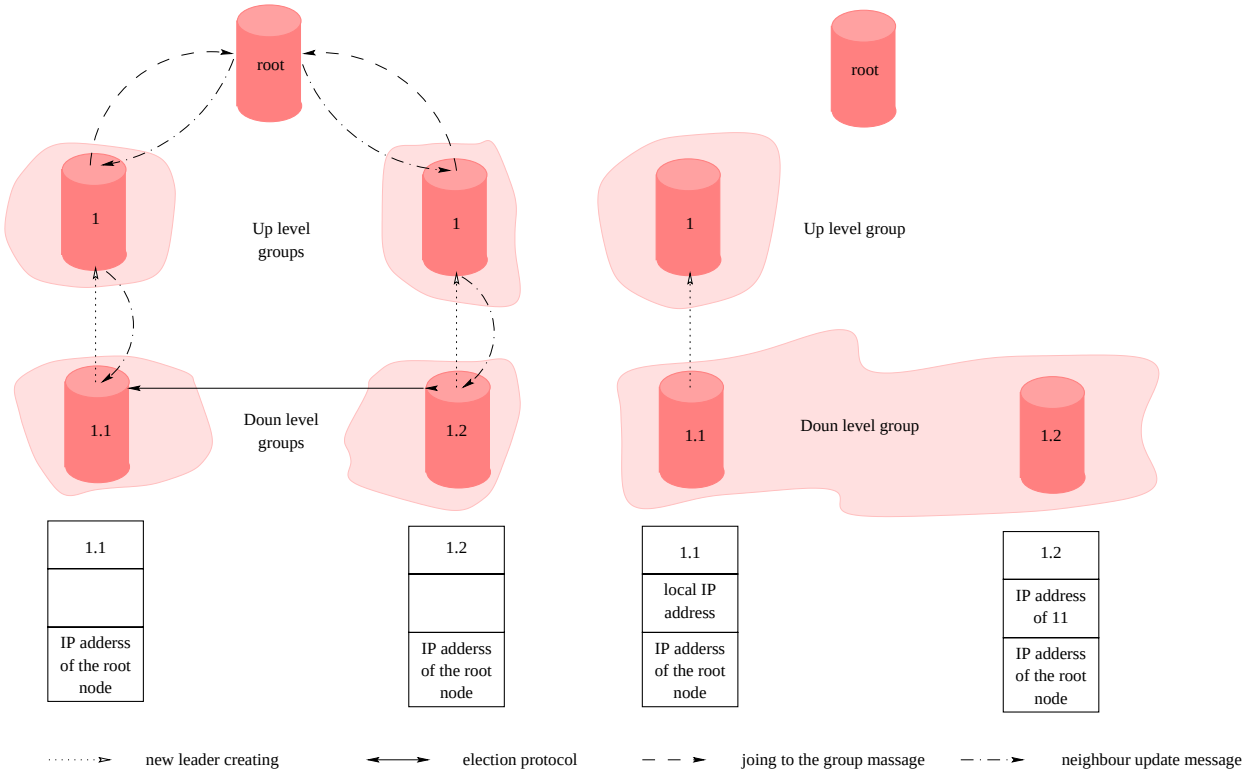


Figure 7: Example of node location

# 5 System Implementation

We have implemented all management activities of the DVS system with Java 1.1 [17, 18]. Given that the computation cost is relatively small compared to the communication cost involved in the management activities

(movie distribution and server selection), the current performance of Java did not present a practical problem. In contrast, we benefited greatly from its platform independence, built-in multi-threading support, and networking support. In particular, we made heavy use of the Remote Method Invocation (RMI) [15] package, which enabled us to program the system as a uniform set of communicating objects regardless of whether they are co-located or physically distributed. In addition, we exploited the powerful RMI capability to pass remote object references as parameters. This significantly improved connection time and eased the programming effort since it eliminated the need to establish network connection for each interim interaction between servers or between clients and servers.

Unlike management and distribution functions, the actual delivery of the video to users must be performed in real-time with performance being a critical factor. Early experiments that we conducted with Java-based video-servers [19] showed an intolerable delay, among other reasons due to the transfer of data from the native calls to Java. Given that video playing over the network per-se was not a research goal in this project, we used the public domain VCR MPEG-1 player from the Oregon Graduate Institute [20] and extended it with the capability to redirect its output to ATM (to the proxy). Although VCR is implemented in C and runs only on Unix, the rest of the system is platform-independent and thus can be deployed over the Web as long as the VCR client is pre-installed as a "helper" application on the browser. In any case VCR is loosely coupled with the rest of the system and could be replaced with the upcoming Java Media Framework [21], resulting in a fully portable system.

Finally, an important component of the DVS system is its runtime monitor, which is capable of tracking dynamically various properties such as changing user demands in a server, load on the servers, and automatic restructuring, in any of the servers that comprise an instance of DVS. Figure 8 shows a snapshot of the monitor's GUI. The left frame lists the server groups and their hierarchical structure (two groups of 4 and 1 members, respectively). Notice that manager nodes in the hierarchy (e.g., `comnet9`) appear also as edge nodes, as explained in Section 4. If some manager fails and a new node is elected, the hierarchy is refreshed and redrawn in the monitor. The right frame shows the movie distribution in a selected server, `tochna11`, and the demand for each movie in that server. For example, movie `T41` is local, which is not surprising given its relatively high demand (indicated by the value 133 in the `Request` column). In contrast, there is no local copy of movie `T3` in that site, reflecting the low demand (6) in this server. The monitor is also capable of manually transferring movies for testing and experimentation purposes.

# 6 Future Work

One of the possible extensions to our system is to improve the movie distribution algorithm, for example by splitting the movie space into geographical regions, since many movies have only local popularity and should not affect other areas. Another improvement is to optimize the movie and load distribution according to the private network's link capacity and cost.

We also plan to explore and incorporate security and robust fault-tolerance components to the next version of DVS. Finally, a major extension is to develop a more general framework that will support distribution and management of electronic comodities (besides VOD) in the general context of network computing, for example to support commerce in software components [22] or purchase services such as remote file storage.

# References

[1] George Young and Lanie Kurata. Building advanced-media solutions, February 1998.

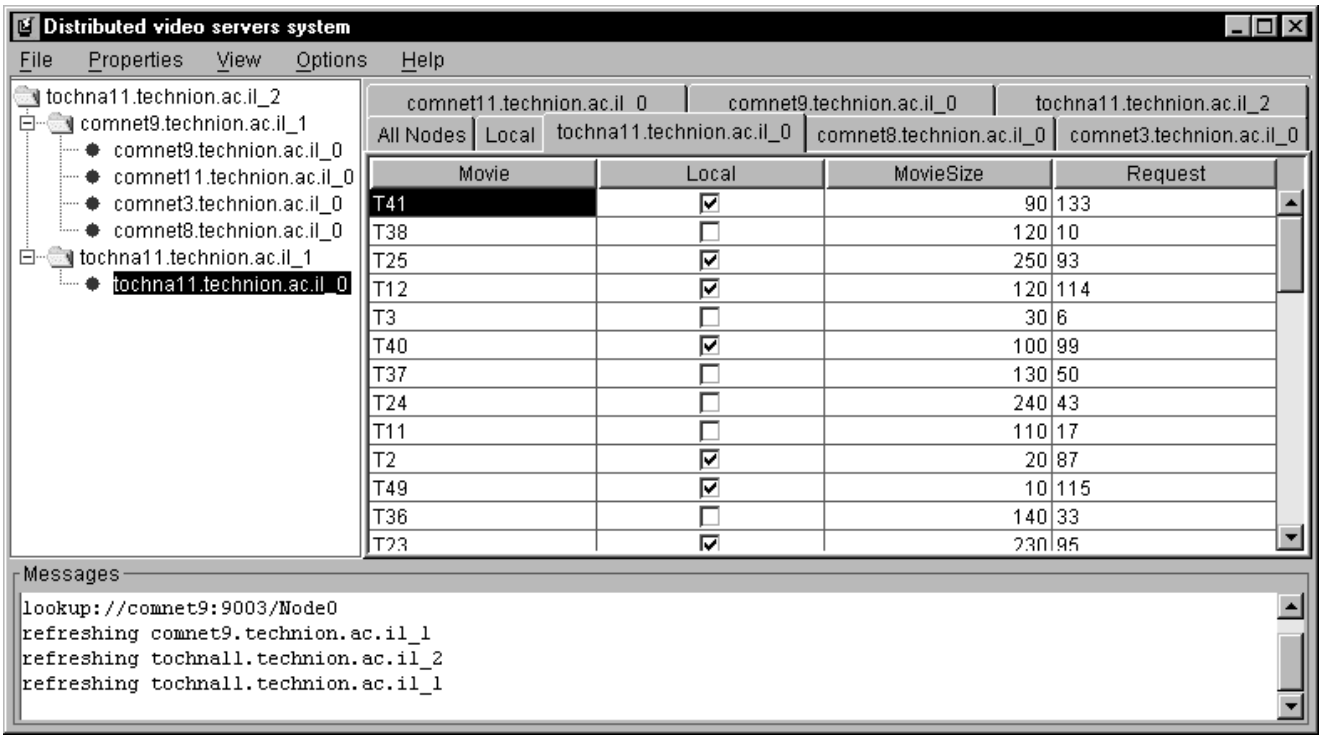


Figure 8: The DVS Monitor

[2] J.L. Wolf, P.S. Yu, and H. Shachnai. Disk load balancing for video-on-demand systems. *ACM Multimedia Systems Journal*, 5(6), 1997.

[3] R. Staehli. *Quality of Service Specification for Resource Management in Multimedia Systems.* PhD thesis, OGI, January 1996.

[4] S.Cen, C.Pu, R.Staehli, C.Cowan, and J.Walpole. A distributed real-time mpeg video audio player. In *Appeared at the Fifth International Workshop on Network and Operating System Support of Digital Audio and Video (NOSSDAV'95)*, Durham, New Hampshire, USA, Aprill 1995.

[5] T. Mojsa and K. Zielinski. Web-enabled, corba driven, distributed videotalk environment on the java platform. In *Proceedings of the 6th International World Wide Web Conference*, pages 143–151, Santa Clara, CA, April 1997.

[6] Renu Tewari, Daniel Dias, Rajat Mukherjee, and Harrick Vin. High availability for clustered multimedia servers. In *In the Proceedings of International Conference on Data Engineering. New Orleans*, February 1996.

[7] Renu Tewari, Daniel Dias, Rajat Mukherjee, and Harrick Vin. Design and performance tradeoffs in clustered multimedia servers. In *In the Proceedings of IEEE-ICMCS - Tokyo*, June 1996.

[8] Jim Gemmell, Harrick M. Vin, Dilip D. Kandlur, and P. Venkat Rangan. Multimedia storage servers: A tutorial and survey. *IEEE Computer*, May 1995.

[9] Eenjun Hwang, B. Prabhakaran, and V.S. Subrahmanian. Presentation planning for distributed video systems. Technical report, University of Maryland, Computer Science Division, December 1996. Report CS-TR-3723.

[10] Silvano Martello and Paolo Toth. *Knapsack Problems. Algorithms and Computer Implementations.* Wiley, 1990.

[11] Silvano Martello and Paolo Toth. Heuristic algorithms for the multiple knapsack problem. *Computing*, 27:93–112, 1981.

[12] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, MCGraw-Hill Book Company, 1989.

[13] A. Roytman. Dvs. a system for distribution and management of global vodeo on demand services. Master's thesis, Technion - Israel Institute of Technology, October 1998.

[14] Gerard Tel. *Introdaction to Distributed Algorithms.* Cambridge University Press, 1994.

[15] javasoft. *RMI - Remote Method Invocation.* http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/.

[16] The ATM Forum Technical Committee. Atm forum pnni specification, march 1996. Version 1.0 af-pnni-0055.000.

[17] Mary Campione and Kathy Walrath. *The Java Tutorial : Object-Oriented Programming for the Internet (Java Series).* Addison-Wesley Pub Co, http://www.javasoft.com/docs/books/tutorial/index.html, second edition, March 1998.

[18] Ken Arnold and James Gosling. *The Java Programming Language (Java Series).* Addison-Wesley Pub Co, second edition, 1997.

[19] Abraham Baum. Talking head, one-to-many real-time video over the internet using java. Technical report tcs-9000ltd, Software Laboratory Departments of Electrical Enginnering of Technion - Israel Institute of Technology, 1998.

[20] Shanwei Cen. *A Distributed Real-Time MPEG Video Audio Player.* ftp://ftp.cse.ogi.edu/pub/dsrg/Player.

[21] javasoft. *Java Media Framework API.* http://java.sun.com/products/java-media/jmf/index.html.

[22] I. Ben-Shaul, A. Cohen, O. Holder, and B. Lavva. Hadas: A network-centric framework for interoperability programming. *International Journal of Cooperetive Information Systems*, 6(3-4):293–314, December 1997.