Scalable Computing: Practice and Experience
Volume 6, Number 2, pp. 83–92. http://www.scpe.org
ISSN 1895-1767
© 2005 SWPS

A CORBA-BASED MIDDLEWARE FOR AN ADAPTIVE STREAMING SERVER

BALÁZS GOLDSCHMIDT† , ROLAND TUSCH‡ , AND LÁSZLÓ BÖSZÖRMÉNYI‡

Abstract. A CORBA-based infrastructure for an adaptive multimedia server is introduced, enabling dynamic migration or replication of certain multimedia applications among a set of available server nodes. The requirements from both the server's and the middleware's point of view are discussed. A specification of a CORBA interface and a corresponding implementation is presented. Finally, as a proof of concept, measurements of the implementation are shown and discussed. The measurements show the importance of the related servers' distribution on the network.

Key words. Multimedia, CORBA, proactive adaptation, mobile agents

1. Introduction. In [15, 16] we presented a distributed multimedia streaming server architecture, which builds upon the concepts for distributed multimedia servers described in [7]. The key extension of the presented architecture therein is the capability that server components (also referred to as applications) may be dynamically migrated or replicated to other server nodes on demand. Program and data dependencies are also considered. The proposed adaptive multimedia server architecture (ADMS) mainly consists of four distinguished types of components: one *cluster manager*, a set of *data managers*, *data collectors*, and *data distributors*. The cluster manager controls the layout of the ADMS cluster concerning the locations of instances of the three other component types (see figure 2.3). Data managers provide facilities for efficient storage and retrieval of units of media data. A data distributor distributes a media stream received from a production client to a given set of data managers. Finally, data collectors perform inverse operations to distributors. A collector retrieves stripe units from data managers, reassembles and streams them to the requesting client(s). Thereby it can serve as a proxy which caches media streams possibly in different qualities, in order to satisfy a number of client requests with varying QoS demands.

Since the proposed architecture employs the full server utility model [14], each server node can run at most one component at a time. Following this model and focusing on especially the data collector component, there are two main questions regarding an optimal provision of quality of service to the requesting clients: (1) Does the component run on optimal locations? (2) If not, what locations should be recommended to run the component, and what are the costs for performing a component migration or replication? These questions typically can not be answered by the server nodes themselves, since for an answer both local and global views on the performance behaviour of the distributed server architecture are needed. Thus, a kind of infrastructure is required which monitors the resource usage on each server node, as well as the quality of the connection links between the nodes.

Infrastructures for managing adaptive multimedia servers are rather spare. Adaptive servers usually support code migration or replication, however, they hardly provide QoS support. An example for such infrastructures is Symphony [3], which provides a number of services for executing virtual servers in internet settings. Symphony does not support a QoS-constrained replication/migration of a multimedia service. The same is valid for Jini[17], a middleware for maintaining a dynamic network of resources and users. Jini's object replication algorithms rely on a network of reasonable speed and latency. QoS-aware middleware frameworks enabling QoS-driven adaptations, as presented in [8], focus on resource management dynamics, rather than on application-level dynamics.

In [1] it is illustrated that mobile agents are very well suited for network management issues like fault, configuration and performance management. Monitoring application demands, network and server loads are predestinated tasks for a mobile agent-based middleware. These lead us to the idea of using such a middleware for an adaptive multimedia server. Since the process of replication/migration is not intended to happen in real-time, an extended version of the CORBA-based mobile agent system Vagabond [4] is used. Vagabond is a 100% pure Java mobile agent system, first developed to be used in distributed multimedia database systems. However, its modular design allows to modify and use it as a middleware for an adaptive multimedia server, resulting in its successor *Vagabond2*.

[1]Budapest University of Technology and Economics, Department of Control Engineering and Information Technology; balage@inf.bme.hu
[2]Klagenfurt University, Institute of Information Technology; {roland, laszlo}@itec.uni-klu.ac.at

This paper addresses the requirements of the adaptive distributed multimedia streaming server ADMS to the underlying infrastructure Vagabond2, as well as the needs of Vagabond2 to ADMS. The remainder of this paper is organized as follows. Section 2 discusses the requirements of ADMS to the underlying infrastructure. In section 3 the needs of Vagabond2 to ADMS are disputed. Section 4 presents the CORBA interface specification between ADMS and Vagabond2. Section 5 deals with the internal architecture and implementation of Vagabond2. In section 6 the results of first measurements in the ADMS testbed are presented. Section 7 concludes this paper including a plan on future work.

2. Requirements to the Underlying Infrastructure. In [16], the limitations of a static distributed multimedia streaming server (SDMS) are discussed. Consider figure 2.1, which illustrates a retrieval scenario of an MPEG-4[9] presentation (Sample.mp4) from a SDMS. The presentation consists of only one video elementary stream, whose data is striped among two data manager instances *dm1* and *dm2*. The meta data of all available presentations on the SDMS is stored in an MPEG-7 meta data database, and can be queried by a so-called *meta data manager* (mdm). The *mdm* is not a separated server component, but simply an object which is typically integrated into the cluster manager component. Further, there exists one instance of a cluster manager (*cm*), data collector (*dc1*), and adaptation engine (*ae*). The adaptation engine tries to find an optimum data collector for a given client request, taking into account the locations of the data managers, among which the data of the requested media stream is striped. Similar to the *mdm*, the *ae* needs not to be a separated server component, and can also be an integral object of the *cm*. However, for the reason of executing computation-intensive optimization algorithms, it might be designed as a separated component, which can also be migrated or replicated on demand.

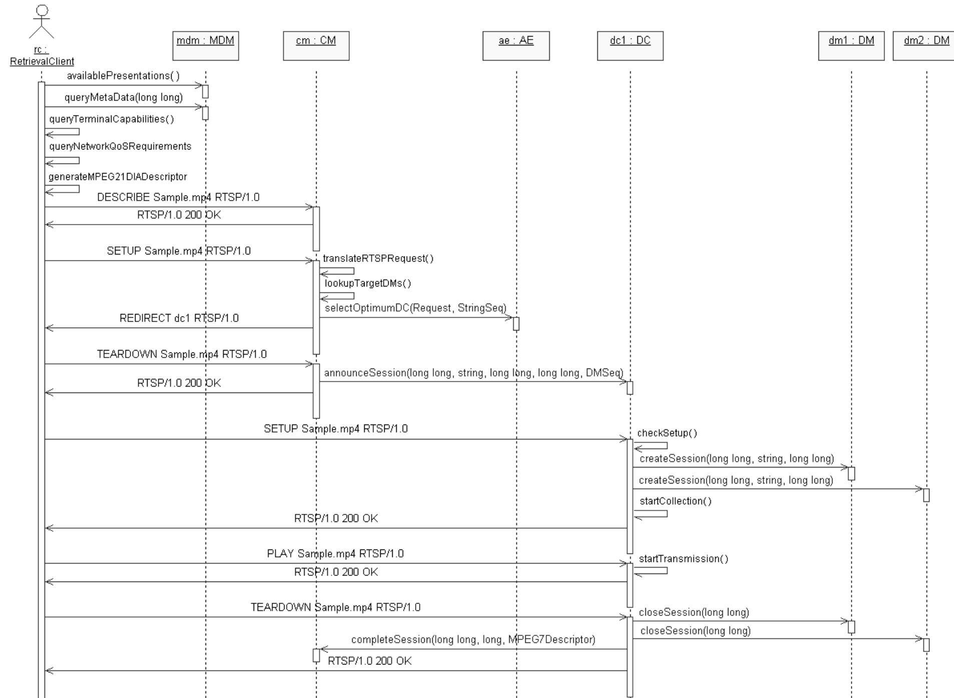


Fig. 2.1. *Interaction Scenario for Retrieving a Presentation from a SDMS*

A retrieval client initially queries the *mdm* for a set of available presentations on the SDMS. For a particular presentation the client asks for metadata (in MPEG-7[10] format), describing for example a temporal decomposition of the stream into segments. Afterwards, the client collects the terminal capabilities of the client device and the network QoS requirements imposed by the requested presentation. Since QoS constraints are used for data collector selection by the *ae*, they have to be communicated using a standardized descriptor. For example this can be an MPEG-21 descriptor [11, 12], as contained in the RTSP SETUP message illustrated in figure 2.2.

The MPEG-21 descriptor is rooted by the *DIA* element. The capability element of the descriptor says that

```
SETUP rtsp://sdms.itec.uni-klu.ac.at/Sample.mp4 RTSP/1.0
CSeq: 2
Range: npt=20-40;time=20030301T140000Z
Content-Type: application/mpeg21_dia
Content-Length: 557
<?xml version="1.0" encoding="UTF-8"?>
<DIA xmlns="urn:mpeg:mpeg21:dia:schema:2003" xmlns:mpeg7="urn:mpeg:mpeg7:schema:2001"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="urn:mpeg:mpeg21:dia:schema:2003 UsageEnvironment.xsd">
   <Description xsi:type="UsageEnvironmentType">
      <Network>
         <Capability maxCapacity="128000" minGuaranteed="32000"
               inSequenceDelivery="false" errorDelivery="true"/>
         <Condition>
            <Utilization maximum="1.0" average="0.5" avgInterval="1"/>
            <Delay packetOneWay="500" delayVariation="100"/>
            <Error packetLossRate="0.02"/>
         </Condition>
      </Network>
   </Description>
</DIA>
```

Fig. 2.2. *A RTSP SETUP Message Including an MPEG-21 DIA Descriptor*

a bandwidth range between 32 and 128 kbit/sec is acceptable, packets might be delivered out of order and may be lost. The condition element specifies that the 128 kbit/sec link may be fully utilized, but on average should only get 64 kbit/sec. The packet delay from a data collector to the client should not be greater than 500 msec, and delay variation not greater than 100 msec. Finally, the packet loss rate must be below two percent.

The RTSP message further indicates that the client wants to get the stream segment from the 20th to the 40th second, relative to the media time. Additionally, the request should be scheduled for March 1st, 2003 at 14 o'clock. This time indication at session setup enables the *ae* to proactively schedule the request to an appropriate data collector, taking into account the given QoS constraints, and the locations of the required data managers. The *cm* achieves this by calling the method *selectOptimumDC(Request,StringSeq)* on the *ae*. The *ae* solves the dynamic server selection problem based on a statically configured set of data collector hosts. It achieves this by communicating with a resource broker (not illustrated in the figure), which provides load information of each server host, as well as host distance information between server-server, and server-client hosts. If there exists at least one data collector which can handle the request, the *cm* redirects the client to the recommended data collector (*dc1* in this case). The limitation of a SDMS lies exactly in the result of this method call. If the method call results in an empty set, the client request has to be rejected, due to server resource limitations. In an adaptive distributed multimedia streaming server (ADMS) environment, where a new facility (e.g. data collector) may be created on demand, it may be possible to accept the client request.

The key objective of ADMS is to maximize the number of acceptable client requests by replicating/migrating server components and media objects on demand to idle server nodes. These actions allow to minimize startup latencies and to reduce network traffic along the delivery paths to the clients. The goal can be achieved by applying a number of adaptation strategies, such as keeping server nodes load balanced, or letting clients be served by their nearest data collector. To perform these adaptation steps, a set of services has to be provided by the underlying infrastructure, which can be divided into two major classes: *application services* and *adaptation services*.

Consider figure 2.3, where an ADMS cluster C_j consists of a set of nodes M either being idle or running a certain server component. Each node is equipped with the ADMS runtime environment. Consider also 1 node being reserved for the *cluster manager*, k nodes running the *data manager* component, and l nodes running a *data collector* component ($k + l + 1 < M$ must hold). The *application service* of the underlying infrastructure has to provide means for replicating or migrating a server component from a host to one of the $M - k - l - 1$ remaining hosts, which are in an idle state. This is illustrated in figure 2.3 by replicating the *data collector* component to cluster node CN_{M-1}. Since a multimedia component may be dependent on program and data files, it also has to provide facilities for carrying those dependencies during the replication/migration processes. Migration or replication of stateful components in real-time is not required.

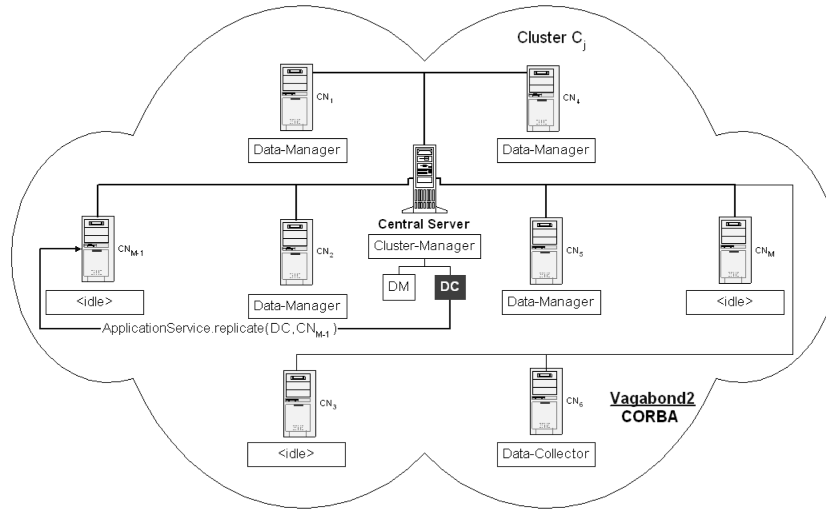


Fig. 2.3. *Replication of a Data Collector to the Idle Node* CN_{M-1}

The *adaptation services* have to provide means for automatically controlling an optimal distribution of server components, as well as host recommendations in the case of a required component adaptation. Thus, the adaptive streaming server must also be able to delegate the adaptation control over a certain component (typically the data collectors) to the underlying middleware. On the other hand, the given recommendations have to take into account server resource information (e.g. CPU, network, disk and memory usage) of each server node, global information concerning the quality of the network connections between the cluster's server nodes, as well as the topological distances and QoS parameters of the clients served by ADMS.

3. Needs by the Underlying Infrastructure. The major problem for the middleware—concerning the requirements of the adaptive server—is the one of recommending a set of server nodes for a required application-adaptation. The data collectors described in §2 are distributed in the network using the services of Vagabond2. One central component of the infrastructure, namely the so-called *host recommender*, should recommend host machines (*Harbour*s) where these applications should be loaded. In particular, the host recommender should (1) provide recommendations concerning the set of hosts that should be used by the data collectors, (2) give hints on the costs (in time) for a certain adaptation, and finally (3) dynamically adapt an already loaded application according to the previous two offers.

The problem of recommending the optimum set of server nodes can be modelled as a capacitated facility location problem (or k-median problem, where k is a given constraint). In [6] it is shown that this problem is NP-hard for general graphs. However, several approximation algorithms have been studied recently and the best published approximation factor to date is 4, with a running time of $O(N^3)$ [2].

In order to perform approximative recommendations for application locations the host recommender needs information about the server nodes, the network, as well as the current and future set of requesting clients. Concerning the server nodes the infrastructure itself can monitor the usage of the server's resources (CPU-load, disk and memory usage). Regarding the quality of the network links between the server nodes a network monitoring agent can be used to periodically perform observations on network throughput and latency times. The major requirements of Vagabond2 to the managed multimedia components are (1) the estimated resource usage of each component, (2) the topological locations of the current and future set of clients serviced by the components, and (3) the QoS-parameters of the client requests (bandwidth, delay, loss rate). Section 4 provides a more detailed specification of these requirements.

4. Interfacing ADMS and Vagabond2. Our intention when designing the interface between ADMS and Vagabond2 was to minimize the coupling between them. Some parts of the interface are implemented on the side of the media server, some on the side of Vagabond2. Either party only depends on the interface when communicating with the other one. There are three major parts of the interface definition (given in CORBA IDL syntax): *basic*, *application-specific*, and *adaptation-specific*.

The *basic part*—illustrated in figure 4.1—holds interfaces that are implemented on the media server's

side. They provide information about the applications Vagabond2 should run, and the clients using these applications. Date defines a structure to hold data about a time instance. RequestInfo contains QoS requirements of a client request. TimedRequest includes a set of requests scheduled for the same date (e.g. a video and an audio elementary stream of an MPEG-4 presentation). Client defines the information to be provided about an ADMS client. Interface *ApplicationInfo* provides all required information for running and managing a server application on a remote host.

```
module vagabond2 {
  // exceptions and basic type declarations

  struct Date {
    short seconds;
    short minutes;
    short hours;
    short day;
    short month;
    long year;
  };

  struct RequestInfo {
    long long avgBitRate;
    long long maxBitRate;
    float maxLossRate;
    long maxDelay;
    long maxJitter;
    long maxLatency;
    long long duration;
  };
  typedef sequence<RequestInfo> RequestInfoSeq;

  struct TimedRequest {
    Date startDate;
    RequestInfoSeq requestInfos;
  };
  typedef sequence<TimedRequest> TimedRequestSeq;

  struct Client {
    string hostAddress;
    TimedRequestSeq timedRequests;
  };

  interface ApplicationInfo {
    string getApplicationName();
    string getMainClassName();
    OctetSeq getApplicationJAR()
      raises (ServerIOException);
    OctetSeq getDependentFilesZIP()
      raises (ServerIOException);
  };
  typedef sequence<Client> ClientSeq;

  interface AdaptiveApplication {
    void start(in StringSeq params)
      raises (ServiceAlreadyStartedException);
    void stop()
      raises (ServiceNotStartedException);
    boolean isIdle();
    ApplicationInfo getApplicationInfo();
    ClientSeq getClients();
    void setLocked(in boolean locked);
    boolean isLocked();
  };
}; // vagabond2
```

FIG. 4.1. *The Basic Part of Vagabond2's CORBA-based Interface*

The interfaces in the *application specific part* are implemented in Vagabond2. They provide services for loading, evacuating and locating adaptive server applications. Interface *ApplicationLocator* provides means for locating all hosts on which a certain application currently runs on. Each application that has been loaded on at least one host has an own application locator. Interface *ApplicationService* provides means for loading a given application on a given host, evacuating an application from a given host, locating server application instances, and retrieving all applications on a given host. It is the centre of the application part, whose IDL specification is shown in figure 4.2.

The interfaces of the *adaptation specific part*—presented in figure 4.3—are implemented in Vagabond2. They provide services for adapting applications loaded by Vagabond2 with respect to an optimal distribution. Interface *HostRecommender* is responsible for recommending a set of hosts for a certain application by using topological and resource usage information. It also checks whether an application has optimal distribution. AdaptationPolicy represents an adaptation policy to be used by the adaptation service. Interface *AdaptationService* provides access to objects being responsible for and dealing with application-adaptational issues. It can return a host recommender, may suggest a policy by taking into account the data of clients, and can adapt an application by reorganizing its distribution.

We are currently working on adaptation strategies based on statistical information about the nodes and network. This information is collected by a separate part of the system, called ResourceBroker. We have recently tested a greedy algorithm that proved that the TCP-based connections between data collectors and data managers should be regarded differently from the RTP/UDP-based connections between the data collectors and the clients. We are now testing how different weights on different link-properties can improve the adaptation process of the media server architecture.

```
module vagabond2 {
  module services {
    module management {
      module application {

        interface ApplicationLocator {
          string getApplicationName();
          StringSeq getLocations();
          AdaptiveApplication getApplicationObject(in string hostAddreess)
            raises (NoSuchHostException);
        };

        interface ApplicationService {
          boolean load(in vagabond2::ApplicationInfo ai, in string hostAddress)
            raises (ApplicationAlreadyLoadedException, NoSuchHostException);
          boolean evacuate(in string appName, in string hostAddress,
            in boolean force)
            raises (NoSuchApplicationException, ApplicationNotLoadedException,
              NoSuchHostException, ApplicationNotIdleException);
          StringSeq getApplications(in string hostAddress)
            raises (NoSuchHostException);
          ApplicationLocator getApplicationLocator(in string appName)
            raises (NoSuchApplicationException);
          boolean runsOn(in string appName, in string hostAddress)
            raises (NoSuchApplicationException, NoSuchHostException);
        };
      }; // application
    }; // management
  }; // services
}; // vagabond2
```

FIG. 4.2. *The Application Module of Vagabond2*

5. The Architecture of Vagabond2. Vagabond2 is based on Vagabond, a mobile agent system developed at the Budapest University of Technology and Economics [4]. It is a CORBA-based[13] mobile agent system written in Java and was originally developed to be used in multimedia database environments [5]. Vagabond2 is also CORBA-based and written in Java. It is able to migrate CORBA objects, and pass their references to their clients. It keeps track of the available server hosts, the applications they are running and the CORBA objects that incarnate the applications. These functionalities are hidden behind the interfaces described in §4.

Figure 5.1 illustrates the implementation class hierarchy of Vagabond2. For application/component management two public interfaces are provided: AppService and AppLoc ("Application" is abbreviated to "App", "Locator" to "Loc" in this section). The system consists of server hosts that are able to load and run Java-CORBA objects, and applications that run on some of the servers. An application therefore consists of several CORBA objects on different hosts, where the objects can be thought of as 'incarnations' of the application.

The two major responsibilities are (1) providing the server hosts that can load Java objects and publish them as CORBA objects, and (2) maintaining the information of the relations among applications, their incarnations, and server hosts. The former required a simple modification of the Vagabond system. We defined a new interface *Harbour*, which will take care of this responsibility. Concerning the second responsibility it is clear that there are one-to-many relations between an application and its incarnations, and between a host and the incarnations running on it. For the sake of efficiency we decided to introduce a set of redundant relations: {application, hosts}, {application, incarnations}, {host, applications}.

The *Harbour* CORBA interface defines the methods provided by objects that are able to receive, run, and evacuate Java objects, and connect them to the ORB. It represents the runtime environment for adaptive server applications. Every server host must run a Harbour. Recently new extensions were added to this interface. We have defined several time points when migrating an application: download, local storing, class loading and object incarnation times. A Vagabond2 client can access these times for every server application loaded on a Harbour. Furthermore, host profile information can be obtained about the host on which a Harbour is running.

The classes starting with "Host" are responsible for storing the information about available server hosts. A HostData object stores information about a single host: a reference to an object implementing the *Har-*

```
module vagabond2 {
  module services {
    module management {
      module adaptation {
        interface HostRecommender {
          boolean hasOptimalDistribution(in string appName,
            in ClientSeq clients);
          StringSeq recommendHosts(in ApplicationInfo ai,
            in ClientSeq clients);
        };

        enum AdaptationPolicy {
          MINIMUM_STARTUP_DELAY,
          LOAD_BALANCED,
          MINIMUM_NETWORK_CONSUMPTION
        };

        interface AdaptationService {
          HostRecommender getHostRecommender();
          AdaptationPolicy getAdaptationPolicy();
          AdaptationPolicy suggestPolicy();
          StringSeq getAvailableHosts(in ApplicationInfo ai);
          void adaptApplication(in string appName)
            raises (NoSuchApplicationException);
          void releaseApplication(in string appName)
            raises (NoSuchApplicationException);
          StringSeq getAdaptedApplications();
        };
      }; // adaptation
    }; // management
  }; // services
}; // vagabond2
```

Fig. 4.3. *The Adaptation Module of Vagabond2*

bour CORBA interface, and the names of the applications currently running on that host. HostStore manages {hostname, HostData} couples. *HostService* provides a CORBA interface for updating the HostStore remotely.

The AppData objects store the {hostname, *Object*} couples for a given application, where the Object is a CORBA object that was loaded on the given host. It also stores a reference to the *AppInfo* object that describes the given application. AppStore stores {appname, AppData} couples.

The AppService_impl implements the *AppService* CORBA interface, has access to the HostStore and AppStore objects, and has a list of objects implementing the *AppLoc* CORBA interface for every loaded application. It will load and evacuate the objects incarnating an application on Harbours that are registered by *HostService*. The load() and evacuate() methods (1) manage the information about newly loaded or removed objects in HostStore and AppStore; (2) call the load() and evacuate() methods of the Harbour object concerned.

6. Proof of Concept: Measurements. We evaluated the system in a testbed shown in figure 6.1. The testbed consists of eight servers in two LANs (B-LAN, I-LAN), interconnected via the Internet. The B-LAN is located in Budapest (Hungary), and the I-LAN in Klagenfurt (Austria). The data collector runs on a client in the I-LAN. Each performance test consists of five test runs and is repeated 50 times. In each run, a sample media stream of 12,5 MB size is retrieved from the data manager instances. In test run 0 all data manager instances are running in the remote B-LAN. In test run 1 the data manager from host 8 is replicated to host 4, meaning that one data manager has been moved closer to the data collector. In particular both, the component code and the requested media stream are replicated, since hosts 1—4 are in an idle state initially. In each further test run one additional data manager is replicated from a host in the B-LAN to a host in the I-LAN.

Figure 6.2 illustrates error plots of the mean retrieval times and throughputs for the test runs in the test scenario. It clearly shows that the more data managers are replicated to the I-LAN, the shorter the retrieval times, and the higher the throughput become. Whereas the variations of retrieval times in the B-LAN are quite high, they get much less in the I-LAN.

An interesting property of data managers can be derived from figure 6.3. It tells the relative gain on

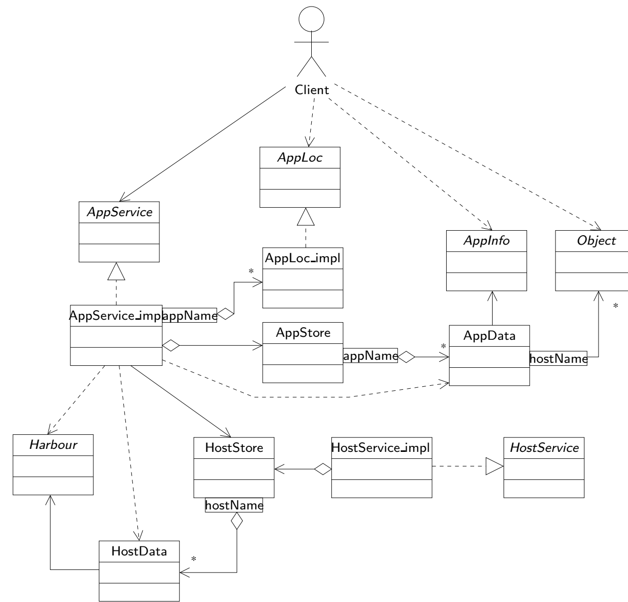
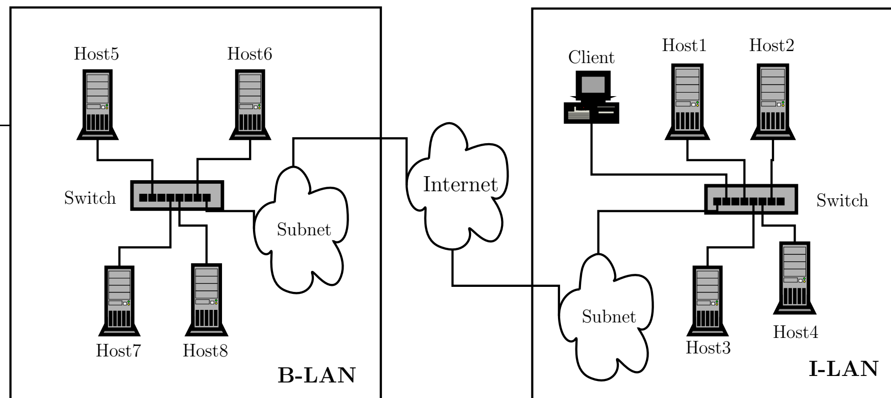


Fig. 5.1. *Implementation Class Hierarchy*

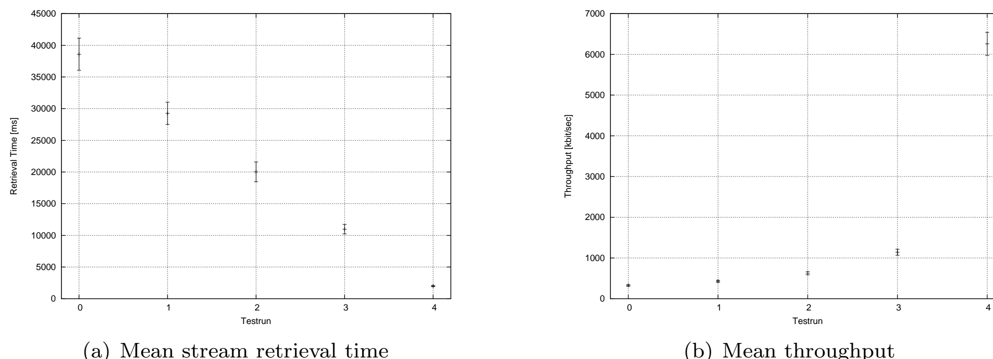


Testrun	DM Hosts
0	Host5, Host6, Host7, Host8
1	Host5, Host6, Host7, Host4
2	Host5, Host6, Host3, Host4
3	Host5, Host2, Host3, Host4
4	Host1, Host2, Host3, Host4

Fig. 6.1. *The Testbed Setup*

throughput, if a certain amount of stripe units is replicated from one LAN to another (including the time for component code replication), based on a simple model[1]. We can see that as the maximum throughput gain increases, the elbow of the curves gets closer to the bottom right corner. This means, that if we want to have a significant increase in relative gain, we have to replicate a high percentage of data managers—usually almost all of them. Thus, the best choice is to keep them together.

[1] Let t_f and t_c be throughputs when all the managers are on the farther LAN, or the closer LAN, respectively. The maximum gain is the ratio of them, $g_m = t_c/t_f$. Let t_x be the throughput, if x part of the managers are on the closer LAN. The relative gain is $g_r = t_x/t_c$. The model states that the relative gain is a function of x and g_m: $g_r(x, g_m) = \frac{1}{x+(1-x)g_m}$. In the case of the measurement, g_m was about 20.



(a) Mean stream retrieval time (b) Mean throughput

Fig. 6.2. *Performance of test runs in the test scenario*

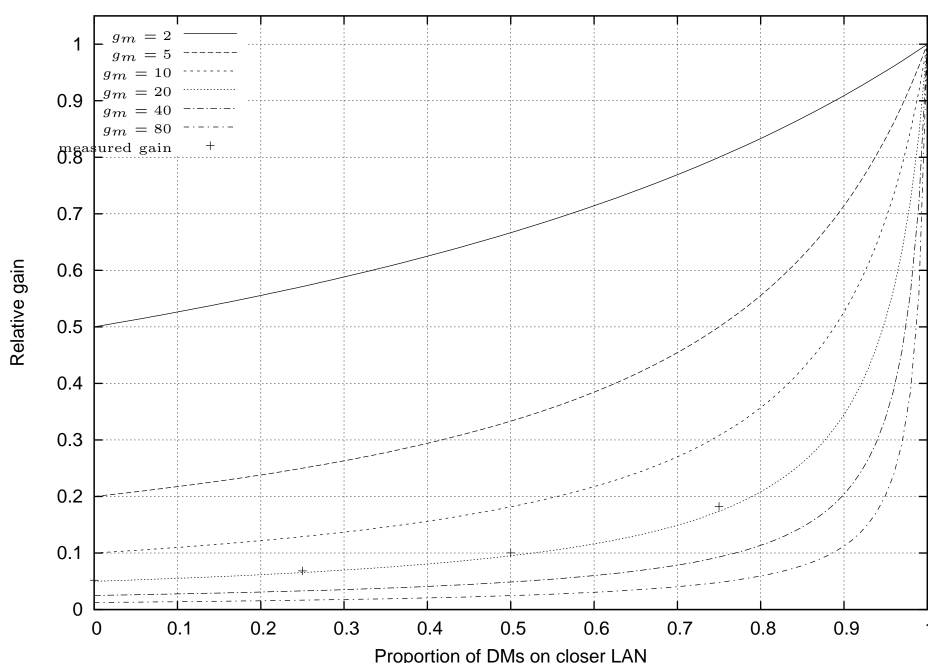


Fig. 6.3. *The results of the model and the measurements*

7. Conclusion and Future Work. We addressed the requirements of the adaptive distributed multimedia streaming server architecture *ADMS* to its underlying infrastructure Vagabond2 and vice versa, regarding the management and adaptation of multimedia components. From the point of the server the middleware has to provide two major classes of services, namely *application* and *adaptation* services. On the other hand, for Vagabond2 performing its tasks of managing applications and recommending hosts, the server has to provide detailed information on the applications to manage, as well as on the clients requesting services from those applications (i.e. information on location and request parameters). This set of bilateral requirements resulted in the specification of a set of CORBA-based interfaces, which all together define the interface between ADMS and Vagabond2. Using this interfaces Vagabond2 provides means for (1) loading and starting CORBA components written in Java on any host that runs Vagabond2, (2) evacuating a component, and (3) querying the location and distribution of components. All of these services can be accessed by CORBA method invocations.

We proved our concept by evaluating the replication of data managers between two LANs interconnected by the Internet. The results have shown that it is recommended to keep the data managers as close to each other, as possible. We are currently working on a support for adaptation using the knowledge gained from the network topology, the load of links and hosts, and the needs of the clients. This knowledge is used to implement

different adaptation strategies in the host recommender. Recently a greedy algorithm was tested with promising results. In this test we realized that a distinction has to be made when considering the TCP-based connections between data collectors and data managers, and the RTP/UDP-based connections between data collectors and clients.

A resource broker subsystem is already working and gives the adaptation service the information about the statistical properties of server nodes and networks. Furthermore, a complete implementation and detailed evaluation of a distributed adaptive multimedia streaming application based on MPEG4-encoded media streams is planned, using Vagabond2 as the infrastructure for performing adaptational issues. In particular, it should be figured out which adaptation strategy performs best under a given set of constraints (e.g. the application distribution and the request parameters). Moreover, it should be investigated whether Vagabond2 would be able to perform *on the fly* migration or replication of server components, and under which restrictions this would be the case.

REFERENCES

[1] A. Bieszczad, B. Pagurek, and T. White, *Mobile Agents for Network Management*, IEEE Communication Surveys, 1 (1998), pp. 2–9.
[2] M. Charikar and S. Guha, *Improved Combinatorial Algorithms for the Facility Location and k-median Problems*, in IEEE Symposium on Foundations of Computer Science, 1999, pp. 378–388.
[3] R. Friedman, E. Biham, A. Itzkovitz, and A. Schuster, *Symphony: An Infrastructure for Managing Virtual Servers*, Cluster Computing, 4 (2001), pp. 221–233.
[4] B. Goldschmidt and Z. László, *Vagabond: A CORBA-based Mobile Agent System*, in Object-Oriented Technology ECOOP Workshop Reader, A. Frohner, ed., Springer Verlag, 2001.
[5] B. Goldschmidt, Z. László, M. Döller, and H. Kosch, *Mobile Agents in a Distributed Heterogeneous Database System*, in Euromicro Workshop on Parallel, Distributed and Network-based Processing, 2002.
[6] O. Kariv and S. L. Hakimi, *An Algorithmic Approach to Network Location Problems. II: The p-medians*, SIAM Journal of Applied Mathematics, 37 (1979), pp. 539–560.
[7] J. Y. Lee, *Parallel Video Servers: A Tutorial*, IEEE Multimedia, 5 (1998), pp. 20–28.
[8] B. Li and K. Nahrstedt, *A Control-based Middleware Framework for Quality of Service Adaptations*, IEEE Journal of Selected Areas in Communications, (1999), pp. 17(9):1632–1650.
[9] Moving Pictures Experts Group, *ISO/IEC JTC1/SC29/WG11 N4668: Overview of the MPEG-4 Standard*, 2002. `http://mpeg.telecomitalialab.com/standards/mpeg-4/mpeg-4.htm`
[10] ———, *ISO/IEC JTC1/SC29/WG11 N4980: MPEG-7 Overview*, 2002. `http://mpeg.telecomitalialab.com/standards/mpeg-7/mpeg-7.htm`
[11] ———, *ISO/IEC JTC1/SC29/WG11 N5231: MPEG-21 Overview*, 2002. `http://mpeg.telecomitalialab.com/standards/mpeg-21/mpeg-21.htm`
[12] ———, *ISO/IEC JTC 1/SC 29 N 5204: Text of ISO/IEC 21000-7 CD - Part 7: Digital Item Adaptation*, 2003. `http://www.itscj.ipsj.or.jp/sc29/open/29view/29n5204c.htm`
[13] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6 ed., December 2001.
[14] J. Rolia, S. Singhal, and R. Friedrich, *Adaptive Internet Data Centers*, SSGRR'00, (2000).
[15] R. Tusch, *Towards an Adaptive Distributed Multimedia Streaming Server Architecture Based on Service-oriented Components*, Tech. Report TR/ITEC/03/2.02, Institute of Information Technology, Klagenfurt University, Klagenfurt, Austria, February 2003. Paper submitted to the Joint Modular Languages Conference (JMLC) 2003.
[16] R. Tusch, L. Böszörményi, B. Goldschmidt, H. Hellwagner, and P. Schojer, *Offensive and Defensive Adaptation in Distributed Multimedia Systems*, Tech. Report TR/ITEC/03/2.03, Institute of Information Technology, Klagenfurt University, Klagenfurt, Austria, February 2003. Paper submitted to the Journal of Systems and Software, Special Issue on Multimedia Adaptation.
[17] J. Waldo, *The Jini Architecture for Network-centric Computing*, Communications of the ACM, 42 (1999), pp. 76–82.

Edited by: Zsolt Nemeth, Dieter Kranzlmuller, Peter Kacsuk, Jens Volkert
Received: March 24, 2003
Accepted: June 5, 2003