AMS: An Adaptive Multimedia Server Architecture

Roland Tusch

Institute of Information Technology
University Klagenfurt
Technical Report No TR/ITEC/02/2.06
April 2002

Abstract

In most existing distributed multimedia architectures the term *adaptivity* covers techniques for increasing or (more often) reducing the quality of transmitted or presented media streams, in order to cope with different network and client capabilities. This kind of *defensive* adaptivity, also called *stream-level adaptivity*, is typically achieved by varying the streams' spatial and/or temporal resolution. However, for multimedia servers stream level adaptivity might not be sufficient in cases where computational resources become bottlenecks or latency times to the requesting clients become critical. In order to avoid increasing numbers of admission rejections or client reneging, the server should apply an *offensive* adaptation strategy.

This paper presents a distributed multimedia streaming server architecture that builds upon the idea of *server-level adaptivity*, where new computational resources are acquired on demand and certain server applications are migrated or replicated to recommended locations. The goal is to maximize the number of admittable client requests by reducing network traffic, startup latencies and/or keeping subsets of server nodes load-balanced. The architecture consists of three distinguished interacting applications that can be migrated or replicated on demand, resulting in a virtual server spawning a dynamic number of server nodes over time. As an infrastructure for application migration and replication, a CORBA-based mobile agent system (*Vagabond2*) is used. This infrastructure also manages the set of nodes belonging to the media server cluster and provides recommendations for a proper placement of multimedia server applications. For adaptational issues the server architecture provides information on currently admitted client requests, and future (not yet admitted) requests.

Keywords: distributed multimedia server, server-level adaptivity, CORBA, mobile agents

1 Introduction

Adaptivity has turned out to be one of the key characteristics of future distributed multimedia systems, enabling them to provide distinguished levels of quality of service for different computational environments. Especially the heterogenity of today's consumer end devices concerning different qualities of their network links (wired or wire-less), as well as their varying display and computational capabilities, requires the underlying system to perform proper adaptations to the transmitted media streams.

Recently, there have been a number of research efforts in internet streaming architectures concerning especially two areas of adaptation, namely the *object-level adaptivity* and the *location-level adaptivity* of media streams. These types of adaptivity are also referred to as *stream-level adaptivity*. Object-level adaptivity of a media stream is achieved by varying the temporal or spatial resolution of it, either on the server-side or somewhere in the network along the delivery path to the client. This is typically achieved by either encoding a media stream in different qualities for different client and network capabilities, or by applying scalable coding techniques as e.g. multilayered coding techniques [10, 9, 11] or the *fine granular scalability* coding technique standardized in MPEG-4[18, 17, 26]. On the other hand, location-level adaptivity bases on migrating and/or replicating multimedia content from one server to another, depending on varying demands on different locations. This approach is commonly followed by content distribution networks and distributed caching architectures[4, 12, 14].

However, stream-level adaption strategies might not be sufficient for an adaptive multimedia streaming architecture for the following reasons: (1) The object-level adaptivity does not take into account the locations of the client requests. Thus, it might e.g. reduce the quality of the transmitted media stream due to bandwidth and client constraints, but it is not able to reduce the transmission distance to the client. (2) The location-level adaptivity allows for a dynamic distribution of media objects over a static number of server nodes. It is able to reduce the transmission distance to the client by migrating or replicating media objects to server nodes in the vicinity of the client, but its scalability is usually limited by the static number of server nodes. Thus, location-level adaptivity of media streams ends where computational and storage resources of server nodes become bottlenecks. This is also valid for a combination of object-level and location-level adaptivity, although it results in a possibly better resource utilization and therefore in an increased number of admitable client requests.

This paper proposes an adaptive distributed multimedia streaming server architecture which adapts itself on the level of server applications, resulting in a virtual multimedia server spawning a dynamic number of server nodes over time[25]. This kind of adaptation, called *server-level adaptation*, tries to acquire new computational resources by allocating new server nodes in cases where server resources or transmission distances to clients become bottlenecks. This is achieved by defining a multimedia streaming server cluster as a set of closely and loosely coupled server nodes, whose number of nodes may grow and shrink over time, depending on the service demand. The goal of server-level adaptation is to maximize the number of admittable client requests by applying different adaptation strategies like e.g. to minimize network consumption, to minimize startup delays to the clients or to keep the server nodes load-balanced. To achieve this goal, three distinguished but interacting server applications are defined, which can be loaded, evacuated, migrated or replicated on demand. These operations are executed using the CORBA-based mobile agent system Vagabond2[8] as an underlying infrastructure. Vagabond2 also manages the set of current server nodes available to the streaming server cluster, as well as provides host recommendations especially in cases were a proactive server-level adaptation needs to take place. The proposed architecture is conceptually similar to the demand adaptive and locality aware clustered streaming server architecture proposed in [6], with the difference that the set of possible server nodes is also kept dynamically. Moreover, the middleware Vagabond2 uses a number of parameters regarding client request, server load and network load during its host recommendation process.

The remainder of this paper is organized as follows. Section 2 presents a static distributed multimedia server architecture in internet settings and discusses its shortcomings. In section 3, an adaptive multimedia streaming server architecture (AMS) is presented, which builds upon the concepts of the static architecture, but with the possibility to dynamically adapt AMS server applications. It also presents the

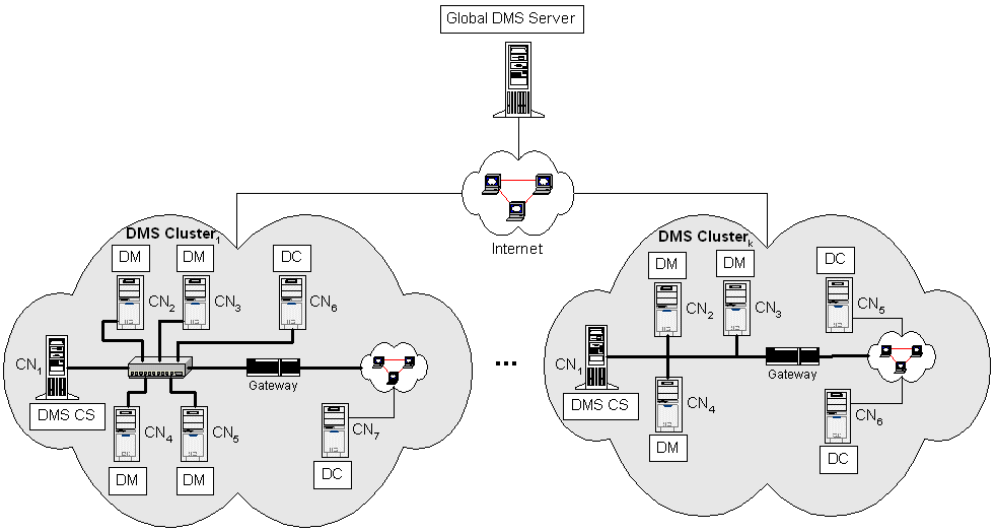

Figure 1: A static distributed multimedia server architecture

AMS runtime environment, the requirements to application management and adaptation, as well as the adaptation policies which might be followed during adaptational processes. The public interfaces between adaptive applications like the AMS server applications and its underlying infrastructure Vagabond2 are topic of section 4. Section 5 discusses the major interaction scenarios between the controlling instances of the AMS server architecture and Vagabond2, and section 6 concludes this paper with a brief summary and perspective on future work.

2 Static Distributed Multimedia Server Architectures

2.1 A Static Design Approach

The major benefits of a distributed multimedia server architecture (also referred to as a parallel multimedia server) in comparison to a high-speed single-server approach lie in an improved scalability and server-level fault tolerance. Thus, following Lee's[15] proposed framework for the design of a parallel video server, a static design approach for a server architecture in global dimensions could look like the one illustrated in figure 1.

The overall controlling instance of the globally distributed multimedia server (*DMS*) is represented by the *Global DMS Server*. The global server's task are (i) to manage a set of pre-installed *DMS Clusters*, (ii) to keep track of which multimedia object is located on which clusters and (iii) to redirect an incoming client request to an appropriate server cluster.

Each DMS cluster involves a certain set of server nodes, which might be interconnected in different topologies and with different "qualities" concerning the network bandwidth available to them. Nodes running highly interacting applications should be coupled more closely to each other (in the sense of network topology). Thus, the underlying network topology represents an important decision factor for placing a certain server application on a certain node. A server cluster can be seen as an autonomous unit running a dedicated application type - in this case the DMS application type. Each DMS application type runs a multiset of pre-installed multimedia server applications. For the reason of simplifying resource management, the server architecture follows the *full server utility* model[22]. Thus, it is assumed that each server node runs exactly one server application at a time, whereas one DMS cluster might run multiple instances of one server application on different server nodes. The three distinguished multimedia server

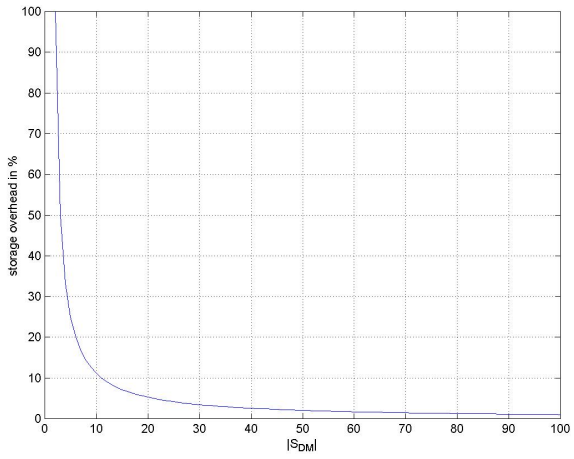

Figure 2: Storage overhead caused by striping techniques using simple parity units

applications of a multimedia server cluster are the following: the *cluster server* (DMS CS), the *data manager* (DM), and the *data collector* (DC) application.

The cluster server application (i) has to maintain knowledge on which server node runs which of the other two server applications, (ii) has a cluster-based view on the distribution of multimedia objects (in particular audio and video elementary streams) among the cluster nodes, and (iii) performs a server-level narrow striping strategy for distributing uploaded multimedia objects over a subset of cluster nodes S_{DM}, which runs the data manager application (also referred to as *data nodes*). Thus, server-level striping techniques (in particular by using stripe units of constant data length) allow data nodes to scale with reduced storage overhead and without any major load-imbalances due to varying popularities of its stored media objects [15]. However, the storage overhead ($O_{Striping}$) caused by parity stripe units depends on the number of existing stripes ($N_{Stripes}$), which themselves depend on the number of data manager nodes ($|S_{DM}|$) a media object is striped across. Equation 1 illustrates these relationships using S_{Stream} as the media object's size, and S_{SU} as the size of a fixed-size stripe unit. It has to be noted that a *xor*-based parity unit calculation can only be applied, if and only if the stripe units are of constant size.

$$\begin{aligned} O_{Striping} &= N_{Stripes} \times S_{SU} \\ N_{Stripes} &= \lceil \frac{N_{SU}}{|S_{DM}| - 1} \rceil \\ N_{SU} &= \lceil \frac{S_{Stream}}{S_{SU}} \rceil \end{aligned} \tag{1}$$

As an implication of the equation, $|S_{DM}| > 1$ must also hold. Figure 2 illustrates the characteristic decrease of storage overhead for parity units with an increasing number of data manager nodes. Assuming that e.g. a storage overhead of at most 20% is acceptable, at least six server nodes have to be pre-configured to run the data manager application. In figure 1 it is illustrated that the vicinity of striping media objects may also vary from cluster to cluster.

Summarizing the tasks of the data manager application it is clear that a data manager has to provide means for (i) storing and (ii) retrieving stripe units for a certain media object.

Finally, there is a set of nodes S_{DC} running the data collector application (also referred to as data collectors), where $|S_{DC}| \geq 1$ must hold. A data collector is responsible for (i) requesting stripe units of a certain media object from the cluster's data nodes, (ii) resequencing and merging them into a coherent stream, and (iii) streaming the buffered media stream out to the requesting client. Since a data collector

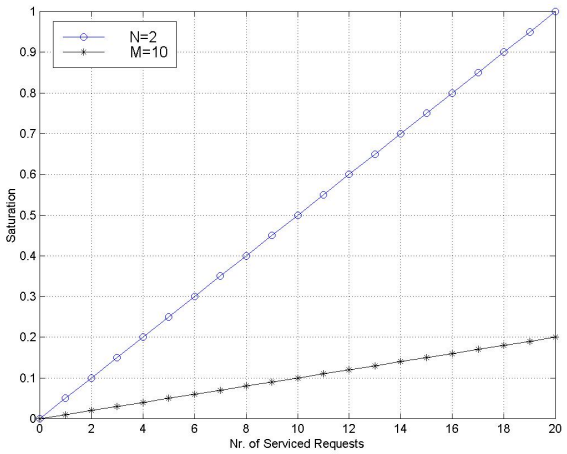

Figure 3: Saturation progression for demand ratio $\delta = 0.1$ per request

in this architecture is an integrated part of a DMS cluster, it can be considered as a proxy module located at the server side. Following Lee's[15] framework it would implement the *independent proxy* architecture.

2.2 Discussion of the Static Design Approach

Although it is claimed that a statical distributed multimedia server architecture is load balanced regardless of the skew in video popularities[15], this is only valid for an architecture where the data collectors are integrated into the client application (i.e., each client has its own data collector). This *proxy-at-client* architecture has the advantage that a failure in running the data collector application would only disrupt the service for the client itself, and not for a set of clients, as it is the case in the *independent proxy* model. However, integrating a data collector module into a client application has the two main disadvantages that (i) the client has to know the internal distribution of the server, and (ii) one media stream delivery could not be shared by a set of clients.

Thus, running the data collector applications on separate nodes as illustrated in figure 1 still remains the most wide-spread technique for resequencing and streaming multimedia data to the clients. However, under this static configuration the distributed media server is not load-balanced anymore. The reason lies in the load-imbalance between the data nodes and the collector nodes.

Consider the case where there are M data nodes and N collector nodes pre-installed. Assuming that each node has the same capacity C and that all clients request media objects with the same bit rate, each request demands c resource units on one collector node for serving it. It is also assumed that the data collectors do not perform any buffering regarding distinct popularities of the media objects, and each request is serviced independently. Thus, in an optimistic resource reservation schema and with a uniform distribution of requests to the data collectors, N data collectors can admit at most $\lfloor \frac{N \cdot C}{c} \rfloor$ simultaneous requests for being serviced. On the other hand a requested media object is striped among the M data nodes, resulting in a capacity demand of $\frac{c}{M}$ on each data node on average. Thus, the total number of admitable client requests on one data node is bound to $\lfloor \frac{C \cdot M}{c} \rfloor$. Since M is a multiple of N in a typical static server configuration, the data nodes and the collector nodes are not load-balanced anymore. Figure 3 illustrates this imbalance in terms of comparing the resource saturations of two data collectors and ten data nodes, with a resource demand ratio $\delta = 0.1$ per request on a data collector. Thus, in this scenario a DMS cluster's load balance is influenced by the number data nodes and collector nodes involved.

As a result, one solution within the static architecture would be to pre-install at least as many data collector nodes as there are data nodes. However, this approach does not take into account regional

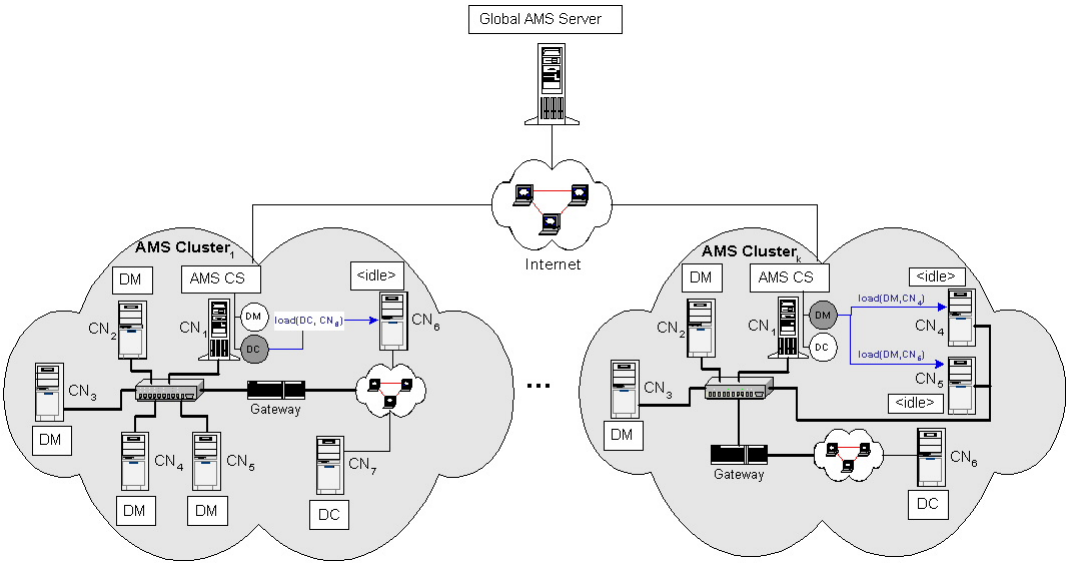

Figure 4: An adaptive multimedia server architecture

variations in media object popularity, as well as the locality and QoS parameters of client requests. If e.g. a set of topological near clients requests the same media objects, it would we better to allocate a new data collector node in the vicinity of those clients, in order to increase the number of possible admissions for a given set of constraints and thereby reducing network load. Since a data collector application can also provide means for caching media objects, QoS parameters like e.g. a given maximum startup delay of D_{max} are also decision factors for placing a data collector application. The major difference of dynamically distributing the data collector application in comparison to dynamically placing web servers[21] or web proxies[16] is that the former is completely under control of the cluster manager application.

3 AMS: An Adaptive Multimedia Server Architecture

3.1 An Adaptive Design Approach

The adaptive multimedia streaming server (AMS) architecture builds upon the concept of defining an AMS cluster as a set of interconnected nodes, which are able to run any of the distinguished applications described in section 2.1. The interconnection can reach from a local area network (LAN) to a set of autonomous networks connected via a high speed backbone. Since one of the major application areas of multimedia streaming applications is the internet, the adaptive multimedia server architecture needs to be designed for running in internet settings.

In the AMS architecture each node is equipped with a special runtime environment which allows for loading or evacuating one of the described multimedia server applications to or from it. Thus, there is a controlling instance containing the binaries of all the applications, which controls the current application distribution within its cluster. Figure 4 shows the components of a globally distributed adaptive multimedia server architecture. Basically, the components of the AMS architecture are quite similar to the DMS architecture, with the difference that the cluster server application also has to keep track of the current distribution of server applications within the cluster. Thus, the cluster server application has access to the binaries of the other two types of server applications (data manager (DM) and data collector (DC) application) and controls their distribution by using Vagabond2[8], a middleware especially designed for distributed application adaptations. The interface specification to Vagabond2 is described in section 4.

Figure 4 also illustrates two of the three major adaptation scenarios within an AMS cluster. In *AMS*

Cluster 1 the data collector application is replicated to the idle cluster node CN_6. In *AMS Cluster k* two new data managers are acquired by loading the data manager application on these nodes. These two nodes could belong to the same striping group as the data nodes CN_2 and CN_3, or define a new striping group for specially handled media objects. The third scenario would be the evacuation of a server application from one cluster node, meaning that this node returns to the set of idle nodes for future use. These scenarios show that on the one hand it must be somehow feasible to move a server application from one server node to another, and on the other hand some heuristics are required for selecting the most appropriate idle server node in the case of a required server adaptation.

An important aspect of the replication/migration process is that even if all the discussed AMS server applications need to deal with real-time constraints concerning the submission of media streams to the clients (at least belonging to the server-side), the migration/replication process itself need not happen in real-time. This relaxation results from the constraint that *on-the-fly* application adaptation is currently not supported. Thus, an ongoing adaptation of a multimedia streaming server application needs to be well planned, using statistical information from previously served clients, as well as estimated future client requests.

3.2 The AMS Runtime Environment

As already mentioned, the major requirement of a distributed adaptive multimedia server architecture is to migrate or replicate one of the distinguished server applications *cluster server*, *data manager* and *data collector*, including their data depencencies, from one cluster node to another. None of these AMS server applications requires a location adaptation in real-time. Although - when running and processing - client request parameters as e.g. maximum startup latency or maximum packet loss, as well as stream-level parameters like average and peak bitrate have to be guaranteed within the scope of the distributed server. In internet settings this means that the guaranteed service ends at the data collector, since the clients themselves are not integral part of the server architecture.

Following these application characteristics a runtime environment is required, which on the one hand provides non-real-time management facilities concerning application adaptation, and on the other hand offers means for executing and controlling real-time server applications - especially the data manager and data collector applications. This results in an runtime environment architecture as presented in figure 5. Using a real-time operating system as the core runtime component, the environment is further divided into a *management plane* and a *service plane*. The management plane includes the non-real-time management infrastructure *Vagabond2*[8] for application adaptation issues. Since Vagabond2 is a CORBA-based mobile agent system completely written in Java, it depends on the Java2 Runtime Environment as well as on a Java-based CORBA-ORB. The advantage of implementing and using a middleware for server-level adaptations lies in the reuseablity of code-migration and code-replication algorithms for other types of applications, like e.g. an adaptive web-server. However, in the case of an *on-the-fly* adaptation an adaptive multimedia streaming server has different requirements concerning the adaptation process, as it is the case with an adaptive web-server.

On the other hand, the service plane provides means for a pattern-oriented cross-platform server application development using the *Adaptive Communication Environment* (ACE) [23] and the real-time ACE-CORBA-ORB *TAO*[24]. Both, ACE and TAO, enable for a distributed object-oriented multimedia server implementation providing guaranteed services, if provided by the underlying infrastructure.

The AMS server applications communicate with both planes through two dedicated interfaces. The interface specification for communicating with the management plane is described in section 4.

The following section focuses on the design considerations and requirements concerning the management plane. Thus, issues regarding application management and application adaptation are discussed.

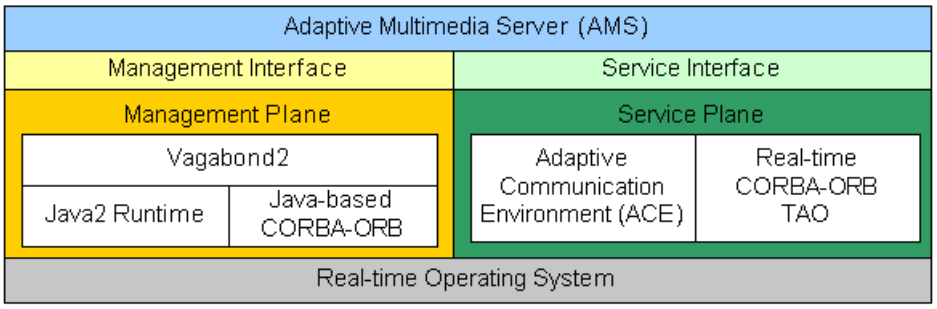

Figure 5: The AMS runtime environment

3.3 Application Management and Adaptation

There are two major categories of functionalities for dealing with adaptive server applications, namely *application management* and *application adaptation*. The application management module has to provide means for loading, evacuating and locating each of the distinguished server applications discussed in section 2.1. In particular, the controlling instance of the AMS architecture, namely the cluster server application, must have the possibility to load and evacuate a data manager or a data collector application to or from an AMS cluster node. Since server applications also may have data dependencies (e.g. all stream data files in the case of a data manager application), the application management module has also to provide means for carrying these data dependencies during a server application migration or replication. The composition of this set of dependent files also has to happen dynamically, since the number of uploaded partial media streams on a data manager node usually varies over time. However, due to the restriction that *on-the-fly* server migrations are not supported, an application evacuation (which denotes the first operation during a migration process) only can take place, if the server application itself is in idle state, meaning that it is not processing client requests. Furthermore, it must be feasible to locate an instance (or incarnation) of any of the AMS cluster applications running on a dedicated cluster node. Since all the distinguished applications will be implemented as TAO-CORBA-objects using the service plane, the locator is required to return a reference to such an object for being able to directly communicate with it.

Migrating or replicating a server application from one cluster node to another one is only the executive part of a server-level adaptation process. The much more interesting question is when and under which circumstances such an adaptation process should take place. These issues have to be dealt within the application adaptation module.

Consider one of the AMS clusters C_j in figure 4, which interconnects a set of $N = |S_{C_j}|$ nodes either being idle or running a certain AMS server application. Furthermore each cluster node is equipped with the AMS runtime environment described in section 3.2. Each AMS cluster requires one node for being reserved for the cluster manager application. Assuming that there are m nodes running the data manager application, and n nodes executing the data collector application (where $m+n+1 \leq N$ must hold), there are $I = N - m - n - 1$ remaining cluster nodes, which are in an idle state. Assuming that there is a set of near-neighbored clients S_Cl requesting a set of multimedia objects S_O at some (partially overlapping) future time. Here the question arises, if this set of client requests can be served by the existing application distribution, regarding their desired quality of service. Otherwise it has to be checked, if S_{Cl} or at least a subset of it can be served, if a server-level adaptation takes place before (like e.g. in the first scenario of figure 4, where the data collector application is loaded to an idle cluster node). Thus, given the set of idle nodes I, the set of client requests S_{Cl} including the set of request parameters R_i for each client request i, the capacity Cap_k on each idle cluster node k, the mean capacity requirement of each AMS server application, and the distance $d(i, j)$ between any two cluster nodes, the problem of finding the optimum set of idle server nodes for running data collectors can be modelled as a dynamic capacitated facility location problem using these parameters as input. For general graphs this problem is known to

be NP-hard[13, 20]. However, there are a number of recently studied approximation algorithms for the uncapacitated facility location problem and the best published approximation factor to date is 1.728 (i.e. the cost is at most 1.728 times the minimum)[1]. However, the worst-case performance bounds for the capacitated version is considerably worse than for the non-capacitated version[3].

In the second scenario of figure 4, the cluster server chooses to load the data manager application to two idle cluster nodes. Assuming the maximum number of data manager nodes is bound by $N_{DM_{max}}$, the problem of finding the optimum K cluster nodes for running the data manager application can be modelled as a dynamic capacitated k-median problem. The major input parameters for this problem are: (i) the set of possible data collector nodes S_{DC}, the capacity Cap_k of each candidate node k, the distance $d(i, j)$ between a demanding node (data collector node) i and a facility node (data manager node) j in terms of network latency or throughput. This kind of location problem is also known to be NP-hard for general graphs[13]. The best approximation algorithm for the uncapacitated k-median problem in the metric space results in an approximation factor of 4, with a running time of $O(N^3)$[1]. Again, the worst-case performance bounds for its capacitated version is considerably worse[2].

In both location problems, (i) finding the optimum locations for running at most $N_{DM_{max}}$ data manager applications, and (ii) finding the optimum number and optimum locations of cluster nodes running the data collector application, the underlying infrastructure Vagabond2 is required to recommend the AMS cluster manager the optimum set of server nodes for a particular server application at a specific time instance (this is why it is called a *dynamic* location problem). In order to perform such recommendations, the responsible component within Vagabond2 requires both, local and global information on client requests, server capacities and network topology. In particular, the following input parameters can be taken into account during the recommendations:

- **client request parameters:**
  - *bit-rate*: the average and/or peak bit-rate required to serve the stream in kbit/sec, depending on whether it is a CBR or VBR stream. The QoS parameter bit-rate may be explicitly specified by the client, or read from the header of the requested bitstream otherwise.
  - *loss-rate*: the maximum accepted percentage on packet loss from the data collector to the client. The effects of packet loss are very coding dependent. In MPEG-4 e.g. there are a number of error correction codes allowing for packet loss up to 2 percent and even higher, without losing any information at decoding time.
  - *delay*: the maximum delay between two consecutive packets in msecs. This parameter is usually not that relevant in internet streaming applications, since the client always has to keep a buffer of packetized audio or video access units, allowing for smoothing out packet delays and network jitter.
  - *jitter*: the maximum allowed standard deviation in delay in msecs. As valid for the delay parameter, the jitter parameter is not that relevant in internet streaming applications, since the client's buffering techniques usually have to compensate the network jitter.
  - *latency*: the maximum startup latency the client is willing to wait (in seconds), before he may initiate a reneging process. This parameter is very crucial for placing an AMS server application (in particular a data collector) near to the client, in the case, if no such data collector can currently fulfill this value.
  - *batch date*: the time instance when the request has to be served. This parameter is provided when the request has to be served in future. This value can be used for planning a server-level adaptation in the meantime, if the current application distribution would cause a rejection of the request at the specified time. Furthermore, the data collector application can use this value for batching client requests which target to the same media stream objects.

- **network parameters:**
  - *latency*: the current, average and maximum round-trip time (in msecs) of packets between any two nodes within the application cluster, or between a server node and a client node - e.g. between a candidate node for running a data collector application and a client. The average and maximum values are calculated over a given sequence of time-intervals T_{seq}.
  - *bandwidth*: the actual, average and maximum available bandwidth in kbit/sec between any two nodes within the application cluster, or between a server node and a client node. Again, the average and maximum values are calculated over a given time-interval sequence T_{seq}.
  - *loss-rate*: the average percentage of packet loss between any two server nodes of the AMS cluster, or between a cluster node and a client node. The values are calculated over a given sequence of time-intervals T_{seq}.
  - *hops*: the number of gateways a packet has to pass between any two given nodes within the AMS cluster, or between an AMS server node and a client.
- **server parameters:**
  - *CPU load*: the current CPU load in percent.
  - *free memory*: the available physical memory on the server node.
  - *free disk space*: the available amount of secondary storage on the host.
  - *disk throughput*: the current, average and maximum measured throughput to the the server's disk subsystem. Average and maximum values are calculated considering all performed measurements so far.
  - *network throughput*: the current, average and maximum measured throughput to the the server's network subsystem. Again, average and maximum calculations include all performed network measurements so far.

The first set of input parameters - the *client request parameters* - is provided by the AMS cluster server application, which handles client requests and delegates these requests to an appropriate data collector node, if such one is available. The second set of input parameters - the *network parameters* - needs to be provided by a network monitoring layer below Vagabond2. And finally, the third parameter set - the server parameters - has to be provided by each AMS server node, allowing for a capacity estimation of each cluster node, which further can be used in solving the capacitated facility location and k-median problems.

Similar to the adaptational concepts implemented in Symphony[5], the underlying infrastructure Vagabond2 should even be able to automatically migrate or replicate an AMS server application from one cluster node to another idle cluster node, without requiring the AMS cluster manager application to control this process. This means that both, the AMS cluster manager and Vagabond2 itself, should be able to take decisions on when and where to perform a server-level adaptation. This capability requires the definition of adaptation policies, which themselves can be mapped to differently optimized cost functions within the recommending component of Vagabond2.

3.4 Application Adaptation Policies

Having defined the set of possible input parameters for the process of recommending a set of idle cluster nodes for a certain AMS server application, there is still the question remaining on how to retrieve this set of nodes, i.e. regarding to which policy the applied cost function should be optimized. Basically, the following adaptation policies (also referred to as *objective functions*) are of relevance for an adaptive multimedia streaming server architecture:

- *minimum startup delay*: under this adaptation policy the number and locations of cluster nodes running the data collector application are selected with regard to minimizing the overall startup latencies for the whole current or future set of clients served by the AMS server. However, under this policy the probability for a load-imbalance among server nodes (especially those nodes which are running the data collector applications) might be high.

- *load balanced*: this policy optimizes the cost function regarding an equal utilization of all server nodes running the data collector application. The utilization function v should reflect a cumulated normalized value of the server parameters described in section 3.3. This adaptation policy might increase the number of acceptable client requests by the AMS server, if startup latencies are not crucial. However, in cases where there are many client requests defining short-termed startup latencies, the number of admittable client requests may decrease due to missing alternatives in selecting further candidate nodes for running data collector applications.

- *minimum network consumption*: under this policy the optimum location of data manager nodes and the optimum number and location of data collector nodes is determined by (i) minimizing the network distances between data manager nodes and the cluster server node, and (ii) minimizing the overlappings of used network paths between data collector nodes and data manager nodes. This policy improves the utilization of the resource *network* and thereby can strongly influence the number of acceptable client requests on the network layer. However, increased numbers of server overloads and startup delay misses may occur, when there is an unequal distribution of client requests to the data collector nodes.

Choosing the right objective function for certain network topologies within an AMS cluster, as well as certain client request patterns concerning media objects and request locations is one of the key issues for the future work.

4 Interfacing AMS and Vagabond2

After having discussed the services required for application management issues from both sides - the AMS and Vagabond2 side - an interface specification was developed, which allows for the management and adaptation of any AMS server application. Moreover, the design of the interface to the middleware is kept generic, which allows for the proactive management and adaptation of any type of adaptive application. Thus, Vagabond2 will not only be able to manage and adapt AMS server applications, but also e.g. an adaptive web-server application. Figure 6 illustrates the public interfaces provided by Vagabond2 in UML[19] notation, denoting an API between Vagabond2 and adaptive server applications. This is a slightly modified version to the model presented in [8].

In general there are three major modules or layers constituting the API between Vagabond2 and an adaptive server application like AMS. The bottom layer (also referred to as *application layer*) represents the set of interfaces which need to be provided by an adaptive application. The intermediate layer (also referred to as application service layer) provides interfaces for loading, evacuating and locating adaptive server applications. And finally, the top-most layer (also referred to as *adaptation service layer*) enables for automatically distributing instances of a certain adaptive application over a set of server nodes, or provides recommendations for allocating a certain set of server nodes for a certain type of application. Each of these layers is briefly described in the following subsections.

4.1 The Application Layer

The interfaces of the application layer represent the basic components of the Vagabond2 architecture, upon which the other two layers are building on. The central interface of the application layer is the *AdaptiveApplication* interface. Each server application which wants itself to become an adaptive server

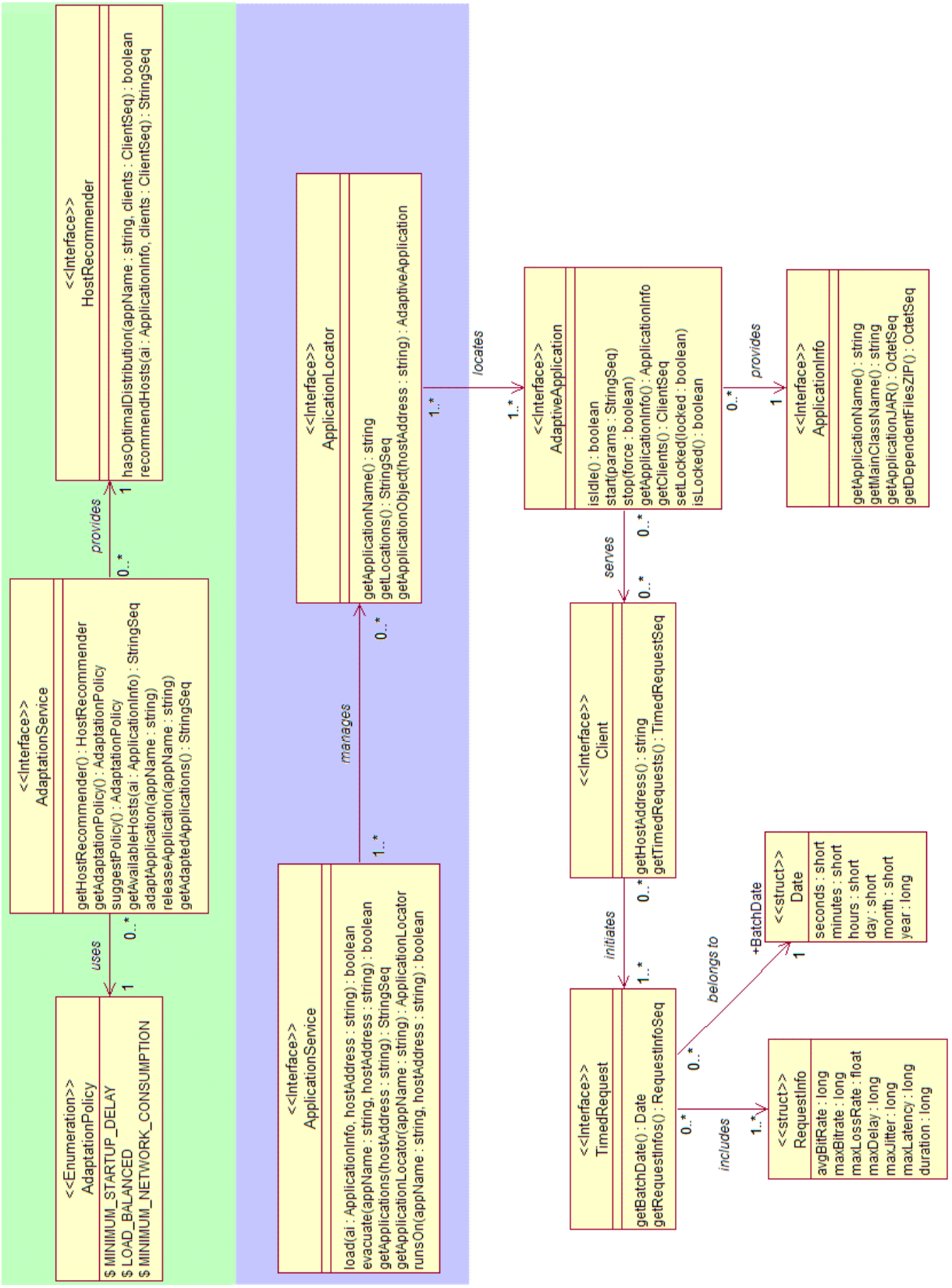

Figure 6: The specification of public interfaces between Vagabond2 and adaptive server applications

application has to implement this interface. In the case of the AMS architecture all three AMS server applications as described in section 3.1, namely the *cluster manager*, the *data manager* and the *data collector* application, will implement this interface.

Basically, the service of each adaptive application can be started and stopped remotely, and it can be checked, if the server application is in an idle state, or not. A server application is called to be idle, if it is currently not serving any requests and it also has no pending client requests, which already have been admitted. This information is especially necessary in the case of a required application migration, where an adaptive server application can only be evacuated from a server node, if and only if it is in an idle state. For exceptional cases it is also possible to force the stopping of the service of an adaptive server application, regardless whether there are pending client requests, or not. However, in the usual case of only being able to evacuate an idle application, it must somehow be possible to guarantee that an adaptive application will become idle in some calculated or predefined time. In other words, the server application needs to be told that it must not accept any future client requests, regardless whether it is able to, or not. This is achieved by putting an adaptive application into the locked state. Once an adaptive application is locked, it has to guarantee that it only finishes serving its pending client requests and does not accept any new incoming requests.

For adaptational purposes each adaptive application also has to provide an *ApplicationInfo* object, which contains all the required information to load and start the application on another server node. An important aspect herein is that although it is not possible to migrate or replicate the transient state of an adaptive server application, it is possible to do it with its persistent state. Thus, in the case of an AMS data manager application e.g. it would be possible to fetch all its partial media stream objects into the dependent files archive and migrate or replicate this archive along with the binaries of the application. Finally, an adaptive application also provides read access to all its currently accepted client requests. Client requests which are served at same time instances are batched together to so-called timed requests (interface *TimedRequest*). This usually makes sense for a multimedia streaming service where e.g. both, a video and an audio stream are supplied at the same time. In a streaming application environment as it is the case in the AMS architecture, each media stream is described by an own object specifying the required QoS parameters (structure *RequestInfo*) and its duration.

Since Vagabond2 is a successor of the CORBA-based mobile agent system Vagabond[7], it is also implemented using CORBA technology. The following code listing shows the public interfaces of the application layer in CORBA-IDL syntax, including all possible exceptions which might occur during Vagabond2 operations.

```
1  module vagabond2 {

2     // specifications of exceptions
3     /** Indicates that the service of an adaptive application has already been started. **/
4     exception ServiceAlreadyStartedException {};
5     /** Indicates that the service of an adaptive application has not yet been started. **/
6     exception ServiceNotStartedException {};
7     /** Thrown, when a given host address does not exist.  **/
8     exception NoSuchHostException {};
9     /** Thrown, if an adaptive application has already been loaded on a host. **/
10    exception ApplicationAlreadyLoadedException {};
11    /** Thrown, if an adaptive application has not yet been loaded on a host. **/
12    exception ApplicationNotLoadedException {};
13    /** Indicates that an adaptive application name is not known. **/
14    exception NoSuchApplicationException {};

15   typedef sequence<octet> OctetSeq;
16   typedef sequence<string> StringSeq;
```

```
17  /** Defines the information to be provided by a date object.  */
18  struct Date {
19    short seconds;
20    short minutes;
21    short hours;
22    short day;
23    short month;
24    long year;
25  }; // Date

26  /** The value of unspecified positive amounts */
27  const short UNSPECIFIED = -1;

28  /** A request info contains detailed information about a client request. **/
29  struct RequestInfo {
30    /** The requested average bitrate (in kbit/sec). **/
31    long avgBitRate;
32    /** The peak bitrate to expect (in kbit/sec). **/
33    long maxBitRate;
34    /** The maximum allowed packet loss rate (in 1/100). **/
35    float maxLossRate;
36    /** The maximum allowed delay between two consecutive packets (in msec). **/
37    long maxDelay;
38    /** The maximum allowed std. deviation in delay (in msec). **/
39    long maxJitter;
40    /** The maximum allowed startup latency the client is willing to wait (in secs). **/
41    long maxLatency;
42    /** The duration of serving the request in seconds. **/
43    long duration;
44  }; // RequestInfo

45  typedef sequence<RequestInfo> RequestInfoSeq;

46  /** An interface to hold a timed client request. Time information is provided in CET. **/
47  interface TimedRequest {
48    /**
49     * Delivers the time instance when the batched requests have to be served.
50     * If not specifed, the corresponding requests have to be served immediately.
51     * @return the batch date of this timed request
52     **/
53    Date getBatchDate();
54    /**
55     * Returns the set of request info objects describing media and QoS parameters
56     * of served media objects batched at the date of this timed request.
57     * @return the request infos of media streams served at the batch date
58     ***/
59    RequestInfoSeq getRequestInfos();
60  }; // TimedRequest

61  typedef sequence<TimedRequest> TimedRequestSeq;
```

```
62  /** An interface providing access to client request information. **/
63  interface Client {
64    /**
65     * Delivers the client's IP address.
66     * @return the client's IP address.
67     **/
68    string getHostAddress();
69    /**
70     * Delivers the sequence of timed requests of this client, ascending ordered
71     * by their batch dates.
72     * @return the timed requests of this client
73     **/
74    TimedRequestSeq getTimedRequests();
75  }; // Client

76  /** Provides all required informations for running and managing an
77   *  adaptive application on a remote host.  */
78  interface ApplicationInfo {
79    /**
80     * The name of the application to which this info belongs.
81     * @return the application's name
82     **/
83    string getApplicationName();
84    /**
85     * The name of the class that will be the CORBA object.
86     * @return the main class name
87     **/
88    string getMainClassName();
89    /**
90     * Returns a byte stream of a JAR file containing all classes
91     * needed by the application.
92     * @return the application's JAR file as a stream of bytes
93     **/
94    OctetSeq getApplicationJAR();
95    /**
96     * Returns a byte stream of a ZIP file containing all files needed
97     * by the application.
98     * @return the dependent files archive as a stream of bytes
99     **/
100   OctetSeq getDependentFilesZIP();
101 }; // ApplicationInfo

102 typedef sequence<Client> ClientSeq;

103 interface AdaptiveApplication {
104   /**
105    * Checks, whether the application is idle, meaning that it is not occupied
106    * with serving any client requests. This method is useful for checking, if
107    * e.g. the application can be evacuated without breaking client services.
108    * @return true, if the application is idle, false otherwise
109    **/
110   boolean isIdle();
```

```
111     /**
112      * Starts the service process of this application using the given set of
113      * parameters for initializing it.
114      * @param params the set of parameters to use for initialization
115      * @exception ServiceAlreadyStartedException if the application has already
116      *            been started
117      **/
118     void start(in StringSeq params) raises (ServiceAlreadyStartedException);
119     /**
120      * Stops the service process of this application. This method blocks until
121      * all scheduled client requests are completely finished. If the <i>force</i>
122      * option is enabled, all client requests are subsequently descheduled and the
123      * service process stops immediately.
124      * @param force indicates, if the service process should be stopped,
125      *        regardless of whether clients are currently served, or not.
126      * @exception ServiceNotStartedException if the adaptive application has not
127      *            yet been started
128      **/
129     void stop(in boolean force) raises (ServiceNotStartedException);
130     /**
131      * Delivers the descriptive object for this application including its
132      * executables and dependencies.
133      * @return the descriptive object
134      **/
135     ApplicationInfo getApplicationInfo();
136     /**
137      * The sequence of clients which is served by this adaptive application.
138      * @return the set of served clients
139      **/
140     ClientSeq getClients();
141     /**
142      * Locks or unlocks this adaptive application. Locking the application means
143      * preventing it from accepting any further incoming client requests.
144      * @param locked if true, this adaptive application is prevented from accepting further
145      * incoming client requests. If false, the application may accept further requests.
146      **/
147     void setLocked(in boolean locked);
148     /**
149      * Indicates, if this application has been locked or not.
150      * @return true, if the application is being prevented from accepting
151      * incoming client requests, false otherwise.
152      **/
153     boolean isLocked();
154   }; // AdaptiveApplication
155 }; // vagabond2
156
```

4.2 The Application Service Layer

The set of public interfaces defined in the application service layer provide means for migrating, replicating and locating instances of an adaptive server application. The migration process of an adaptive application is an atomic operation composed of first evacuating it from the server node it is running on, and second loading it on the target server node. As described in section 4.1, the application instance must be in an idle state in order to guarantee that no currently served or future admitted client requests are broken. To ensure that an adaptive application will become idle its service needs to be stopped and the application

instance must be locked in order to not accept any future incoming requests. Furthermore, an actual image of the persistent state of the application instance has to be taken by retrieving the application instance' current application info object. The replication process is a special kind of the migration process, with the difference that the target application instance is not evacuated from its origin server node.

The application service, which represents the central interface of this layer, provides and maintains access to all instances of an adaptive application through a so-called application locator (interface *ApplicationLocator*). This locator allows for performing the operations on an adaptive application instance during a migration or replication process, as described before. The following code listing illustrates the public interfaces of the application service layer in CORBA-IDL syntax.

```
1  module vagabond2 {
2    module services {
3      module management {
4        module application {
5          /** Provides means for locating instances of adaptive applications. **/
6          interface ApplicationLocator {
7            /**
8             * Delivers the name of the located adaptive application.
9             * @return the name of the located adaptive application
10            **/
11           string getApplicationName();
12           /**
13             * Returns the host addresses which currently run instances of
14             * applications managed by this application locator.
15             * @return the host address running the located application
16             **/
17           StringSeq getLocations();
18           /**
19            * Returns the CORBA reference of the application running on the given host.
20            * @return the CORBA object referene of the application running on the given
21            * host, or null, if the host does not run such an adaptive application.
22            * @exception NoSuchHostException if the host does not exist
23            **/
24           AdaptiveApplication getApplicationObject(in string hostAddreess)
25                                                    raises (NoSuchHostException);
26         }; // ApplicationLocator

27         /** Provides means for migrating, replicating and locating applications. **/
28         interface ApplicationService {
29           /**
30            * Loads the given application on the given host.
31            * @param ai the descriptive info object of the adaptive application
32            * @param hostAddress the IP address of the target host
33            * @return true on success, false otherwise
34            * @exception ApplicationAlreadyLoadedException if the given application
35            *            has already been loaded on the target host
36            * @exception NoSuchHostException if the host does not exist
37            **/
38           boolean load(in vagabond2::ApplicationInfo ai, in string hostAddress)
39                        raises (ApplicationAlreadyLoadedException, NoSuchHostException);
```

```
40          /**
41           * Evacuates the given application from the given host.
42           * @param appName the name of the application to evacuate
43           * @param hostAddress the IP address of the target host
44           * @return true on success, false otherwise
45           * @exception NoSuchApplicationException if the application name is unknown
46           * @exception ApplicationNotLoadedException if the application name is known,
47           *            but the application has not been loaded on the target host
48           * @exception NoSuchHostException if the host does not exist
49           **/
50          boolean evacuate(in string appName, in string hostAddress) raises
51            (NoSuchApplicationException, ApplicationNotLoadedException, NoSuchHostException);
52          /**
53           * Delivers the names of all applications running on the given host.
54           * @return the application names running on the given host
55           * @exception NoSuchHostException if the host does not exist
56           **/
57          StringSeq getApplications(in string hostAddress) raises (NoSuchHostException);
58          /**
59           * Delivers the application locator referring to all locations,
60           * where instances of the given application currently run on.
61           * @param appName the name of the adaptive application
62           * @return the locator providing access to all instances of the given application
63           * @exception NoSuchApplicationException if the given application name is unknown
64           **/
65          ApplicationLocator getApplicationLocator(in string appName)
66                                                  raises (NoSuchApplicationException);
67          /**
68           * Checks, if the given application runs on the given host.
69           * @param appName the name of the application to check
70           * @param hostAddress the IP address of the target host
71           * @return true, if the given application runs on the given host, false otherwise
72           * @exception NoSuchApplicationException if the given application name is unknown
73           * @exception NoSuchHostException if the host does not exist
74           **/
75          boolean runsOn(in string appName, in string hostAddress)
76                          raises (NoSuchApplicationException, NoSuchHostException);
77        }; // ApplicationService
78      }; // application
79    }; // management
80  }; // services
81 }; // vagabond2
82
```

4.3 The Adaptation Service Layer

The main task of the interfaces provided by the adaptation layer is the one described in section 3.3, namely to find the optimum number and/or the optimum locations for running a certain adaptive server application. This optimum distribution of an adaptive application has to be suggested by a so-called *HostRecommender*, which is provided by the central interface of this layer, namely the *AdaptationService* interface. In particular, the host recommender allows for checking the quality of the current application distribution and recommends a set of server nodes for a given application and a given set of client requests. There is an important difference between the set of client requests served by an adaptive application, and the set of client requests provided to the host recommendation process. The latter denotes a set of requests which currently has not yet been accepted by the adaptive server application. However, the

former set has already been accepted and needs to be taken into account during the recommendation process as a fixed input parameter.

The recommendations of the host recommender are based on the current adaptation policy used by the adaptation service. As in the case of the AMS server architecture, different adaptation policies might be followed, which have been described in section 3.4. The adaptation service also allows for taking over and releasing control of a given application. In this scenario, the adaptation service itself controls the distribution of a server application without requiring explicit instructions from a controlling instance outside Vagabond2. The public interfaces of the adaptation service layer in CORBA-IDL syntax are presented within the following code listing.

```
1  module vagabond2 {
2    module services {
3      module management {
4        module adaptation {
5          /** Recommends a set of hosts for a certain adaptive application. **/
6          interface HostRecommender {
7            /**
8             * Checks, whether the given application is optimally distributed.
9             * @param appName the name of the adaptive application
10            * @param clients the set of clients whose requests have not yet been admitted
11            *        to be served by the adaptive application
12            * @return true, if the given application is optimally distributed using the
13            *         current adaptation policy, false otherwise
14            **/
15           boolean hasOptimalDistribution(in string appName, in ClientSeq clients);
16           /**
17            * Recommends the optimum set of host addresses for the given application and
18            * the given set of not yet accepted client requests.
19            * @param ai the descriptive object of the application
20            * @param clients the set of clients whose requests have not yet been admitted
21            *        to be served by the adaptive application
22            * @return the optimum set of hosts for running instances of the application
23            **/
24           StringSeq recommendHosts(in ApplicationInfo ai, in ClientSeq clients);
25         }; // HostRecommender

26         /** Represents an adaptation policy to be used by the adaptation service. **/
27         enum AdaptationPolicy {
28           MINIMUM_STARTUP_DELAY,
29           LOAD_BALANCED,
30           MINIMUM_NETWORK_CONSUMPTION
31         }; // AdaptationPolicy

32         /** Provides means for explicitly or automatically adapt applications. **/
33         interface AdaptationService {
34           /**
35            * Delivers the host recommender to be used for suggesting ideal hosts
36            * for a certain application.
37            * @return the host recommender
38            **/
39           HostRecommender getHostRecommender();
40           /**
41            * Delivers the adaptation policy currently followed by this adaptation service.
42            * @return the current adaptation policy
43            **/
44           AdaptationPolicy getAdaptationPolicy();
```

```
45          /**
46           * Suggests the ideal adaptation policy by taking into account the current and
47           * especially future (estimated) set of client requests.
48           * @return the suggested adaptation policy
49           **/
50          AdaptationPolicy suggestPolicy();
51          /**
52           * Delivers the addresses of all available hosts belonging to the
53           * application cluster, which are able to run the given application.
54           * @param ai the descriptive object of the application to run
55           * @return the cluster's set of host addresses which can run instances
56           *         of the given application
57           **/
58          StringSeq getAvailableHosts(in ApplicationInfo ai);
59          /**
60           * Tells the adaptation service to take over control of the distribution
61           * of instances of the given application.
62           * @param appName the name of the application to control
63           * @exception NoSuchApplicationException if the given application name is unknown
64           **/
65          void adaptApplication(in string appName) raises (NoSuchApplicationException);
66          /**
67           * Releases the given application from being automatically adapted by this
68           * adaptation service.
69           * @param appName the name of the application to release control
70           * @exception NoSuchApplicationException if the given application name is unknown
71           **/
72          void releaseApplication(in string appName) raises (NoSuchApplicationException);
73          /**
74           * Delivers the set of application names which are currently
75           * automatically adapted by this adaptation service.
76           * @return the names of the automatically adapted applications
77           **/
78          StringSeq getAdaptedApplications();
79        }; // AdaptationService
80      }; // adaptation
81    }; // management
82  }; // services
83 }; // vagabond2
84
```

5 AMS-Vagabond2 Interaction Scenarios

The major classes and interfaces of the AMS server architecture being involved in interaction scenarios with the Vagabond2 middleware, and thereby with the management plane, are illustrated in figure 7.

The *AdaptiveApplication* interface of Vagabond2 is specialized into an *AMSServerApplication*, which allows for a general creation and management of AMS server sessions, as well as for transactional issues in the case where media objects are distributed among server nodes. Furthermore, the three AMS server applications *cluster manager*, *data manager* and *data collector* are derived from this *AMSServerApplication* interface. Each of these CORBA interfaces has an associated implementation class, which is contained in an implementation of the Vagabond2 interface *ApplicationInfo*.

The main controlling instance of the AMS server implementation is the application manager (class *AMSApplicationManager*). It provides access to the application and adaptation services of Vagabond2, and manages three application controllers which provide means for controlling the distribution of an ap-

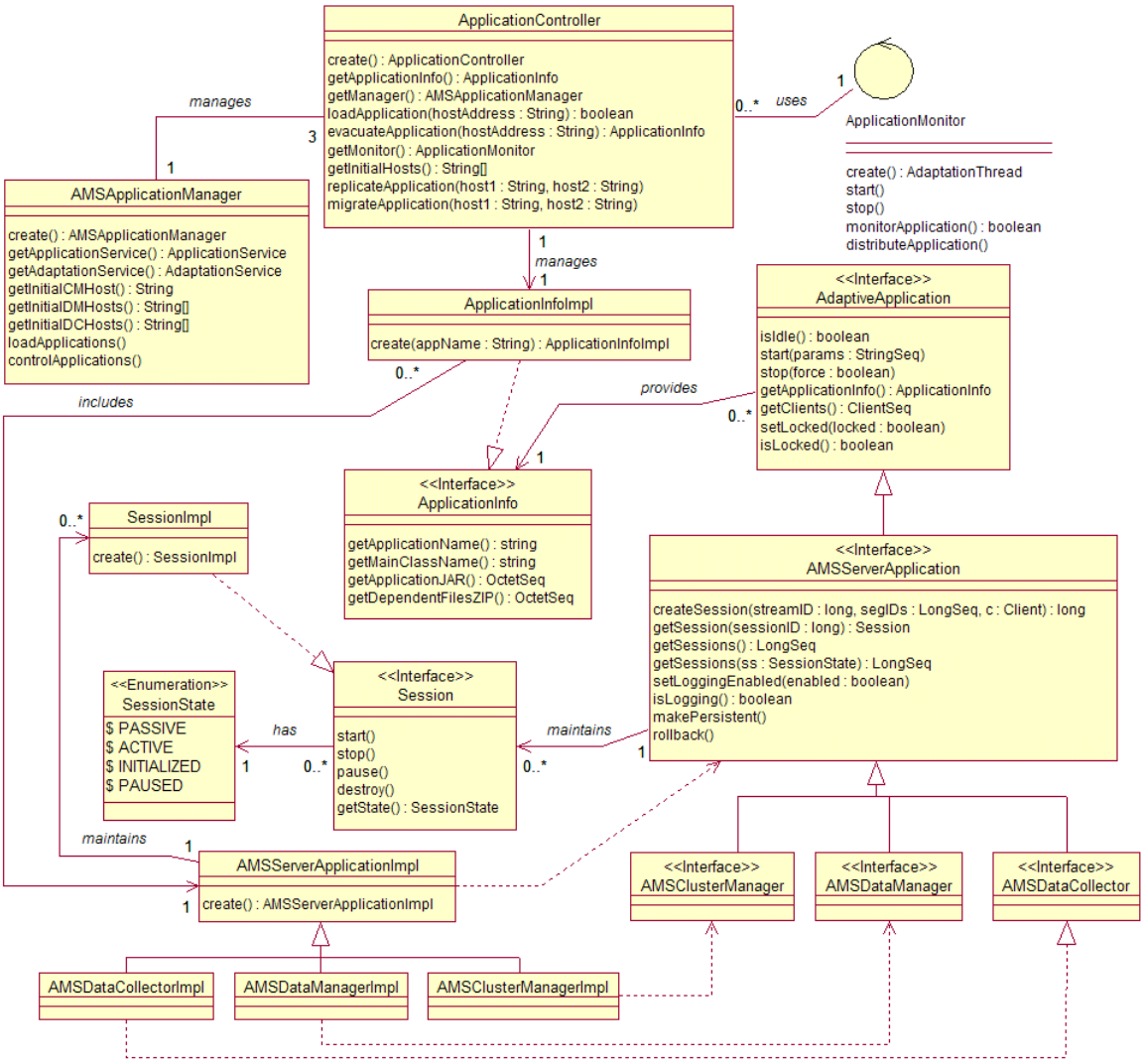

Figure 7: AMS entities involved in interactions with Vagabond2 interfaces

plication. There is exactly one application controller for each distinguished AMS server application. An application controller itself only provides the methods for loading, evacuating, replicating or migrating an application of its managed application type. Issues on monitoring and reorganizing the distribution of instances of its applications are handled by a corresponding application monitor (class *Application-Monitor*). Initially, for each distinguished application there must be a set of server nodes known by the application manager, in order to load the applications on these hosts. The required descriptive application info objects are provided by the corresponding application controllers. The following sequence diagrams illustrate the most common interaction scenarios of AMS server instances with the Vagabond2 middleware. The sequence diagrams only show the use cases for managing and adapting an AMS *cluster manager* application. However, these diagrams are also valid for a *data manager* and a *data collector* application.

5.1 AMS Server Initialization

Figure 8 covers the steps taken during the first initialization of the distributed AMS server. During the creation of the application manager instance an application controller and an application monitor are created for each of the three distinguished AMS server applications[1]. The application manager also retrieves the CORBA object references to the application and adaptation services provided by Vagabond2.

The server initialization process requires the set of initial host addresses for running the cluster manager, the data manager and the data collector applications to be given as input parameters. Based on these sets of addresses the application manager initiates the loading of the applications on the corresponding hosts. This is achieved by using management facilities provided by the controller of each application type. After having loaded the applications on the desired hosts, the application manager tells the application monitor of each controller to monitor and reorganize the distribution of its application instances, if necessary.

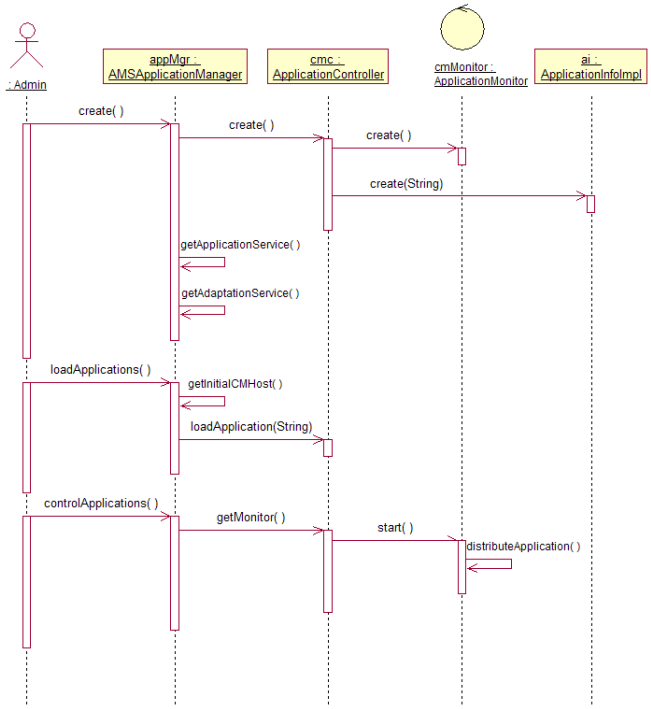

Figure 8: Initializing a distributed AMS server

5.2 Managing an AMS Server Application

5.2.1 Loading an Application

Loading an AMS application on a given host always refers to loading it based on the initial application info object, which is maintained by the corresponding application controller. The application controller of a certain AMS application initially creates a static image of the application's program and data binaries, which are further used during application loading processes.

An application loading process is always directly initiated by the application manager, or indirectly by an application monitor when performing a application migration. As illustrated in figure 9, the

[1]Due to space shortage only the creation of a controller and a monitor for the *cluster manager* application are demonstrated.

application controller of the cluster manager application is ordered to load its static application image on a given host. The controller achieves this by (i) stopping its associated monitor in order to not interfere with the current process, (ii) retrieving the application service reference from the application manager, and (iii) loading the application on the given host using the application service. Afterwards, the service of the new instance of an *AMSClusterManager* is explicitly started, using the application locator provided by the application service. Finally, the application controller's monitor is started again.

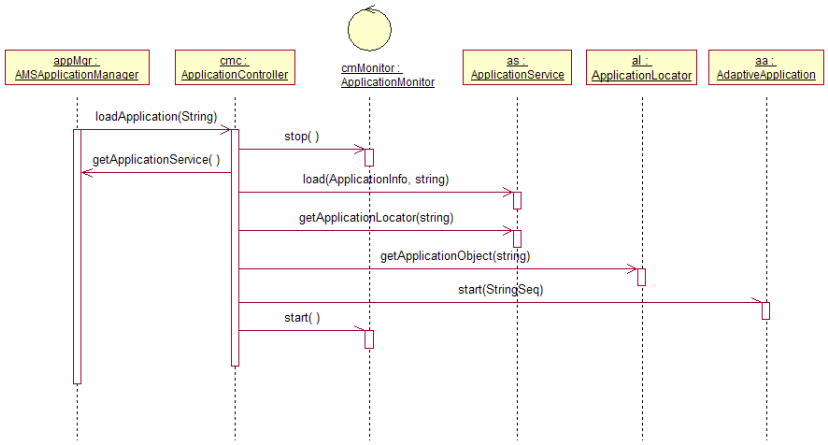

Figure 9: Loading the cluster manager application on a given host

5.2.2 Evacuating an Application

Figure 10 shows the steps taken during the evacuation of a cluster manager application. The evacuation of an AMS server application is always initiated by its corresponding application monitor. After retrieving the application's name from the descriptive application info object, the instance of the application on the target host is retrieved by using an application locator of Vagabond2's application service. The first operation on the target application instance is to lock it, in order to prevent it from accepting further client requests. Second, the service of the target instance is stopped. Usually, a stop should not enforce an abrupt termination of all current connections of this application instance. Thus, the target application finishes serving all its accepted client requests till now and a call of the stop operation blocks until the application finishes processing its admitted requests. Finally, the application instance is evacuated from the target host using Vagabond2's application service.

5.2.3 Replicating an Application

Replicating an AMS server application, as illustrated in figure 11 for the cluster manager application, slightly differs from loading an adaptive application. During a replication, the dynamic application image of the source application instance is taken, rather than the static image maintained by application controller. Here, the term *image* always refers to the persistent state of the application, including its binaries and data dependencies. This means that e.g. a cluster manager application is replicated including its latest data repository file, which itself includes the knowledge which media objects are distributed among which data manager nodes. Having retrieved the dynamic application info object of the source application instance, a new application instance is loaded on the target host, using the application service.

5.2.4 Migrating an Application

The migration of an AMS application instance from one server node to another consists of the two sub-processes of evacuating the old instance from the source node and loading the new instance on the target

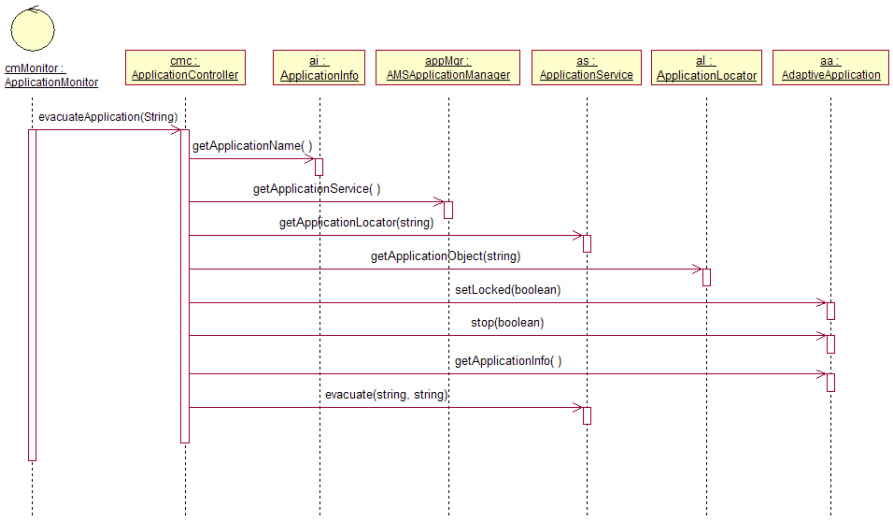

Figure 10: Evacuating the cluster manager application from a given host

node, as shown in figure 5.2.2 in the case of a cluster manager application. It is important to mention that the evacuation process of the application controller returns an application info object, which provides the latest image of the evacuated application. This ensures a consistent migration of the persistent state of an application instance.

5.3 Adapting an AMS Server Application

5.3.1 Monitoring an Application

Monitoring the distribution of application instances is a typical task of an application monitor. Figure 13 illustrates the steps taken during the monitoring process of the cluster manager monitor. After retrieving the adaptation service via the application manager, the monitor periodically queries the host recommender to check, whether the given application is optimally distributed, or not. As an input to this verification process, the monitor provides a set of not yet admitted future client requests, which is retrieved from the current cluster manager instance.

5.3.2 Distributing an Application

Distributing the instances of an AMS server application denotes the general process of application adaptation. As illustrated in figure 14, the distribution process is performed by the application monitor of the associated application. If the application monitoring subprogram indicates that the instances of the application are not optimally distributed, it calls the host recommender of the adaptation service to recommend the optimum set of hosts (S_o) for the application. Again, the monitor provides a set of not yet admitted future client requests as input to the recommendation process. In the next step, the current set (S_c) of server nodes running instances of the monitored application is retrieved by querying the corresponding application locator provided by the application service. These two sets S_o and S_c enable for the calculation of the adaptation sets defined in equation 2.

$$\begin{aligned} S_k &= S_o \cap S_c \\ S_{ec} &= S_c \setminus S_k \\ S_n &= S_o \setminus S_k \end{aligned} \tag{2}$$

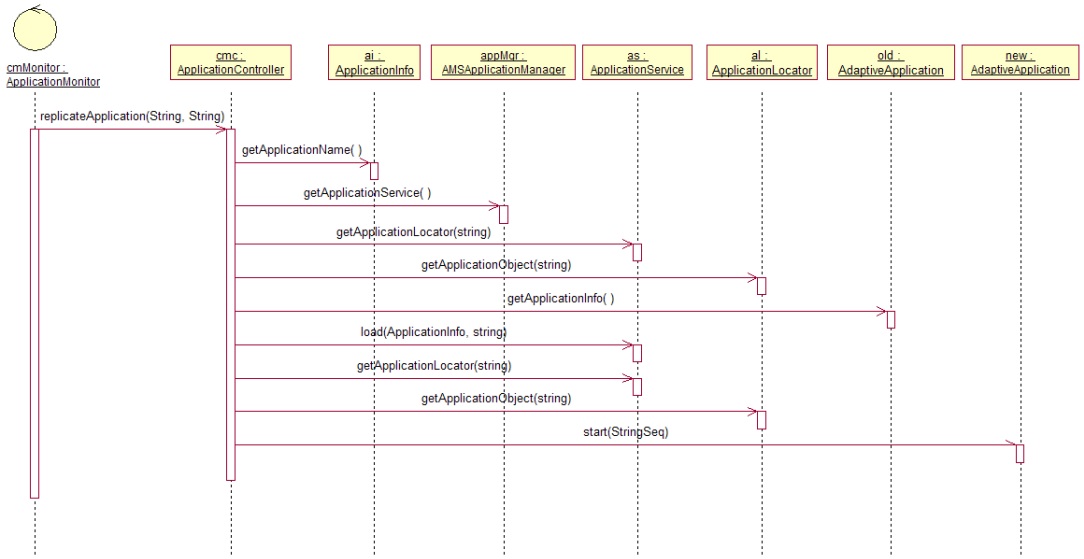

Figure 11: Replicating the cluster manager application from one host to another

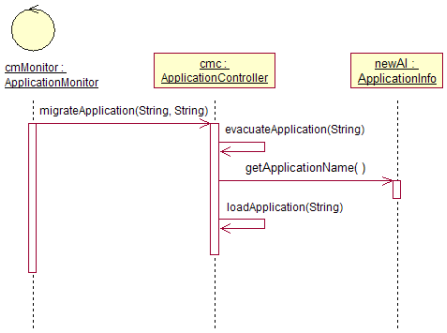

Figure 12: Migrating the cluster manager application from one host to another

S_k represents the set of server nodes which remains unchanged. S_{ec} includes all server nodes which are candidates for application evacuation. S_n denotes the set of new server nodes which need to be equipped by instances of the application. However, it is in unclear until now which applications on nodes in S_{ec} are replicated or migrated to nodes in S_n, and which applications are simply evacuated from nodes in S_{ec}. It is obvious that the host recommendation process should provide some hints concerning the future steps to take for the nodes in S_{ec}.

6 Conclusion

This paper presented the requirements and components of a multimedia streaming server architecture that supports proactive server-level adaptations, in cases where server resources or startup latencies to clients become bottlenecks. It defined the three distinguished streaming server applications *cluster manager*, *data manager* and *data collector*, which interact together for serving client requests. Beside the cluster manager application, each of the two other applications might have multiple instances running on a variable number of server nodes, resulting in a virtual server environment. This allows to optimize server resource usage and to overcome the shortcomings of a static distributed server environment.

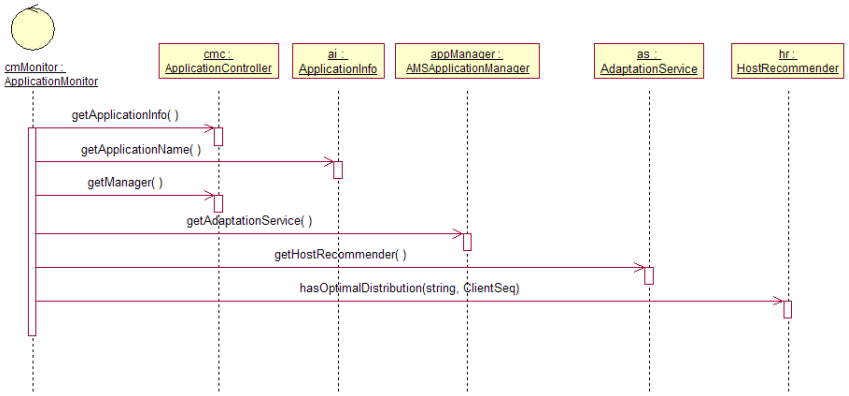

Figure 13: Monitoring the distribution of the cluster manager application

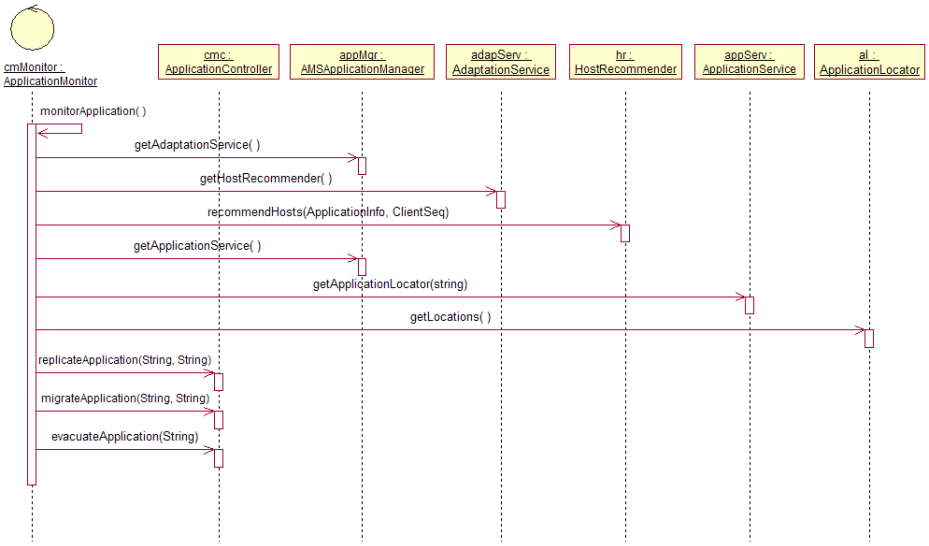

Figure 14: Distributing the cluster manager application on recommended hosts

The distribution of each server application is controlled by an associated application controller, which itself cooperates with the underlying CORBA-based middleware Vagabond2. The middleware therefore provides means for both, application management and adaptation management issues. It allows to load static and to evacuate, replicate or migrate dynamic persistent images of server applications. In cases of a required application adaptation Vagabond2 also provides recommendations concerning the optimum set of server nodes to acquire for it. Host recommendations are based on the sets of current (already admitted) and future (not yet admitted) client requests, as well as the estimated server and network loads. The current and future set of client requests is provided by the server architecture as input to the host recommender.

In the near future a complete implementation and detailed evaluation of an adaptive multimedia streaming application based on MPEG-encoded media streams is planned. It has to be figured out, which of the discussed adaptation policies performs best under a given set of constraints (like e.g. the distribution of application instances and the distribution of client requests).

References

[1] Moses Charikar and Sudipto Guha, *Improved Combinatorial Algorithms for the Facility Location and k-median Problems*, IEEE Symposium on Foundations of Computer Science, 1999, pp. 378–388.

[2] Moses Charikar, Sudipto Guha, E. Tardos, and D. B. Shmoys, *A Constant-factor Approximation Algorithm for the k-median Problem*, Proceedings of the 31st Annual ACM Symposium on Theory of Computing, 1999.

[3] F. A. Chudak and D. B. Shmoys, *Improved Approximation Algorithms for the Capacitated Facility Location Problem*, Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, 1999, pp. 875–876.

[4] Israel Cidon, Shay Kutten, and Ran Soffer, *Optimal Allocation of Electronic Content*, Proceedings of the IEEE Infocom, April 2001.

[5] Roy Friedman, Eli Biham, Ayal Itzkovitz, and Assaf Schuster, *Symphony: An Infrastructure for Managing Virtual Servers*, Cluster Computing **4** (2001), no. 3, 221–233.

[6] Zihui Ge, Ping Ji, and Prashant Shenoy, *A Demand Adaptive and Locality Aware (DALA) Streaming Media Server Cluster Architecture*, 12th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'02), May 2002.

[7] Balázs Goldschmidt and Zoltán László, *Vagabond: a CORBA-based Mobile Agent System*, Object-Oriented Technology ECOOP Workshop Reader (A. Frohner, ed.), Springer Verlag, 2001.

[8] Balázs Goldschmidt, Roland Tusch, and László Böszörményi, *A Mobile Agent-based Infrastructure for an Adaptive Multimedia Server*, 4th Austrian-Hungarian Workshop on Distributed and Parallel Systems, September - October 2002.

[9] Frank Hartung, Bernd Girod, and Uwe Horn, *Multiresolution Coding of Image and Video Signals*, EUSIPCO'98, September 1998.

[10] Uwe Horn and Bernd Girod, *Scalable Video Transmission for the Internet*, Computer Network and ISDN Systems **29** (1997), no. 15, 1833–1842.

[11] Uwe Horn, K.W. Stuhlmuller, M. Link, and Bernd Girod, *Robust Internet Video Transmission Based on Scalable Coding and Unequal Error Protection*, Image Communication **15** (19999), no. 1-2, 77–94.

[12] J. Kangasharju, J. Roberts, and K. W. Ross, *Object Replication Strategies in Content Distribution Networks*, Proceedings of WCW'01: Web Caching and Content Distribution Workshop, June 2001.

[13] O. Kariv and S. L. Hakimi, *An Algorithmic Approach to Network Location Problems. II: The p-Medians*, SIAM Journal on Applied Mathematics **37** (1979), no. 3, 539–560.

[14] Bong-Jun Ko and Dan Rubenstein, *A Greedy Approach to Replicated Content Placement Using Graph Coloring*, SPIE ITCom Conference on Scalability and Traffic Control in IP Networks II, July 2002, to appear.

[15] Jack Y.B. Lee, *Parallel Video Servers: A Tutorial*, IEEE Multimedia **5** (1998), no. 2, 20–28.

[16] Bo Li, Mordecai Golin, Giuseppe Italiano, Xin Deng, and Kazem Sohraby, *On the optimal placement of web proxies in the internet*, Proceedings of IEEE Infocom, March 1999.

[17] Weiping Li, *Overview of Fine Granular Scalability in MPEG-4 Video Standard*, IEEE Transactions on CSVT. **11** (2001), 301–317.

[18] Moving Picture Experts Group, ISO/IEC JTC1/SC29 WG11, http://mpeg.telecomitalialab.com/, *MPEG-4 Visual Amendment 4: Streaming video profile*, 2000.

[19] Object Management Group (OMG), http://cgi.omg.org/docs/formal/01-09-67.pdf, *OMG Unified Modeling Language Specification*, 1.4 ed., 2001.

[20] C. Papadimitriou, *Worst-case and Probabilistic Analysis of a Geometric Location Problem*, SIAM Journal on Computing **10** (1981), 542–557.

[21] Lili Qiu, Venkata Padmanaban, and Geoffrey M Voelker, *On the Placement of Web Server Replicas*, Proceedings of IEEE Infocom, April 2001.

[22] Jerome Rolia, Sharad Singhal, and Richard Friedrich, *Adaptive Internet Data Centers*, Intl. Conf. on Advances in Infrastructure for Electronic Business, Science, and Education on the Internet (SSGRR), July 2000.

[23] Douglas C. Schmidt, *The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software*, Washington University, 1994.

[24] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, *The Design of the TAO Real-Time Object Request Broker*, Computer Communications, Elsivier Science **21** (1998), no. 4.

[25] Roland Tusch, *Design of a Modular Adaptive Virtual Video Server Architecture for On-Demand Video Services*, Object-Oriented Technology ECOOP Workshop Reader (A. Frohner, ed.), Springer Verlag, 2001.

[26] Michaela van der Schaar and Hayder Radha, *A Hybrid Temporal-SNR Fine Granular Scalability for Internet Video*, IEEE Transactions on CSVT **11** (2001), 318–331.

Institute of Information Technology
University Klagenfurt
Universitaetsstr. 65-67
A-9020 Klagenfurt
Austria

http://www-itec.uni-klu.ac.at

University Klagenfurt