A MetaArchitecture for QoS-Based Distributed Resource Management

Nalini Venkatasubramanian[*] Carolyn Talcott[**] Gul Agha[***]

No Institute Given

Abstract: *Systems that provide distributed multimedia services are subject to constant evolution - customizable middleware is required to effectively manage this change. In this paper we present a meta-architectural framework for customizable QoS-based middleware using Actors, a model of concurrent active objects. Middleware services for resource management execute concurrently with each other and with application activities— scheduling, protocols providing security and reliability, load balancing and stream synchronization can therefore potentially interfere with each other. To ensure cost-effective QoS in distributed multimedia systems, safe composability of resource management services is essential. For instance, system protocols and activities must not cause arbitrary delays in the presence of timing based QoS constraints. Using TLAM, a semantic model for specifying and reasoning about components of open distributed systems, we show how a QoS brokerage service can be used to coordinate multimedia resource management services in a safe, flexible and efficient manner.*

In the coming years, distributed multimedia servers will be deployed to deliver a variety of interactive, digital multimedia(MM) services over emerging broadband (wide-area) networks [**?**] to form a wide-area infrastructure. Applications such as telemedicine, distance learning and electronic commerce exhibit varying requirements such as timeliness, security, reliability and availability. Many MM applications can tolerate minor, infrequent violations of their performance requirements specified as a quality-of-service (QoS) parameter, for e.g., tolerable jitter in a video frame. Systems that provide distributed multimedia services are continuously changing and evolving. For instance, the set of servers, clients, user requirements, network and system conditions, in a wide area infrastructure are changing continuously. Multiple protocols that cater to a wide range of network configurations and software applications must be supported seamlessly and handle dynamically changing QoS requirements. In order to manage the distributed components and adapt to the above dynamic changes in multimedia applications, customizable middleware services are required. Today, the task of distributed systems management is performed in middleware layers using frameworks such as CORBA and DCOM. Such frameworks are designed for heterogeneous interoperability but are limited in the degree of flexibility and customizability of services. They provide only limited capabilities for the specification and adaptation of end-to-end QoS properties. Future applications will require dynamic invocation and revocation of services distributed in the network without violating QoS constraints of ongoing applications. Customizable middleware allows us to deal with changes in systems and applications in a non-intrusive way.

In this paper, we describe techniques for developing components of customizable component based middleware infrastructures. We develop a model for customizable, cost-effective middleware to enforce QoS requirements in multimedia applications. For instance, we illustrate a placement policy that determines the degree of replication necessary for popular MM objects using a cost-based optimization procedure based on *a priori* predictions of expected subscriber requests. For scheduling MM requests, we represent an adaptive scheduling policy that compares the relative utilization of resources in a multimedia server to determine an assignment of requests to replicas. To optimize storage utilization, we introduce methods for dereplication of MM objects based on changes in their popularity and in server usage patterns. The meta-architectural middleware framework developed in this paper ensures composability of concurrently executing system components. This meta-architecture is the basis of experimental implementation of resource management

[*] Contact Author: Nalini Venkatasubramanian Dept. of Information and Computer Science, Univ. of California Irvine, Irvine, CA 92697-3425, Email: nalini@ics.uci.edu

[**] Department of Computer Science, Stanford University, Stanford, CA 94305, Email: clt@cs.stanford.edu

[***] Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, Email: agha@cs.uiuc.edu

mechanisms for distributed MM servers. Details of the mechanisms implemented and the performance evaluation of the composite environment is described in [**?**]; detailed formal specification and correctness reasoning of system activities is described in [**?**].

Performance studies show that application objects can be managed effectively by composing multiple resource management activities managed at the metalevel [**?**]. Figures **??** and **??** illustrate the performance, measured by request rejection rate, of various policies for load management - (a) purely adaptive (on-the-fly) scheduling and placement(P1), (b) purely predictive (decided apriori) scheduling and placement(P2), (c) composite policies that provide adaptive scheduling and predictive placement(P3 and P4, an optimized version of policy P3). Figure **??** illustrates the request rejection rate under purely adaptive policies for placement and scheduling. *Startup latency* is a QoS factor that indicates how long the user is willing to wait for a replica to be created adaptively. The graph demonstrates that when the startup latency is below a threshold value (2 min), the purely adaptive mechanisms, represented by P1 force a very large fraction of the requests received to be rejected. Assuming that startup latency is sufficiently large, Figure **??** depicts the inadequacy of P2, that relies on only predictive policies for scheduling and placement. In comparison, the other 3 policies (P1, P3 and P4), show hardly any rejects (indicated by the overlapping lines in the graph). As can be observed from the performance results, the ability to run multiple policies simultaneously (as in cases P3 and P4) reduced the total number of rejected requests in the overall system.

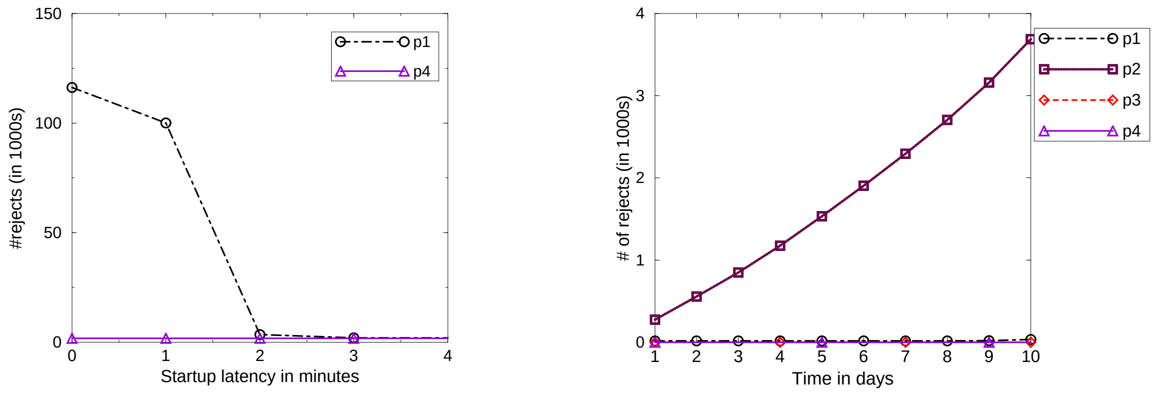

Fig. 1. Comparison of the performance of load management policies for request scheduling and video placement in a distributed video server. In the graph on the left hand side, Policy P1 which represents purely adaptive policies has a much higher rejection rate than policy P4 that represents composite adaptive and predictive mechanisms. In the right hand side graph, Policy P2 which represents purely predictive mechanisms has a much higher rejection rate than P1 (purely adaptive mechanisms), P3 and P4(composite adaptive and predictive mechanisms) for data placement and request scheduling.

The rest of this paper is organized as follows. Section 1 reviews the two level semantic framework for distributed resource management based on Actors, a model of concurrent objects. Section 2 describes a multimedia metaarchitecure, its components and interactions among resource management services. Section 3 presents the several policies and optimizations for load management represented in the meta-architecture. Section 4 briefly describes the formalization of the multimedia meta-architecture and the representation of the policies discussed in the two level semantic framework. Section 5 discusses related work and outlines areas for future research.

1 The Two Level MetaArchitecture

The Actor model is a model of concurrent active objects that has a built-in notion of encapsulation and interaction and is thus well-suited to represent evolution and co-ordination among interacting components in distributed multimedia applications. In the actor paradigm, the universe contains computational agents called *actors*, distributed over a network. Traditional passive objects encapsulate state and a set of procedures that manipulate the state; actors extend this by encapsulating a thread of control as well. Each actor potentially executes in parallel with other actors and interacts only by sending and receiving messages. On receiving a communication, an actor processes the message and as a result may do one or more of the following: (a) create new actors; (b) change its state; (c) send messages to actors that it knows about. (See [**?**,**?**] for more discussion of the actor model, and for many examples of programming with actors.)

In [**?**,**?**], we presented the TLAM(Two Level Actor Machine) framework for specifying, composing and reasoning about resource management services in open distributed systems. In the TLAM, a system is composed of two kinds of actors, base actors and meta actors, distributed over a network of processing nodes. *Base-actors* carry out application level computation, while *meta-actors* are part of the runtime system which manages system resources and controls the runtime behavior of the base level. A TLAM provides an abstract characterization of actor identity, state, messages, and computation, and of the connection between base and meta level computation. Meta-actors communicate with each other via message passing as do base-actors, and they may also examine and modify the state of the base actors located on the same node. Base level actors and messages have associated runtime annotations that can be set and read by meta actors, but are invisible to base level computation. Actions which result in a change of base-level state are called events. Meta actors may react to events occurring on their node. A TLAM configuration, C, has a set of base and meta level actors and a set of undelivered messages. The actors are distributed over the TLAM nodes. Each actor has a unique name (address) and the configuration associates a current state to each actor name. The undelivered messages are distributed over the network – some are traveling along communication links and others are held in node buffers. The semantics of a TLAM is given by a labeled transition relation on configurations. There are two kinds of transitions: communication and execution. Communication transitions move undelivered messages around the network and are the same in every TLAM. An execution transition is a computation step taken by a base or meta level actor. These transitions are described by *reaction rules* specific to a particular TLAM that determine how individual actors react to messages received, and in the case of meta actors how they react to events. A *computation path* for a configuration is a possibly infinite sequence of labeled transitions. The semantics of a configuration is the set of fair computation paths starting with that configuration. A *system* is a set of configurations closed under the transition relation.

Core Services for Resource Management: Meta-level actors provide services that customize various aspects of distributed systems management. In practice, multiple system and application activities occur concurrently in a distributed system, e.g. scheduling, load balancing, protocol processing, stream synchronization – such compositions can lead to complex interactions. Consider the following example of a system where distributed garbage collection and process migration can proceed concurrently. If processes in transit are not accounted for, the garbage collection process can potentially destroy accessible information. In general, risks that arise due to composition of services include loss of information, possible nonterminations that cause deadlocks and livelocks, dangling resources, inconsistencies, and incorrect execution semantics. Current approaches to deal with such interference include: serializing or delaying activities, which may cause QoS violations; and designing a closed system, which reduces portability of code.

Composability of resource management activities is not just desirable, it is *essential* to ensure cost-effective QoS in distributed multimedia systems. For instance, system protocols and activities must not enforce arbitrary delays in the presence of timing based QoS constraints. This adds complexity to the design of such systems. We would like to be able to consider separately issues such as: functional behavior of an application; and resource management issues such as memory management, load balancing, QoS specification and enforcement. To ensure non-interference and manage the complexity of reasoning about components of open distributed systems in general, our strategy is to identify key system services where non-trivial interactions between the application and system occur, i.e. base-meta interactions. We refer to these key services as *core services*. As a starting point, we have identified three core services:

- Recreation of services/data at a remote site - *remote creation*

- Capturing information at multiple nodes/sites - *distributed snapshot*
- Interactions with a global repository - *directory services*.

We have studied non-interference requirements that would allow these services to safely operate concurrently. A detailed case study of the composition of remote creation and a distributed snapshot formalized in the TLAM framework appears in [**?**]. Core services may be used in specifying and implementing more

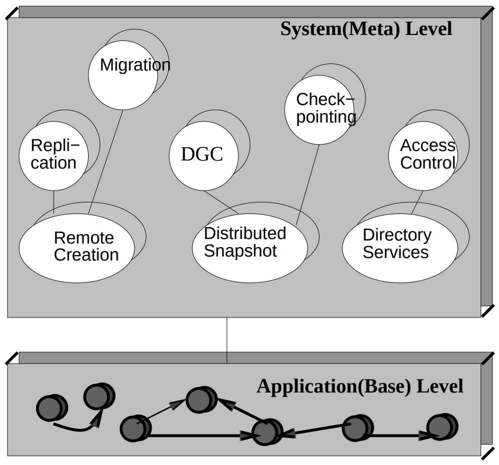

Fig. 2. Classification of Core Services

complex activities within the actor framework as purely meta-level interactions (See Figure **??**). For example, remote creation can be used as the basis for designing algorithms for activities such as migration, replication and load balancing. Distributed snapshots (cf. [**?**]). can also be used as the basis for global activities like quiescence detection, checkpointing and recovery and distributed garbage collection (cf. [**?**]). Similarly, a directory service can be used to design access control mechanisms, routing policies and group based communication.

2 A Meta-Architecture for Multimedia Servers

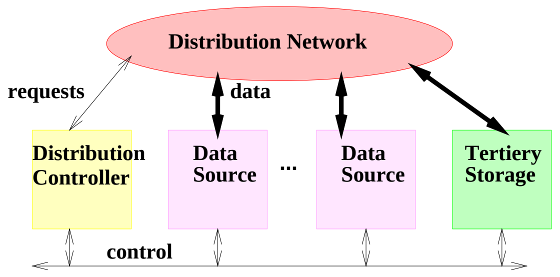

Fig. 3. Architectural View of a Networked Multimedia System

Building on the two-level model described in the previous section, we develop a meta-architectural model of a multimedia server that provides QoS based services to applications. The physical architecture of the MM server (See Figure **??**) consists of:

- A set of data sources (DS) that provide high bandwidth streaming MM access to multiple clients. Each independent data source includes high capacity storage devices (e.g. hard-disks), a processor, buffer memory, and high-speed network interfaces for real-time multimedia retrieval and transmission

- a specific node designated as the distribution controller (DC) that coordinates the execution of requests on the data sources and
- a tertiary storage server that contains the MM objects; replicas of these MM objects are placed on the data source nodes.

All the above components are interconnected via an external distribution network that also transports multimedia information to computers and set-top devices at the client end. A lower speed back-channel conveys subscriber commands back to the data sources via the DC.

The meta-architecture model of a multimedia server consists of two subsystems - the base level and meta-level subsystems corresponding to the application and system management components respectively. The base level component implements the functionality of the MM application and models both MM data objects and their replicas (e.g. video and audio files), and MM requests to access this data via *sessions*. The corresponding base-level entities are *replica actors* and *request actors*.

The Meta-Level System: The meta-level component deals with the coordination of multiple requests and sharing of existing resources among multiple requests. To provide coordination at the highest level and perform admission control for new incoming sessions, we introduce a meta-level entity, the *QoS Broker* meta-actor, QB. The two main functions of the QoS Broker are data management and request management. The organization of meta-level services provided by a QoS broker is shown in Figure **??**. Each of these services in turn can be based on one or more of the core services - remote creation and distributed snapshot.

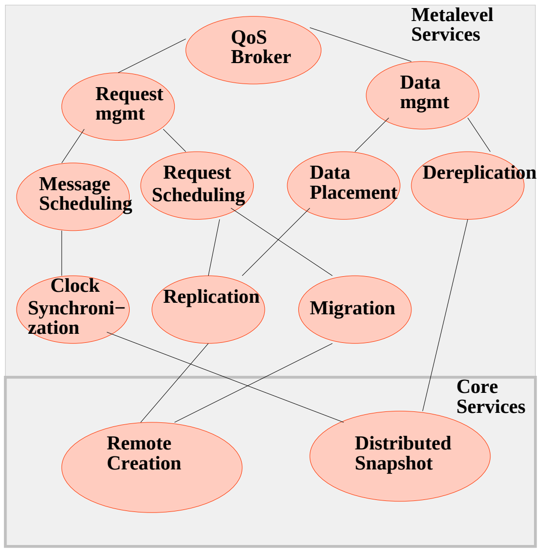

Fig. 4. Detailed Architecture of the Meta-level System

We describe some of the metalevel services in more detail. *Data Management* – The data management component decides on the placement of replica actors in the distributed MM system, i.e. it decides when and where to create additional replicas of data. It also determines when additional replicas are no longer needed and can be garbage collected/dereplicated. Data Placement can be performed adaptively or offline using a predefined placement strategy. The adaptive data placement policy dynamically initiates data placement when data or resources that are required to satisfy a request are not available.A predictive placement policy determines the degree of replication necessary for popular objects using a cost-based optimization procedure based on *a priori* predictions of expected subscriber requests.

Request management – Request arrival is unpredictable; often, it is not possible to know when requests will arrive and for how long they will last. This creates a need for adaptive admission control. The feasibility of QoS constraints for a MM session is determined during the *request scheduling* process when a new request arrives. For scheduling requests, we use an adaptive request scheduling policy that compares the relative utilization of resources in a MM server and assigns the request to a replica on a DS node. Once a request has been admitted, a *message scheduling* component assures the satisfaction of QoS constraints for service within the session.

Media data and request management functions in turn require a number of services. We describe a representative sample below.

Synchronization service: schedules message service in order to maintain a uniform notion of time among multiple nodes and thereby among actors in the session that are distributed across multiple nodes. The notion of synchronization among audio and video actors can be built by using a uniform time value implemented as a snapshot broadcast phase that informs nodes about a (approximation of) current value of time. In [**?**], we specify the notion of a session in the context of MM actors and describe a construct, *QoSSynchronizer*, for expressing session specific time and QoS constraints.

Replication service: creates replicas to support availability, load-balancing and fault tolerance. When the existing copies of a MM object are not capable of supporting additional subscriber requests, replication of the MM object becomes necessary. At the time of replication it is necessary to choose a data source on which a new replica should be created. The rate at which replication proceeds also has a direct impact on system performance and application interactivity. Replication at a rate that is faster than real-time minimizes the time for replication, but consumes greater server resources.

Dereplication service: can dereplicate data. In many time periods, a MM object may drop in popularity. By tracking the demand for different MM objects and determining when copies of a MM object can be dereplicated, the load management procedure can optimize utilization of storage space on a MM server. The copy to be dereplicated must be carefully chosen based on current load on the different data sources, as well as the expected future demands for other MM objects currently being stored or that are expected to be stored on the MM server. Dereplication cannot occur instantly - the service must ensure that a copy that has been chosen for dereplication is removed only after all requests that are currently being serviced by that copy have completed.

Migration service: migrates data or requests to support load balancing, availability and locality. Often, instead of replicating a MM object to service a new request, one can effectively migrate an existing request to another data source. Such *copy-free* migration is an attractive alternative for Internet Web servers and other services that handle non-real-time data. Often, this requires explicit tear-down and reestablishment of network connections as well as exchange of state information between data sources, both of which can cause significant playback jitter in the case of a video stream.

In order to map the QoS meta-architecture to the physical system architecture, we distinguish between local and global components and define interactions between local resource managers on nodes and the global resource management component. The global component which includes the QoS broker and associated meta-actors resides on the distribution controller node. The node local components of the QoS broker metaarchitecture implementation include, for each DS node:

- a DS meta-actor for load-management on that node. The DS meta-actor contains state information regarding the current state of the node in terms of available resources, replicas, ongoing requests and replication processes etc.
- Request base actors corresponding to the requests assigned to that node.
- Replica base actors that correspond to the replicas of data objects currently available on that node.

3 Load Management in MM Servers

Apart from the QoS broker, the MM system contains a number of meta-actors whose behaviors implement the specific resource management policies discussed. In this section, we describe mechanisms that provide a modular and integrated approach to managing the individual resources of a MM server so as to effectively utilize all of the resources such as disks, CPU, memory and network resources. A MM request specifies a client, one or more multi-media objects, and a required QoS. The QoS requirement in turn entails resource allocation requirements. The ability of a data source to support additional requests is dependent not only on the resources that it has available, but also on the MM object requested and the characteristics of the request (e.g., playback rate, resolution). We characterize the degree of loading of a data source DS with respect to request R in terms of its load factor, $LF(R, DS)$, as: $LF(R, DS) = max(\frac{DB^R}{DB^{DS}}, \frac{M^R}{M^{DS}}, \frac{CPU^R}{CPU^{DS}}, \frac{NetBW^R}{NetBW^{DS}})$ where $DB^{R(DS)}$, $M^{R(DS)}$, $CPU^{R(DS)}$, and $NetBW^{R(DS)}$ denote the disk bandwidth, memory buffer space, CPU cycles, and network transfer bandwidth, respectively, that are necessary for supporting request R (available on data source DS). The load factor helps identify the critical resource in a data source, i.e., the resource that limits the capability of the data source to service additional requests. By comparing the load factor values for different servers, load management decisions can be taken by the QoS brokerage service.

Scheduling of Multimedia Requests: The Request Scheduling meta-actor (RS) implements an adaptive scheduling policy that compares the relative utilization of resources at different data sources to generate an assignment of requests to replicas, so as to maximize the number of requests serviced. The data source that contains a copy of the MM object requested and which entails the least overhead (as determined by the computed load factor) is chosen as the candidate source for an incoming request.

If no candidate data source can be found for servicing request R, then the RS meta-actor can either reject the incoming request or initiate *replication on demand* - implemented via a replication on demand meta-actor (ROD). In the former case, the RS meta-actor analyzes the rate of rejections over longer time frame and triggers appropriate placement policies to reduce the rate of rejection. With *replication on demand*, the RS meta-actor decides to create a new replica of the MM object on one of the sources on the fly. The source DS on which the new replica is to be made can be determined to be the one that has minimum load-factor with respect to R, i.e., with the minimum value of $LF(R, DS)$. By doing so, the QoS broker attempts to maximize the possibility of servicing additional requests from the same replica. In order for this approach to be feasible and attractive, the replication must proceed at a very high rate, thereby consuming vital server resources for replication.

Placement of MM Objects: The predictive placement meta-actor(PP) implements a placement policy that determines in advance when, where, for how long, and how many replicas of each MM object should be placed in a MM server so as to maximize the number of requests that can be serviced by it. Predictive placement can be initiated periodically. The periodicity can be varied and the frequency with which the predictive placement process is executed governs its effectiveness. Smaller the time period, greater the computational overheads and greater the difficulty in predicting the request rates more accurately. Infrequent execution of predictive placement may result in under-utilization of the MM server's resources. The two main tasks of the predictive placement module are replication of MM objects from one data source to another, or from tertiary storage to secondary storage, and dereplication of MM objects from the data sources. The placement module must determine how many replicas of each MM object are necessary, and which of the data sources these replicas should be allocated to. The number of replicas possible for each MM object is dependent on the data source(s) to which the replicas are allocated, and upon the degree of loading of the data sources. In the process of determining the number and location of replicas that are necessary, the predictive placement module has to derive an allocation of MM objects to data sources, as well as a tentative assignment of expected requests to data sources; we term this the *pseudo-allocation* procedure.

Placement mechanisms must be designed to work effectively with request scheduling. The goal of the pseudo-allocation procedure is to facilitate the task of the adaptive scheduler module, by allocating MM objects in such a way as to maximize system-wide revenue, by permitting a maximum number of requests to be admitted and scheduled for service. In order for predictive placement to execute concurrently with the adaptive scheduling process, a current snapshot of the system is taken prior to initiating predictive placement to provide a consistent view of the system state. The predictive placement module does not consider the exact times at which requests may arrive; the adaptive scheduling module makes assignment decisions based on the exact arrival times of requests. The pseudo-allocation procedure only determines when and to where replication must be initiated. The predictive placement module then has to figure out at what rate the replication should proceed. Replication may be initiated at the maximum feasible replication rate and this rate may even be changed dynamically, depending on the instantaneous load on the data sources and the number of replications scheduled to be performed from the same data source In [**?**,**?**], we propose a greedy matrix algorithm to implement the pseudo-allocation process for predictive placement.

Optimizing the Load Management Policies: We discuss two possible optimizations that can be used to further enhance resource utilization in a distributed MM server - *eager replication* and *lazy dereplication*. Eager replication is a technique to make use of idle resources effectively to reduce overhead and provide good performance during periods of high demand. The eager replication mechanism creates additional replicas of popular MM objects so that they are readily available when needed. Eager replication is assigned lowest priority compared to the other load management tasks in the MM server and is initiated only during periods when it is not likely to impact MM server performance and throughput. Eager replication can potentially not only enhance MM server utilization and throughput, but it can also lower the latency between reception of a subscriber request and servicing of that request.

In the *lazy dereplication* strategy, when a MM object MM_i is dereplicated, the storage resources that were being taken up by MM_i are released and marked as available for other objects. However, the disk blocks that were being used for MM_i are rewritten only if there is an immediate need to reuse these blocks for storage of some other object. In the interim period, between the time dereplication is initiated and the time when the disk blocks of MM_i are overwritten, MM object MM_i exists on the data source and can be reclaimed if so desired.

4 Specifying and Reasoning about QoS-Based MM Services

Assuring safe composability of resource management services is essential for efficient management of distributed systems with widely varying and dynamically changing requirements. To analyze designs, clarify assumptions that must be met for correct operation, and establish criteria for non-interference, it is important to have a rigorous semantic model of the system: the resources, the management processes, the application activities, and the sharing and interactions among these.

In this section we briefly and mostly informally describe how to model the multimedia meta-architecture and resource management policies presented above in the TLAM framework. We state the key theorems relating QoS requirements to the QoS broker architecture and discuss reasoning about correctness and non-interference. More detail can be found in [**?**,**?**]. Properties of a system modeled in the TLAM are specified as properties of computation paths. A property can be a simple invariant that must hold for all configurations of a path, a requirement that a configuration satisfying some condition eventually arise, or a requirement involving the transitions themselves. Such properties are checked using the properties of the building blocks for configurations – message contents and actor state descriptions – and of the TLAM reaction rules that determine the behavior of actors in the system.

To set the stage we informally describe the notion of a system providing QoS-based MM Service ($MMService_{QoS}$). We then map QoS requirements to resource requirements and focus on modeling and reasoning about the resource management underlying a QoS-based service. We define the notions of a *Resource-based MM Service* ($MMService_{Resource}$) and of a system having *Resource Based MM Behavior* ($MMBehavior$). $MMService_{Resource}$ reflects the chosen system resource architecture and allows us to reason about the availability and use of resources. $MMBehavior$ reflects the QoS broker software architecture and models the resource management algorithms as specific meta actor behaviors. Such a behavior specification can serve as a first stage in refining a service specification into an implementation. The main result is that a

MM system meeting the Resource Based MM Behavior specification provides Resource-based MM Service, which in turn implies (under the assumptions of the mapping from QoS requirements to Resource requirements) that it provides QoS based MM Service.

We assume that there is a fixed set $MMObjects$ of MM objects available in the system and let MM range over $MMObjects$. A MM request message specifies a triple $(\alpha_{cl}, \langle MM_0, \ldots, MM_n \rangle, QoS)$ which is interpreted as a request to initiate a MM streaming service from the server receiving the request, to the client actor α_{cl}, using the MM objects $\langle MM_0, \ldots, MM_n \rangle$, and obeying the QoS requirement QoS. To simplify the presentation, we shall assume that there is only one MM object involved in each request.

Definition 1 ($MMService_{QoS}$) A system S provides a QoS-based MM Service over the set of MM objects, $MMObjects$, iff for a configuration $C \in S$, if there is an undelivered message $MMreq$, then along any path π from C there are configurations such that one and only one of the following properties hold: (1) the request is accepted for service and service is provided with the required QoS until complete, or (2) the request is rejected, for this to happen it must be the case that the requested QoS cannot be provided at the time the request arrives.

4.1 Resource Based MM Service

We assume given a function $QoSTranslate$ which maps MM requests to resource requirements which, if met, will ensure the requested QoS (See [**?**] for examples of such QoS translation functions). The function $QoSTranslate$ maps MM requests to 4-tuples representing resource allocation requirements for the four managed resources: network bandwidth ($NetBW$), CPU cycles (CPU), disk bandwidth ($DiskBW$), and memory buffer ($BufMem$).

MM nodes (data source, distribution controller, and client) are modeled as TLAM nodes. We assume given a function $Capacity$ such that $Capacity(DS, Res) \in Unit_{Res}$ for any data source node DS and resource Res. To provide an abstraction of the admission and placement processes, we introduce functions characterizing the state of replicas on each data source node and functions characterizing the state of each request that has arrived. For example, $ReplClass(C, DS, MM)$ is the replication class of the multimedia

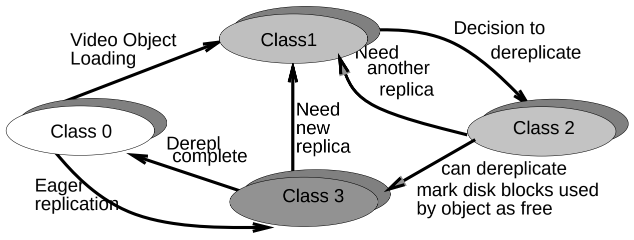

Fig. 5. State Transition Diagram. $Class_0$ indicates that the replica is not present on the node. A $Class_1$ replica is guaranteed to be available. A $Class_2$ replica is marked as dereplicable but remains available until all requests assigned to it have completed. A $Class_3$ replica exists on the node in the sense that it has not been overwritten, but there is no guarantee it will remain that way and can not be considered available until its class is changed. The transitions specify the possible phase changes in the life of a MM object.

object MM on node DS. The constraints on this function are given by the class transition diagram in Figure **??**. Each request is uniquely associated to a base-level request actor. This relies on the uniqueness of messages and of newly created actors in the TLAM model. $ReqState(C, \alpha^{req})$ is the request status of request actor α^{req}. When initially created a request actor is in state $WaitForAdm$. As the system progresses, the value of this function will eventually change from $WaitForAdm$ to $admGranted$ or $admDenied$. From $admGranted$ it moves to $Servicing$ and then to $Completed$. After the value reaches $Completed$ or $admDenied$ it remains constant. The functions $ReqObjId(C, \alpha^{req})$ and $ReqReplica(C, \alpha^{req})$ identify the replica to which the request has been assigned (the MM object and the DS node containing the assigned replica). $ReqQoS(C, \alpha^{req})$ is the tuple of resources assigned to the request. The *total resource property* $\phi_{res}^{total}(S)$ states that for every configuration in the system and every data source node the sum of the resources allocated to the requests on the node will not exceed the total capacity of resources on the node.

Using the characterizing functions and the corresponding constraints on their values, we define a Resource-based MM service as follows.

Definition 2 ($MMService_{Resource}$) A system S providesResource-based MM service with respect to functions $QoSTranslate$, $Capacity$, and the functions characterizing replica and request state as specified above iff S obeys the constraints given for the replica and resource functions, $\phi_{res}^{total}(S)$ holds, and for $C \in S$, if there is an undelivered message, $MMreq = (\alpha_{cl}, MM, QoS)$, then along any computation path π from C there is one of the following segments:

(D) $C_{\text{start}} \xrightarrow{+} C_{\text{deny}}$

(G) $C_{\text{start}} \xrightarrow{+} C_{\text{grant}} \xrightarrow{+} C_{\text{serve}} \xrightarrow{+} C_{\text{complete}}$

such that

1. C_{start} is the result of delivery of $MMreq$ and there is a new request actor α^{req} such that: $ReqClientId(C_{\text{start}}, \alpha^{req}) = \alpha_{cl}$, $ReqObjId(C_{\text{start}}, \alpha^{req}) = MM$, $ReqQoS(C_{\text{start}}, \alpha^{req})_{Res}$ is greater than $QoSTranslate(QoS)_{Res}$ for each Res, and $ReqState(C_{\text{start}}, \alpha^{req}) = WaitForAdm$.
2. $ReqState(C_{\text{deny}}, \alpha^{req}) = admDenied$, and there is a message to α_{cl} notifying of rejection of $MMreq$. In this case the system had insufficient resources to schedule $MMreq$ in C_{start}.
3. $ReqState(C_{\text{grant}}, \alpha^{req}) = admGranted$, and $ReqReplica(C_{\text{grant}}, \alpha^{req}) = DS$ for some DS node such that $ReplClass(C_{\text{grant}}, DS, MM) = Class_1$.
4. $ReqState(C_{\text{serve}}, \alpha^{req}) = Servicing$ and $ReqState(C_{\text{complete}}, \alpha^{req}) = Completed$ (allocated resources remain allocated during servicing).

Theorem 1 (QoS2Resource) If a system S provides $MMService_{Resource}$, and $QoSTranslate$ satisfies the stated requirements, then the system provides $MMService_{QoS}$.

4.2 QoS Broker MM Behavior

A system with MM behavior has meta actors: QB (a QoS Broker), RS (a Request Scheduler), ROD (Replication-On-Demand), DR (Dereplication), PP (Predictive Placement), located on the DC node, and a data source manager $DSma(DS)$ on each data source node DS. Information about resource capacity, replication, and request allocation is distributed on the data source nodes. The replica of an MM object MM on data source node DS, if present, is represented by a replica base actor $NodeReplica(C, DS, MM)$ and the request and replica state information is kept in annotations of the base actors state.

The QoS broker maintains, as part of its current state beh^{qb}, a model $MMstate(beh^{\text{qb}})$ of the distributed MM state which is used by the resource managers to make decisions. There are analogs of the configuration based functions characterizing request and replicas in which QoS broker state plays the role of configuration. To support more sophisticated resource management processes, the QoS broker maintains a current prediction/statistics model, $Pmodel(beh^{\text{qb}})$, and knowledge of what management processes are ongoing at any time.

The dynamic behavior of the QoS meta actors is given by TLAM rules specifying the effects of reaction to events and messages: updating the actors state, creating new actors, sending messages, modifying base-level annotations. There are QoS broker rules for receiving requests, invoking resource management processes (request scheduling, predictive placement, dereplication, ...), and updating the broker model in response to reports from these processes. In addition there are rules specifying the behavior of DS node meta actors that manage the replicas and local request actors.

All such broker processes are required to terminate and to accurately report effects to QB. DR can only change the class of replicas and can only change class 1 to class 2. ROD and PP can change change class annotations and initiate replication (changing replication status and bandwidth). They can only change a replicas class from 0, 2, or 3 to 1. Processes that can change resource allocation are required to maintain the total resource invariant of the model they are provided with. A replication in progress must complete and we require that request servicing is a finite process.

We assume that only the MM meta actors set or modify the MM annotations of replica and request base actors. The QoS broker must coordinate the management activities sufficiently to assure that the model used is accurate. Thus we require that in any configuration, the QOS brokers model agree with the distributed state, modulo the effects of delivery of pending update notifications from the managers.

Theorem 2 If a system S has QoSBroker MM behavior as specified above, then S provides $MMService_{Resource}$.

The shared information used and modified by the resource managers— resources allocated to requests, replication status and replica class— is distributed on the DS nodes as base actor annotations. Interleaving of incremental modifications keeps the QoS brokers model sufficiently accurate to prevent interference among these processes. In fact $\phi^{total}_{res}(S)$ and the class transition constraints provide simple criteria for admitting other management processes in the interleaved setting.

A rigorous framework such as we have outlined is crucial in analyzing and preventing interference between concurrent activities. As an example of problems that arise, we might imagine that DR and RS could be allowed to proceed concurrently. An attempt to establish that the system invariants are maintained reveals the following problem. Suppose MM on DS is $Class_1$, RS is invoked with request α^{req} requiring a copy of MM and concurrently DR is invoked. Then we have the following possible scenario: DR decides to mark MM on DS as $Class_2$ and RS decides to allocate MM on DS to α^{req}; the notification from DR reaches $DSma(DS)$ first and all requests previously allocated to MM complete so MM becomes $Class_3$; then the notification from RS arrives and now we have a request allocated to a $Class_3$ replica which can be overwritten. Protocols and annotations that support increased concurrency of MM resource management are the topic of current investigations.

5 Related Work and Future Research Directions

A preliminary implementation of the QoS broker and its associated components is discussed in [**?**]. Using the two level multimedia metaarchitecture described in this paper, a QoS-enabled customizable middleware framework, called **CompOSE|Q** is currently being developed at the University of California, Irvine [**?**]. **CompOSE|Q** includes modules that implement the 3 basic composable core services with specific interface definitions and interaction constraints built in, common services built out of the core services - object migration, distributed snapshot, directory services, scheduling etc. The **CompOSE|Q** environment consists of the following components: (1) a programming environment based on concurrent objects (2) a middleware library that manages execution of applications on nodes, coordinates distribution and provides services. (3) a compact runtime component that resides on the nodes of the distributed system which interacts with the distributed component of the middleware.

Commercially available object-based middleware infrastructures such as CORBA and DCOM represent a step toward compositional software architectures but do not support the development and maintenance of large applications. Specifically, they do not deal with interactions of multiple object services executing at the same time, or the implication of composing object services. For instance, the Electra framework [**?**] extends CORBA to provide support for fault tolerance using group-communication facilities and protocols like reliable multicast. Architectures that provide real-time extensions to CORBA [**?**,**?**] necessary to support timing-based QoS requirements [**?**] have been proposed. TAO is a framework that supports real-time CORBA extensions to provide end-to-end QoS; it has been used to study performance optimizations [**?**], and patterns for extensible middleware [**?**]. Similarly, real-time method invocations have been explored by transmitting timing information in CORBA data structures [**?**].

The Java Development Environment is a distributed object implementation framework that provides mobility; however the semantics of interaction with other customizations is dependent on the implementation.

The ability to deal with the management of thread priorities for real-time thread management is dependent on the underlying threads implementation, making QoS support complicated to achieve. Various systems such as the Infospheres Infrastructure [**?**] and the Globe System [**?**] explore the construction of large scale distributed systems using the paradigm of distributed objects. Globus, a metacomputing framework for networked virtual machines defines a QoS component called Qualis [**?**] where low level QoS mechanisms can be integrated and tested. *Reflection* allows application objects to customize the system behavior [**?**] as in Apertos [**?**] and 2K [**?**]. The Aspect Oriented Programming paradigm [**?**] makes it possible to express programs where design decisions can be appropriately isolated permitting composition and re-use. Some of the more recent research on actors has focused on coordination structures and meta-architectures [**?**] and runtime systems such as Broadway [**?**] and the Actor Foundry [**?**].

Multimedia QoS enforcement has been a topic of extensive research. The Omega architecture [**?**] developed end-to-end real-time communication protocols and QoS brokers at the endpoints to supply end-to-end QoS. *QualMan* [**?**] is a QoS aware resource management platform which contains a set of resource brokers that provides negotiation, admission and reservation capabilities for sharing end-system resources such as CPU, memory and network bandwidth. Work on resource management mechanisms for multimedia servers has focussed on placement of media on disk to ensure real-time retrieval [**?**], admission control procedures to maximize server throughput [**?**], replication and striping strategies for optimizing storage across disk arrays [**?**], batching mechanisms that group closely spaced requests for the same objects [**?**], load balancing mechanisms for effective utilization [**?**,**?**].

Much of the work on formal models for QoS has been in the context of QoS specification mechanisms and constructs. In some implementation driven methods of QoS specification, the specification of QoS requirements is intermixed with the service specification [**?**,**?**]. Other approaches address formal description methodologies based on synchronous communication for specifying QoS via dual language techniques that specify functional behavior and QoS constraints distinctly using two different languages [**?**]. *add more current work reference on multiparadigm representation, aspect oriented representation etc.* Synchronizers [**?**] allow us to express QoS constraints via coordination constraints in the actor model either as local synchronization constraints or multi-actor coordination constraints. Real-time constraints are described by synchronization code between the interfaces of actors [**?**] using a high-level programming language construct called *RTsynchronizer*. A number of language independent formalisms have been developed for specifying and reasoning about concurrent systems such as Unity [**?**] and I/O automata [**?**]. The TLAM approach models runtime services and the application using a single framework and uses the framework to reason about interactions between application actors, between metalevel services as well as ensure the correct behavior of base-meta interactions.

We are actively working on extending the existing meta-architecture to support more services. Modeling client interaction requires a notion of session and resources within a session. Further work is required to provide a generalized model that captures the architectural resources required in the server and network to support the session connection. *Dynamic negotiation protocols* involves negotiation of resources on the fly to degrade QoS of ongoing requests in order to admit more requests. In addition to providing a clear model of negotiation, this requires developing a mechanism to allow requests to decide when and how to renegotiate. For end-to-end QoS, it is necessary to determine how *real-time scheduling* strategies for time constrained task management interact with strategies for other tasks such as CPU intensive calculations, or network communication with clients. Also, further work is required in order to model the request migration service in the meta-architecture and develop strategies for its effective use.

In general, the dynamic nature of applications such as those of multimedia under varying network conditions, request traffic, etc. imply that resource management policies must be dynamic and customizable. Current mechanisms, which allow arbitrary objects to be plugged together, are not sufficient to capture the richness of interactions between resource managers and application components. For example, they do not allow customization of execution protocols for scheduling, replication, etc. This implies that the components must be redefined to incorporate the different protocols representing such interaction. We believe that a cleanly defined meta-architecture which supports customization and composition of protocols and services is needed to support the flexible use of component based software.