A Formal Model for Reasoning About Adaptive QoS-Enabled Middleware

NALINI VENKATASUBRAMANIAN
Department of Computer Science, University of California, Irvine
CAROLYN TALCOTT
SRI International
and
GUL A. AGHA
Department of Computer Science, University of Illinois at Urbana-Champaign

Systems that provide distributed multimedia services are subject to constant evolution; customizable middleware is required to effectively manage this change. Middleware services for resource management execute concurrently with each other, and with application activities, and can, therefore, potentially interfere with each other. To ensure cost-effective QoS in distributed multimedia systems, safe composability of resource management services is essential. In this article, we present a meta-architectural framework, the Two-Level Actor Model (TLAM) for customizable QoS-based middleware, based on the actor model of concurrent active objects. 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 particular, we show a system in which the multimedia actor behaviors satisfy the specified requirements and provide the required multimedia service. The behavior specification leaves open the possibility of a variety of algorithms for resource management. Furthermore, constraints are identified that are sufficient to guarantee noninterference among the multiple broker resource management services, as well as providing guidelines for the safe composition of additional services.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming*; D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; *correctness proofs*; D.2.12 [**Software Engineering**]: Interoperability—*distributed objects*

General Terms: Algorithms, Verification, Theory, Performance

This research was supported by funding from an NSF Career Award ANI-9875988 and from an ONR MURI grant NO0014-02-1-0715.

Authors' addresses: N. Venkatasubramanian, Dept. of Information and Computer Science, ICS 464D, University of California, Irvine, Irvine, CA 92697; email: nalini@ics.uci.edu; C. Talcott, SRI International, Room EK274, 333 Ravenswood Avenue, Menlo Park, CA 94025; email: cit@csl.sri.com; G. Agha, University of Illinois at Urbana-Champaign, Department of Computer Science, 1304 West Springfield Avenue, MC 258, Urbana, IL 61801; email: agha@ca.uiuc.edu.

Additional Key Words and Phrases: Middleware services, reflection, theoretical foundations, meta-object models, actors, multimedia, quality-of-service

1. INTRODUCTION

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 [Buddhikot and Parulkar 1995] to form a wide-area infrastructure. Applications such as telemedicine, distance learning, and electronic commerce have varying requirements such as timeliness, security, reliability and availability.

Systems that provide distributed multimedia services are continuously changing and evolving. For instance, the set of servers, clients, user requirements, and network and system conditions in a wide-area infrastructure are changing continuously. Many MM applications can tolerate minor, infrequent, violations of their performance requirements, specified as a quality-of-service (QoS) parameter, for for example, tolerable jitter in a video frame. Future applications will require dynamic invocation and revocation of services distributed in the network without violating QoS constraints of ongoing applications. 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. Customizable middleware allows us to deal with changes in systems and applications in a nonintrusive way.

To assure safe adaptation to dynamically changing requirements, it is important to have a rigorous semantic model of the system: the resources, the middleware that provides system management, the application activities, and the sharing and interactions among these elements. Using such a model, designs can be analyzed to clarify assumptions that must be met for correct operation, and to establish criteria for noninterference. In Venkatasubramanian and Talcott [1995] and Venkatasubramanian [1998], we presented the TLAM (Two-Level Actor Machine) semantic framework for specifying, composing, and reasoning about resource management services in open distributed systems.

In this article, we use our framework to develop a model for customizable, cost-effective middleware to enforce QoS requirements in multimedia applications. Specifically, we deal with the modeling of high-level behaviors that describe the distributed middleware components and reason about the noninterference requirements of modules implementing these behaviors. The sample application used to drive the specification and reasoning process is a video-on-demand application [Venkatasubramanian and Ramanathan 1997], using a commercial VOD architecture as a basis. Key middleware services for QoS-based resource provisioning include request scheduling, data placement

(replication), and dereplication. For scheduling MM requests, we use an adaptive scheduling policy that compares the relative utilization of resources in a multimedia server to determine an assignment of requests to replicas. We discuss 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. To optimize storage utilization, we introduce methods for dereplication of MM objects based on changes in their popularity and in server usage patterns. While the sample application used to drive the specification and reasoning process is a video-on-demand application [Venkatasubramanian and Ramanathan 1997], the architectural specification can be used to describe any multimedia delivery application where the underlying system supports resource reservation.[1]

1.1 The Two-Level Actor Framework

The TLAM is based on the actor computation model [Hewitt et al. 1973; Baker and Hewitt 1977; Agha 1986], a model of concurrent active objects that has a built-in notion of encapsulation of state and control. Traditional passive objects encapsulate state and a set of procedures that manipulate the state; an actor (active object) encapsulates a thread of control as well. Each actor potentially executes in parallel with other actors and interacts only by asynchronous message-passing. Each actor has a unique name (mail address) and a mailbox to buffer incoming messages. Actors compute by serially executing the messages queued in their mailboxes. As a general model of concurrency, the actor model can be used to represent and build various procedural and functional architectural components at both the application and system-level. A variety of coordination and interaction mechanisms can be modeled using actors, including RPC, messaging, transactions, and other forms of object synchronization [Hewitt 1977; Frølund 1996]. In the TLAM, the actor model is used to represent the application behavior: interactions between applications and middleware services as well as interactions between the middleware services themselves. Elsewhere, the actor model has been used to model architectural abstractions such as components and connectors [Astley 1999; Astley and Agha 1998], coordination constructs [Frølund 1996], and real-time constraints [Ren 1997].

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 run-time system which manages system resources and controls the run-time behavior of the base level. Specifically, metalevel actors are used to model and represent the resource management (middleware) functionality. The TLAM framework provides an abstract characterization of actor identity, state, messages, and computation, and of the connection between base-level and metalevel computation. Meta-actors communicate with each other via message-passing as do

[1]Resource reservation ensures that once resources are allocated to an admitted request, they will remain allocated until the broker decides otherwise, or the request completes. We specifically do not deal with applications that involve updates of multimedia data on the fly (i.e., read/write data) and the concurrency control issues therein.

base-actors, and meta-actors may also examine and modify the state of the base-actors located on the same node. Base-actors and messages have associated run-time *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. The behavior of base-actors and meta-actors are specified by local reaction rules describing the result of receiving a message or event notification.

In the development of the TLAM we have restricted attention to a two-level system rather than modeling a reflective tower (n-level system) in order to simplify the model and focus on the key relation between base-actors and meta-actors. In addition the two-level system is a good match for the middleware services that we wanted to study. A fundamental problem in reasoning about metalevel services and their composition is managing the complexity of base-meta-level interactions (in the context of already complex interactions in distributed systems), and to develop principles to ensure noninterference between metalevel services. Part of the TLAM development has been to identify key system services where nontrivial interactions between the application and system occur, that is, base-meta interactions. We refer to these key services as *core services* and represent them as TLAM metalevel entities. As a starting point, we have identified three core services:

—Creation of services/data at a remote site—*remote creation*,
—Capturing information at multiple nodes/sites—*distributed snapshot*,
—Interactions with a global repository—*directory services*.

Higher-level services can be built using these services to isolate potentially conflicting interactions with the base level. Principles for noninterfering composition of these services have been developed to aviod such conflicts.

Our general approach to modeling middleware components is to develop a family of specifications from different points of view and at different levels of abstraction. From a high-level point of view, we specify the end-to-end service provided by a system in response to a request. This can be refined by expressing system-wide properties in terms of abstract properties of the underlying network. From a low-level point of view, we specify constraints on the behavior and distribution of a group of actors. This local behavior point of view can be further refined by specifying protocols and algorithms for the actions of individual actors. The local behavior and system-wide points of view are related by the notion of a group of meta-actors providing a service in a system satisfying suitable initialization and noninterference conditions. The staging and refinement of specifications provides a form of modularity, scalability, and reusability. It reduces the task of implementation to that of implementing individual abstract behaviors. Behavior-level specifications can be used to guide or check implementations or even serve as executable prototypes.

As multimedia applications begin to execute in environments that are increasingly distributed and mobile, middleware services to support these applications become increasingly important. As dependence on middleware services increases, noninterfering execution of the services are of paramount

importance. To enable a system designer to reason about the overall correctness of the system, it becomes more important to have clear semantic models and be able to carry out a variety of analyses based on these models in order to increase assurance of correct and expected behavior.

System specifications such as the one presented here can serve as a valuable form of documentation of requirements, design, and implementation decisions. The relations between viewpoint provide a systematic and rigorous mechanism to relate system requirements and the design/implementation descriptions. Having this form of multiple-view specification and documentation can be used in configuration management to help isolate effects of change, to propagate effects of change, and to reason about the effects of adaptation on application behavior. Furthermore, such a methodology for the integration of middleware services encourages modular specification of middleware services, resulting in effective reuse of both the middleware services and the composability reasoning .

Prior work has addressed the development of the TLAM model, specification and reasoning about other resource management services in the TLAM style, and the implementation of a reflective middleware framework [Venkatasubramanian et al. 2001] using this architectural model. Previous major case studies carried out using the TLAM framework include: (1) distributed garbage collection [Venkatasubramanian et al. 1992; Venkatasubramanian 1992]; (2) composition of migration and reachability services [Venkatasubramanian and Talcott 1995, 2001a]; (3) a rudimentary logging service [Venkatasubramanian and Talcott 2001b]; (4) a service replication framework [Venkatasubramanian and Talcott 1993]. This article presents, in some detail, the application of the TLAM methodology to formally specify and reason about QoS-based resource management for multimedia servers.

Based on the two-level architecture, we are developing a customizable and safe distributed systems middleware infrastructure, called **CompOSE|Q** (Composable Open Software Environment with QoS) [Venkatasubramanian 1999], at the University of California, Irvine, that has the ability to provide cost-effective and safe QoS-based distributed resource management. The semantic model is used to verify safe interaction of the implemented mechanisms. Details of the mechanisms implemented and the performance evaluation of the composite environment are described in Venkatasubramanian and Ramanathan [1997] and Venkatasubramanian [1998].

1.2 Contributions of the Article

Our TLAM semantic framework [Venkatasubramanian and Talcott 1995; Venkatasubramanian 1998] provides a basis for modular reasoning about properties of resource management algorithms and their composition. We begin by informally describing the notion of a system providing QoS-based MM Service. We then map QoS requirements to resource requirements and focus on modeling and reasoning about the resource management underlying a QoS-based service. For this purpose, we define, in a rigorous manner, the notions of a system providing Resource-based MM Service, of a system having Resource-based MM

Behavior, and, finally, refining the system with an Adaptive Request Scheduling Policy. The Resource-based MM Service specification reflects the chosen physical system resource architecture and allows us to reason about the availability and use of resources. The Resource-based MM Behavior specification models the QoS broker software architecture presented in Venkatasubramanian [1998] and places constraints on the actions of the QoS meta-actors. Such a behavior specification can serve as a first stage in refining a service specification into an implementation. The Adaptive Request Scheduling Policy illustrates such refinement. It specifies one of the resource management policies developed in Venkatasubramanian and Ramanathan [1997] and Venkatasubramanian [1998] and implemented in Venkatasubramanian [1999] by giving high-level algorithms for determining meta-actor behavior. The main results are:

(1) if a system provides Resource-based MM Service, then (under the assumptions on the mapping from QoS requirements to Resource requirements) it provides QoS-based MM Service;
(2) if a system has Resource-based MM Behavior, and meets certain initialization and noninterference conditions, it provides Resource-based MM Service;
(3) if a system is refined with specific policies for QoS-based resource management, for example, the Adaptive Request Scheduling Policy, then the system implements Resource-based MM Behavior.

A consequence of (2) is that new broker policies can be safely installed as long as they satisfy the behavior constraints. (3) is an example of such a policy.

The rest of this article is organized as follows. Section 2 describes a multimedia server: its physical configuration and software architecture, based on the TLAM two-level model, and a mapping of the software architecture onto the physical architecture. Section 3 describes several resource management policies that we have used in realizing the software architecture. Section 4 gives a brief summary of the TLAM semantic framework, covering concepts and notations needed to understand the formal model of the QoS Broker. Section 5 shows how we use the TLAM framework to formally model and reason about systems such as QoS Broker. Section 6 discusses related work. We conclude in Section 7 with future research directions.

2. A META-ARCHITECTURE FOR QOS-BASED SERVICES

Using the TLAM framework, we develop a meta-architectural model of a multimedia server that provides QoS-based services to applications. The physical architecture of the MM server corresponds to the underlying network layer of the TLAM (see Figure 1). We chose a commercial scalable distributed video server as our design platform [Thapar and Koerner 1994; Venkatasubramanian and Ramanathan 1997]. It consists of:

—a set of data sources (DS) that provide high bandwidth streaming MM access to multiple client nodes outside the server complex (distributed across a wide-area network). Each independent data source includes high capacity

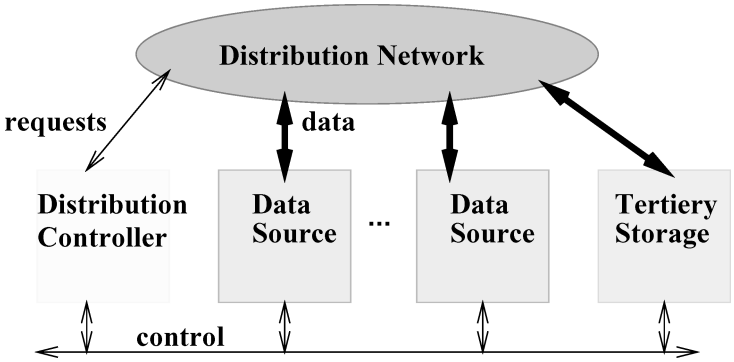

Fig. 1. Physical architecture of the QoS brokerage service mapped.

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.

—a tertiary storage server that contains the passive MM objects (e.g., read-only video data). Replicas of these MM objects are placed on secondary storage (disk subsystems) 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. Note that in this architecture, scheduling a request on a data source implies that the request is serviced using storage on that data source. The scalability of this architecture is attributable to the ability to add additional data sources to the server in order to provide increased storage capacity and transfer bandwidth. The rationale for nonshared storage subsystems across servers is that management mechanisms will scale to server complexes distributed over a wide-area network easily. In contrast, other architectures [Dan et al. 1995; Dan and Sitaram 1995] assume the availability of shared storage among the servers. Alternative nonlocal storage architectures, such as the SAN (Storage Area Networks) and NAS (Network Attached Storage) models, are also possible. While the specifics of the proposed policies and their performance may vary depending on the architectural configuration, our specification and reasoning techniques apply equally well to these alternate architectures. For a detailed description of design choices, see Thapar and Koerner [1994] and Venkatasubramanian and Ramanathan [1997].

The software architecture of a multimedia server consists of two subsystems—the base level and metalevel subsystems corresponding to the application and system-level resource 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 metalevel component deals

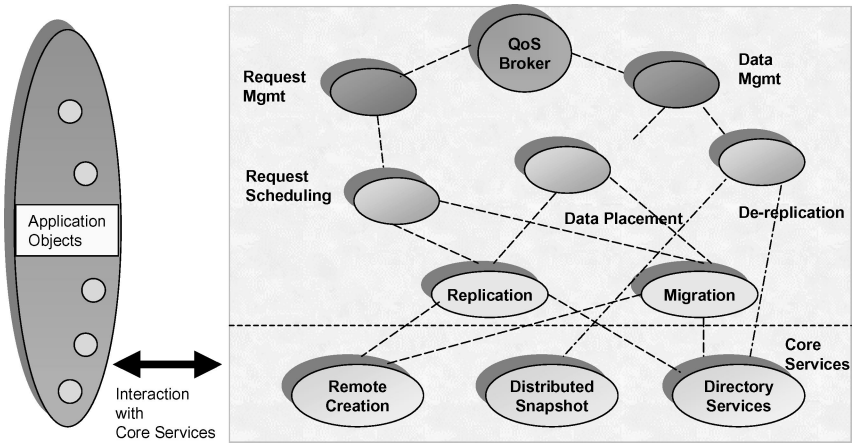

Fig. 2. Detailed architecture of the QoS meta-architecture system. The dotted arrows indicate the possible flow of an incoming request through the different middleware modules.

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 metalevel entity, the *QoS Broker* meta-actor, *QB*.

The two main functions of the QoS Broker are data management and request management. The multimedia data and request management functions of the QoS broker, in turn, use a number of simpler services. The organization of metalevel services provided by a QoS broker is shown in Figure 2. We now describe some of the main metalevel services and the supporting services in the above QoS broker system and discuss some of the issues that must be considered in establishing their correctness, when operating possibly concurrently with other services.

The *data management* component decides the placement of data in the distributed system, that is, it decides when and where to create additional replicas of data based on various factors, for example, popularity of a specific video object. Such replication ensures that heavily accessed information is available at multiple data sources.[2] It also determines when additional replicas of data actors are no longer needed and can be garbage-collected/dereplicated. The *request management* component performs the task of admission control for incoming requests and ensures the satisfaction of QoS constraints for requests that are ongoing in the system. We have developed adaptive admission control mechanisms [Venkatasubramanian and Ramanathan 1997] in the *request scheduling* module that assigns requests to servers and ensures cost-effective utilization of resources. The data and request management functions, in turn, require basic services such as replication, dereplication, and migration. In this article, we restrict attention to replication and dereplication service.

[2]One may also use the replicas to provide fault tolerance in the event of a DS failure, although failures are not considered in this article.

—*Replication.* To replicate data and request actors using adaptive and predictive techniques for selecting where, when, and how fast replication should proceed. The rate at which replication proceeds also has a direct impact on system performance and application interactivity (QoS).

—*Dereplication.* To dereplicate/garbage-collect data or request actors and optimize utilization of storage space in the distributed system, based on current load in the system as well as expected future demands for the object. 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.

In order to map the QoS software 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, including the QoS broker and associated meta-actors, reside on the distribution controller node. The node local components include, for each DS node:

—a DS meta-actor for resource and request servicing 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, replication processes and so forth.

—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. RESOURCE MANAGEMENT POLICIES FOR MM SERVERS

Apart from the QoS broker, *QB*, the MM system contains a number of meta-actors whose behaviors are coordinated by *QB* and combine to provide the resource management services discussed above. In this section, we describe some of the load management policies that have been treated in the formal model. The policies are implemented as metalevel actors and 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, is translated into 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:

$$LF(R, DS) = max\left(\frac{DiskBW^R}{DiskBW^{DS}}, \frac{BufMem^R}{BufMem^{DS}}, \frac{CPU^R}{CPU^{DS}}, \frac{NetBW^R}{NetBW^{DS}}\right),$$

where $DiskBW^R$, $BufMem^R$, CPU^R, and $NetBW^R$ denote the disk bandwidth, memory buffer space, CPU cycles, and network transfer bandwidth,

respectively, that are necessary for supporting request R and similarly Res^{DS} denotes the amount of resource Res available on data source DS. The load factor helps identify the critical resource in a data source, that is, the resource that limits the capability of the data source to service additional requests.[3] By comparing the load factor values for different servers, load management decisions can be taken by the QoS brokerage service. We briefly describe mechanisms for the scheduling of MM requests and placement of MM data. Optimizations to the basic mechanisms are outside the scope of this article and are presented elsewhere [Venkatasubramanian and Ramanathan 1997; Venkatasubramanian 1998].

3.1 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 is impacted the least (i.e., the data source with the least 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 meta-actor *RS* can either reject the incoming request or initiate *replication on demand*—implemented via a replication on demand meta-actor (*ROD*). The replication on demand meta-actor *ROD* attempts to create a new replica of a requested MM object on the fly. The source *DS* on which the new replica is to be made is one that has minimum load-factor with respect to the request, R, that is, with the minimum value of $LF(R, DS)$. By doing so, *ROD* attempts to maximize the possibility of the QoS broker 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.

3.2 Placement of MM Objects

The QoS broker analyzes the rejections over time and triggers appropriate placement policies, implemented via predictive placement and dereplication meta-actors (*PP* and *DR*) to reduce the rate of rejection. The predictive placement and dereplication meta-actors (*PP* and *DR*) implement a placement policy that determines in advance when, where and how many replicas of each MM object should be placed in a MM server, and when to dereplicate an existing replica. In particular, the goal of the predictive placement procedure is to facilitate the task of the adaptive scheduler meta-actor, 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 principle, the term revenue can be used to mean overall system throughput in terms of the

[3]Note that diskspace usage on a data source for a MM object does not change with increasing number of requests to that MM object—a single copy on disk is sufficient; hence, diskspace is not accounted for in the load-factor formulation.

number of requests serviced. In the studies presented, we associate a revenue generated (to the service provider) by each incoming request. The placement algorithm being used is designed to maximize revenue to the service provider. In the case that all requests are identical in terms of revenue, the placement problem reduces to one of maximizing throughput. Placement mechanisms must be designed to work effectively with request scheduling so that maximum system throughput can be achieved. In order for predictive placement to execute concurrently with the adaptive scheduling process, a current snapshot of the system is given to the placement processes to provide a consistent view of the system state. The predictive placement and dereplication meta-actors do not consider the exact times at which requests may arrive; the adaptive scheduling meta-actor makes assignment decisions based on the exact arrival times of requests.

We also propose optimizations to the basic adaptive and predictive phases—eager replication and lazy dereplication. With eager replication, replication of video objects occurs in anticipation when there are idle resources, even though the demand for these objects may not be apparent immediately (e.g., replication of video objects during non peak-hours). Video objects slated for eager replication may be chosen as part of the predictive placement procedure, but may actually be replicated only if sufficient video server resources become available. In the lazy dereplication strategy, when a MM object MM_i is dereplicated, MM_i's storage resources are released and marked as being available. 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_i exists on the data source and can be reclaimed if so desired. The implementation of eager replication and lazy dereplication is supported by appropriate classification of video objects in the system (Figure 7 of Section 5).

3.2.1 *Interaction of QoS Auxiliary Services.* In order to ensure noninterference among the auxiliary services that are used to provide QoS, the specific mechanisms implemented for placement and scheduling must be designed not to conflict with each other. Currently ROD (replication-on-demand meta-actor), PP (predictive placement meta-actor), and DR (dereplication meta-actor) operate on the basis of a (conservative) snapshot of the current resource allocation and use. However, without appropriate constraints on the usage of these services, inconsistencies can arise due to their interaction. Currently a snapshot of the system state is obtained at every prediction period. This includes the requests currently being serviced, the replicas currently available (including completed replications and dereplications from prior periods) and current resource availabilities.

Some examples of constraining the behavior of the auxiliary placement and scheduling services are:

—DR should not dereplicate a replica that RS (request scheduling meta-actor) is making an assignment to. (Also a replica assigned to an active request should not be physically dereplicated.)

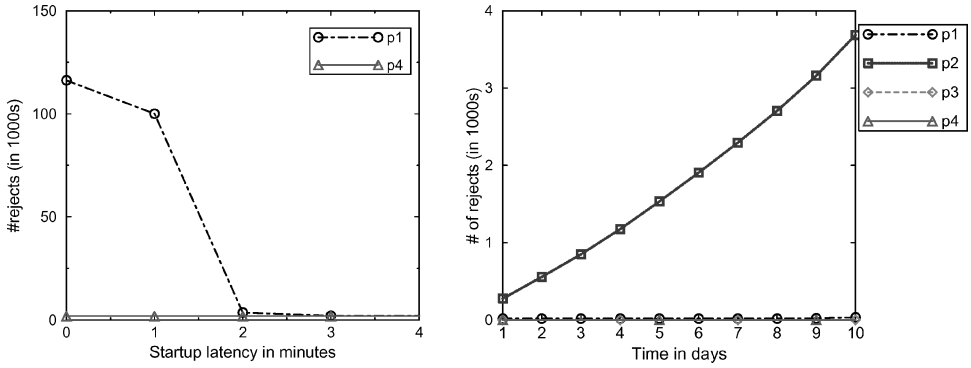

Fig. 3. Comparison of the performance of load management policies for request scheduling and video placement in a distributed video server.

—With the concurrent execution of the ROD and PP policies, each process initiates replica creation based on a current snapshot of available systems resources. Without proper constraints, this snapshot will not account for replicas being created dynamically by the other process and thus the resource capacity of a node may be exceeded by the combined effects. A naive solution is to disable ROD when PP is initiated.

—DR and PP should not cancel one another out. The interaction between DR and PP is not a functional correctness issue, it has to do with cost-effective performance of the overall system. By using compatible prediction models for the two services, the broker coordinates the service interaction. For instance, a lazy DR is initiated prior to the execution of a PP process to make replica space available for the creation of new replicas if needed.

The main objective here is to exploit concurrency without sacrificing consistency. Increased concurrency allows for improved performance and reduces unnecessary delays caused by serizalized execution on middleware services, which in turn may impact user QoS. What is therefore required, is a reasonable set of constraints that ensure the desired system behavior.

3.2.2 *Some Performance Results.* Performance studies show that application objects can be managed effectively by composing multiple resource management activities managed at the meta level [Venkatasubramanian and Ramanathan 1997]. Figure 3 illustrates 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 a priori) scheduling and placement (P2), (c) composite policies that provide adaptive scheduling and predictive placement (P3 and P4, an optimized version of policy P3). The graph on the left-hand side 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, the right hand side depicts the inadequacy of P2, that relies on only predictive policies for scheduling and placement. In comparison, the other three 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. In this paper, we study complex interactions that can arise due to the simultaneous execution of multiple system policies.

4. THE TWO-LEVEL METAARCHITECTURAL FRAMEWORK

As mentioned earlier, in the TLAM framework, 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 run-time system which manages system resources and controls the run-time behavior of the base level. Base-level actors (and messages) may have associated *annotations*, that is, metadata in the form of finite maps, that meta-actors can read and write.

4.1 TLAM Models

A TLAM model is a structure of the form

$$TLAM = \langle Net, TLAS, loc \rangle,$$

where *Net* is the underlying network, with processor nodes and communication links, *TLAS* is a *two-level actor system*, and *loc* is a function that specifies how the actors are distributed over the network. The *TLAS* component specifies the following sets: actor names, actor states, messages, annotation tags and and annotation values, with names, states, and messages partitioned into base- and metalevels. These sets are typically presented as algebraic data types. The *TLAS* also specifies a set of reaction rules that determine the actions of actors when messages are delivered, or in the case of meta-actors, when notifications of base-level events are delivered.

We illustrate these concepts with examples from the QoSB TLAM specification of Section 5.3 where QoS specific notational details are more fully explained. The actor names of the QoSB TLAM include, among others, *QB*, the name of the broker meta actor, and a set *ReqActors* of names of base-level actors created to represent incoming MM requests. The broker state is of the form $QBB(mms, status)$, where *mms* and *status* are data structures used to represent the broker's model of the MM state and resource management activity respectively. A client request message sent to *QB* has the form

$$QB \lhd \texttt{mmReq}(\alpha_{cl}, MM, qs),$$

where α_{cl} is the name of the client, *MM* is the requested MM object, and *qs* is the requested QoS. When such a request is received by the broker, a request actor is created with annotations

$$anot_{\text{req}} = [ClientId = \alpha_{cl}, ObjId = MM, Qos = qt, State = Waiting],$$

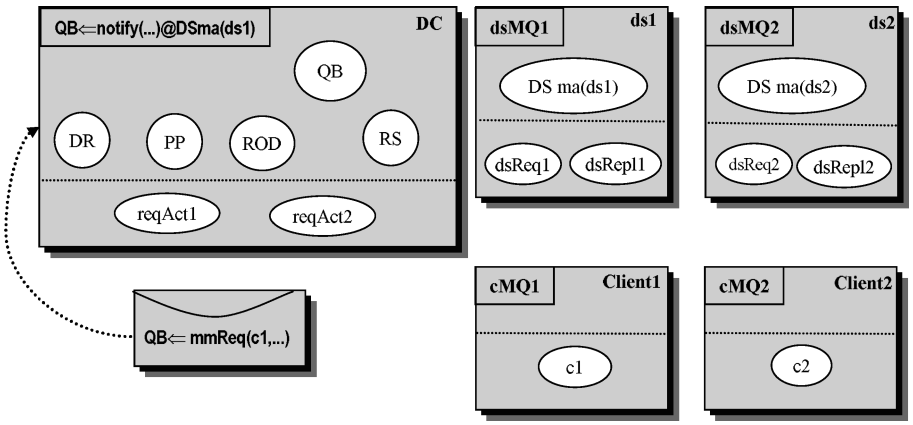

Fig. 4. Example TLAM configuration for the QoS brokerage service showing components on DS and DC nodes.

where *qt* describes the resources needed to provide the requested QoS. The annotations of an actor or message are treated as finite functions from tags to annotation values. The tags of $anot_{\text{req}}$ are *ClientId*, *ObjId*, *Qos*, and *State*. The value is associated to *ClientID* by $anot_{\text{req}}$, that is, $anot_{\text{req}}(ClientId)$, is α_{cl}.

4.2 TLAM System Configurations

A TLAM *configuration*, C, represents a snapshot of the system state. It has a set of base-level actors, a set of metalevel actors and a set of undelivered messages. Each actor has a unique name, a state, and resides on the node given by the *loc* function. The undelivered messages are distributed over the network—some are travelling along communication links and others are held in node buffers. We represent a system configuration concretely by a set of node configurations together with the set of messages travelling along network links, suppressing details of which links these messages are travelling along. A node configuration has a node name, a message buffer, and its set of base and meta actors.

Figure 4 shows a possible configuration, C_x, of a QoS Broker system with a DC node, two DS nodes and two client nodes. Two requests have been admitted, one is being serviced, and one has completed (as indicated by the *state* annotation). There are two undelivered messages: a notification in the DC message buffer to *QB* that the request on one DS node has completed, and an MM request message travelling in the network. There are two request actors on the DC node, corresponding to the two admitted MM requests, one from each client. Each DS node is serving one of the requests with a local request actor and replica representing the assigned request. The mathematical notation for this configuration is given in Figure 5.

The meaning of the QoSB specific notation, such as *mms*, *status*, and the roles of the QoS annotation tags, used above is discussed in more detail in Section 5.3.

$C_x =$

$\{Client_1 \mid cMQ_1 \mid \ldots; \langle c_1 : clientB(\alpha_1^{req})\rangle \ldots\} \{Client_2 \mid cMQ_1 \mid \ldots; \langle c_2 : clientB(\alpha_2^{req})\rangle \ldots\}$

$\{ds_1 \mid dsMQ_1 \mid \langle DSma(ds_1) : DSB(ds_1)\rangle; \langle \alpha_1^{\mathrm{req,ds}} : dsReqB[dsreqa_1]\rangle\langle \alpha_1^{\mathrm{repl}} : dsReplB[dsrepla_1]\rangle\}$

$\{ds_2 \mid dsMQ_2 \mid \langle DSma(ds_2) : DSB(ds_2)\rangle; \langle \alpha_2^{\mathrm{req,ds}} : dsReqB[dsreqa_2]\rangle\langle \alpha_2^{\mathrm{repl}} : dsReplB[dsrepla_2]\rangle\}$

$QB \triangleleft \texttt{mmReq}(c_1, MM, qs)$

$\{\ DC \mid QB \triangleleft \texttt{notify}([\alpha_1^{req} = [State = reqCompleted]]) \ @\ DSma(ds_1) \mid$
$\langle QB : QBB(mms, status)\rangle\langle RS : IdleB_{\mathrm{rs}}\rangle\langle ROD : IdleB_{\mathrm{rod}}\rangle\langle PP : IdleB_{\mathrm{pp}}\rangle\langle DR : IdleB_{\mathrm{pp}}\rangle;$
$\langle \alpha_1^{req} : dcreqB_1[annots_1]\rangle\langle \alpha_2^{req} : dcreqB_2[annots_2]\rangle\ \}$

where

$dsreqa_1 = [ClientId = c_1, ObjId = MM_1, QoS = qt_1, State = reqCompleted, ReqId = \alpha_1^{req}]$

$\ldots$

$annots_1 = [ClientId = c_1, ObjId = MM_1, QoS = qt_1, State = Servicing, Replica = ds_1]$

$\ldots$

$mms = [\alpha_1^{req} = annots_1, \alpha_2^{req} = annots_2, (ds_1, MM_1) = \ldots, (ds_2, MM_2) = \ldots]$

$status = [PP = \texttt{false}, DR = \texttt{false}, RS = \texttt{nil}, ROD = \texttt{true}]$

Fig. 5. Example specification of the TLAM configuration for the QoS brokerage service.

4.2.1 *Functions to Access Configuration Information.* We use the following functions to extract information from a configuration C:

—$Cast(C)$ is set of names of base- and metalevel actors that exist in C.
—$getState(C, a)$ is the state of actor a in C for $a \in Cast(C)$, thus $getState(C, a) = s$ just if $\langle a : s\rangle$ occurs in C
—$getA(C, a, t)$ is the value in C of the annotation with tag t of actor a.
—$setA(C, a, t, v)$ sets the value of the annotation with tag t of actor a, returning the updated configuration. Thus $getA(setA(C, a, t, v), a, t) = v$.

For example,

—$Cast(C_x) = \{QB, RS, PP, DR, ROD, DSma(ds_1), DSma(ds_2), \alpha_1^{req}, \alpha_2^{req}, \alpha_1^{\mathrm{req,ds}}, \alpha_2^{\mathrm{req,ds}}, \alpha_1^{\mathrm{repl}}, \alpha_2^{\mathrm{repl}}\}$
—$getState(C_x, QB) = QBB(mms, status)$
—$getA(C_x, \alpha_1^{req}, State) = Servicing$ and $getA(C_x, \alpha_1^{\mathrm{req,ds}}, State) = reqCompleted$

4.3 Rules, Transitions, and Computation Paths

The QoS meta-actor rules have the general form:

$$(\dagger)\quad \langle a : s\rangle[,\ a \triangleleft M\,] \xrightarrow[effect]{trigger} \langle a : s'\rangle\ ,\ MC \quad \text{if } cond,$$

where $\langle a : s\rangle$ is a QoS meta-actor with name a in state s and $a \triangleleft M$ is a message to a with content M. ([. . .] indicates that the message part may be empty.) s' is the new state of actor a and MC is a possibly empty set of messages sent In

the QoS rules, the trigger *trigger*, if not empty, describes a base level event. In the QoS Broker TLAM these events correspond to progress in replication and request servicing. The effect *effect* may include an effect the form $setA(mmsU)$ meaning that annotations of base-level actors on the same node are modified by applying the annotation update, $mmsU$, or an effect $\texttt{new}(\alpha)$ indicating creation of a new base-level actor α.

Transitions yield new configurations by application of enabled rules, or by moving messages to and from node message buffers. A rule for receiving a message is enabled in a configuration, if the message is present on the node of the receiver, the receiving actors state matches that in the left-hand side of the rule, and the rule condition is satisfied. When such a rule is applied to a configuration, only the node where the receiving actor is located is changed. The message delivered is removed, and any messages sent are added to the node message buffer, the receiving actors state is replaced by the state specified by the right-hand side of the rule, and any base-level effects specified by the rule are applied. For example the rule **QBdsNotify** (Section 5.3.4) is enabled in C_x. If this rule is applied, the DC message buffer becomes empty, and the annotation update contained in the message is applied, thus the *State* annotation of α_1^{req} becomes *reqCompleted*. Call this C_1. In configuration C_1, the MM request can be moved to the DC message buffer, thus enabling the request delivery rule. If this rule is now applied, the result is a configuration, C_3, in which the MM request message is removed from the DC message buffer, the state of QB is given by the right-hand side of the rule, α^{req} is added to the set of base-level actors, and the message to RS is put in the DC message buffer. Note that another possible sequence of steps leading to C_3 is to move the MM request message into the DC message buffer, then deliver the notification, then deliver the MM request message.

In general, there are also transitions corresponding to application of base-level rules. In addition to changing the base-level state, these transitions may also generate base-level events to be handled by metalevel event rules. We leave base-level rules unspecified here. As an example of metalevel event rules, the notification delivered to QB above would have been generated by the DS node manager rule node manager rule, DSreplComplete, for handling a service completion trigger.

$$\langle DSma(DS) : DSB(DS)\rangle \xrightarrow[setA(reqU)]{\texttt{reqCompletes}\left(\alpha_1^{\mathrm{req,ds}}\right)} \langle DSma(DS) : DSB(DS)\rangle \ ,\ QB \lhd \texttt{notify}(reqU') \mathbin{@} DSma(DS),$$

where

$$reqU = \left[\alpha_1^{\mathrm{req,ds}} = [State = reqCompleted]\right]$$

$$reqU' = \left[\alpha_1^{req} = [State = reqCompleted]\right] \wedge \alpha_1^{req} = getA\left(\alpha_1^{\mathrm{req,ds}}, ReqId\right).$$

In this case, the DS node manager updates the annotation of the request actor, $\alpha_1^{\mathrm{req,ds}}$, that has signalled completion and sends a notification to the broker QB.

Formally, a transition τ has the form $C \xrightarrow{l} C'$ where C is the source ($C = \mathit{source}(\tau)$) and C' is the target ($C' = \mathit{target}(\tau)$) and l is the label of the rule applied. A *computation path* π is a possibly infinite sequence of transitions in which the source of each transition other than the first is the same as the target of the previous transition.

$$\pi = \left[C_i \xrightarrow{l_i} C_{i+1} \;\middle|\; i \in \mathbf{Nat}\right]$$

The ith transition of π, $\pi(i)$ is $\tau_i = C_i \xrightarrow{l_i} C_{i+1}$ and C_i, also called the ith stage of π. When focusing on the sequence of configurations rather than transitions, we represent a computation path as an alternating sequence of configurations and rule labels.

$$\pi = C_0 \xrightarrow{l_0} C_1 \cdots C_i \xrightarrow{l_i} C_{i+1} \cdots .$$

The example sequence above is part of a computation path with initial configuration C_0 in which there are two undelivered MM request messages and there is no ongoing activity. Omitting repeat of the matching final and source configurations of adjacent transitions, this would be written

$$C_0 \cdots C_x \xrightarrow{\textbf{QBdsNotify}} C_1 \xrightarrow{\textbf{toDC}} C_2 \xrightarrow{\textbf{QBreq}} C_3 \cdots .$$

The event diagrams shown in Section 5.3.1 illustrate segments of additional possible computations in a QoSB system.

A computation path is fair if for any transition that becomes enabled at some point (communication or execution step), then either that step eventually happens, or it becomes permanently disabled. (Only execution steps can become disabled, for example if a actor state changes in such a way that a rule no longer applies.) The semantics of a configuration is the set of fair computation paths starting with that configuration.

A TLAM *system* is a set of configurations closed under the transition relation. 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. For example, replica-request constraints (ϕ_{rr}—Definition 5.9) and the total resource property (ϕ_{res}—Definition 5.10) are simple invariants that must hold for all configurations of the system, while the request constraints (ϕ_{req}—Definition 5.6) and the replica constraints (ϕ_{repl}—Definition 5.8 combine configuration invariants and properties of transitions constraining the ways in which configurations may change.

End-to-end service specifications constrain the patterns of interactions between clients and servers: request messages and replies sent in a computation path. System-wide service specifications express constraints on the computation paths of a system in terms of abstract functions on configurations measuring possibly global properties. Behavior specifications express constraints on the cast of a configuration, including types and locations of actors, actor states, and transitions.

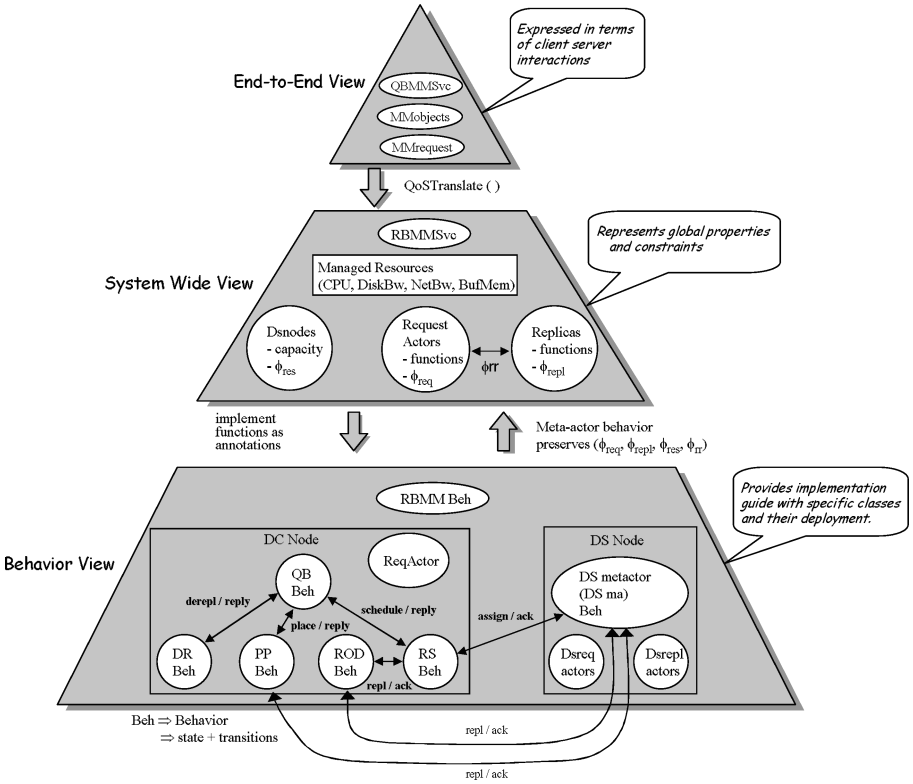

Fig. 6. Pyramid showing summary of formal specification. The three levels indicate (a) the end-to-end viewpoint from the perspective of the end-to-end service provided by the system in response to a request, (b) a system-wide view with abstract properties that must be obeyed by the underlying components and (c) a detailed behavioral view that specifies component behaviors and constraints on their behaviors.

5. REASONING ABOUT QOS-BASED MM SERVICES

As mentioned in Section 1, 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 noninterference, 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 describe how to model the multimedia meta-architecture and resource management policies presented above using the TLAM framework. Following our basic approach to modeling systems in the TLAM framework, we specify QoS services from different viewpoints, define functions relating elements of the different viewpoints and establish theorems relating the different specifications.

Figure 6 summarizes concepts formalized in the different viewpoints and their interrelationships. It can be used as a reference to guide the reader through the different levels of specification. In Section 5.1, we informally

describe the notion of a system providing *QoS-based MM Service*. This is the high-level end-to-end request-based service that the client sees. In Section 5.2, we define the notion of a system providing *Resource-based MM Service*. This reflects the relevant system resources and expresses high-level resource management requirements that must be met in order to provide the QoS-based MM Service. In particular, functions characterizing resources, replicas, and requests are introduced and predicates constraining the system are defined in terms of these functions. We postulate a function, *QoSTranslate*, that maps QoS requirements to resource requirements and show that:

—if a system provides Resource-based MM Service, then under the given assumptions on the mapping from QoS requirements to Resource requirements, the system provides QoS-based MM Service.

In Section 5.3, we define the notion of a system having *Resource-based MM Behavior*. This viewpoint reflects the QoS broker software architecture. It specifies the different types of QoS meta-actors and their deployment, and places constraints on the actions of the QoS meta actors. We define initial and noninterference conditions for a system, and show that

—if a system has Resource-based MM Behavior, then if the initial and noninterference conditions hold, the system provides Resource-based MM Service.

In Section 5.5, we refine the behavior by requiring the system to act according to given *Resource-based MM Broker Policies*. Here we focus on one specific policy, the *Adaptive Request Scheduling* Policy. We show that

—if a system acts according to the Adaptive request Scheduling Policy, then it has Resource-based MM Behavior.

5.1 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*. We also assume given a set *MMreqset* of MM requests—messages used to request MM service—and let *MMreq* range over *MMreqset*. A MM request message *MMreq* determines a triple (α_{cl}, MM, qs), interpreted as a request to initiate a MM streaming service from the server receiving the request, to the client α_{cl}, using the MM object *MM*, and obeying the QoS requirement *qs*. (More generally a MM request could involve more than one MM object. For simplicity we restrict attention to the single object case.)

Definition 5.1 (*QoS-Based MM Service*). A system *S* provides a QoS-based MM Service over the set of MM objects, *MMObjects*, and request messages *MMreqset* iff for every configuration *C* of *S*, if there is an undelivered request message *MMreq* in *C*, then along any path π from *C* exactly one of the following properties hold:

(1) there is a unique transition in π where *MMreq* is accepted for service, and service is provided with the required QoS until complete, or

(2) there is a unique transition in π where *MMreq* is rejected, and for this to happen it must be the case that the requested QoS cannot be provided at the time that *MMreq* arrives.

5.2 Specifying a Resource-Based MM Service

5.2.1 *Roadmap to the Resource-MM Service Specification.* As shown in Figure 6, the Resource-Based MM Service viewpoint concerns a set of managed resources, DS Nodes, request and replica objects. Nodes of the physical architecture of the MM server (Figure 1) are represented by TLAM nodes and the MM state of the system by TLAM system configurations. In the remainder of this section, the set of managed resources is defined and constraints are given on the function *QoSTranslate* that relates the QoS requirement part of an MM request in the QoS MM Service viewpoint to correspoonding resource requirements in the Resource Based MM Service viewpoint.

In Section 5.2.2, functions are introduced to abstractly characterize relevant features of a system configuration. DS node capacity is introduced in Definition 5.4. Each MM request is represented by a freshly created request base-actor and functions characterizing the state of a request are introduced in Definition 5.5. Since there is at most one replica of an MM object on a given DS node, replicas are represented by pairs giving the DS node where the replica is located and the replicated MM object. In Definition 5.7, functions characterizing replicas are introduced. The resource-based MM service is then specified in terms of constraints on the values of these given functions and the way the values may change (Definitions 5.6, 5.8, 5.9, and 5.10). If the MM system obeys these constraints, then the QoS-based service requirements will be met and the underlying streaming mechanisms will be able to provide the desired QoS-based service (Section 5.2.3). In particular, as will be shown in the Section 5.3, any admission and placement policies that obey the constraints can be used to implement the MM service. Note that the replica constraints (5.8) constitute noninterference constraints associated with the management of replicas by the middleware services.

5.2.1.1 *Managed Resources.* To specify the system-wide MM service requirements in more detail we assume given a function *QoSTranslate*, which maps MM requests to resource requirements which, if met, will ensure the requested QoS. Thus, real-time requirements typical of MM applications, for example, required bit-rate of video, are translated into corresponding resource requirements, for example, a bandwidth requirement. (See Nahrstedt [1995] for examples of such QoS translation functions). For the purposes of this specification, we assume that if resources are allocated for a request, then they are used (as needed) to provide the requested QoS.

Definition 5.2 (*Managed Resources*). We consider four managed resources: network bandwidth (*NetBW*), CPU cycles (*CPU*), disk bandwidth (*DiskBW*), and memory buffer (*BufMem*). We let *Resources* denote this set of resources and let *Res* range over *Resources* and we use the notation $Unit_{Res}$ for the units in which we measure the resource *Res*. We let $QoSTuple = Unit_{DiskBW} \times Unit_{CPU} \times$

$Unit_{NetBW} \times Unit_{BufMem}$, For an element qt of $QoSTuple$, we write qt_{Res} to select the component associated to Res.

Disk space on an individual node is considered only during the placement of data objects. However, the consumption of disk space does not change dynamically as requests are being serviced and scheduled. To simplify the model, we do not include disk space as a managed resource. It could be included with no significant change in the overall organization of the specification, theorems, or proofs.

Definition 5.3 (*QoSTranslate Requirements*). The function *QoSTranslate* maps MM requests to 4-tuples representing resource allocation requirements for the four managed resources. Thus, for any MM request, *MMreq*,

$$QoSTranslate(MMreq) \in QoSTuple.$$

We require that $QoSTranslate(MMreq)$ be such that the QoS requirement of *MMreq* is met if

(*a*) the resources allocated to *MMreq* are at least those specified by $QoSTranslate(MMreq)$ and

(*b*) the allocated resources and a copy of the MM object of *MMreq* are continuously available on the assigned node during the service phase for *MMreq*.

Availability means that the MM object replica is not deleted (or overwritten) and that the total allocation never exceeds capacity, since over allocation implies that the extra resources must be taken from some already admitted request thereby possibly violating the QoS constraints for that request.

5.2.2 *Constraints on Resources, Requests and Replicas.* Nodes in the MM server physical architecture (Figure 1) are modeled as TLAM nodes. Recall that there are several kinds of nodes: A set of data source nodes that holds replicas and provides the actual MM streaming, a distribution controller node responsible for coordinating the data source nodes, and a set of client nodes from which MM requests arise. There is also a tertiary storage node that contains the MM objects; however, we do not model this explicitly.

Definition 5.4 (*DSnodes*). We let *DSnodes* be the set of data source nodes and let *DS* range over *DSnodes*. We assume, given, a function *capacity* such that $capacity(DS, Res) \in Unit_{Res}$ is the total *Res*-capacity of node *DS* for any data source node *DS* and resource *Res*.

Now we introduce functions on system configurations characterizing what can be observed about requests and replicas. Predicates are defined constraining system behavior. A predicate on the system is defined in terms of a predicate (with the same name) on the computation paths of the system, which in turn is defined in terms of predicates on transitions and configurations. The definitions are first stated in English, followed by a mathematical formula.

Definition 5.5 (*Functions Characterizing Requests*). Each MM request that has been delivered in a system has a uniquely associated base actor that represents the request during admission control and servicing. We let

$ReqActors \subset Act_{\mathrm{b}}$ be a subset of the base actor identifiers set aside for association with MM requests and let α^{req} range over $ReqActors$. (Note that the ability to form these unique associations relies on uniqueness of messages and newly created actors in the TLAM model.) There are five functions characterizing the state of a request actor α^{req} in a system configuration C:

—$reqClientId(C, \alpha^{req})$—identifies the client making the request.
—$reqObjId(C, \alpha^{req})$—the MM object requested.
—$reqQoS(C, \alpha^{req})$—the 4-tuple returned by $QoSTranslate$.
—$reqState(C, \alpha^{req}) \in \{Waiting, Granted, Denied, Servicing, reqCompleted\}$—the request status.
—$reqReplica(C, \alpha^{req}) \in DSnodes + \{\mathtt{nil}\}$—the DS node to which the request has been assigned if not `nil`.

Definition 5.6 (*Request Constraints* (ϕ_{req})). The predicate ϕ_{req} constrains the way in which the request functions are allowed to change as a system evolves.

(1) A system S satisfies the request function constraints, $\phi_{\mathrm{req}}(S)$ just if $\phi_{\mathrm{req}}(\pi)$ holds for each computation π of S.

(2) $\phi_{\mathrm{req}}(\pi)$ holds for a computation path π of S just if $\phi_{\mathrm{req}}(\tau)$ holds for every transition of π, and (a) the state of every request actor waiting for admission eventually changes (to being admission granted or denied), and (b) granted requests eventually are serviced and completed.

$$(a)\ (\forall i \in \mathbf{Nat})(C = source(\pi(i)) \wedge reqState(C, \alpha^{req}) = Waiting \Rightarrow (\exists j \in \mathbf{Nat}, C' = source(\pi(i+j+1))) \; reqState(C', \alpha^{req}) \in \{Granted, Denied\}.)$$

$$(b)\ (\forall i \in \mathbf{Nat})(C = source(\pi(i)) \wedge reqState(C, \alpha^{req}) = Granted \Rightarrow (\exists j_s, j_c \in \mathbf{Nat}, C_s = source(\pi(i+j_s+1)), C_c = source(\pi(i+j_c+1))) \; (reqState(C_s, \alpha^{req}) = Servicing \wedge reqState(C_c, \alpha^{req}) = reqCompleted).$$

(3) $\phi_{\mathrm{req}}(\tau)$ holds for a transition $\tau : C \longrightarrow C'$ in S just if

(a) The values of $reqClientId$, $reqObjId$, and $reqQoS$ are unchanged.

$$(\forall \alpha^{req} \in ReqActors \cap Cast(C)) \; (reqClientId(C, \alpha^{req}) = reqClientId(C', \alpha^{req}) \wedge reqObjId(C, \alpha^{req}) = reqObjId(C', \alpha^{req}) \wedge reqQoS(C, \alpha^{req}) = reqQoS(C', \alpha^{req})).$$

(b) The state of request actor can only move from $Waiting$ to $Granted$ or $Denied$, from $Granted$ to $Servicing$ to $reqCompleted$.

$$(\forall \alpha^{req} \in ReqActors \cap Cast(C)) \; reqState(C, \alpha^{req}) = Waiting \Rightarrow reqState(C', \alpha^{req}) \in \{Waiting, Granted, Denied\}$$

$reqState(C, \alpha^{req}) = Denied \Rightarrow reqState(C', \alpha^{req}) = Denied$
$reqState(C, \alpha^{req}) = Granted \Rightarrow reqState(C', \alpha^{req}) \in \{Granted, Servicing\}$
$reqState(C, \alpha^{req}) = Servicing \Rightarrow$
$reqState(C', \alpha^{req}) \in \{Servicing, reqCompleted\}$
$reqState(C, \alpha^{req}) = reqCompleted \Rightarrow reqState(C', \alpha^{req}) = reqCompleted.$

Furthermore, if a request state moves from *Waiting* to *Denied*, it must be the case that there are not sufficient resources available at the time the request is processed.

(c) Once defined, the replica associated to a request actor remains constant.

$$reqReplica(C, \alpha^{req}) \neq \texttt{nil} \Rightarrow reqReplica(C, \alpha^{req}) = reqReplica(C', \alpha^{req}).$$

Note that (3a) for all transitions of all computations of a system implies that the values of *reqClientId*, *reqObjId*, and *reqQoS* are constant throughout the life of a request actor.

Definition 5.7 (*Functions Characterizing Replicas*). There is at most one replica of a given MM object on any DS node, and thus it can be uniquely identified by the node and MM object. Unlike request object, there is no need for explicit instantiation of replica objects at this stage; this allows us to state abstractly the notion of a replica and leave its implementation to the behavior specification. In addition to either being on the node or not, a MM object may be in the intermediate state of being replicated. There are three functions characterizing the state of the replica of a MM object *MM* on a DS node *DS* in system configuration *C*:

—$replState(C, DS, MM) \in ReplStates = \{InQueue, InProgress, replCompleted\}$ is the replication status of the multimedia object *MM* on node *DS*. *InQueue* indicates that replication has been requested but not initiated, *InProgress* indicates that replication is in progress, and *replCompleted* indicates that replication is complete.

—$replnBW(C, DS, MM) \in Unit_{NetBW}$ is the minimum bandwidth available to complete a replication, meaningful only if replication is in progress or in queue.

—$replClass(C, DS, MM) \in \{0, 1, 2, 3\}$ is the replication class of the multimedia object *MM* on node *DS*. This is used to coordinate replication and dereplication activities. Class 0 indicates that the replica is not present on the node. A replica of class 1 is guaranteed to be available. A replica of class 2 is considered marked as dereplicable but remains available until all requests assigned to it have completed. A replica of class 3 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.

Definition 5.8 (*Replica Constraints* (ϕ_{repl})). The predicate ϕ_{repl} constrains the way in which the replica functions are allowed to change as a system evolves.

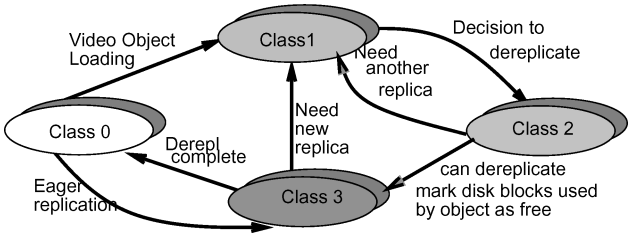

Fig. 7. State Transition Diagram depicting the life of a media replica actor. In this diagram, states are labelled by the value of the function *replClass* and the arrows indicate allowed changes in the value as the system evolves.

(1) A system S satisfies the replica function constraints, $\phi_{\text{repl}}(S)$, just if for all computations π of S, $\phi_{\text{repl}}(\pi)$ holds.

(2) $\phi_{\text{repl}}(\pi)$ holds for a computation π of S if $\phi_{\text{repl}}(\tau)$ holds for all transitions τ of π, and replication progresses, that is for C a configuration of π

(a) if $replState(C, DS, MM) = InQueue$, then there is a later configuration C' such that

$$replState(C', DS, MM) = InProgress$$

(b) if $replState(C, DS, MM) = InProgress$, then there is a later configuration C' such that

$$replState(C', DS, MM) = replCompleted.$$

(3) $\phi_{\text{repl}}(\tau)$ holds for a transition $\tau : C \longrightarrow C'$ just if

(a) The replication state only moves from *InQueue* to *InProgress* to *replCompleted* to *InQueue* (the last allowed only if the replica class is 0).

$$replState(C, DS, MM) = InQueue \Rightarrow$$
$$replState(C', DS, MM) \in \{InQueue, InProgress\}$$
$$replState(C, DS, MM) = InProgress \Rightarrow$$
$$replState(C', DS, MM) \in \{InProgress, replCompleted\}$$
$$replState(C, DS, MM) = replCompleted \Rightarrow$$
$$replState(C', DS, MM) \in \{replCompleted, InQueue\}$$
$$replState(C', DS, MM) = InQueue \Rightarrow replClass(C, DS, MM) = 0$$

meaning the replica is not present in C.

(b) the *replClass* function must satisfy the constraints specified by the transition diagram given in Figure 7. For example the diagram specifies that if

$$replClass(C, DS, MM) = 0 \quad \text{and} \quad C \to C',$$

then

$$replClass(C', DS, MM) \in \{0, 1, 3\}.$$

Also, if

$$replClass(C, DS, MM) = 2 \quad \text{and} \quad replClass(C', DS, MM) = 3,$$

then there are no active requests assigned to this replica.

The function *replnBW* is only constrained by the network bandwidth availability. This constraint appears as part of the *Total Resource* Property (Definition 5.10 below).

The predicate ϕ_{rr} constrains the relation between the replica and request functions.

Definition 5.9 (*Replica-Request Constraints* (ϕ_{rr})). A system S satisfies the replica-request function constraints, $\phi_{\text{rr}}(S)$ just if $\phi_{\text{rr}}(C)$ holds for each configuration C in S where $\phi_{\text{rr}}(C)$ holds iff:

(1) $reqReplica(C, \alpha^{req}) \neq \texttt{nil} \Leftrightarrow reqState(C, \alpha^{req}) \in \{Granted, Servicing, reqCompleted\}$
(2) $reqState(C, \alpha^{req}) = Servicing \Rightarrow replState(C, reqReplica(C, \alpha^{req}), reqObjId(C, \alpha^{req})) = replCompleted$
(3) $reqState(C, \alpha^{req}) \in \{Granted, Servicing\} \Rightarrow replClass(C, reqReplica(C, \alpha^{req}), reqObjId(C, \alpha^{req})) \in \{1, 2\}$

Note that if replication for a granted request is still ongoing, the request state remains *Granted* until replication is complete, after which point it must eventually become *Servicing*.

The final definition needed before we state the full specification deals with the use of resources as determined for a given configuration by the replica and request functions.

Definition 5.10 (*TotalResource Property* (ϕ_{res})). $\phi_{\text{res}}(S)$ states that for every data source node and every managed resource, the sum of the resources allocated to the requests on the DS node in any configuration of the system will not exceed the node's capacity for that resource. The resources currently allocated on a DS node include resources allocated to streaming accepted MM requests, as given by the *reqQoS* function for requests assigned to that node, as well as replications that are currently ongoing. More precisely, define $ReplAlloc(C, DS, Res)$, the resource allocated to ongoing replications on node DS in configuration C by

$$ReplAlloc(C, DS, Res) = 0 \quad \text{if } Res \neq NetBW$$

$$ReplAlloc(C, DS, NetBW) = \sum_{MM \in replicating(C,DS)} replnBW(C, DS, MM)$$

and define $ResAlloc(C, DS, Res)$ the amount of resource Res on node DS currently allocated in configuration C by

$$ResAlloc(C, DS, Res) = \sum_{\alpha^{req} \in Streaming(C,DS)} reqQoS(C, \alpha^{req})_{Res} + ReplAlloc(C, DS, Res),$$

where

$$Replicating(C, DS) = \{MM \in MMObjects \mid replState(C, DS, MM) = InProgress\}$$

$$Streaming(C, DS) = \{\alpha^{req} \in ReqActors \cap Cast(C) \mid reqState(C, \alpha^{req}) = Servicing \wedge reqReplica(C, \alpha^{req}) = DS\}$$

Then ϕ_{res} is defined for configurations C and systems S by

$$\phi_{\text{res}}(C, DS, Res) \Leftrightarrow ResAlloc(C, DS, Res) \leq capacity(DS, Res)$$

$$\phi_{\text{res}}(C) \Leftrightarrow (\forall DS \in DSnodes, Res \in Resources)\phi_{\text{res}}(C, DS, Res)$$

$$\phi_{\text{res}}(S) \Leftrightarrow (\forall C \in S)\phi_{\text{res}}(C)$$

5.2.3 *Specification, Theorem, Proof.* Using the characterizing functions and constraints defined above, we now define the requirements for a Resource-based MM service, and showing that such a service provides a QoS-based MM Service.

Definition 5.11 (*Resource-Based MM Service Specification*). A system S provides Resource-based MM service with respect to requests in *MMreqset*, functions *QoSTranslate*, *capacity*, and the functions characterizing replica and request state as specified above iff

(1) S satisfies the constraints in replica and request functions
   (a) $\phi_{\text{repl}}(S)$—S satisfies the replica constraints of Definition 5.8
   (b) $\phi_{\text{req}}(S)$—S satisfies the request constraints of Definition 5.6
   (c) $\phi_{\text{rr}}(S)$—S satisfies the replica-request constraints of Definition 5.9
   (d) $\phi_{\text{res}}(S)$—S satisfies the total resource requirement (Definition 5.10)
(2) for $C \in S$, if there is an undelivered message, *MMreq*, with parameters (α_{cl}, MM, qs), then along any path π from C there is a (unique) stage i such that the transition $\pi(i) = C \longrightarrow C'$ delivers *MMreq* and there is a newly created request actor, α^{req}, ($\alpha^{req} \in Cast(C')$—$Cast(C)$) such that
   (a) $reqClientId(C', \alpha^{req}) = \alpha_{cl}$
   (b) $reqObjId(C', \alpha^{req}) = MM$
   (c) $reqQoS(C', \alpha^{req}) = QoSTranslate(qs)$
   (d) $reqState(C', \alpha^{req}) = Waiting$
   (e) $reqReplica(C', \alpha^{req}) = \texttt{nil}$.

THEOREM 5.12 (QOS2RESOURCE). *If a system S provides Resource-based MM service as defined in* 5.11 *and the function QoSTranslate satisfies the requirements of Definition* 5.3*, then S provides QoS Based Service according to Definition* 5.1.

PROOF (OF THEOREM 5.12). We first observe that if a system S provides Resource-based MM service, then the following holds for any $C \in S$. If there is an undelivered message, *MMreq*, in C with parameters (α_{cl}, MM, qs), then,

along any path π from C, there is one of the following segments:

(Case-Denied) $C_{\text{start}} \xrightarrow{+} C_{\text{deny}}$

(Case-Granted) $C_{\text{start}} \xrightarrow{+} C_{\text{grant}} \xrightarrow{+} C_{\text{serve}} \xrightarrow{+} C_{\text{complete}}$,

such that the following hold:

(1) C_{start} is the result of delivery of *MMreq* and there is a newly created request actor $\alpha^{req} \in ReqActors \cap Cast(C_{\text{start}})$ such that Eqs. (2) of Definition 5.11 hold.

(2) $reqState(C_{\text{deny}}, \alpha^{req}) = Denied$, 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) $reqState(C_{\text{grant}}, \alpha^{req}) = Granted$, and $reqReplica(C_{\text{grant}}, \alpha^{req}) = DS$ for some DS node such that $replClass(C_{\text{grant}}, DS, MM) = 1$.

(4) $reqState(C_{\text{serve}}, \alpha^{req}) = Servicing$ and $reqState(C_{\text{complete}}, \alpha^{req}) = reqCompleted$.

(5) From C_{serve} to C_{complete} the requested *MM* is available—$replState(C, DS, MM) = replCompleted$ and $replClass(C_{\text{grant}}, DS, MM) \in \{1, 2\}$.

(Case-Denied) and (Case-Granted) guarantee that each request is accepted or rejected. Furthermore there is at most one occurrence of (Case-Denied) or (Case-Granted) for a given request message, since each message is consumed upon delivery. What remains is to show that the required QoS is provided during the service period. For this we show that the assumptions for *QoSTranslate* (Definition 5.3) correctness hold. Definition 5.3(a) follows from Eq. (2)(c) of Definition 5.11 and the request function constraints ϕ_{req} (3)(a), that *reqQoS* is constant. Definition 5.3(b) follows from ϕ_{res}, ϕ_{req}, ϕ_{rr}, and the class transition constraint of ϕ_{repl}. □

5.3 A Resource-Based MM Behavior

In the behavior viewpoint the QoS meta-actors that cooperate to provide the Resource-based MM service are made explicit. The QoS base-level annotations, meta-actor states, and messages are defined and transition rules given that specify the meta-actor behaviors.

5.3.1 *Roadmap of the Resource-Based MM Behavior View.* In Section 5.3.2, the annotations used by the QoS meta-actors are introduced. The functions characterizing request and replica state abstractly in the system-wide view are defined in terms of information held in base-level annotations.

In Section 5.3.3, the structure of metalevel actor states and messages is specified. As discussed in Section 2, there are five QoS meta-actors on the controller node, and a DSNode manager meta-actor on each DS node.

—*QB*. The QoS broker that accepts MM requests, maintains a model of the MM state, and coordinates the remaining broker services. The state of *QB* has two components denoted by *mms* and *status*. *mms* represents the MM state as a finite map from requests and replicas to their QoS annotations. *status* is used by *QB* to keep track of ongoing scheduling, replication and dereplication activities. Note that the Status Invariant (Definition 5.15) ensures noninterference (safe concurrent execution) of these ongoing middleware services.

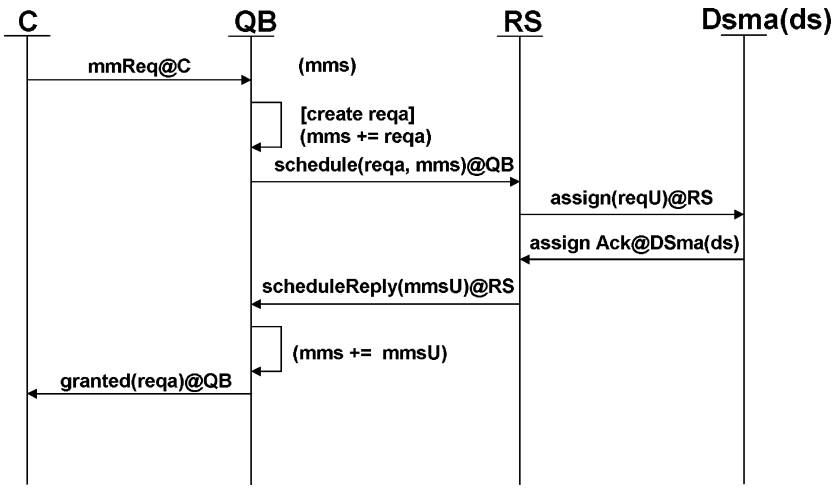

Fig. 8. A simple request admission scenario. Vertical lines represent event timelines of meta-actors named at the head. A message M from A to A' is represented by M@A labelling an arrow from A to A'. In this scenario, a request arrives at QB, QB sends a schedule message to RS, RS picks a DS node, ds, sends an assign message to the DS node manager DSma(ds), waits for an assignAck, the sends a scheduleReply to QB.

—*RS*. The request scheduler, that either idle or waiting for an acknowledge to a DS node assignment request (remembering the associated update to report to *QB*) or waiting for a reply to replication on demand request (remembering the scheduling request data).

—*ROD*. The replication on demand server, that is either idle or waiting for an acknowledge to a DS node replication request (remembering the associated update to report its customer).

—*DR*. The dereplication server, that is either idle or waiting for acknowledgments for dereplication updates sent to DS nodes (remembering the combined updates to report to *QB*).

—*PP*. The predictive placement server, that is either idle or waiting for acknowledgments for replication updates sent to DS nodes (remembering the combined updates to report to *QB*).

—*DSma*(*DS*). The DSNode manager on *DS*. Its state is embodied in the state of the requests and replicas on *DS*.

Figures 8, 9, and 10 are event diagrams illustrating sample scenarios showing interaction of the QoS meta-actors.

The transition rules formalizing the behavor shown in the event diagrams are given in Section 5.3.4. Full details are given here for a subset of the rules, the rest are summarized with details available in the Appendix.

Using the state, message and rules definitions, the definition of Resource-Based MM Behavior is given in Section 5.3.5. The system-wide view places constraints on the entire system including meta-actors not among the QoS broker meta-actors. The behavior view only makes local constraints on deployment and behaviors of individual QoS meta-actors. Typically, there is an

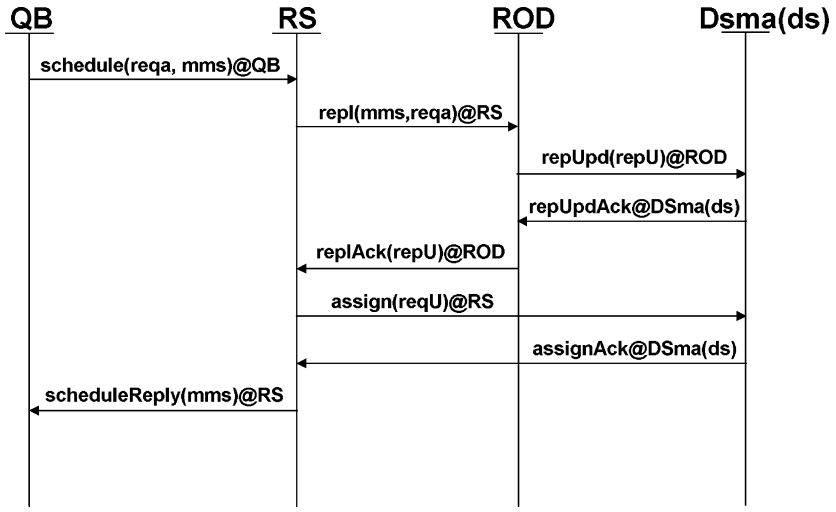

Fig. 9. Alternative RS subscenario using replication on demand.

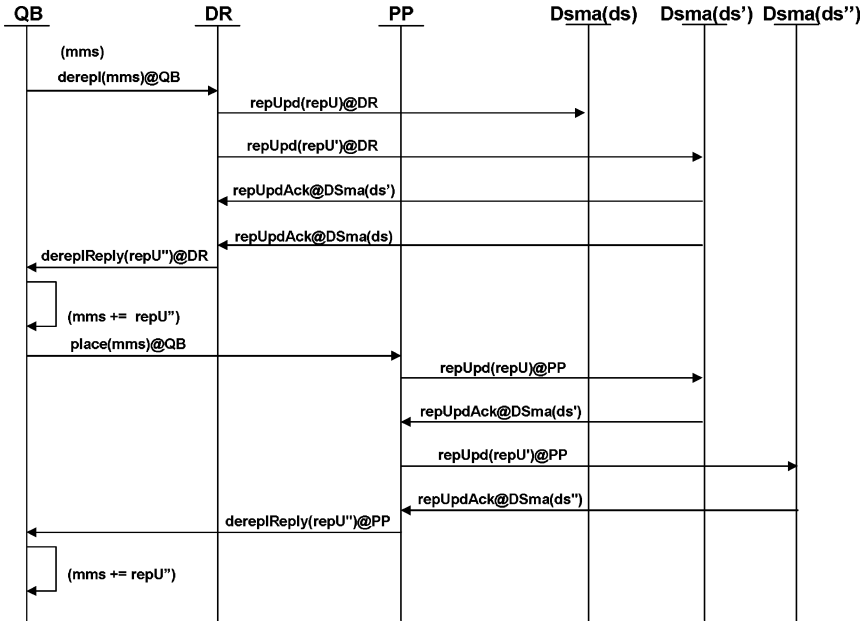

Fig. 10. Scenario showing QB management of background dereplication and replication.

implicit assumption that the system has started in some clean initial state, and that the environment will not interfere. The meaning of "not-interfere" is made explicit and precise by stating noninterference requirements and the meaning of clean initial state is made explicit and precise by stating initial state requirements and only considering system configurations reachable from such states. Using the behavior definition and initial and noninterference conditions the key theorem connecting the behavior view and the system-wide

view is stated and a sketch of the proof is given. Proof details can be found in Section 5.4.

It is possible to avoid the initial conditions by introducing a startup message an adding rules that allow the meta-actors concerned to reach a suitable state before accepting any service requests. To avoid added detail and complexity, we have chosen to assume that a suitable system configuration can be reached.

5.3.2 *QoS Annotations and Base-Level Request and Replica Functions.* In the behavioral specification, the MM resource state of a configuration, modeled previously by the abstract request and replica functions, is recorded in annotations of base-level actors implementing the MM streaming service. We partition the MM base actors of a configuration into three groups.

—*ReqActors*. Corresponding to delivered MM requests as for the resource based service. These actors are located on the Controller node.

—*DSReqActors*. Corresponding to the granted requests. These actors are located on their assigned DSnodes.

—*DSReplActors*. Corresponding to the replicas on the DS nodes. Given the assumption that there is at most one replica of any MM object on a DS node we define
NodeRepl(C, DS, MM) to be a function that gives the replica base actor associated to *MM* on node *DS* in *C*. If there is no such replica, then we take the value to be `nil`.

Definition 5.13 (*QoS Annotations*). For each system-wide replica function *replX*, *X* one of *State*, *BW*, *Class*, a replica actor has an annotation tag *X* to represent that function. Similarly, for each system-wide request function *reqX* (for *X* one of *ClientId*, *ObjId*, *QoS*, *State*, *Replica*), a request actor (on DC or a DS node), there is an annotation tag *X* to represent that function. In addition each DS request actor has an annotation tag *ReqId*, with value in *ReqActors*, used to link DS node request actors to the original request. If the value, $getA(C, \alpha^{\mathrm{req,ds}}, ReqId)$ of the *ReqId* annotation on DS request actor $\alpha^{\mathrm{req,ds}}$ in configuration *C* is α^{req}, then $\alpha^{\mathrm{req,ds}}$ is the DSnode request actor for α^{req}. Furthermore, there is a unique such $\alpha^{\mathrm{req,ds}}$ for each granted request and none for a waiting or denied request. Thus, we define

$$NodeReq(C, \alpha^{req}) = \alpha^{\mathrm{req,ds}} \quad \text{if } getA(C, \alpha^{\mathrm{req,ds}}, ReqId) = \alpha^{req}$$
$$= \texttt{nil} \quad \text{if no such } \alpha^{\mathrm{req,ds}} \text{ exists}$$

Definition 5.14 (*Replica and Request Functions*). Using the QoS annotations and functions accessing requests and replicas defined above, we define the system-wide functions on requests and replicas as follows:

—For *X* one of *State*, *BW*, *Class*, $replX(C, DS, MM)$ is the value of the *X* annotation of $NodeRepl(C, DS, MM)$ (when this is non-nil) in *C*. For example,

$$replState(C, DS, MM) = getA(C, NodeRepl(C, DS, MM), State).$$

—For *X* one of *ClientId*, *ObjId*, *QoS*, *State*, *Replica*, $replX(C, \alpha^{req})$ is defined to be the value of the *X* annotation of $NodeReq(C, \alpha^{req})$ in *C* for a granted

request, and to be the X annotation of α^{req} in C otherwise. For example,

$$reqReplica(C, \alpha^{req}) = loc(NodeReq(C, \alpha^{req})) \quad \text{if } NodeReq(C, \alpha^{req}) \neq \texttt{nil}$$
$$= \texttt{nil} \quad \text{otherwise.}$$

Recall that *loc* is the TLAM function the gives the node on which an actor is located.

5.3.3 *QoS Meta-Actor States and Messages.* We represent the QoS meta actors' knowledge of the MM state as a finite function from received requests (represented by actors in *ReqActors*) and replicas (represented by pairs (DS, MM)) to a function from annotation tags to corresponding values or `nil` if undefined. We let *MMState* denote this set of functions:

$$MMState \subset ReqActors \cup (DSnodes \times MMObjects) \xrightarrow{f} Tag \xrightarrow{f} TagValue \cup \{\texttt{nil}\},$$

where *TagValue* is the set containing the possible QoS annotation values (not `nil`). We let mms, mms' range over *MMState*. We define the updating operation mms `modby` mms' by

$$(mms \texttt{ modby } mms')(x,t) = \texttt{if } mms'(x,t) \neq \texttt{nil then } mms'(x,t) \texttt{ else } mms(x,t).$$

We also let *reqU*, *replU*, and *mmsU* range over *MMState* where *reqU* is used to update request actor tags, *replU* is used to update replica actor tags, and *mmsU* is used to indicate a general update. For any set of request actors A, mms/A is the restriction of mms to $x \in A$ and for any DS node DS, mms/DS is the restriction of mms to replicas on DS. Thus,

$$(mms/A)(x,t) = \texttt{if } x \in A \texttt{ then } mms(x,t) \texttt{ else nil}$$

$$(mms/DS)((DS', MM), t) = \texttt{if } DS = DS' \texttt{ then } mms((DS, MM), t) \texttt{ else nil}.$$

We use a (nested) finite map representation for MM states and updates where the outer level maps requests and replicas to their associated annotation tag maps. For example,

$$[a = [t_1 = v_1, t_2 = v_2]]$$

denotes the function mms where

$$mms(a, t_1) = v_1$$

$$mms(a, t_2) = v_2$$

$$mms(x,t) = \texttt{nil} \quad \text{if } x \neq a \quad \text{or} \quad t \notin \{t_1, t_2\}.$$

Following the QoS Broker software architecture discussed in Section 2, there are five broker meta-actors residing on the DC node: (a) the main QoS broker *QB*, (b) a request scheduler *RS*, (c) a replication on demand server *ROD*, (d) a dereplication server *DR* and (e) a predictive placement server *PP*. In addition there is a DSNode manager meta-actor $DSma(DS)$ on each DS node *DS*. We represent the state of a QoS broker meta actor using elements of an abstract data type given by their constructors. The dynamic state used by a DS node QoS manager $DSma(DS)$ is stored in base actor annotations. The static state depends only on characteristics of the DS node on which it resides. Thus we

ActorName (Service)	States
QB (QoS Broker)	$QBB(MMState, Status)$
RS (Request Scheduler)	$IdleB_{\mathrm{rs}}, WaitB_{\mathrm{rs}}(MMState, ReqActors, MMState)$
ROD (Replication on Demand)	$IdleB_{\mathrm{rod}}, WaitB_{\mathrm{rod}}(MMState)$
DR (DeReplication)	$IdleB_{\mathrm{dr}}, WaitB_{\mathrm{dr}}(MMState, \mathbf{P}_{\omega}(DSnodes))$
PP (Predictive Placement)	$IdleB_{\mathrm{pp}}, WaitB_{\mathrm{pp}}(MMState, \mathbf{P}_{\omega}(DSnodes))$
$DSma(DS)$ (DSnode QoS mgr)	$DSB(DS)$

Fig. 11. QoS meta actors, services and states.

assume there is a function *DSB* on DS nodes such that $DSB(DS)$ denotes the state of the manager on *DS*. The QoS meta actors, their services and their possible states are summarized in Figure 11.

The QoS broker *QB* coordinates the QoS resource management services: scheduling, replication, predictive placement, and dereplication. Since these activities use and modify the actual MM state, care must be taken to avoid interference among these activities. For this purpose, we define the notion of status function, *status*, that the QoS broker uses to keep track of which processes are ongoing. A status function *status* has domain $\{RS, ROD, DR, PP\}$ and is such that:

—$status(RS) \in ReqActors \cup \{\texttt{nil}\}$
—$status(X) \in \{\texttt{true}, \texttt{false}\}$ for $X \in \{PP, DR, ROD\}$

$status(RS) = (\alpha^{req})$ indicates that *RS* has been requested to schedule α^{req} and scheduling is in progress, with *RS* allowed to invoke *ROD* only if $status(ROD) = \texttt{true}$. $status(RS) = \texttt{nil}$ indicates that there is no outstanding request from *QB* to *RS* and consequently no undelivered messages to or from *RS*. For $X \in \{PP, DR\}$, $status(X) = \texttt{true}$ indicates the process *X* is ongoing and $status(X) = \texttt{false}$ indicates the process *X* is not active and there are no outstanding requests from *QB* to *X* and in fact no undelivered messages to or from *X*. We let *Status* denote the set of status functions and let *status* range over *Status*.

As we will show, the rules for *QB* behavior assure noninterference amongst the QoS broker auxiliary services by maintaining the status invariant property Φ_{status}.

Definition 5.15 (*Status Invariant,* $\Phi_{\mathrm{status}}(status)$). To assure noninterference amongst the QoS broker services, the QoS broker does not allow two services that implement replication to run concurrently or a dereplication service to run concurrently with either scheduling or replication. Note, however, that multiple replications initiated by the same replication service can go on concurrently. Thus the broker status function *status* will satisfy status invariant, $\Phi_{\mathrm{status}}(status)$, which holds just if the following conditions hold:

—$status(PP) = \texttt{true} \Rightarrow status(ROD) = \texttt{false} \wedge status(DR) = \texttt{false}$
—$status(DR) = \texttt{true} \Rightarrow status(ROD) = \texttt{false} \wedge status(PP) = \texttt{false} \wedge status(RS) = \texttt{nil}$

request	reply
$RS \lhd \texttt{schedule}(\alpha^{req}, mms, b) \ @ \ QB$	$QB \lhd \texttt{scheduleReply}(mmsU) \ @ \ RS$
$DR \lhd \texttt{derepl}(mms) \ @ \ QB$	$QB \lhd \texttt{dereplReply}(replU) \ @ \ DR$
$PP \lhd \texttt{place}(mms) \ @ \ QB$	$QB \lhd \texttt{placeReply}(replU) \ @ \ PP$
$DSma(DS) \lhd \texttt{assign}(reqU) \ @ \ RS$	$RS \lhd \texttt{assignAck}() \ @ \ DSma(DS)$
$ROD \lhd \texttt{repl}(mms, MM, qt) \ @ \ RS$	$RS \lhd \texttt{replAck}(replU) \ @ \ ROD$
$DSma(DS) \lhd \texttt{repUpd}(replU) \ @ \ X$	$X \lhd \texttt{repUpdAck}() \ @ \ DSma(DS)$
$DSma(DS) \lhd \texttt{repUpd}(replU) \ @ \ X$	$X \lhd \texttt{repUpdAck}() \ @ \ DSma(DS)$
for $X \in \{ROD, DR, PP\} \ \wedge \ DS \in DSnodes$	

notifications
$QB \lhd \texttt{notify}([(DS, MM) = [State = replCompleted]]) \ @ \ DSma(DS)$
$QB \lhd \texttt{notify}([\alpha^{req} = [State = reqCompleted]]) \ @ \ DSma(DS)$

Fig. 12. Internal QoS Messages. These are classified either as requests with corresponding reply messages, or as notifications, which need no reply. The general form of a message is $X \lhd \texttt{mid}(\cdots) \ @ \ Y$ where X is the intended receiver of the message, Y is the sender, mid is the message type, and $(\cdots)$ contains parameters.

Although simple and effective, this constraint reduces the potential concurrency of the QoS activities. Defining formal restrictions that allow this constraint to be safely and efficiently relaxed is a topic of future work.

Definition 5.16 (*QoS Message*). A *QoS message* is either an internal QoS message or a client interaction message. Internal QoS messages are either requests, replies or notifications sent to a QoS meta actor by a QoS meta actor. The possible forms for internal QoS messages are given in Figure 12. A client interaction message is either a client MM request of the form $QB \lhd \texttt{mmReq}(\alpha_{cl}, MM, qs)$ or a reply of one of the following forms:

$$\alpha_{cl} \lhd \texttt{granted}(MM, qs) \ @ \ QB \quad \text{or} \quad \alpha_{cl} \lhd \texttt{denied}(MM, qs) \ @ \ QB.$$

The replica update message, repUpd, is the only message received by the DSma from ROD, DR and PP. It carries along with it a set of replicas and the replstate to which they must be altered.

The notify messages are just to inform *QB* that some resources are released. Notification is needed because the resources cannot be considered available for reuse until an appropriate notification is received by *QB*. However, a reply not required as there is no need to synchronize changes with DS node managers.

5.3.4 *QoS Meta-Actor Transition Rules.* The transition rules specify the reaction of a meta-actor upon receiving a QoS message, or a base-level event notification, thus defining the possible behaviors of the QoS Broker meta-actors. The rules are expressed both in English and in formal symbolic notation. The reader should feel free to read only one form according to taste. The QoS meta-actor

rules have the general form:

$$(\dagger)\quad \langle a : s\rangle[,\, a \lhd M\,] \xrightarrow[effect]{trigger} \langle a : s'\rangle\ ,\ MC \quad \text{if } cond$$

where $\langle a : s\rangle$ is a QoS meta-actor with name a in state s and $a \lhd M$ is a message to a with content M. ($[\cdots]$ indicates that the message part may be empty.) s' is the new state of actor a and MC is a possibly empty set of messages sent In the QoS rules, the trigger *trigger* is empty except for DS meta actors, where triggers signal progress in replication as request servicing. *effect* is only nonempty for QB and DS meta-actors, where it may include an effect the form $setA(mmsU)$ meaning that annotations of base-level actors on the same node have been modified by applying $mmsU$, and/or an effect $\texttt{new}(\alpha)$ indicating creation of a new base-level actor α. Recall that the events in event diagram figures in Section 5.3.1 are labelled with names of transition rules, to illustrate the emergent control flow.

Transition Rules for QB. The transition rules for the QoS broker, QB, provide the overall organization of the QoS service activities.

5.3.4.1 ($QBmmReq$). If neither dereplication nor request scheduling are in progress then an MM request, $QB \lhd \texttt{mmReq}(\alpha_{cl}, MM, qs)$, can be processed. The rule (QBmmReq) for receipt of an MM request by QB says that upon receipt of an MM request, a request actor α^{req} is created (the effect $\texttt{new}(\alpha^{req})$), its annotations are initialized using $reqU$ (the effect $setA(reqU)$), and a scheduling request message is sent to RS. In the new state of QB, mms' is the result of updating mms using $reqU$.

A message $RS \lhd \texttt{schedule}(mms', \alpha^{req}, rod)$ is sent, where mms' is QB's current model of the MM state augmented with request information associated to the new request actor α^{req}, and rod indicates whether replication-on-demand is enabled for the scheduler. Replication-on-demand may be enabled for the scheduler, if predictive-placement is not in progress.

$$\langle QB : QBB(mms, status)\rangle\ ,\ QB \lhd \texttt{mmReq}(\alpha_{cl}, MM, qs)$$

$$\xrightarrow[\texttt{new}(\alpha^{req});setA(reqU)]{}$$

$$\langle QB : (mms', status[RS = \alpha^{req}][ROD = rod])\rangle\ ,\ RS \lhd \texttt{schedule}(mms', \alpha^{req}, rod)$$

$$\text{if } status(DR) = \texttt{false} \wedge status(RS) = \texttt{nil}$$

$$\begin{aligned}\text{where}\quad & status(PP) = \texttt{true} \Rightarrow rod = \texttt{false}\\ & mms' = mms\ \texttt{modby}\ reqU\\ & reqU = [\alpha^{req} = [ClientId = \alpha_{cl},\\ & \qquad\qquad ObjId = MM,\\ & \qquad\qquad Qos = QoSTranslate(qs),\\ & \qquad\qquad State = Waiting]]\end{aligned}$$

As discussed above, the effect label, $\texttt{new}(\alpha^{req})$, on the transition arrow means that $\alpha^{req} \in ReqActors$ is newly created, and the label, $setA(reqU)$, says that the QoS annotations of α^{req} have been initialized according to the values given in $reqU$.

5.3.4.2 (*QBdsNotify*). A notification from the DS arrives when an ongoing request or an ongoing replication has completed. When a processable DS notification arrives, the QoS broker updates its MM state. The effect of the transition below is to apply the annotation update contained in the message.

$$\langle QB : QBB(mms, status)\rangle \ , \ QB \lhd \texttt{notify}(mmsU) \,@\, DSma(DS)$$

$$\longrightarrow$$

$$\langle QB : QBB(mms \ \texttt{modby} \ mmsU, status)\rangle$$

if notification is processable

A notification is processable unless it is a notification of a change of state for a request before the admission granted update has arrived, or it is a notification of replication completion before the update that starts the replication arrives.

5.3.4.3 (*QBpp*). The QoS broker may initiate predictive placement if dereplication is not in progress and replication-on-demand is not enabled. Note that the enabledness of this rule (and of the **QBdr** rule), depends only on the state of the QoS broker QB, since no message is delivered.

$$\langle QB : QBB(mms, status)\rangle$$

$$\longrightarrow$$

$$\langle QB : (mms, status[PP = \texttt{true}][ROD = \texttt{false}])\rangle \ , \ PP \lhd \texttt{place}(mms)$$

if $status(PP) = status(DR) = \texttt{false}$
$\wedge \ (status(RS) = \texttt{nil} \ \vee \ status(ROD) = \texttt{false})$

5.3.4.4 (*QBdr*). The QoS broker may invoke dereplication if neither predictive placement nor request scheduling is in progress (again, no message is consumed).

$$\langle QB : QBB(mms, status)\rangle$$

$$\longrightarrow$$

$$\langle QB : (mms, status[DR = \texttt{true}])\rangle \ , \ DR \lhd \texttt{derepl}(mms)$$

if $status(PP) = status(DR) = \texttt{false} \ \wedge \ status(RS) = \texttt{nil}$

5.3.4.5 (*QBrsReply*). When a reply to an outstanding scheduling request arrives, QB updates its state using the update MM state contained in the reply message and sends a reply to the requesting client indicating whether the request has been granted or denied.

$$\langle QB : QBB(mms, status[RS = \alpha^{req}])\rangle \ , \ QB \lhd \texttt{scheduleReply}(mmsU) \,@\, RS$$

$$\xrightarrow[setA(mmsU_{/\alpha^{req}})]{}$$

$$\langle QB : QBB(mms \ \texttt{modby} \ mmsU, status[RS = \texttt{nil}])\rangle \ , \ \alpha_{cl} \lhd X(\alpha^{req})$$

where

$mms(\alpha^{req})(ClientId) = \alpha_{cl}$

$mmsU(\alpha^{req})(State) = Granted \Rightarrow X = \texttt{granted}$

$mmsU(\alpha^{req})(State) = Denied \Rightarrow X = \texttt{denied}$

5.3.4.6 (*QBdrReply*) (*QBppReply*). When a reply to a predictive placement or dereplication request arrives the QoS broker updates its MM state and status.

$$\langle QB : QBB(mms, status[DR = \texttt{true}])\rangle \ , \ QB \lhd \texttt{dereplReply}(mmsU) \ @ \ DR$$
$$\longrightarrow$$
$$\langle QB : QBB(mms \ \texttt{modby} \ mmsU, status[DR = \texttt{false}])\rangle$$
$$\langle QB : QBB(mms, status[PP = \texttt{true}])\rangle \ , \ QB \lhd \texttt{placeReply}(mmsU) \ @ \ PP$$
$$\longrightarrow$$
$$\langle QB : QBB(mms \ \texttt{modby} \ mmsU, status[PP = \texttt{false}])\rangle$$

Transition Rules for Scheduling and Placement. We provide the rules for *RS* in detail and summarize the behavior specified by the transition rules for *ROD*, *PP* and *DR*. The complete rules for *ROD*, *PP* and *DR* are given in the appendix.

To simplify the statement of the rules for scheduling and placement, we will define some auxiliary notions that characterize the allowed actions. (Auxiliaries for *ROD*, *PP* and *DR* appear in the appendix along with the rules.) These definitions rely on an extension of the total resource property to MM states. $\phi_{\text{res}}(mms)$ is defined by replacing the request and replica functions by lookup of the corresponding tags in *mms*. The servicing state for streaming is replaced by the admission granted state, since the broker must assume that the allocated resources are in use once admission is granted.

Definition 5.17 (*Scheduling Auxiliaries*).

$$ReplAlloc(mms, DS, Res) = 0 \quad \text{if } Res \neq NetBW$$
$$ReplAlloc(mms, DS, NetBW) = \sum_{MM \in Replicating(mms, DS)} mms((DS, MM), BW)$$
$$ResAlloc(mms, DS, Res) =$$
$$\sum_{\alpha^{req} \in Streaming(mms, DS)} mms(\alpha^{req}, QoS)_{Res} \ + \ ReplAlloc(mms, DS, Res)$$
$$\phi_{\text{res}}(mms, DS, Res) \ \Leftrightarrow \ ResAlloc(mms, DS, Res) \leq capacity(DS, Res)$$
$$\phi_{\text{res}}(mms) \ \Leftrightarrow \ (\forall DS \in DSnodes, Res \in Resources)\phi_{\text{res}}(mms, DS, Res),$$

where

$$Replicating(mms, DS) = \{MM \in MMObjects \mid$$
$$mms((DS, MM), State) = InProgress\}$$

$$Streaming(mms, DS) = \{\alpha^{req} \in ReqActors \cap \mathrm{Dom}(mms) \mid mms(\alpha^{req}, State) = Granted \wedge mms(\alpha^{req}, Replica) = DS\}.$$

To state the rules for RS, we introduce two functions. $assignTo(mms, \alpha^{req}, DS)$ is the request and replication update for assigning the request represented by α^{req} in mms to DS.

$$assignTo(mms, \alpha^{req}, DS) = (mms/\alpha^{req})[\alpha^{req} = [State = Granted, Replica = DS], (DS, mms(\alpha^{req}, ObjId)) = [Class = 1]]$$

$assignable(mms, \alpha^{req})$ is the set of DS nodes to which the request represented by α^{req} in mms could be assigned without violating any resource constraints using only existing replicas.

$$assignable(mms, \alpha^{req}) = \{DS \in DSnodes \mid \phi_{\mathrm{res}}(mms \texttt{ modby } assignTo(mms, \alpha^{req}, DS)) \wedge mms(DS, mms(\alpha^{req})(ObjId))(Class) \neq 0\}.$$

5.3.4.7 *Rules for RS.* The transition rules for the request scheduler, *RS*, deal with scheduling requests, and interactions of *RS* with the DS node managers and with *ROD*.

5.3.4.8 (*RSscheduleGrant*). If there is some DS node to which the request represented by α^{req} can be assigned without violating resource constraints then RS picks one and notifies the DS manager of that node.

$$\langle RS : IdleB_{\mathrm{rs}} \rangle \ , \ RS \lhd \texttt{schedule}(mms, \alpha^{req}, rod)$$

$$\longrightarrow$$

$$\langle RS : WaitB_{\mathrm{rs}}(mms, \alpha^{req}, mmsU) \rangle \ , \ DSma(DS) \lhd \texttt{assign}(mmsU) \ @ \ RS$$

if $DS \in assignable(mms, \alpha^{req}) \ \wedge \ mmsU = assignTo(mms, \alpha^{req}, DS)$.

5.3.4.9 (*RSscheduleDeny*). If there is no DS node on which the request represented by α^{req} can be scheduled without violating resource constraints and ROD is disabled, then RS sends the broker a denied update.

$$\langle RS : IdleB_{\mathrm{rs}} \rangle \ , \ RS \lhd \texttt{schedule}(mms, \alpha^{req}, \texttt{false})$$

$$\longrightarrow$$

$$\langle RS : IdleB_{\mathrm{rs}} \rangle \ , \ QB \lhd \texttt{scheduleReply}([\alpha^{req} = [State = Denied]]) \ @ \ RS$$

if $assignable(mms, \alpha^{req}) = \emptyset$.

5.3.4.10 (*RSscheduleRod*). If there is no DS node on which the request represented by α^{req} can be scheduled without violating resource constraints and ROD is enabled, then RS requests replication meeting the QoS requirements

of the request specified by qt.

$$\langle RS : IdleB_{\mathrm{rs}} \rangle \ , \ RS \lhd \texttt{schedule}(mms, \alpha^{req}, \texttt{true})$$

$$\longrightarrow$$

$$\langle RS : WaitB_{\mathrm{rs}}(mms, \alpha^{req}, \texttt{nil}) \rangle \ , \ ROD \lhd \texttt{repl}(mms, MM, qt) \ @ \ RS$$

$$\text{if } MM = mms(\alpha^{req}, ObjId) \ \wedge \ qt = mms(\alpha^{req}, QoS) \wedge assignable(mms, \alpha^{req}) = \emptyset.$$

5.3.4.11 (*RSdsAck*). When RS receives an acknowledgment to an outstanding assignment request to a DS node, then it sends a granted reply to the broker with the updated MM state, $mmsU$. We assume $mmsU$ contains the update information for α^{req} and any replication update that has been done.

$$\langle RS : WaitB_{\mathrm{rs}}(mms, \alpha^{req}, mmsU) \rangle \ , \ RS \lhd \texttt{assignAck}() \ @ \ DSma(DS)$$

$$\longrightarrow$$

$$\langle RS : IdleB_{\mathrm{rs}} \rangle \ , \ QB \lhd \texttt{scheduleReply}(mmsU) \ @ \ RS.$$

5.3.4.12 (*RSrodReplyOk*). If RS receives a replication update from an outstanding request to ROD that allows the request it is attempting to schedule to be granted, then RS picks a suitable DS node and sends a corresponding assign request to the nodes DS manager.

$$\langle RS : WaitB_{\mathrm{rs}}(mms, \alpha^{req}, \texttt{nil}) \rangle \ , \ RS \lhd reply(replU) \ @ \ ROD$$

$$\longrightarrow$$

$$\langle RS : WaitB_{\mathrm{rs}}(mms, \alpha^{req}, replU \ \texttt{modby} \ mmsU) \rangle \ , \ DSma(DS) \lhd \texttt{assign}(mmsU)$$

$$@RS \quad \text{if } DS \in assignable(mms \ \texttt{modby} \ replU, \alpha^{req})$$

$$\text{and} \quad mmsU = assignTo(mms \ \texttt{modby} \ replU, \alpha^{req}, DS).$$

5.3.4.13 (*RSrodReplyFail*). When RS receives a replication update from ROD that does not allow its request to be granted, then RS replies to QB with an MM state indicating request denied.

$$\langle RS : WaitB_{\mathrm{rs}}(mms, \alpha^{req}, \texttt{nil}) \rangle \ , \ RS \lhd \texttt{reply}(replU) \ @ \ ROD$$

$$\longrightarrow$$

$$\langle RS : IdleB_{\mathrm{rs}} \rangle \ , \ QB \lhd \texttt{scheduleReply}(replU \ \texttt{modby} \ [\alpha^{req} = [State = Denied]])$$

$$@RS \quad \text{if } assignable(mms \ \texttt{modby} \ replU, \alpha^{req}) = \emptyset.$$

5.3.4.14 *Rules for PP, ROD, DR.* When ROD receives a request, $ROD \lhd \texttt{repl}(mms, MM, qt) \ @ \ RS$ for replication of MM object, MM, with QoS resource requirements qt, it looks, using mms, for a DS node that doesn't have the needed MM object and that has the required resources available. If one is found, a replication request is sent to that DS node and ROD waits for an acknowledgment. When the acknowledgment message is received, a reply is sent to RS with MM state containing the replica update information. If no such DS node is found, then a failure reply is sent to RS.

Based on the information in the MM state of a `place` request, *PP* may decide to reclassify some replicas from 0, 2, or 3 to 1, and in the case of moving from 0 to 1 initiate replication. It then notifies each DS node of any changes on that node, waits for acknowledgments from these DS nodes, and then sends a reply to *QB* containing the updated replica state. Similarly, upon receiving a dereplication request, *DR* may decide to reclassify some replicas from 1 to 2. It then notifies each DS node of any changes on that node, waits for acknowledgments, and then sends a reply *QB* containing the replica state update. (Note that in practice, *PP* and *DR* also use other information for predicting requests that we do not model at this level of abstraction.)

Transition Rules for DS Manager

5.3.4.15 (*DSassign*). When a DS node manager $DSma(DS)$ receives an assignment request with MM state $mmsU$ it creates a new request actor, sets the annotations of this actor and of the replica actor for the requested MM object using $mmsU$ (which contains the MM request information, and name of the associated request actor), and sends an `assignAck` reply to RS.

$$\langle DSma(DS) : DSB(DS)\rangle, DSma(DS) \lhd \texttt{assign}(mmsU) \mathbin{@} RS$$

$$\xrightarrow[\texttt{new}(\alpha^{\mathrm{req,ds}}),setA(mmsU')]{}$$

$$\langle DSma(DS) : DSB(DS)\rangle, RS \lhd \texttt{assignAck}(reqU) \mathbin{@} DSma(DS)$$

$$\text{if } mmsU' = [\alpha^{\mathrm{req,ds}} = reqU(\alpha^{req}) \texttt{ modby } [ReqId = \alpha^{req}]] \texttt{ modby } mmsU/DS$$

$$\wedge\, \alpha^{req} \in \mathrm{Dom}(mmsU).$$

5.3.4.16 (*DSrepUpd*). When a DS node manager $DSma(DS)$ receives a replication request it uses the MM state replica information to update the annotations of its replica actors and then sends a `replAck` reply to the requester (which could be *ROD*, *PP*, or *DR*).

$$\langle DSma(DS) : DSB(DS)\rangle\ ,\ DSma(DS) \lhd \texttt{repUpd}(replU) \mathbin{@} X$$

$$\xrightarrow[setA(replU)]{}$$

$$\langle DSma(DS) : DSB(DS)\rangle\ ,\ X \lhd \texttt{replAck}() \mathbin{@} DSma(DS)$$

$$\text{for} \quad X \in \{ROD, DR, PP\}.$$

5.3.4.17 (*DSstartRepl*). Similarly, when replication of an MM object on a DS nodes starts (or completes) the replica actor annotations are updated, and a notification is sent to *QB*.

$$\langle DSma(DS) : DSB(DS)\rangle$$

$$\xrightarrow[setA(reqU)]{\texttt{replStarts}(\alpha)}$$

$$\langle DSma(DS) : DSB(DS)\rangle\ ,\ QB \lhd \texttt{notify}(reqU') \mathbin{@} DSma(DS)$$

where

$reqU = [\alpha = [State = InProgress]]$

$reqU' = [(DS, MM) = [State = InProgress]] \wedge NodeRepl(DS, MM) = \alpha.$

5.3.4.18 (*DSreplComplete*). When replication of an MM object on a DS node completes, the replica actor annotations are updated, and a notification is sent to QB. Also, in the case of completion, if any requests are waiting for this completion they are moved from *Granted* state to *Servicing* state.

$$\langle DSma(DS) : DSB(DS)\rangle \xrightarrow[setA(replU\ \texttt{modby}\ reqU)]{\texttt{replCompletes}(\alpha)} \langle DSma(DS) : DSB(DS)\rangle, QB \lhd notify(replU) @ DSma(DS),$$

where

$\alpha = NodeRepl(DS, MM)$

$replU = [(DS, MM) = [State = replCompleted]]$

$$reqU = [\ \alpha^{\mathrm{req,ds}} = [State = Servicing] \mid \alpha^{\mathrm{req,ds}} \in DSReqActors \\ getA(\alpha^{\mathrm{req,ds}}, State) = Granted \\ \wedge getA(\alpha^{\mathrm{req,ds}}, Replica) = DS \\ \wedge getA(\alpha^{\mathrm{req,ds}}, ObjId) = MM].$$

5.3.4.19 (*DSreqComplete*). When servicing of a request with request actor $\alpha^{\mathrm{req,ds}}$ completes on a DS node, an event $\texttt{reqCompletes}(\alpha^{\mathrm{req,ds}})$ is signaled. The DS node manager then updates the annotations of $\alpha^{\mathrm{req,ds}}$ to record the completion, and sends a notification to QB with the state update for the request actor associated to this request.

$$\langle DSma(DS) : DSB(DS)\rangle \xrightarrow[setA(reqU)]{\texttt{reqCompletes}(\alpha^{\mathrm{req,ds}})} \langle DSma(DS) : DSB(DS)\rangle\ ,\ QB \lhd \texttt{notify}(reqU') @ DSma(DS),$$

where

$reqU = [\alpha^{\mathrm{req,ds}} = [State = reqCompleted]]$

$reqU' = [\alpha^{req} = [State = reqCompleted]] \wedge \alpha^{req} = getA(\alpha^{\mathrm{req,ds}}, ReqId.)$

5.3.5 *Formal Specification of the Resource-Based MM Behavior.* Given the above definitions, we can now define the Resource-Based MM Behavior Specification, the initial and noninterference conditions, and the theorems connecting the behavior, system-wide, and end-to-end viewpoints.

Definition 5.18 (*Resource-Based MM Behavior*). A system S has *Resource-Based MM Behavior* with respect to the underlying system architecture (DS

Nodes and capacity function), the QoS meta actors QB, RS, ROD, DR, PP, and the DS node managers
$DSma(DSnodes)$, if

—for $C \in S$, the state of X in C is appropriate for X in accordance with Figure 11, for X one of QB, RS, ROD, DR, PP, or $DSma(DS)$) for $DS \in DSnodes$. For example, $getState(C, QB) = QBB(mms, status)$ for some MM state function mms and status function $status$.
—every computation π of S obeys the transition rules for QoS meta-actors discussed above and guarantees termination of servicing and replication-in-progress states;
—for any transition $\tau = C \longrightarrow C'$ of S and any replica actor α and request actor $\alpha^{\mathrm{req,ds}}$
  —`replStarts`(α) is an event of τ only if $getA(C, \alpha, State) = InQueue$,
  —`replCompletes`(α) is an event of τ only if $getA(C, \alpha, State) = InProgress$,
  —`reqCompletes`($\alpha^{\mathrm{req,ds}}$) is an event of τ only if $getA(C, \alpha^{\mathrm{req,ds}}, State) = Servicing$.

To state the "Resource-based MM Behavior provides QoS-based MM Service" theorem, it is necessary to specify the conditions under which this resource-based MM behavior is expected to provide QoS-based MM service. For this purpose, we define the requirements for QoS Initial configurations and QoS Non-Interference. QoS Initial configurations are those in which no QoS meta-activity is going on. We will restrict attention to system configurations reachable from an initial QoS state and analyze the properties of such configurations.

Definition 5.19 (*QoS Initial Condition*). $QoSInitial(C)$ holds just if

—If mms is the status function of QB in C, then mms says that there are no active processes:
$status(RS) =$ `nil` and $status(DR) = status(PP) =$ `false`,
—RS, ROD, DR, PP are Idle,
—there are no undelivered internal QoS messages (see Definition 5.16) in C.

QoS nonInterference expresses constraints on the environments in which the QoS system can operate safely. In particular, it constrains the activity of meta-actors other than the QoS meta-actors.

Definition 5.20 (*QoS Noninterference Requirement*). S satisfies the *QoS-Broker Noninterference Requirement* iff meta transitions with focus actor not a QoS meta-actor obey the following constraints:

—Neither QoS annotations nor the state of actors in
$ReqActors$, $DSReqActors$, or $DSReplActors$ are modified.
—No resource dedicated to QoS is used.
—No internal QoS messages (Definition 5.16) are sent.

THEOREM 5.21 (RESOURCE-BASED MM BEHAVIOR IMPLIES RESOURCE-BASED MM SERVICE). *If a system S satisfies the following requirements:*

—*S has Resource-based MM behavior (Definition 5.18),*

—*S satisfies the QoS Noninterference Requirement (Definition 5.20),*

—*Every $C \in S$ is reachable from a configuration satisfying the QoS Initial conditions (Definition 5.19),*

—*QoSTranslate satisfies the QoSTranslate requirements (Definition 5.3),*

then S provides Resource-based MM Service (Definition 5.11) with respect to the given functions QoSTranslate and capacity, with MMreqset being messages of the form $QB \lhd \texttt{mmReq}(\alpha_{cl}, MM, qs)$, and replica and request functions defined in terms of annotations according to Definition 5.14.

PROOF (SEE SECTION 5.4 FOR DETAILS). The proof is based on two key properties of a system satisfiying the hypotheses of the theorem. The first is that the model QB has of the system state (embodied in the two parameters, *mms* and *status*, of the *QB* state) is sufficiently accurate. The second is that *QB* is always eventually enabled to receive an MM request. The first property is established by Lemma 5.24, Corollary 5.26, and Lemma 5.27. These lemmas combine to show that (for X one of *RS*, *PP*, *DR*, or *ROD*)

—If *QB* thinks that X not active, then X is idle and there are no undelivered messages between *QB* and X,

—If *QB* thinks X is active, then either

  —X is idle and there is an undelivered request to X from *QB*, or there is an undelivered reply from X to *QB*, or

  —X is waiting and there is an undelivered message to or from the actor that X is waiting for.

—Furthermore, the *mms* component of *QB*'s state, combined with any pending updates gives an accurate representation of the request and replica states on the *DS* nodes.

These lemmas further show that the *mms* component of *QB*'s state and the changes due to receipt of updates satisfy the configuration and transition constraints on resource allocation, request and replicat state. Lemma 5.24 is typical of lemmas that must be established in reasoning about TLAM behaviors. It spells out detailed invariants on meta-actor states, base-level annotations, and information contained in undelivered messages.

The second property is established by showing that any ongoing scheduling, replication, and dereplication activities eventually terminate (Lemma 5.28) and thus *QB* will eventually be enabled (Lemma 5.29). The proof of these lemmas depends crucially on the invariants lemma (Lemma 5.24). □

THEOREM 5.22 (RESOURCE-BASED MM BEHAVIOR IMPLIES QOS-BASED MM SERVICE). *If a system S satisfies the premises of Theorem* 5.21*, then S provides QoS-based MM Service (Definition* 5.1*) with respect to the given functions QoSTranslate and capacity, with MMreqset being messages of the form $QB \lhd \texttt{mmReq}(\alpha_{cl}, MM, qs)$.*

PROOF By Theorems 5.21 and 5.12. □

5.4 Proving Resource-Based MM Behavior Implies Resource-Based Service

In the following, we assume S is a system that satisfies the hypotheses of Theorem 5.21 and we restrict C to range over S. Thus, S has Resource-based MM behavior, all configurations are reachable from a QoS initial configuration, and there is no interference from non-QoS meta actors. We begin with some lemmas. Lemma 5.24 states some MM state invariants of configurations of systems satisfying the above hypotheses. It is the key to establishing that the state of QB represents a sufficiently accurate model of the system MM state. Lemma 5.29 states that under these conditions QB will always eventually be ready to receive a MM request. Lemma 5.28 says that if a scheduling request has been sent to RS then eventually a reply will be sent by RS. This is used to establish Lemma 5.29.

Definition 5.23 (*Auxiliaries*). To state the lemmas, we introduce some notation for extracting information from a configuration. *getMms* and *getStatus* extract the two parameters of the state of QB. *QoSannot* extracts the the QoS annotations from a base actor, *getRepl* extracts the QoS annotations for replica. *rsGranted* characterizes MM state updates that correspond to possible request grants by RS. Recall that $getState(C, \alpha)$ is the state of actor α in configuration C.

—$getMms(C) = mms$ if $getState(C, QB) = QBB(mms, status)$.
—$getStatus(C) = status$ if $getState(C, QB) = QBB(mms, status)$.
—$QoSannot(C, \alpha)$ is the annotation function mapping QoS annotation tags to their values as annotations of α in C.
—$getRepl(C, DS) = [(DS, MM) = QoSannot(C, NodeRepl(C, DS, MM)) \mid MM \in MMObjects]$.
—$getRepl(C) = [getRepl(C, DS) \mid DS \in DSnodes]$.
—$rsGranted(mms, \alpha^{req}, mmsU)$ holds just if $mmsU = reqU$ `modby` $replU$ such that
  —$replU =$ `nil` and $reqU = assignTo(mms, \alpha^{req}, DS)$
    for some $DS \in assignable(mms, \alpha^{req})$, or
  —$assignable(mms, \alpha^{req}) = \emptyset$ and
    $replU = rodRepl(DS', MM, bw)$ for some $(DS', bw) \in rodOk(mms, MM, qt)$
    and
    $reqU = assignTo(mms$ `modby` $replU, \alpha^{req}, DS)$
    for some $DS \in assignable(mms$ `modby` $replU, \alpha^{req})$
    where $MM = mms(\alpha^{req}, ObjectId)$ and $qt = mms(\alpha^{req}, QoS)$.

Recall that *assignable* and *assign To* are defined in Definition 5.17. Definitions of *rodOk*, and *rodRepl* can be found in the appendix. Note that with notation as in *rsGranted*, $reqU(\alpha^{req}, State) = Granted$, and given the rules for RS and ROD behavior, it will be the case that $DS = DS'$ since no other assignment will be possible.

Lemma 5.24 lists out MM state invariants of configurations needed to establish a sufficiently accurate model of the system MM state at the QoS broker.

We enumerate three specific conditions—the status invariants, the idle condition invariants and the active state invariants. (I) expresses constraints on concurrent scheduling, replication, and dereplecation activity. (II) says that if *QB* thinks that a QoS Broker meta-actor is not active, then it is idle and there are no undelivered messages to or from that actor. (III) says that if *QB* thinks that a QoS Broker meta-actor is active, then either there is an undelivered message to or from that actor, or it is waiting for a response from some other meta-actor before replying to QB and again there is exactly one undelivered message to/from that meta-actor and the one it is waiting for.

LEMMA 5.24 (QOSB INVARIANTS). *Assume $C \in S$ with S satisfying the hypotheses of Theorem* 5.21. *Let status = getStatus(C), mms = getMms(C), and U be the multiset of undelivered messages of C targeted to QoS meta actors, then* (I), (II), (III) *below must hold, where:*

(I) [status invariant] one of the following three cases holds
  (1) [enabled] $status(RS) = \texttt{nil}$ and $status(DR) = \texttt{false}$, or
  (2) [DRon] $status(RS) = \texttt{nil}$, $status(PP) = \texttt{false}$ and $status(DR) = \texttt{true}$
  (3) [RSon] $status(RS) = \alpha^{req}$, $status(DR) = \texttt{false}$, $mms(\alpha^{req}, State) = Waiting$,
  and $mms(\alpha^{req}, Replica) = \texttt{nil}$
  Furthermore, (3) splits into two subcases:
  (3.1) [RODenabled-PPoff] $status(ROD) = \texttt{true}$ and $status(PP) = \texttt{false}$, or
  (3.2) [RODdisabled] $status(ROD) = \texttt{false}$
  Note that if $status(RS) = \texttt{nil}$ then $status(ROD)$ is not relevant. Also in (1) and (3.2) $status(PP)$ is not constrained, it can be `true` or `false`.

(II) [idle conditions]
  —if $status(PP) = \texttt{false}$, then $getState(C, PP) = IdleB_{\text{pp}}$ and U contains no message to or from PP
  —if $status(DR) = \texttt{false}$, then $getState(C, DR) = IdleB_{\text{dr}}$ and U contains no message to or from DR
  —if $status(RS) = \texttt{nil}$, then $getState(C, RS) = IdleB_{\text{rs}}$, $getState(C, ROD) = IdleB_{\text{rod}}$ and U contains no message to or from RS or ROD

(III) [active conditions]
  (pp) if $status(PP) = \texttt{true}$, then one of the following cases holds
    (pp.1) $getState(C, PP) = IdleB_{\text{pp}}$, and U contains exactly one message to/from PP. This message has one of the forms

$$PP \lhd \texttt{place}(mms) \,@\, QB \quad \text{or} \quad QB \lhd \texttt{placeReply}(replU) \,@\, PP$$

    where $replU = ppRepl(mms, P)$ and mms `modby` $replU$ is $getRepl(C)$ [up to pending notifications] for some $P \subseteq (DSnodes \times MMObjects \times Unit_{NetBW})$ such that $ppOk(mms, P)$.
    (pp.2) $getState(C, PP) = WaitB_{\text{pp}}(replU, dsout)$,
    U contains no messages to/from PP that are from/to QB, and $replU = ppRepl(mms, P)$ for some $P \subseteq DSnodes \times MMObjects \times Unit_{NetBW}$ such that $ppOk(mms, P)$, and

—for each $DS \in dsout$, U contains exactly one message to/from PP from/to $DSma(DS)$, either
$DSma(DS) \lhd \texttt{repl}(replU/DS) @ PP$ and mms/DS is $getRepl(C, DS)$, or
$PP \lhd \texttt{replAck}() @ DSma(DS)$ and mms/DS is $getRepl(C, DS)$, and
—for $DS \notin dsout$, $(mms \texttt{ modby } replU)/DS$ is $getRepl(C, DS)$ [with all replica state comparisons modulo outstanding replica notifications]

(dr) the case $status(DR) = \texttt{true}$ is similar to $status(PP) = \texttt{true}$

(rs) if $status(RS) = \alpha^{req}$, then one of rs.1, rs.2, rs.3 holds, where

$(rs.1)$ $getState(C, RS) = IdleB_{\text{rs}}$, and U contains exactly one message to or from RS:
—$RS \lhd \texttt{schedule}(\alpha^{req}, mms, status(ROD)) @ QB$, or
—$QB \lhd \texttt{scheduleReply}(mmsU) @ RS$
and if U contains $QB \lhd \texttt{scheduleReply}(mmsU) @ RS$,
then there is some $(reqU, replU)$
such that $mmsU = reqU \texttt{ modby } replU$ and rs.1.d or rs.1.g holds, where
$(rs.1.d)$ $reqU(\alpha^{req}, State) = Denied$, $replU = \texttt{nil}$ and $assignable(mms, \alpha^{req}) = \emptyset$
$(rs.1.g)$ $rsGranted(mms, \alpha^{req}, mmsU)$

$(rs.2)$ $getState(C, RS) = WaitB_{\text{rs}}(mms, \alpha^{req}, reqU \texttt{ modby } replU)$ with $reqU \neq \texttt{nil}$ (granted, waiting for DSack) such that
—$rsGranted(mms, \alpha^{req}, mmsU)$, and
—U contains exactly one message to/from RS which, letting $DS = mmsU(\alpha^{req}, Replica)$, is either
$DSma(DS) \lhd \texttt{assign}(mmsU) @ RS$ or
$RS \lhd \texttt{assignAck}() @ DSma(DS)$

$(rs.3)$ $getState(C, RS) = WaitB_{\text{rs}}(mms, \alpha^{req}, \texttt{nil})$, $status(ROD) = \texttt{true}$ $assignable(mms, \alpha^{req}) = \emptyset$, and one of two cases holds:
—$getState(C, ROD) = IdleB_{\text{rod}}$ and U contains exactly one message to/from RS which is either
$ROD \lhd \texttt{repl}(mms, MM, qt) @ RS$, or
$RS \lhd \texttt{replyAck}(replU) @ ROD$
where $replU = rodRepl(DS, MM, bw)$
for some $(DS, bw) \in rodOk(mms, MM, qt)$, $MM = mms(\alpha^{req}, ObjectId)$ and $qt = mms(\alpha^{req}, QoS)$
—$getState(C, ROD) = WaitB_{\text{rod}}(replU)$ with $replU$ as above, and U contains exactly one message to/from ROD which is either
$DSma(DS) \lhd \texttt{repl}(replU) @ ROD$ or
$ROD \lhd \texttt{replAck}() @ DSma(DS)$.

Furthermore, if (rs.1) or (rs.2) holds, then $getState(C, ROD) = IdleB_{\text{rod}}$ and there are no undelivered messages to or from ROD,

PROOF (OF LEMMA 5.24). Let C be reachable from C_0 via the transitions

$$[\tau_i : C_i \longrightarrow C_{i+1} \mid i < n]$$

where C_0 satisfies the QoS Initial conditions and $C_n = C$. We show the conditions hold for C by induction on n. If $n = 0$, we are done by definition of QoS Initial since in this case (I.1) holds with $status(PP) = \texttt{false}$.

Now assume that the invariants hold for C_j. We must show they hold for C_{j+1}. If τ_j is a non-QoS transition then by the noninterference assumption we are done, since nothing that the invariants depend upon can be changed. Thus, we need only consider transitions arising from application of one of the QoS meta-actor rules. We will discuss only a representative sample of such transitions.

QBmmreq. *QB* received an MM request. For Lemma 5.24(I), we see that (1) holds for C_j and (3) holds for C_{j+1}. For (II, III), the conditions for *PP*, *DR*, *ROD* are unchanged and *RS* must be idle in both configurations. In C_j, there are no undelivered messages for *RS*, so the one message clause holds in C_{j+1}.

QBdsNotify. *QB* receives a notification. The updates contained in a processable notification do not invalidate any ongoing scheduling or replication decisions, and they do not otherwise effect the invariant properties.

RSschedule. *RS* receives a scheduling request. For a scheduling request to be in the set of undelivered messages, (Lemma 5.24, I.3) must hold. The properties relevant for (I), (II), and (III—PP,DR) are unchanged by the transition. Furthermore, in C_j, (rs.1) must be the case. If the rule applied is **RSscheduleGrant**, then (rs.2) holds in C_{j+1} (using the property that the schedule request was the only undelivered message to/from *RS* in C_j). If the rule applied is **RSscheduleDeny**, then (rs.1) holds in C_{j+1}. If the rule applied is **RSscheduleRod**, then (rs.3) holds in C_{j+1} (using the fact that *ROD* is idle with no undelivered messages to/from *ROD* in C_j).

RSdsReply. *RS* receives an assignment acknowledgment from some *DS*. Again (I.3) must hold, and the properties relevant for (I), (II), and (III—PP,DR) are unchanged by the transition. If there is an undelivered message from a *DS* manager to *RS*, then (rs.2) must hold in C_j and in C_{j+1} (rs.1g) must hold.

DSassign. *DSma*(*DS*) receives an assignment request from *RS*. The only property effected is (III.rs). If there is an undelivered message to *DSma*(*DS*) from *RS* then (rs.2) must hold in C_j and continues to hold in C_{j+1} with the message to *DSma*(*DS*) from *RS* then replace by a message from *DSma*(*DS*) to *RS*. □

Definition 5.25 (*Pending Update*). A pending update is a change in the MM state at some DS node but the update report is in an undelivered message or is being held in the state of some QoS broker meta-actor, and thus has not yet been reported to *QB*. More precisely, we say that *mmsU* is pending in C (with U the undelivered QoS messages of C) if

(1) $mmsU$ is the update part of a message in U to QB from RS, PP, or DR, or
(2) $getState(C, RS) = WaitB_{\mathrm{rs}}(mms, \alpha^{req}, mmsU)$ and there is an `assignAck` to RS in U, or
(3) $getState(C, RS) = WaitB_{\mathrm{rs}}(mms, \alpha^{req}, replU$ `modby` $reqU)$ and $mmsU = replU$ and there is an `assign` from RS in U ($replU$, if nonempty, has been done by ROD), or
(4) $getState(C, ROD) = WaitB_{\mathrm{rod}}(replU)$ and there is a `replAck` to ROD in U, or
(5) $getState(C, X) = WaitB_{\mathrm{x}}(replU, dsout)$ and $mmsU$ is $replU/(DSnodes—D)$ where $DS \in D$ if there is a `repl` to DS (from X) in U (the rest of $replU$ has been done) for $X \in \{PP, DR\}$
(6) $mmsU$ is the update of a notification to QB in U.

COROLLARY 5.26 (QoSB INVARIANTS). *Let C be a configuration in S, $mms = getMms(C)$, and let $mmsU$ be the combined pending updates in C. Then,*

—*there are never any conflicting pending updates in C.*
—*for any pending $mmsU$ the transition $mms \longrightarrow mms$* `modby` *$mmsU$ obeys the request-replica transition constraints.*
—*if $mmsU$ is the union of the pending updates, then mms* `modby` *$mmsU$ satisfies the request, replica, and resource configuration constraints.*

LEMMA 5.27 (MMS ACCURACY). *Let C be a configuration in S, $mms = getMms(C)$, and let $mmsU$ be the combined pending updates in C. Then mms* `modby` *$mmsU$ gives an accurate model of the MM state.*

—$replX(C, DS, MM) = (mms$ `modby` $mmsU)(DS, MM)(X)$ *for X one of State, BW, Class*
—$reqX(C, \alpha^{req}) = (mms$ `modby` $mmsU)(\alpha^{req})(X)$ *for X one of ClientId, ObjId, QoS, State, Replica and α^{req} an existing request actor.*

PROOF (OF LEMMA 5.27). By induction on the number of transitions from a QoS Initial configuration using the QoSB invariants Lemma (5.24) and Corollary (5.26). □

To prove the QB progress Lemma 5.29, we need to establish progress properties for RS, ROD, and DR.

LEMMA 5.28 (RS,ROD,DR PROGRESS). *For any computation path π of S, and for any stage i of π, with source configuration C_i the following hold:*

(DR) If $getState(C_i, DR) = WaitB_{\mathrm{dr}}(replU, dsout)$, then there is a stage $i + j$ such that $getState(C_i, DR) = IdleB_{\mathrm{dr}}$ and a message $QB \lhd$ `dereplReply`*$(replU)$ @DR is pending.*
(ROD) If $getState(C_i, ROD) = WaitB_{\mathrm{rod}}(replU)$, then there is a stage $i + j$ such that $getState(C_i, ROD) = IdleB_{\mathrm{rod}}$ and a message $RS \lhd$ `replAck`*$(replU)$@ROD is pending.*

(RS) If $getState(C_i, RS) = WaitB_{\mathrm{rs}}(mms, \alpha^{req}, mmsU)$*, then there is a stage* $i + j$ *such that* $getState(C_i, RS) = IdleB_{\mathrm{rs}}$ *and a message* $QB \triangleleft \texttt{scheduleReply}\ (mmsU)\ @\ RS$ *is pending.*

PROOF OF LEMMA 5.28. Using the Invariants Lemma (5.24), we see that, if *ROD* or *DR* is waiting, then there is a pending message that will cause that process to send a reply to *RS* or *QB*, respectively, and become idle, or there are pending messages to some *DSma*(*DS*) whose replies will do this. Since *DSma*(*DS*) messages are never disabled they will be delivered and replied to (by system fairness and the transition rules). If *RS* is waiting with $mmsU = \texttt{nil}$, then by the the Invariants lemma and the above argument, it will eventually receive a reply from *ROD* and either become idle after replying to *QB* (request denied), or become waiting with $mmsU \neq \texttt{nil}$. If *RS* is waiting with $mmsU \neq \texttt{nil}$, then there is either an undelivered `assignAck` for *RS* from some *DSma*(*DS*) or an undelivered `assign` request to some *DSma*(*DS*) which will eventually be delivered and whose delivery will result in the sending of an `assignAck`. By system fairness and the rules for *RS*, pending `assignAck` for *RS* will be delivered, and *RS* will become idle, having sent a `scheduleReply` to *QB*. □

LEMMA 5.29 (QB PROGRESS). *For any computation path* π *of S, and for any stage i of* π *there is a stage* $i + j$ *such that invariant case (Lemma* 5.24, I.1*) holds (thus QB is enabled for request delivery).*

PROOF OF LEMMA 5.29. Now we prove Lemma 5.29. Let *C* be the configuration at stage *i* in π and let $status = getStatus(C)$. If the invariant case (Lemma 5.24, I.1) does not hold, then either $status(DR) = \texttt{true}$ or $status(RS) \in ReqActors$, but not both. Using Lemma 5.28, the argument for the two situations is similar. We consider the situation in which $status(DR) = \texttt{true}$. There are three subcases:

(1) $getState(C, DR) = IdleB_{\mathrm{dr}}$ and there is an undelivered `derepl` to *DR*.
(2) $getState(C, DR) = WaitB_{\mathrm{dr}}(replU, dsout)$.
(3) $getState(C, DR) = IdleB_{\mathrm{dr}}$ and there is an undelivered `dereplReply` to *QB*.

If (1) holds, then by message fairness, the `derepl` request will be delivered and then (2) holds. If (2) holds, then by Lemma 5.28, eventually (3) holds. If (3) holds, then by message fairness the `dereplReply` will be delivered and the status of *DR* becomes `false` as desired. □

PROOF OF THEOREM 5.21. Now we complete the proof that the hypotheses of Theorem 5.21 imply that the system provides Resource-based service. By Lemma 5.29 and the rule **QBreqrcv**, requirement (2) of Resource-based service holds. Thus, we need only show that for any configuration *C* of *S*, any transition $\tau = C \longrightarrow C'$, and any computation path π for *C*, requirement (1) holds:

(1.1) ϕ_{repl} holds
  (1.1.1) τ obeys the class diagram requirement
  (1.1.2) τ obeys the rules for *replState* change

(1.1.3) π obeys the progress constraints—by QoS Resource Behavior definition.

(1.2) ϕ_{req} holds

(1.2.1) τ obeys the constancy constraints and rules for change of *reqState*

(1.2.2) π obeys the progress constraints—by QoS Resource Behavior definition, (1.1.3), and transition rules.

(1.3) $\phi_{\text{rr}}(C)$ holds.

(1.4) $\phi_{\text{res}}(C)$ holds.

These follow from the Invariant Lemma (5.24) and Corollary (5.26), the MMS accuracy Lemma (5.27) and transition rules. □

5.5 Implementation of Algorithms within the Broker Framework

In this section, we illustrate how we introduce the implementation of a specific algorithm for resource management into the Resource-Based MM Behavior Specification. As an example, we show how to use the load-factor based adaptive scheduling algorithm to refine the behavior of the request scheduler meta-actor and ensure that required constraints are not violated. To show this is correct, we need only show that the resource-based behavior requirements are met. This follows from the fact that the algorithm meets the constraints implicit in the request scheduler transition rules.

The adaptive scheduling algorithm performs the task of data server selection. It is executed as a single transition step when the request scheduling meta-actor *RS* receives a scheduling request from the QoS Broker *QB*. The algorithm takes as input a MM request and dynamically chooses the best data source (server) on which to schedule that request. The choice of data server is adaptive in that the current load on the DS's in the system when the request arrives is used as the basis for server selection.

5.5.1 *Obtaining Node State Information for Determining Load Factor.* Recall that the request scheduler receives scheduling requests of the form

$$RS \lhd \texttt{schedule}(mms, \alpha^{req}, rod)$$

from the QoS broker, where *mms* is the QoS broker's perception of the current MM state. In particular, *mms* contains information about existing replicas and request assignments that reflects the availability of resources at each data source. We adapt the load-factor function (Section 3) to calculate the load-factor obtained upon assigning the request to that particular DS using an MM state, a request actor and a DS node as arguments, and using correspondingly adapted functions for calculating the resource allocation in a given configuration. This is then used to define the *Candidates* function that determines the best candidate DS node(s) for assigment of the request according to the load factor criteria.

Definition 5.30 (*Candidate Function*). We calculate the available resources on each node for the load factor calculation by subtracting the resources

currently allocated from the total capacity of a data source.

$$Available(mms, DS, Res) = capacity(DS, Res)—ResAlloc(mms, DS, Res),$$

where $ResAlloc(mms, DS, Res)$ is amount of resource Res allocated to requests on DS, according to the information in mms. Recall the four managed resources: network bandwidth ($NetBW$), CPU cycles (CPU), disk bandwidth ($DiskBW$), and memory buffer ($BufMem$). The adaptive load factor calculation is then defined on MM states as follows.

$$LF(mms, \alpha^{req}, DS) = max\left(\frac{qt_{DiskBW}}{DiskBW^{DS}}, \frac{qt_{BufMem}}{BufMem^{DS}}, \frac{qt_{CPU}}{CPU^{DS}}, \frac{qt_{NetBW}}{NetBW^{DS}}\right),$$

where

$$qt = mms\left(\alpha^{req}, QoS\right)$$

$$Res^{DS} = Available(mms, DS, Res) \quad \text{for} \quad Res \in Resources.$$

The candidate data sources, $Candidates(mms, \alpha^{req})$ based on the adaptive load factor calculation are those such that the load factor is minimal and not infinite, and a replica of the requested MM object exists on the data source.

LEMMA 5.31 (CORRECTNESS OF *Candidates*). *The Candidates set is a valid replacement for the assignable set of DSnodes for the purpose of request scheduling. Specifically,*

—$Candidates(mms, \alpha^{req}) \subseteq assignable(mms, \alpha^{req})$.
—$Candidates(mms, \alpha^{req}) = \emptyset \Rightarrow assignable(mms, \alpha^{req}) = \emptyset$.

In the generic request scheduling behavior (RSB), a request represented by α^{req} must be scheduled without violating resource constraints. If many such data servers DS are possible, the RS picks one and notifies the DS manager of that node. With the refined adaptive request scheduling behavior, the value of DS is the candidate data source returned by the adaptive scheduling algorithm. If there is no candidate value possible, a value nil is returned by the adaptive scheduling algorithm

We define Adaptive Request Scheduling MM Behavior by replacing the rules for scheduling requests discussed in Section 5.3 by rules based on the adaptive load factor calculation. We replace

—**(RSscheduleGrant)** with **(AdaptiveRSscheduleGrant)**,
—**(RSscheduleDeny)** with **(AdaptiveRSscheduleDeny)**,
—**(RSscheduleROD)** with **(AdaptiveRSscheduleROD)**,

where the new rules are defined below. The rules for ROD are similarly modified to use the load factor calculation to find a canditate node for replication.

AdaptiveRS. Suppose RS receives a scheduling request, `schedule`(mms, α^{req}, rod) and let $D = Candidates(mms, \alpha^{req})$. If D is nonempty, then RS picks a a node from D on which the request represented by α^{req} can be scheduled and notifies the DS manager of that node **(AdaptiveRSscheduleGrant)**. If D is empty (there is no DS node on which the request represented by α^{req}

can be scheduled without violating resource constraints) and *rod* is false (i.e., disabled), then *RS* sends the broker a denied update **(AdaptiveRSscheduleDeny)**. If *D* is empty and *rod* is true, then *RS* sends a request to *ROD* containing *mms*, along with the requested MM object and QoS requirement of the request **(AdaptiveRSscheduleROD)**.

5.5.2 (*AdaptiveRSscheduleGrant*)

$$\langle RS : IdleB_{\mathrm{rs}}\rangle, RS \lhd \texttt{schedule}(mms, \alpha^{req}, rod)$$

$$\longrightarrow$$

$$\langle RS : WaitB_{\mathrm{rs}}(mms, \alpha^{req}, mmsU)\rangle, DSma(DS) \lhd \texttt{assign}(mmsU) \,@\, RS$$

$$\text{if } DS \in Candidates(mms, \alpha^{req}) \neq \emptyset$$

$$\text{where} \quad mmsU = assignTo(mms, \alpha^{req}, DS)$$

5.5.3 (*AdaptiveRSscheduleDeny*)

$$\langle RS : IdleB_{\mathrm{rs}}\rangle \ , RS \lhd \texttt{schedule}(mms, \alpha^{req}, \texttt{false})$$

$$\longrightarrow$$

$$\langle RS : IdleB_{\mathrm{rs}}\rangle \ , QB \lhd \texttt{scheduleReply}([\alpha^{req} = [State = Denied]]) \,@\, RS$$

$$\text{if } \ Candidates(mms, \alpha^{req}) = \emptyset$$

5.5.4 (*AdaptiveRSscheduleROD*)

$$\langle RS : IdleB_{\mathrm{rs}}\rangle \ , RS \lhd \texttt{schedule}(mms, \alpha^{req}, \texttt{true})$$

$$\longrightarrow$$

$$\langle RS : WaitB_{\mathrm{rs}}(mms, \alpha^{req}, \texttt{nil})\rangle \ , ROD \lhd \texttt{repl}(mms, MM, qt) \,@\, RS$$

$$\text{where} \quad MM = mms(\alpha^{req}, ObjId), t = mms(\alpha^{req}, QoS)$$

$$\text{if } \ Candidates(mms, \alpha^{req}) = \emptyset$$

Adaptive Request Scheduling MM Behavior is defined by a simple modification of the definition of Resource-based MM Behavior.

Definition 5.32 (*Adaptive RS MM Behavior*). A system *S* has *Adaptive Request Scheduling MM Behavior* with respect to the underlying system architecture (nodes and capacity function), the QoS meta actors *QB*, *RS*, *ROD*, *DR*, *PP*, and the DS node managers *DSma*(*DSnodes*), if it satisfies the conditions for Resource-based MM behavior, modified by replacing the request scheduler rules as discussed above.

The correctness theorem for Adaptive RS MM Behavior is the following.

THEOREM 5.33 (ADAPTIVE RS MM BEHAVIOR IMPLIES QOS-BASED MM SERVICE). *If*

—*S has Adaptive RS MM behavior (Definition* 5.32*),*
—*S satisfies the QoS NonInterference Requirement (Definition* 5.20*),*

—*every* $C \in S$ *is reachable from a configuration satifying the QoSInitial conditions (Definition* 5.19*),*

—*QoSTranslate satisfies the QoSTranslate requirements (Definition* 5.3*),*

then S provides QoS-based MM Service (Definition 5.1*).*

PROOF. By Theorem 5.22, we only need to show that under the QoS-Initial and QoS-NonInterference assumptions a system that has Adaptive RS MM Behavior also has Resource-based MM Behavior. For this, it is sufficient to check that for each of the three Adaptive RS rules replacing a generic Resource-based rule, every transition arising from a appliction of an Adaptive RS rule is a transition allowed by the corresponding Resource-based rule. This follows from Lemma 5.31. □

6. RELATED WORK

In this section, we compare our work to related research in object-based middleware, reflective middleware and QoS-based multimedia systems.

Commercially available object-based middleware infrastructures such as CORBA represent a step toward compositional software architectures but do not support the development and maintenance of applications in highly dynamic environments. Specifically, they do not deal with interactions of multiple object services executing at the same time, or the implication of composing object services that has been the focus of this article. Extended software architectures are required that play a role in building systems that solve and maintain architectural properties (e.g., composability, evolvability, scalability, debuggability) [OMG Workshop on Compositional Software Architectures 1998]. For instance, the Electra framework [Maffeis and Schmidt 1997] 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 [Schmidt et al. 1997; Wolfe et al. 1995] necessary to support timing-based QoS requirements [Zinky et al. 1997] 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 [Gokhale and Schmidt 1997], and patterns for extensible middleware [Schmidt and Cleeland 1998]. Similarly, real-time method invocations have been explored by transmitting timing information in CORBA data structures [Wolfe et al. 1995].

The Java Development Environment is a distributed object implementation framework that transforms a heterogeneous network of machines into a homogeneous network of Java virtual machines. Java's main advantage is that it provides mobility; however, the semantics of interaction with other customizations is dependent on the implementation. For instance, 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.

Dynamic composition of communication protocols to satisfy requirements such as fault tolerance and security has been addressed in systems such as Cactus [He et al. 2001; Hiltunen et al. 1999]. Cactus uses microprotocols and

event-based programming style to provide customized services that can tolerate certain classes of failures. Various systems such as the Infospheres Infrastructure [Chandy et al. 1996] and the Globe System [van Steen et al. 1998] explore the construction of large scale distributed systems using the paradigm of distributed objects. Cactus supports configurability by providing events and shared variables that maximize the independence between micro-protocols.

To our knowledge, none of the above systems provides a formal semantics for the composition of middleware services and protocols. The distinguishing feature of the TLAM architecture on which the **CompOSE|Q** system is based is that it is based on formal semantics that supports safe and correct composability of the services being implemented. This in turn, provides a framework for the dynamic composition and safe adaptation of middleware services that can provide application level QoS guarantees.

6.1 Reflective Middleware

Reflection allows application objects to customize the system behavior [Smith 1982] as in Apertos [ichiro Itoh et al. 1995] and 2K [Kon et al. 1998]. The Aspect Oriented Programming paradigm [Kiczales et al. 1997] 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 [Agha et al. 1993] and runtime systems such as Broadway [Sturman 1996] and the Actor Foundry [Astley 1999]. In other reflective models for distributed object computation [Okamura et al. 1992; Costa et al. 1998; Blair et al. 1998], an object is represented by multiple models allowing behavior to be described at different levels of abstraction and from different points of view.

Adaptability and extensibility are prime requirements of middleware systems, and several groups are doing research on reflective middleware [Kon and Saikoski 2000]. Reflective middleware typically builds on the idea of a metaobject protocol, with a metalevel describing the internal architecture of the middleware, and reflection used to inspect and modify internal components. Many of the middleware systems described focus heavily on implementation issues while the focus of the work presented in this paper is on developing formal semantics and reasoning for a QoS-based middleware environment.

DynamicTao [Kon et al. 2000] is a reflective CORBA ORB [Object Management Group 1999] built as an extension of the Tao real-time CORBA ORB [Schmidt et al. 1997]. DynamicTao supports on-the-fly reconfiguration while maintaining consistency by reifying both internal structure and dependency relations using objects called configurators. In Wang et al. [2000] the use of reflective middleware techniques to enhance adaptability in Quality of Service (QoS)-enabled component-based applications is discussed and illustrated using the Tao ORB. The distributed Multimedia Research Group at Lancaster University has proposed a reflective architecture for next-generation middleware based on multiple metamodels [Blair et al. 1998, 2000], and a prototype has been developed using the reflective capabilities of Python.

Middleware systems often contain components that are reflectively related to the application level and/or the underlying infrastructure. For example

Quo [Zinky et al. 1997; Loyall et al. 1998] has *system condition objects* that provide interfaces to resources, mechanisms, orbs etc. that need to be observed, measured or controlled. Delegates reify method requests and evaluate them according to *contracts* that represent strategies for meeting service level agreements. Another example is the Grid Protocol architecture proposed in Foster et al. [2001], in which the resource level contains protocols for query and control of individual resources. The TLAM approach models run-time 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.

In many of the above reflective models for distributed object computation [Okamura et al. 1992; Costa et al. 1998; Blair et al. 1998], an object is represented by multiple models allowing behavior to be described at different levels of abstraction and from different points of view. In the TLAM, each meta actor can examine and modify the behavior of a group of base level actors—namely, those located on the same node. Other work on using reflective ORBs to customize resource management behavior, for example, scheduling is reported in, Singhai et al. [1997]. The two-level architecture naturally extends to multiple levels, with each level manipulating the level below while being protected from manipulation by lower levels. In practice, however, expressing a computation in terms of multiple metalevels becomes unwieldy. A purely reflective architecture provides an unbounded number of metalevels with a single basic mechanism. The formal verification of interaction semantics between the different layers in the reflective hierarchy can be quite complex and requires further investigation.

6.2 QoS in Multimedia Systems

Multimedia QoS enforcement has been a topic of extensive research. Related work in this area includes projects such as Omega [Nahrstedt and Steinmetz 1995], *QualMan* [Nahrstedt et al. 1998] and several algorithms for MM server management [Yu et al. 1992; Vin and Rangan 1993; Lougher and Shepherd 1993; Keeton and Katz 1993; Dan et al. 1996; Wolf et al. 1995; Dan and Sitaram 1995]. *QualMan* [Nahrstedt et al. 1998] 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.

Much of the work on formal models for QoS has been in the context of QoS specification mechanisms and constructs [Frolund and Koistinen 1998]. In some implementation driven methods of QoS specification, the specification of QoS requirements is intermixed with the service specification [Leydekkers and Gay 1996; Lima and Madeira 1996]. Other approaches address the representation of QoS via multiparadigm specification techniques that specify functional behavior and performance constraints distinctly using multiple languages [Blair et al. 1988; Zave and Jackson 1997; Blair and Blair 1999a, 1999b]. The TLAM approach, on the other hand, models run-time 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. Synchronizers and *RtSynchronizers* [Frølund 1996; Ren et al. 1996] allow us to express QoS constraints via coordination constraints in the actor model either as local synchronization constraints or multi-actor coordination constraints. The TLAM supports both local and multi-actor coordination and supports reasoning about the relation between the model held at the metalevel and the base-level being modeled.

7. CONCLUDING REMARKS

In this article, we have shown, using the QoS broker MM architecture, how the TLAM framework can be used to specify and reason about distributed middleware services and their composition, and have also indicated how specifications in the framework can lead to implementations. A key concern of our work is to exploit concurrency while preserving the desired behavior of applications and systems. To ensure correct concurrent execution of middleware services, we attempt to develop a reasonably minimal set of constraints that ensure consistent system behavior. The constraint set presented in this article is a result of trying to formally analyze the behavior of an actual implemented system. In fact the reasoning process yielded new constraints that were not originally enforced. This demonstrates the utility of a formal model in the development of middleware systems.

The work presented here focused on managing the resources that support real-time and QoS requirements. We are actively working on extending the existing meta-architecture to support more services for MM interaction. Modeling client interaction requires a notion of session and resources within a session. Our next step is to develop constructs and techniques for modeling and reasoning about sessions and session-based services in distributed MM systems. Specifying and reasoning about the enforcement of timing-based QoS requirements of multiple sessions involves a more thorough treatment of time and synchronization. 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. Further work is also required in order to model request migration in the meta-architecture and develop strategies for its effective use.

As applications must run in increasingly distributed and mobile environments, middleware services to support these applications becomes increasingly important. As dependence on middleware services increases it becomes more important to have clear semantic models and to be able to carry out a variety of analyses based on these models in order to increase assurance of correct and expected behavior.

Another important area of future research is the specification of flexible security mechanisms and their integration into middleware frameworks. Security poses unique problems for customizability since many security policies hide system-level information needed for customizability. Locality and security are often closely integrated and this raises issues when migration of objects is

required for load-balancing. Such issues are of particular relevance to providers of electronic commerce services that require both transaction security and effective resource utilization.

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 that supports customization and composition of protocols and services is needed to support the flexible use of component based software.

System specifications such as the one presented here can serve as a valuable form of documentation of requirements, design and implementation decisions. The relations between viewpoints provide a systematic and rigorous mechanism to relate system requirements and the design and implementation descriptions. Having this form of multiple view specification and documentation can be used in configuration management to help isolate effects of change, to propagate effects of change, and to reason about the effects of adaptation on application behavior.

APPENDIX A. TRANSITION RULES FOR ROD, DR, PP

A.1 Transition Rules for ROD

To state the rules for ROD, we define two functions. $rodRepl(DS, MM, bw)$ is the replication update for placement of a new replica of MM on DS with minimum replication bandwidth bw.

$$rodRepl(DS, MM, bw) = [(DS, MM) = [Class = 1, \; BW = bw, \; State = InQueue]].$$

$rodOk(mms, MM, qt)$ is the set of pairs (DS, bw) such that replicating MM on DS is allowed and having done this, assigning a request for MM with QoS qt to DS is OK.

$$\begin{aligned} rodOk(mms, MM, qt) = \{&(DS, bw) \in DSnodes \times Unit_{NetBW} \mid \\ &mms((DS, MM), Class) = 0 \wedge \\ &\phi_{\text{res}}(mms \texttt{ modby } reqU \texttt{ modby } rodRepl(mms, DS, MM, \\ &bw)) \quad \text{where} \quad reqU = [\alpha^{req} = [State = Granted, \\ &\qquad ObjId = MM, \\ &\qquad QoS = qt, \\ &\qquad Replica = DS]]\} \\ &\text{for} \quad \alpha^{req} \notin \text{Dom}(mms). \end{aligned}$$

The α^{req} above represents a hypothetical grantee for purpose of checking constraints.

A.1.1 (*RODreplOk*). When *ROD* receives a request, $ROD \lhd \texttt{repl}(mms, MM, qt) @ RS$ for replication of MM object, *MM*, supporting QoS resource requirements *qt*, it looks, using *mms*, for a DS node that doesn't have the needed MM object and that has the required resources available. If one is found, a replication request is sent to that DS node and *ROD* waits for an acknowledgment.

$$\langle ROD : IdleB_{\text{rod}} \rangle \ , \ ROD \lhd \texttt{repl}(mms, MM, qt) @ \alpha$$

$$\longrightarrow$$

$$\langle ROD : WaitB_{\text{rod}}(C, replU) \rangle \ , \ DSma(DS) \lhd \texttt{repl}(replU) @ ROD$$

$$\text{if } (DS, bw) \in rodOk(mms, MM, qt) \ \wedge \ replU = rodRepl(DS, MM, bw)$$

A.1.2 (*RODreplFail*). If no such DS node is found, then a failure reply is sent to *RS*, which is the client *C*.

$$\langle ROD : IdleB_{\text{rod}} \rangle \ , \ ROD \lhd \texttt{repl}(mms, MM, qt) @ C$$

$$\longrightarrow$$

$$\langle ROD : IdleB_{\text{rod}} \rangle \ , \ C \lhd \texttt{replAck(nil)} @ ROD$$

$$\text{if } rodOk(mms, MM, qt) = \emptyset$$

A.1.3 (*RODrepUpdAck*). When the acknowledgment message is received, a reply is sent to *RS* with MM state containing the replica update information.

$$\langle ROD : WaitB_{\text{rod}}(C, replU) \rangle \ , \ ROD \lhd \texttt{repUpdAck()} @ DSma(DS)$$

$$\longrightarrow$$

$$\langle ROD : IdleB_{\text{rod}} \rangle \ , \ C \lhd \texttt{replAck}(replU) @ ROD.$$

A.2 Transition Rules for DR

To state the rules for *DR* we use two auxiliary functions. $drRepl(DS, MM)$ is the replication update marking *MM* for dereplication on *DS*, and for a set *D* of DS node, MM object pairs, $drRepl(D)$ is the composition of the updates for each element of *D*.

$$drRepl(DS, MM) = [(DS, MM) = [Class = 2]]$$

$$drRepl(D) = [drRepl(DS, MM) \mid (DS, MM) \in D]$$

$drOk(mms)$ is the set of pairs (DS, MM) such that dereplication is a possibility in *mms*.

$$drOk(mms) = \{(DS, MM) \mid mms((DS, MM), Class = 1\}.$$

A.2.1 (*DRderepl*). Upon receiving a dereplication request, based on the information in *mms*, DR may decide to reclassify some replicas from 1 to 2. The DRderepl transition then notifies each DS node of any changes on that node,

that is, it sends an appropriate message to the DSma on that node.

$$\langle DR : IdleB_{\mathrm{dr}}\rangle \ , \ DR \lhd \texttt{derepl}(mms)$$

$$\longrightarrow$$

$$\langle DR : WaitB_{\mathrm{dr}}(replU, dsout)\rangle \ , \ \{DSma(DS) \lhd \texttt{repl}(replU/DS) \mathbin{@} DR \mid$$

$$DS \in dsout\} \quad \text{if } D \subseteq drOk(mms) \wedge dsout = \{DS \in DSnodes \mid$$

$$(\exists MM)(DS, MM) \in D\} \wedge replU = drRepl(D) \wedge \phi_{\mathrm{res}}(mms \ \texttt{modby} \ replU).$$

A.2.2 (*DRdsAck*). The *DR* waits for acknowledgments in the transition DRdsAck that indicates that reclassification of replicas on the DS node have completed.

$$\langle DR : WaitB_{\mathrm{dr}}(replU, dsout)\rangle \ , \ DR \lhd \texttt{replAck}() \mathbin{@} DSma(DS)$$

$$\longrightarrow$$

$$\langle DR : WaitB_{\mathrm{dr}}(replU, dsout - DS)\rangle.$$

A.2.3 (*DRdone*). Once the *DR* receives an acknowledgment from the DSnode indicating change of replica state, *DR* sends a reply *QB* containing the replica state update.

$$\langle DR : WaitB_{\mathrm{dr}}(replU, \emptyset)\rangle$$

$$\longrightarrow$$

$$\langle DR : IdleB_{\mathrm{dr}}\rangle \ , \ QB \lhd \texttt{replAck}(replU) \mathbin{@} DR.$$

A.3 Transition Rules for PP

We first define some auxiliary functions. $ppRepl(mms, (DS, MM), bw)$ is the replication update for reclassification of the replica class of *MM* on *DS* in *mms*, including if needed initiating replication with minimum bandwidth *bw*. For a set P of triples of the form (DS, MM, bw), $ppRepl(mms, P)$ is the union of the replication updates for the elements of P in *mms*.

$$\begin{aligned} &ppRepl(mms, (DS, MM, bw)) = \\ &\quad [(DS, MM) = [Class = 1, \\ &\qquad\qquad BW = bw, \\ &\qquad\qquad State = \texttt{if} \ mms((DS, MM), Class) = 0 \\ &\qquad\qquad\qquad\qquad \texttt{then} \ InQueue \\ &\qquad\qquad\qquad\qquad \texttt{else} \ mms((DS, MM), State)]] \\ &ppRepl(mms, P) = [ppRepl(mms, (DS, MM), bw) \mid (DS, MM, bw) \in P]. \end{aligned}$$

$ppOk(mms, P)$ holds if P has at most one update for a given replica, updated replicas are not class 1, and the overall update respects the total resource constraint.

$$ppOk(mms, P) = (DS, MM, bw) \in P \ \wedge \ (DS, MM, bw') \in P \ \Rightarrow \ bw = bw' \wedge$$
$$(DS, MM, bw) \in P \ \Rightarrow \ mms((DS, MM), Class) \neq 1 \wedge$$
$$\phi_{\text{res}}(mms \ \texttt{modby} \ ppRepl(mms, P)).$$

A.3.1 (*PPplace*). Based on the information in MM state *mms* of `place` request, PP may decide to reclassify some replicas from 0, 2, or 3 to 1 and in the case of moving from 0 to 1 initiates replication. It then notifies each DS node of any changes on that node in the transition PPplace.

$$\langle PP : IdleB_{\text{pp}} \rangle, \ PP \lhd \texttt{place}(mms)$$

$$\longrightarrow$$

$$\langle PP : WaitB_{\text{pp}}(replU, dsout) \rangle \ , \ \{DSma(DS) \lhd \texttt{repl}(replU/DS) \ @ \ PP \ \mid$$

$$DS \in dsout\} \quad \text{if} \ ppOk(mms, P) \wedge$$

$$replU = ppRepl(mms, P) \wedge$$

$$dsout = \{DS \in DSnodes \ \mid \ (\exists MM, bw)(DS, MM, bw) \in P\}.$$

A.3.2 (*PPdsAck*). *P P* waits for acknowledgments from DS nodes indicating completion of reclassification of replica state in (PPdsAck).

$$\langle PP : WaitB_{\text{pp}}(replU, dsout) \rangle \ , \ PP \lhd \texttt{replAck}() \ @ \ DSma(DS)$$

$$\longrightarrow$$

$$\langle PP : WaitB_{\text{pp}}(replU, dsout - DS) \rangle.$$

A.3.3 (*PPdone*). Once the *PP* receives the reclassification acknowledgment from the DSnode, it sends a reply to *QB* containing the updated replica state (PPdone). It then notifies each DS node of any changes on that node (PPplace), waits for acknowledgments (PPdsAck), and then sends a reply to *QB* containing the updated replica state (PPdone).

$$\langle PP : WaitB_{\text{pp}}(replU, \emptyset) \rangle$$

$$\longrightarrow$$

$$\langle PP : IdleB_{\text{pp}} \rangle, \ QB \lhd \texttt{placeAck}(replU) \ @ \ PP.$$

REFERENCES

AGHA, G. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, Mass.

AGHA, G., FRØLUND, S., KIM, W., PANWAR, R., PATTERSON, A., AND STURMAN, D. 1993. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parall. Distrib. Tech.: Syst. Appl. 1*, 2 (May), 3–14.

ASTLEY, M. 1999. Customization and Composition of Distributed Objects: Policy Management in Distributed Software Architectures. Ph.D. thesis, University of Illinois at Urbana-Champaign.

ASTLEY, M. AND AGHA, G. A. 1998. Customization and composition of distributed objects: Middleware abstractions for policy management. In *Proceedings of the 6th International Symposium on the Foundations of Software Engineering (FSE 1998)*.

BAKER, H. G. AND HEWITT, C. 1977. Laws for communicating parallel processes. In *IFIP Congress*. IFIP, 987–992.

BLAIR, G., BLAIR, L., BOWMAN, H., AND CHETWYND, A. 1988. *Formal Specifications of Distributed Multimedia Systems*. UCL Press.

BLAIR, G., CLARKE, M., COSTA, F., COULSON, G., DURAN, H., AND PARLAVANTZAS, N. 2000. The evolution of OpenORB. In *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*. ACM, New York.

BLAIR, G., COULSON, G., ROBIN, P., AND PAPATHOMAS, M. 1998. An architecture for next generation middleware. In *Middleware '98*.

BLAIR, L. AND BLAIR, G. 1999a. Composition in multiparadigm specification techniques. In *IFIP Workshop on Formal Methods for Open Object-based Distributed Systems, FMOODS'99*.

BLAIR, L. AND BLAIR, G. 1999b. The impact of aspect-oriented programming on formal methods: Position paper. In *ECOOP Workshop on Aspect Oriented Programming*.

BUDDHIKOT, M. AND PARULKAR, G. 1995. Efficient data layout, scheduling and playout control in mars. In *Proceedings of NOSSDAV'95*. 339–351.

CHANDY, K., RIFKIN, A., SIVILOTTI, P. A., MANDELSON, J., RICHARDSON, M., TANAKA, W., AND WEISMAN, L. 1996. A world-wide distributed system using java and the internet. In *Proceedings of IEEE International Symposium on High Performance Distributed Computing (HPDC-5)*, (Syracuse, New York).

COSTA, F., BLAIR, G., AND COULSON, G. 1998. Experiments with reflective middleware. In *European Workshop on Reflective Object-Oriented Programming and Systems, ECOOP'98*. Springer-Verlag, New York.

DAN, A., DIAS, D., MUKHERJEE, R., SITARAM, D., AND TEWARI, R. 1995. Buffering and caching in large scale video servers. In *IEEE Compcon*. 217–224.

DAN, A. AND SITARAM, D. 1995. An online video placement policy based on bandwidth to space ratio (bsr). In *SIGMOD '95*. ACM, New York, 376–385.

DAN, A., SITARAM, D., AND SHAHABUDDIN, P. 1996. Dynamic batching policies for an on-demand video server. *ACM Trans. Multimed. Syst. 4*, 112–121.

FOSTER, I., KESSELMAN, C., AND TUECKE, S. 2001. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. Supercomput. Appl.*

FRØLUND, S. 1996. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, Cambridge, Mass.

FROLUND, S. AND KOISTINEN, J. 1998. Quality of Service Specification in Distributed Object System Design. In *USENIX COOTS*.

GOKHALE, A. AND SCHMIDT, D. C. 1997. Evaluating the performance of demultiplexing strategies for real-time CORBA. In *Proceedings of GLOBECOM '97* (Phoenix, Az.).

HE, J., HILTUNEN, M., RAJAGOPALAN, M., AND SCHLICHTING, R. 2001. Providing QoS Customization in Distributed Object Systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms* (Middleware). ACM, New York.

HEWITT, C. 1977. Viewing control structures as patterns of passing messages. *Journal of Artificial Intelligence 8*, 3, 323–364.

HEWITT, C., BISHOP, P., AND STEIGER, R. 1973. A universal modulal actor formalism for artificial intelligence. In *Proceedings of 1973 International Joint Conference on Artificial Intelligence*. 235–245.

HILTUNEN, M., IMMANUEL, V., AND SCHLICHTING, R. 1999. Supporting Customized Failure Models for Distributed Software. *Distrib. Syst. Eng. 6*, 103–111.

ICHIRO ITOH, J., LEA, R., AND YOKOTE, Y. 1995. Using meta-objects to support optimization in the Apertos operating system. In *USENIX COOTS (Conference on Object-Oriented Technologies)*.

KEETON, K. AND KATZ, R. 1993. The evaluation of video layout strategies on a high-bandwidth file server. In *Proceedings of the 4th International Workshop on Network and Operating System Support for Digital Audio and Video* (Lancaster, UK). 237–250.

KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of ECOOP'97 European Conference on Object-Oriented Programming*.

KON, F., ROMÁN, M., LIU, P., MAO, J., YAMANE, T., MAGALHÃES, L. C., AND CAMPBELL, R. H. 2000. Monitoring, security, and dynamic configuration with the dynamic TAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and*

Open Distributed Processing (Middleware'2000). Lecture Notes in Computer Science, vol. 1795. Springer-Verlag, New York, 121–143.

KON, F. AND SAIKOSKI, K. B., Eds. 2000. *IFIP/ACM Middleware'2000 Workshop on Reflective Middleware*. Gordon Blair and Roy Campbell (co-chairs), ACM, New York.

KON, F., SINGHAI, A., CAMPBELL, R. H., CARVALHO, D., MOORE, R., AND BALLESTEROS, F. J. 1998. 2K: A reflective, component-based operating system for rapidly changing environments . In *Proceedings of ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems* (Brussels, Belgium).

LEYDEKKERS, P. AND GAY, V. 1996. ODP view on QOS for open distributed mm environments. In *Proceedings of the 4th International IFIP Workshop on Quality of Service (IwQos96)* (Paris, France). J. D. Meer and A. Vogel, Eds. 45–55.

LIMA, F. AND MADEIRA, E. 1996. ODP based QOS specification for the multiware platform. In *Proceedings of the 4th International IFIP Workshop on Quality of Service (IwQos96)* (Paris, France). J. D. Meer and A. Vogel, Eds. 45–55.

LOUGHER, P. AND SHEPHERD, D. 1993. The design of a storage server for continuous media. *The Comput. J.—Special Issue on Distributed Multimedia Systems 36*, 1 (Feb.), 32–42.

LOYALL, J., SCHANTZ, R., ZINKY, J., AND BAKKEN, D. 1998. Specifying and measuring quality of service in distributed object systems. In *Proceedings of the First International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '98)*.

MAFFEIS, S. AND SCHMIDT, D. 1997. Constructing reliable distributed communication systems with corba. *IEEE Commun. 14*, 2 (Feb.).

NAHRSTEDT, K. 1995. Network Service Customization: End-Point Perspective. Ph.D. thesis, University of Pennsylvannia.

NAHRSTEDT, K., CHU, H.-H., AND NARAYAN, S. 1998. QOS-aware resource management for distributed multimedia applications. *J. High Speed Networking 8*, 304, IOS Press, 225–227.

NAHRSTEDT, K. AND STEINMETZ, R. 1995. Resource management in networked multimedia systems. *IEEE Comput. 28*, 5 (May), 52–65.

OBJECT MANAGEMENT GROUP. 1999. The Common Object Request Broker:Architecture and Specification, 2.3 ed.

OKAMURA, H., ISHIKAWA, Y., AND TOKORO, M. 1992. Al-1/d: A distributed programming system with multi-model reflection framework. In *Reflection and Meta-Level Architetures*, A. Yonezawa and B. C. Smith, Eds. In *ACM SIGPLAN*. ACM, New York, 36–47.

OMG WORKSHOP ON COMPOSITIONAL SOFTWARE ARCHITECTURES. 1998. *Proceedings of the OMG Workshop on Compositional Software Architectures*.

REN, S. 1997. Modularization of time constraint specification in real-time distributed computing (to be published). Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.

REN, S., AGHA, G., AND SAITO, M. 1996. A modular approach for programming distributed real-time systems. *J. Paral. Distr. Comput. 36*, 1 (July).

SCHMIDT, D. C. AND CLEELAND, C. 1998. Applying patterns to develop extensible and maintainable orb middleware. *Commun. ACM*.

SCHMIDT, D. C., LEVINE, D., AND MUNGEE, S. 1997. The design of the Tao real-time object request broker. *Comput. Commun. 21. Special Issue on Building Quality of Service into Distributed System*.

SINGHAI, A., SANE, A., AND CAMPBELL, R. 1997. Reflective ORBs: Support for robust, time-critical distribution. In *Proceedings of the ECOOP'97 Workshop on Reflective Real-Time Object-Oriented Programming and Systems*.

SMITH, B. C. 1982. Reflection and semantics in a procedural langauge. Ph.D. thesis, Massahusetts Institute of Technology.

STURMAN, D. 1996. Modular Specification of Interaction Policies in Distributed Computing. Ph.D. dissertation, University of Illinois at Urbana-Champaign. TR UIUCDCS-R-96-1950.

THAPAR, M. AND KOERNER, B. 1994. Architecture for video servers. In *Proceedings of the 43rd Annual NCTA Convention and Exposition* (New Orleans, La.). 141–148.

VAN STEEN, M., TANENBAUM, A., KUZ, I., AND SIP, H. 1998. A scalable middleware solution for advanced wide-area web services. In *Proceedings of the Middleware '98* (The Lake District, U.K.)

VENKATASUBRAMANIAN, N. 1992. Hierarchical garbage collection in scalable distributed systems. M.S. dissertation, University of Illinois, Urbana-Champaign.

VENKATASUBRAMANIAN, N. 1998. An adaptive resource management architecture for global distributed computing. Ph.D. dissertation, University of Illinois, Urbana-Champaign.

VENKATASUBRAMANIAN, N. 1999. Composeq—A QOS-enabled customizable middleware framework for distributed computing. In *Proceedings of the Middleware Workshop, International Conference on Distributed Computing Systems (ICDCS99)*.

VENKATASUBRAMANIAN, N., AGHA, G., AND TALCOTT, C. L. 1992. Scalable distributed garbage collection for systems of active objects. In *International Workshop on Memory Management, IWMM92* (Saint-Malo). Lecture Notes in Computer Science. Springer-Verlag, New York.

VENKATASUBRAMANIAN, N., DESHPANDE, M., MOHAPATRA, S., GUTIERREZ-NOLASCO, S., AND WICKRAMASURIYA, J. 2001. Design and implementation of a composable reflective middleware framework. In *International Conference on Distributed Computing Systems (ICDCS2001)*.

VENKATASUBRAMANIAN, N. AND RAMANATHAN, S. 1997. Effective load management for scalable video servers. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS97)*.

VENKATASUBRAMANIAN, N. AND TALCOTT, C. L. 1993. A metaarchitecture for distributed resource management. In *Hawaii International Conference on System Sciences, HICSS-26*.

VENKATASUBRAMANIAN, N. AND TALCOTT, C. L. 1995. Reasoning about meta level activities in open distributed systems. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*. ACM, New York, 144–152.

VENKATASUBRAMANIAN, N. AND TALCOTT, C. L. 2001a. Integration of resource management activities in distributed systems. Tech. Rep., Department of Computer Science, UC Irvine.

VENKATASUBRAMANIAN, N. AND TALCOTT, C. L. 2001b. A semantic framework for modeling and reasoning about reflective middleware. *IEEE Distrib. Syst. Online 2*, 7.

VIN, H. M. AND RANGAN, P. V. 1993. Designing a multi-user HDTV storage server. *IEEE J. Select. Areas Commun. 11*, 1 (Jan.), 153–164.

WANG, N., KIRCHER, M., SCHMIDT, D. C., AND PARAMESWARAN, K. 2000. Applying reflective middleware techniques to optimize a QOS-enabled corba component model implementation. In *COMPSAC 2000 Conference*. (Taipei, Taiwan).

WOLF, J. L., YU, P. S., AND SHACHNAI, H. 1995. DASD dancing: A disk load balancing optimization scheme for video-on-demand computer systems. In *Proceedings of ACM SIGMETRICS '95, Performance Evaluation Review*. ACM, New York, 157–166.

WOLFE, V. F., BLACK, J. K., THURAISINGHAM, B., AND KRUPP, P. 1995. Real-time method invocations in distributed environments. In *Proceedings of the HiPC'95 Conference on High Performance Computing*.

YU, P., CHEN, M., AND KANDLUR, D. 1992. Design and analysis of a grouped sweeping scheme for multimedia storage management. *Proceedings of 3rd International Workshop on Network and Operating System Support for Digital Audio and Video* (San Diego, Calif.). 38–49.

ZAVE, P. AND JACKSON, M. 1997. Requirements for telecommunications services: An attack on complexity. In *Proceedings of the IEEE International Symposium on Requirements Engineering*. IEEE Computer Society Press, Los Alamitos, Calif.

ZINKY, J., BAKKEN, D., AND SCHANTZ, R. 1997. Architectural support for quality of service for CORBA objects. *Theory Pract. Obj. Syst*.

Received October 2000; revised November 2002 and October 2003; accepted November 2003