Design and Implementation of a Composable Reflective Middleware Framework

Nalini Venkatasubramanian, Mayur Deshpande, Shivjit Mohapatra, Sebastian Gutierrez-Nolasco & Jehan Wickramasuriya
Department of Information and Computer Science
University of California, Irvine, Irvine CA 92697-3425
{nalini,mayur,mopy,seguti,jwickram}@ics.uci.edu

Abstract

With the evolution of the global information infrastructure, service providers will need to provide effective and adaptive resource management mechanisms that can serve more concurrent clients and deal with applications that exhibit Quality of Service (QoS) requirements. Flexible, scalable and customizable middleware can be used as enabling technology for next generation systems that adhere to the QoS requirements of applications that execute in highly dynamic distributed environments. To enable application aware resource management, we are developing a customizable and composable middleware framework called CompOSE|Q based on a reflective metamodel. In this paper, we describe the architecture and runtime environment for CompOSE|Q and briefly assess the performance overhead of the additional flexibility. We also illustrate how flexible communication mechanisms can be supported efficiently in the CompOSE|Q framework.

1. Introduction

In the future, large scale distributed systems must handle applications that are open and interactive. Applications over the digital wireless Internet such as smart houses, intelligent transportation systems, mobile information appliances such as palmtops running Web-based multimedia applications require dynamic reconfiguration of systems, services and protocols. Open Distributed Systems (ODS) allow services, end-systems and protocols to dynamically attach to and detach from the distributed environment; such systems evolve dynamically and their components interact with an environment that is not under their control. Increasing levels of distribution in information access, openness and dynamism results in increasing complexity. Middleware is the fundamental infrastructure that enables distributed computing. Flexible middleware platforms incorporate the notion of *reflection* in order to provide the desired level of configurability and openness in a controlled manner[SSC97]. While such distributed middleware enables the modular connection of software components to manage the resources of an ODS, it must constrain the global behavior of the distributed system to ensure safety while providing for dynamic reconfiguration. Furthermore, distributed applications have varying requirements often stated as QoS (Quality of Service) parameters that define the extent to which performance specifications such as responsiveness, availability, cost-effective utilization and security may be violated, if at all. To satisfy these application requirements, appropriate resource management policies can be used; e.g. replication, migration & checkpointing. One role of middleware is to abstract over the low level *mechanisms*, which are required to implement these policies. Building a QoS-enabled customizable middleware framework requires characterizing and reasoning about the interactions between multiple resource management activities in ODSs, their dynamic installation and customization in the presence of QoS requirements.

In this paper, we describe the design and implementation a QoS-enabled reflective middleware framework called CompOSE|Q, currently being developed at the University of California, Irvine. CompOSE|Q is based on a meta-architectural model that facilitates specifying and reasoning about the composability of multiple resource management services in ODSs. We illustrate how effective mechanisms can be incorporated into a composable middleware framework to ensure safety and QoS enforcement in distributed environments. The rest of this paper is organized as follows. We begin by briefly describing the TLAM, a formal semantic model for reasoning about middleware in Section 2. Section 3 discusses the basic CompOSE|Q architecture and metalevel services. Section 4 addresses the implementation of the CompOSE|Q system and presents initial performance results. Related work is discussed in Section 5. We conclude in Section 6 with future research directions.

2. The Two Level Meta Architectural Model

To simplify development of applications, we use Actors[A86], a model of concurrent active objects that has a built-in notion of encapsulation and interaction among the concurrent components of an ODS. In the actor paradigm, the universe contains computational agents called *actors*, distributed over a network. Traditional passive objects encapsulate state and a set of procedures that manipulate the state; actors extend this by encapsulating a thread of control as well. Each actor potentially executes in parallel with other actors and may communicate with other actors via asynchronous message passing. Using Actors, we define a meta-architecture framework that permits customization of resource management mechanisms such as placement, scheduling and synchronization.

Ensuring correctness in a purely reflective model involves reasoning about system level interactions by characterizing the semantics of shared distributed resources and understanding what correctness of the overall system means. The TLAM (Two Level Actor Machine) model[VT95] is a first step towards providing a formal semantics for specifying and reasoning about properties of and interactions between components of ODSs. 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 level actors carry out application level computation, while meta-actors are part of the runtime system, which manages system resources and controls the runtime behavior of the base level. Meta-actors communicate with each other via message passing as do base level actors, but they may also examine and modify the state of the base actors located on the same node.

The TLAM uses reification (base object state as data at the meta object level) and reflection (modification of base object state by meta objects) with support for implicit invocation of meta objects in response to changes of base level state. The TLAM provides for full actor-style interaction of meta level objects; we are currently looking into providing restricted support for the *explicit* invocation of meta objects by base objects. Our earlier work has focused on defining a rigorous mathematical semantics and on using these semantics to develop concepts and methods for expressing and reasoning about properties of middleware components and their interactions.

In the TLAM model, meta-level controllers define protocols and mechanisms that customize various aspects of distributed systems management. In practice, multiple system and application activities occur concurrently in a distributed system, e.g. scheduling, protocol processing, stream synchronization etc., and can therefore interfere with each other. Composing multiple resource management mechanisms leads to complex interactions. Consider the following example of a system where distributed garbage collection and process migration can proceed concurrently. If processes in migration (transit) are not accounted for, the garbage collection process can potentially destroy accessible information. Similarly, if the process is continuously migrating, the garbage collection process runs the risk of potential non-termination. In general, risks that arise due to mechanism composition include loss of information, possible non-terminations that cause deadlocks and livelocks, dangling resources, inconsistencies, and incorrect execution semantics.

One approach to deal with interference during mechanism composition in an open system is to serialize or delay activities to ensure overall safety of the system. However, this can result in over-serialization of resource management activities causing performance degradation. Furthermore, global delays and halts may cause violations of timing based QoS constraints. We also argue that composability of resource management activities is not just desirable, but *essential* to ensure cost-effective QoS in distributed systems. For instance, system protocols and activities must not enforce arbitrary delays in the presence of timing based QoS constraints.

To ensure non-interference and manage the complexity of reasoning about components of ODSs in general, our strategy is to identify key system services where non-trivial interactions between the application and system occur, i.e. base-meta interactions. We refer to these key services as *core services*. Core services are used in specifying and implementing more complex activities within the framework as purely meta-level interactions. The development of suitable non-interference requirements allows us to reason about the composition of multiple system services; these services have constraints that must be obeyed to maintain composability (i.e. safe concurrent execution). Examples of such non-inteference requirements have been spelled out in [VT95,V98].

We use commonly observed patterns in distributed systems services to extract implementable abstractions and identify three metalevel core activities(See Figure 1):

- Remote Creation: Recreation of services/data at a remote site. Remote creation can be used as the basis for designing algorithms for activities such as migration, replication and load balancing.
- Distributed Snapshot: Capturing information at multiple nodes/sites used as a basis for

checkpointing, distributed garbage collection (cf.[VAT92]) etc.

- Directory Services: Interactions with a global repository. Directory services can be used to provide access control, resource discovery and implement group communication protocols.

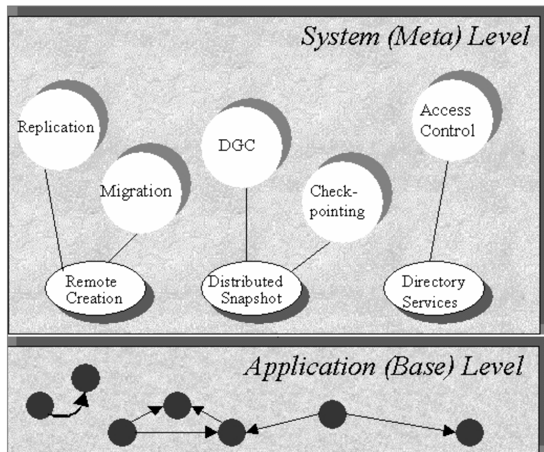

Figure 1. Classification of Core Services. The layering of the system level on top of the application level represents the reflective hierarchy.

3. The CompOSE|Q Architecture

Based on the two level meta-architecture, we are developing a customizable and safe distributed systems middleware infrastructure, called CompOSE|Q (Composable Open Software Environment with QoS) that has the ability to provide cost-effective QoS-based distributed resource management. CompOSE|Q provides *composable distributed resource management*, i.e., it allows the *concurrent execution* of multiple resource management policies in a distributed system in a safe and correct manner. This will allow safe integration of resource management mechanisms for services such as mobility, load balancing, fault tolerance and end-to-end QoS management. The CompOSE|Q architecture and implementation contains:

- Modules that implement the 3 basic composable core services - remote creation, distributed snapshot and directory services with interaction constraints that ensure their concurrent execution with each other and other metalevel services.
- Common services built using core services - actor migration, replication of services and data, actor scheduling, distributed garbage collection, name services etc. Each of these services has its own interface definitions and interaction constraints.
- Reflective communication architecture
- QoS enforcement mechanisms.

3.1 Meta Level Resource Management Services

Implementation of sophisticated policies and mechanisms for QoS management is made possible by providing support for common services in CompOSE|Q. For instance, object scheduling mechanisms use the basic remote creation core service to assign newly created objects/actors on nodes with adequate resources. Using generalized state capture facilities, we are developing a checkpointing service for capturing causal orders of executions in the system that can be used for monitoring and debugging distributed computations. A state broadcast mechanism is used to implement a clock synchronization service, which informs nodes about a global time value that can be used for time related services.

Remote Creation: Remote creation is the process by which actor creation occurs on a specified node other than the node from which creation is being initiated. Remote creation is a basic facility that can be used in other resource management activities like load-balancing, replication and migration. By encapsulating the interactions between the application and system level actors within the remote creation service, we can state requirements that ensure safe and correct composition of other resource management activities with remote creation. In a real TLAM based implementation, the control activities of remote creation are managed by remote creation meta-actors (RCM) residing on every node in the system. A remote creation request has four parameters - a description of the fragment *desc* to be migrated, the remote node *(N),* any initial state the *desc* has to be set to and the initiating-actor 'β'. The initiator actor 'β' is maintained by the RCM to ensure composability with other meta-level services[V98]. If the requester needs to know that the request has been completed, or the names of some of the newly created actors, then this can be achieved by specifying appropriate messages as part of the requested fragment, and observing their delivery.

Distributed Snapshot Services: Global properties like the number of application-actors, the current reachability graph of distributed actors, number of messages being processed and task queue sizes help in making runtime decisions like load balancing, migration and garbage collection, leading to efficient runtime management of a distributed system. To fully represent the global state of the distributed system, we need a mechanism for recording the state of all nodes including the portion of node state being communicated in the network channels[CL85]. As state information is accessible explicitly only in nodes, a snapshot mechanism must ensure that node state information in channels are

recorded at some node in the system (possibly the target node itself). The snapshot mechanisms we have devised are such that application-level computation and system level services proceed concurrently with the snapshot, thereby preserving application and service semantics. In order to initiate snapshot recording on every node and force messages in channels to reach a node, we have defined two wave protocols for message propagation that (a) visit all nodes exactly once, capturing node-resident information and (b) traverse all links in the system exactly once forcing messages on channels to reach nodes (where their state can be recorded). The starting and finishing points of both waves, the propagation path and constraints on message propagation are defined elsewhere[V98]. Termination is signaled when the wave is complete, serving as a synchronization point in the snapshot mechanism. Note that this generalizes the notions of wave and global snapshot discussed in[H89] where the snapshots explicitly do not account for information contained in messages in transit.

Migration: We now illustrate how a migration service can be built using the remote creation core service. By using RC as the basis for migration, we have ensured composability of migration with other meta-level services[V98] such as reachability snapshots and distributed checkpointing. Migration is the process by which actors move from one node to another. The migration service allows for relocation of actors for easier access, availability and load balancing. In this section, we describe a generalized migration service and it's potential implementation using the remote creation (RC) specification. A migration request is given by a pair (α,ν), where α is the actor to be migrated, and ν is the destination node. This is interpreted as a request to move the computation carried out by α to the node ν. In order to state explicitly invariants maintained by the system during the migration process, we classify the migration process into 3 phases wrt the actor being migrated and the node to which it is being migrated. The first phase is the initiation phase and specifies the state of the system when the migration request received can be processed. It determines the computation to be migrated by suspending the computation of the actor and noting its current description. In the second configuration the actual actor migration is performed using the RC service. The last configuration finalizes the migration process and establishes transparent access to the migrated actor.

QoS Brokerage Service: This work illustrates the use of TLAM services in the design of mechanisms and policies needed to enforce QoS constraints in the actor-based runtime environment. We extend the basic meta-architectural framework to provide QoS based services to applications. The base level component of the meta-architecture implements the functionality of the distributed session and deals with (a) data, which includes objects of varying media, types, e.g., video and audio files and (b) requests to access this data via sessions. The meta-level component deals with the coordination of multiple requests and sharing of existing resources among multiple requests. To provide coordination at the highest level and perform admission control for new incoming sessions, a meta-level entity called the *QoS broker*[NS95] is being developed. The organization of the meta-level services in CompOSE|Q is illustrated in Figure 2; the services have been mapped to a specific distributed server architecture[VR97, V98].

The two main functions of the QoS broker are (a) data management and (b) request management. The *data management* component decides the placement of data in the distributed system, i.e. it decides when and where to create additional replicas of data. It also determines when additional replicas of data actors are no longer needed and can be garbage collected/dereplicated. We implement adaptive admission control mechanisms[VR97] in the request-scheduling module that assigns requests to servers and ensures cost-effective utilization of resources. The *message-scheduling* module ensures QoS constraint satisfaction of requests that have already been initiated. The data and request management functions in turn require auxiliary services such as clock synchronization,

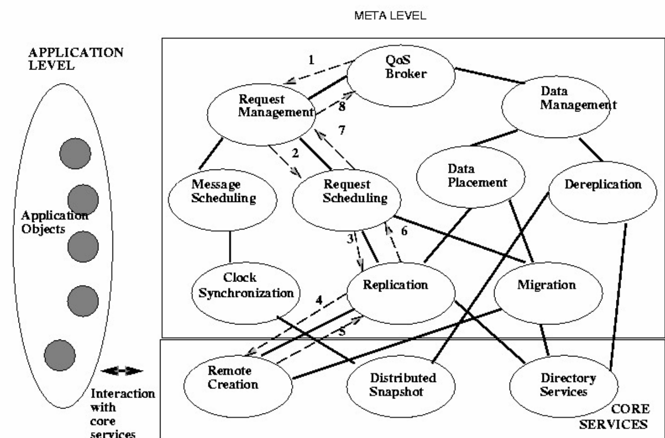

Figure 2. The QoS Broker

replication, dereplication and migration. So far, we have focused on the following services:

- *Replication:* to replicate data and request actors using adaptive and predictive techniques for selecting where, when and how fast replication should proceed.
- *Dereplication:* to dereplicate/garbage-collect data or request actors and optimize utilization of distributed storage based on current system load and expected future demands for the object.

- *Migration:* to migrate data or requests for load balancing, availability and locality. The interaction of migration with timing based QoS constraints is an interesting issue since it can introduce playback jitter in MM applications caused by explicit teardown and re-establishment of network connections.

The auxiliary services described above are developed using one or more of the core services - remote creation, distributed snapshot and the directory service. In order to ensure non-interference 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, placement and dereplication operate on the basis of a (conservative) snapshot of the current resource allocation and use. The placement and dereplication services do not consider the exact times at which requests arrive; in contrast, an adaptive request scheduling process makes decisions based on the exact arrival times of requests. However, without appropriate constraints on the usage of these services, inconsistencies can arise due to their interaction. The broker coordinates the service interaction by constraining the behavior of the auxiliary placement and scheduling services. For instance, the dereplication service does not dereplicate a replica that the request scheduling process is making an assignment to. Furthermore a replica assigned to an active request should not be physically dereplicated. The broker also ensures that the dereplication and placement metalevel services do not cancel one another out. While the interaction between dereplication and placement is not a functional correctness issue, it has to do with cost-effective performance of the overall system. Formal reasoning of QoS properties within a meta-architectural model, in the presence of other activities has been discussed in[V98].

3.2 A Reflective Communication Service Architecture

Although middleware platforms provide some notion of QoS specification[ZBS97], QoS enforcement[WPGS97] and mechanisms to enhance application behavior transparently at run-time[NMS99], they do not deal with flexible and safe composition of communication services. As a result, several (*implementation oriented*) communication frameworks are not well suited in distributed and highly dynamic asynchronous environments, where the communication framework must be able to automatically reconfigure itself in order to respond to application requirements and/or changes in the environment by adding, removing or composing communication services dynamically. A communication service is a mechanism that ensures certain desirable properties about the communication between two or more distributed objects, such as security or reliability. Communication services are composed to obtain their combined benefits. However, their service guarantees may crucially depend on the composition order and a semantic approach to reason about the correct composition of communication services is required. Furthermore, such composition must be constrained in order to prevent functional interference with other services of the system that could lead to an inconsistent configuration state, which may violate the semantics of the basic primitives (*safe flexibility*). Safe flexibility is required to protect the system from security threats and failures while ensuring high performance; policies to enforce this are often hard-wired across different parts of the system

Communication Framework: In order to provide correct composition of communication services to QoS-based applications in a transparent and scalable fashion, while ensuring correctness of basic middleware services in a meta level architecture for distributed resource management (*e.g.* garbage collection, remote creation), the TLAM model is extended with a composable reflective communication framework (CRCF), which customizes the base level communication services among a group of objects as follows (see Figure 3). Each base level actor has a meta level actor, called *messenger*, which serves as the customized and transparent mail queue for that base level actor. There is one *communication manager* in every node of the distributed system, which implements and controls the correct composition of communication services specified by the messenger. The messenger has four message queues: the up and down queues are used to communicate with its base level actor, serving as the actor's send buffer and customized mail queue respectively, while the in and out queues are used for interaction with the communication manager, requesting communication services that satisfy QoS constraints. The up and down queues hold raw messages from and to base level actors, while the out and in queues hold processed messages, which are messages with the required protocols enforced. Furthermore, the communication manager has a set of communication protocol actors, each of them implementing a particular communication service provided by the framework (*e.g.* reliable protocol, in-order protocol).

Communication services can be added (*plugged in*) or removed (*plugged out*) dynamically without side effects. The above scheme allows us to abstract a core set of communication services and share it between the different messengers on a node, simplifying the synchronization and composition process, while encouraging separation of

concerns in the process of message transmission and reception. In order to maintain accurate semantics and provide an efficient implementation of the architecture, the communication manager implements a set of meta level representatives, called *pool-actors*. At any instance, the pool-actor handles the communication services requested by a messenger for an individual message. In other words, every message requiring communication services is assigned a pool actor. The pool actor assures the correct order of composition of required services and provides a coordination mechanism between the messenger that requires the services and the protocols that provide it. This concept of reusable pool-actors is an efficient way to handle the service request of each messenger without having to pay the bottleneck associated with the centralization of the services in the node communication manager. In summary, the notion of pool actors provides separation of concerns and manageable concurrency in the communication process.

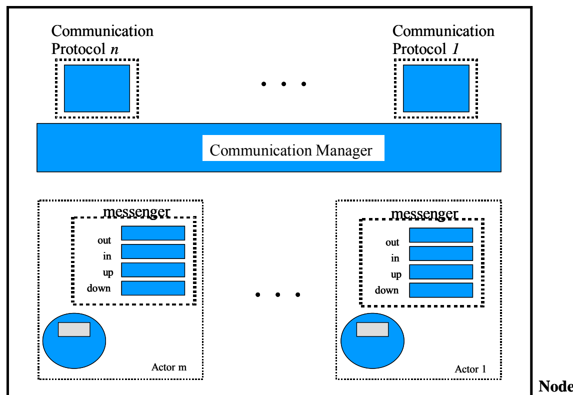

Figure 3. CRCF model: Each base level actor has a meta level entity - its *messenger* that interacts with the node *communication manager*.

4. Implementation & Performance of CompOSE|Q

Our goal is to build a system on which to prototype customizable and composable middleware services and protocols. CompOSE|Q middleware consists of a set of runtime kernels that run on the individual nodes of the distributed system and distributed middleware components that provide the required distributed services. The node runtime kernel is a substrate on which actors execute[K97]. The principal component of the node runtime kernel is a meta-level node manager actor. The node manager interfaces with a set of runtime components that implement basic services such as actor scheduling, message communication and actor creation. The distributed middleware layer of CompOSE|Q contains metalevel services such as remote creation, distributed snapshots, directory services, migration and a QoS brokerage. The distributed middleware components can execute on a single controller node or may themselves be distributed across the nodes in the system, e.g. the QoS broker operates on a single node, whereas the migration meta-actors are distributed across nodes.

4.1 Implementation of the Run Time Architecture

The current middleware environment has been implemented using Java, due to its many favorable features such as its portability across a wide variety of platforms, wide user base and support for flexibility through introspection. In our approach, we suitably 'constrain' the Java programming language to achieve Actor semantics; this methodology is also taken in systems such as the Actor Foundry[AA98]. In order to assist the three core services in achieving their task easily and efficiently, the run-time system consists of:

(1) A NodeManager that manages and coordinates various components on a node.
(2) A NodeInfoManager that manages information needed by the local actors and interfaces with the directory service.
(3) A communication sub-system that handles messaging between actors.

The NodeManager: Each node running CompOSE|Q has one NodeManager to manage actors on that node, as well as to start-up and shutdown various other modules of the run-time system. When a new actor is created it registers itself with the NodeManager and is identified by an Actor ID (aid). The NodeManager enters the new actor into a local-table which helps keep track of the actor for activities such as node checkpointing and node shutdown and notifies the MessageQmanager entity, which allocates a message queue that serves as the “in” Queue for the actor. The MessageQManager is responsible for managing mail queues for all actors on a node. To start CompOSE|Q on a node, the NodeManager has to be started first. It in turn, initiates the various other modules and communication components.

The NodeInfoManager: The NodeInfoManager is a repository of information as well as an interface to the main directory service in the distributed architecture. Currently, the NodeInfoManager implements basic functionality to:

1) Register an actor with the directory service so that it is accessible to all other nodes
2) Search for a particular actor to find out which node that actor is currently on and
3) Search for an actor object given the class name (a rudimentary naming service).

The NodeInfoManager has a local-table, which contains references to all local actors (updated and maintained by the NodeManager) and a remote-actor cache that contains information about recently accessed remote actors.

The Communication Subsystem: The communication transport layer and the CRCF module(above the transport layer), together compose the node communication subsystem. The message transport layer provides a framework for sending the outgoing messages to the appropriate node (routing) and resolving incoming messages to their appropriate actor queues (message-resolution). The CRCF module is responsible for the implementation of communication services (and their composition). Separation of these layers allows for independent customization of protocol implementations and the messaging runtime. This facilitates correct composition of protocols without interfering with the runtime communication semantics.

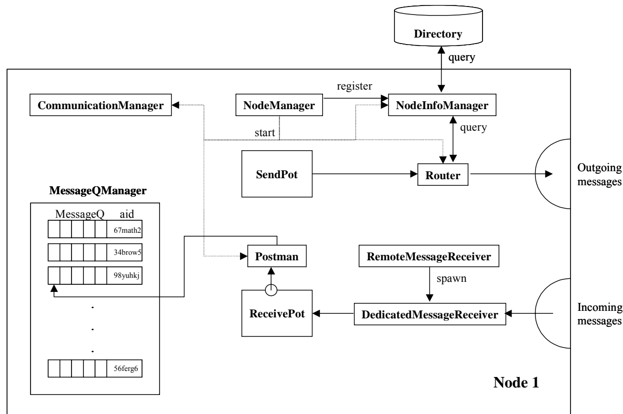

Figure 4. Schematic of the software design for the CompOSE|Q runtime environment.

The communication transport layer consists of the following components: a *Router*, a *Postman* and a *RemoteMessageReceiver* (Fig 4). The transport layer maintains two message queues on a node for incoming and outgoing messages (on that node) called *SendPot* and *ReceivePot* respectively. The *Router* consults the *NodeInfoManager* to obtain the current location (node) of the target actor. If the location of the target actor is local (i.e. on the same node), the *Router* puts the message directly into the node's *ReceivePot*. Otherwise, it sends the message to the remote node. The *RemoteMessageReceiver* (RMR) on the target node extracts the incoming message and puts it into the node's ReceivePot. The *Postman* then picks up the message and adds it to the target actor's "in" Queue.

The communication manager is instantiated in each node during system startup. When an actor is created and protocol composition services are not desired or required, its *messenger* is not created and the actor sends and receives raw messages using the transport layer. When flexible communication is required or desired, an independent *messenger* is created for every base level actor and the entire CRCF functionality is invoked. The overhead of the CRCF module is minimized in case of communications with no protocols attached. In this scenario, we tunnel raw messages through the actor's messenger directly to the transport layer.

Initial performance metrics include timing for basic operations, the overheads of the flexible communication service layer and the directory service overheads. All execution times presented here are average execution times over 100 iterations, without JVM induced overheads. Local actor creation executes in 667 μs. A locality check is used to determine whether an actor is local or remote and executes in 31 μs. A local send/dispatch operation executes in 181 μs. The timings of the remote message send can vary dramatically depending on whether the location of the target actor exists on the remote actor cache on a node (cache hit) or not (cache miss). With a cache hit, a remote send operation executes in 254 μs, whereas a cache miss causes a remote send to take 42,630 μs to execute. The performance impact of caching (at the socket cache and remote actor cache) can be seen in the effective roundtrip latencies for an initial message (approx. 3 secs) and subsequent messages (50ms). The CRCF overhead varies between 28.67–237 μs in message reception and between 209-1296 in message transmission, depending on the number of protocols used (a protocol consumes approx 45 μs) and whether or not tunneling is applied.

4.2 Implementation of Meta-Level Services

Here we describe the implementation of a few meta-level services being developed for the CompOSE|Q framework. For further details on the implementation of metalevel services see[VD00].

Remote Creation. Our initial implementation of remote creation only allows for the creation of a single actor 'α' on a remote node 'ν' with an initial state 'σ''. A remote creation request takes the form RC (α, σ, ν, β). This call creates and starts an actor, α with the specified state, σ on the remote node, ν. The call is made to the RCM, which then uses the NodeManager (NM) to implement remote creation. Remote Creation proceeds in two phases. In the first phase the two NMs (caller and callee) go through a handshake protocol to negotiate loading of classes and execute checks for security and resource availability. If the handshake is successful, then the caller NM transfers the actor-state, σ, to the callee node, ν, which then

initializes a newly created instance of the actor-class with that state.

Actor Migration. As described in the architecture of CompOSE|Q, the migration service is built using the remote creation core service. Migration behavior is specified by assigning to each node a migration Meta actor (MM) that handles migration requests for actors on that node. A remote creation service accessed via a remote creation request *RC* is used to install the migrating actor's state on the remote node. The remote creation request also includes a message to be sent to the original node containing the address of the newly created actor. To avoid confusion with other messages to the migrating actor α, a temporary surrogate actor α_s is created to receive this message. The specification of the migration behavior based on the remote creation service refines the 3 stages of the service specification (initialization, remote creation and rerouting). When the MM receives a request to migrate an actor, MigReq (α,ν), where 'α' is the actor to be migrated and 'ν' is the remote node, the described actions are executed by the migration mechanism.

Message passing between nodes constitutes the main overhead for both migration and remote creation. A round-trip between nodes takes an average of 41ms. and a remote creation call involves at least one round trip message. Thus, excluding messaging overhead, remote creation takes only about 7ms, which is just 1ms more than a local actor create. In migration there are more messages being passed around, so the timings are expectedly higher (82ms). Migration includes the time for a remote creation (48ms) and communication of the new actor address to the surrogate actor.

Directory Services in CompOSE|Q. Our initial implementation of the DS uses the Lightweight Directory Access Protocol (LDAP)[WHK97], which is easily adaptable to many commercial directory services such as NDS (Novell Directory Service). Recent work on efficient implementation of LDAP query processing includes directory caching and flexible management of lists[KNS00]. We utilize the OpenLDAP Group's slapd LDAP server[OLDAP] for our implementation. We used the Berkeley Sleepy Cat Database as the backend for slapd. An abstraction layer was implemented that presented only the necessary functionality via the Netscape Java LDAP API. Following preliminary testing of the DS, tuning was done to ensure faster performance for both access and updates. On our test system (Sun Sparc 5, running Solaris 2.7), priority paging was enabled to enhance system response and reduce the effect of system I/O paging. In terms of improving performance on the directory side, attribute indexing was used to improve search times; we primarily index the commonly queried ActorID attribute. Caching was also used, both for the entries and indexing. These parameters were tuned to the system, and a variety of tests done to determine the optimal settings.

Directory access overhead is substantially high in comparison to the native execution time of basic operations. We are currently working on removing directory access out of the critical data-path for frequently used operations. We are also studying scalability issues in directory services. Our initial attempts at performance optimization have yielded significant improvements over raw access and update overheads. Adding an actor takes 27,463μs, which is an 80% improvement over raw access times. Adding an actor attribute and searching for an attribute take 5,875 & 2814μs respectively (90% improvement over raw overheads).

5. Related Work

Commercially available object-based middleware infrastructures including CORBA and DCOM represent a step toward compositional software architectures but do not deal with interactions of multiple object services executing at the same time, and the implication of composing object services. The Electra framework[MS97] extends CORBA to provide support for fault tolerance using group-communication facilities; real-time extensions to CORBA[SLM97] to support timing-based QoS requirements have been proposed. Systems such as dynamicTAO[KRLM00] explore ways to make the various components of an ORB dynamically configurable as well as componentizing them to achieve minimal footprint for small applications[RMKC00]. Other systems such as Infospheres[CRSM96] and Globe[STKS98] use a distributed object model to construct large scale distributed systems. Globus, a metacomputing framework defines a QoS component called Qualis[LKSR98] where low level QoS mechanisms can be integrated and tested. The QuO project from BBN[ZBS97] deals with the specification, monitoring and control of application level QoS. This framework has been further extended to support flexible caching mechanisms with consistency protocols[KGDA00]. *QualMan*[NCN] is a QoS aware resource management platform that provides negotiation, admission and reservation capabilities for sharing end-system resources.

Traditional reflective languages and systems aim at providing a customizable execution of concurrent systems[S82]. The Aspect Oriented Programming paradigm[KLMM97] makes it possible to express programs where design decisions (aspects) can be appropriately isolated permitting composition and re-use

of code. Reflective systems being developed include Apertos[ILY95], 2K[KSCCMB98] and Broadway[S96]. Recent work integrates the OpenORB architecture[BCRP98] with the Aster system[IB96] to apply ideas from the software architecture community to the field of adaptive component-based middleware. The distinguishing feature of the CompOSE|Q architecture is that it is based on formal semantics that ensures safe and correct composability of the services being implemented. In other reflective models for distributed object computation[OIT92,BCRP98], 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, e.g. scheduling is reported in[SSC97].

In former approaches to dynamic installation of communication services, e.g. Broadway[S96], Actor Foundry[AA98], multiple protocols may be applied to a single component by *stacking* metalevel objects, which implement each protocol. The pluggable protocol framework[ARSK00] addresses the lack of support for multiple inter-ORB protocols and deals with integration and use of multiple ORB messaging and transport protocols, not with the composition of the protocols itself. Theoretical and formal approaches[MTD99][BB98] study modular and adaptable solutions that deal with the problems posed by composition of distributed communication services. Many such approaches assume that communication is point-to-point. Specification and formal characterization of group communication services has been developed in the context of I/O Automata[LT89] and in recent years, several group communication systems (GCS) have been developed to provide additional flexibility of communication services by allowing for dynamic composition of communication protocols[vRBM96][H98]. The CRCF approach presented in this paper enforces formal restrictions on the structure of basic communication primitives to facilitate dynamic installation and composition of communication protocols. CompOSE|Q also facilitates the customization and composition of protocols and services using a single unified framework.

6. Conclusions

The two-level architecture presented in this paper, 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 meta-levels becomes unwieldy. A purely reflective architecture provides an unbounded number of meta-levels 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.

We are currently implementing a number of extensions to the CompOSE|Q runtime. For instance, we are actively working on extending CompOSE|Q to support more distributed services for scheduling, security, mobility and fault-tolerance. We are developing a real-time scheduler in CompOSE|Q that attempts to provide soft real-time guarantees to actors, while still ensuring that the normal time-sharing actors within the system do not starve. A meta-architectural access control model is also being designed to provide the programmer with the mechanisms to implement a variety of access control policies that could be tailored to the needs of the user and application. The generic access control information is flexible enough for the programmer to use capabilities, roles or a custom policy based on the needs of the system.

We are also exploring techniques for designing and managing intelligent network infrastructures using the basic services in CompOSE|Q. These include services to support dynamic network customizations that deal with routing and network management (information collection) that adapt to application and system conditions. These services implement decisions about the degree of network awareness that applications and middleware must possess to ensure performance under varying network conditions. Further work is required to provide a generalized model that captures the architectural resources required in the server and network to support a session connection. We are further refining the concept of QoS Synchronizers[V98] and developing techniques for their specification, use and implementation.

In general, the dynamic nature of applications under varying network conditions and request traffic imply that middleware mechanisms must be dynamic and customizable. We believe that composable and safe middleware frameworks that implement cleanly defined meta-architectures enable customization of applications, protocols and system services; this will provide a foundation for the evolution of large scale distributed computing.

References

[A86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Mass., 1986.

[AA98] Mark Astley and Gul Agha. *Customization and Composition of Distributed Objects: Middleware Abstractions for Policy Management*, 6th Intl. Symposium on the Foundation of Software Engineering, 1998.

[ARSK00] A.B. Arulanthu, Carlos O'Ryan, Doug Schmidt, Michael Kircher and Jeff Parsons. *The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging*, Proc. of Middleware 2000, New York, 2000.

[BB98] Lynne Blair, and Gordon Blair. *The Impact of Aspect-Oriented Programming on Formal Methods*, Proceedings of ECOOP Aspect-Oriented Programming Workshop, Brussels, 1998.

[BCRP98] Gordon Blair, Geoff Coulson, Phillipe Robin and Michael Papathomas. *An Architecture for Next Generation Middleware*. In Proc. of Middleware'98, Lake District, England, November 1998.

[CL85] Chandy, K. M. and Lamport, L. *Distributed Snapshots: determining global states of a distributed system*, ACM Transactions on Computing Systems, 1985.

[CRSM96] K. Chandy, et al. *A Worldwide Distributed System using Java and the Internet*. Symposium on High Performance Computing (HPDC-5), Syracuse, New York, Aug. 1996.

[H89] Helary, J. *Observing global states of asynchronous distributed applications*, Proceedings of the Workshop on Distributed Algorithms, Springer-Verlag, 1989.

[H98] Mark Garland Hayden. *The Ensemble System*, PhD Thesis, Department of Computer Science, Cornell University, 1998.

[IB96] V. Issarny and C. Bidan. *Aster: A Framework for Sound Customization of Distributed Runtime Systems*. Proc. of ICDCS'96.

[ILY95] J. Ichiro Itoh, R. Lea, and Y. Yokote. *Using meta-objects to support optimization in the Apertos operating system*. In USENIX COOTS (Conference an Object-Oriented Technology, June 1995).

[K97] Wooyoung Kim. *Thal: An Actor System for Efficient and Scalable Concurrent Computing*. PhD thesis, UIUC, 1997.

[KGDA00] Vijaykumar Krishnaswamy, Ivan B. Ganev, Jaideep M. Dharap and Mustaque Ahmed. *Distributed Object Implementations for Interactive Applications*. Proceedings of Middleware'2000.

[KLMM97] G. Kiezales, et al.. *Aspect-Oriented Programming*. In Proceedings of ECOOP'97, June 1997.

[KNS00] Olga Kapitskaia, Raymond T. Ng and Divesh Srivastava, *Evolution and Revolutions in LDAP Directory Caches*, Proc. of the EDBT 2000, 2000.

[KRLM00] Fabio Kon, et al. *Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB*. IFIP/ACM Middleware'2000. New York. April 3-7, 2000.

[KSCCMB98] F.Kon, et al. *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, July 1998.

[LKSR98] C. Lee et al. *The Quality of Service Component for the Globus Metacomputing System*. Proceedings of IWQoS '98.

[LT89] Nancy Lynch & M R Tuttle. *An Introduction to Input/Output Automata*, CWI Quarterly, 2(3):219-246, 1989.

[MS97] S. Maffeis and D. Schmidt. *Constructing reliable distributed communication systems with CORBA*. IEEE Communications, 14(2), February 1997.

[MTD99] Jose Meseguer, Carolyn Talcott and Denker G. *Rewriting Semantics of Meta-Objects & Composable Distributed Services*, 1999.

[NCN] K.Nahrstedt, H.-H. Chu, and S.Narayan. *QoS aware resource management for distributed multimedia applications*. Journal on High-Speed Networking, Special Issue on Multimedia Networking, 1998

[NMS99] Priya Narasimhan, Luise E. Moser & P.M. Melliar-Smith. *Using Interceptors to Enhance CORBA*, IEEE Computer, 1999.

[OIT92] Okamura, H. and Ishikawa, Y. and Tokoro, M. *A Distributed Programming System with Multi-Model Reflection Framework*, ACM SIGPLAN, 1992

[S82] B.C. Smith. *Reflection and Semantics in a Procedural Language*. PhD thesis, Massachusetts Institute of Technology, Jan 1982.

[S96] D.Sturman. Modular Specification of Interaction Policies in Distributed Computing, PhD thesis, University of Illinois at Urbana-Champaign, May 1996, TR UIUCDCS-R-96-1950

[SLM97] D.C. Schmidt, D.Levine, and S.Mungee. *The Design of the TAO Real-time Object Request Broker*. Computer Comm. Special Issue -Building Quality of Service into Distributed System, 1997.

[SSC97] Ashish Singhai, Aamod Sane and Roy Campbell, "Reflective ORBs: supporting robust, time-critical distribution", ECOOP 1997.

[STKS98] M. van Steen, A. Tanenbaum, L.Kuz, and H.Sip. *A Scalable Middleware Solution for Advanced Wide-area Web Services*. In Proc. Middleware '98, The Lake District, UK, 1998.

[vRBM96] Robbert van Renesse, Kenneth Birman & Silvano Maffeis. *Horus: A Flexible Group Communication System*, Communication of the ACM, 39(4):76-83, 1996.

[V98] N. Venkatasubramanian. *An Adaptive Resource Management Architecture for Global Distributed Computing*. PhD thesis, UIUC, 1998.

[VAT92] N. Venkatasubramanian, G. Agha, and C.L. Talcott. *Scalable distributed garbage collection for systems of active objects*. In International Workshop on Memory Management, IWMM92.

[VD00] N. Venkatasubramanian et al. *Design & Implementation of CompOSE/Q.* Technical Report, University of California, Irvine, 2000.

[VR97] N. Venkatasubramanian and S. Ramanathan. *Effective load management for scalable video servers*. In Proceedings of ICDCS97, May 1997.

[VT95] N. Venkatasubramanian and C.L. Talcott. *Reasoning about Meta Level Activities In Open Distributed Systems*. In 14th ACM Symposium on Principles of Distributed Computing, 1995.

[WHK97] M Wahl, T Howes and S. Kille. *LDAP (v3)*, IETF RFC 2251, December 1997.

[RMKC00] Manuel Roman et al. *LegORB and Ubiquitous CORBA*. Proceedings of the IFIP/ACM Workshop on Reflective Middleware (RM2000), April 2000.

[ZBS97] J. Zinky, D.Bakken and R. Schantz. *Architectural Support for Quality of Service for CORBA Objects*. In Theory and Practice of Object Systems, pages 41-49, 1997