Design and Implementation of a Composable Reflective Middleware Framework

Nalini Venkatasubramanian, Mayur Deshpande, Shivjit Mohapatra, Sebastian Gutierrez-Nolasco and 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 provide initial performance measurements. We also illustrate how flexible communication mechanisms can be supported efficiently in the CompOSE|Q framework.

1. Introduction

In the next decade, distributed computing will be the default mode of operation. Large scale distributed systems handle applications that are open and interactive; 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 [CBC98][SSC97]. While such distributed middleware enables the modular connection of software components to manage the resources of an open distributed system (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, reliability, 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. The 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 metaarchitectural 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 two level meta architecture, a formal semantic model for reasoning about middleware in Section 2. Section 3 discusses the basic CompOSE|Q architecture and metalevel services that are being developed within the CompOSE|Q framework. Section 3 also describes a reflective communication framework architecture and how it is integrated into CompOSE|Q. Section 4 addresses the implementation of the CompOSE|Q system including the runtime environment, metalevel resource management services and various issues therein. A preliminary look at the performance of the system is given in Section 5. Finally, related work and future directions are discussed in Section 6.

2. The Two Level Meta Architectural Model

To simplify development of applications, we use Actors [G86], 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. In [VT95], the TLAM (Two Level Actor Machine) model was presented as 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 this 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).
We use commonly observed patterns in distributed systems to 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.

### 3. The CompOSE|Q Architecture

Based on the two level metaarchitecture, 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.
• 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 composable with other meta-level services [V98]. No acknowledgement is required for a remote creation request. If the requester needs to know that the request has been met, or the names of some of the newly created actors, then this is implemented by specifying appropriate messages as part of the requested fragment, and observing their delivery.

Migration: 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 $\alpha$ to the node $\nu$. 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. By basing migration on RC we have ensured composability of migration with other Meta-level services [V98]. The final configuration finalizes the migration process and establishes transparent access to the migrated actor.

**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). 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 [NV98]. 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.

The implementation of the snapshot primitives consists of a "Global Snapshot MetaActor" in the system that coordinates the distributed snapshot with local snapshot metaactors on each node responsible for capturing state information on the respective nodes. At present, we allow only one ongoing snapshot process at any point in time. We define two forms snapshot operation - (a) Ordered snapshot and (b) Unordered snapshot. The ordered snapshot assumes a strict ordering of nodes in the system and proceeds to take snapshots on individual nodes in a specific order. The global snapshot metaactor sends a message to the local snapshot worker on a node, waits for a response from that node and then send a snapshot request next node on the list, until the last node returns. The initial implementation assumes that the global snapshot manager takes care of the ordering. In the future, the local snapshot workers will propagate the snapshot message to the nodes that follow it in the ordering. In the unordered snapshot, the global snapshot
meta actor proceeds to simultaneously initiate snapshots on individual nodes. The local snapshot workers on the nodes return completion messages to the global snapshot meta actor where the state of the snapshot is maintained.

**QoS Brokerage Service:** Our work focuses on the design of mechanisms and policies needed to enforce QoS constraints in the actor-based runtime environment. We extend the basic metarchitectural 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 meta-level services in CompOSE|Q is illustrated in Figure 2; the services have been mapped to a specific distributed server architecture [VR97, V98].

![Figure 2. The QoS Broker.](image)

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, 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 storage space in the distributed system based on current load in the system as well as 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. For instance, in MM applications, significant playback jitter that can be introduced by explicit teardown and re-establishment of network connections must be minimized.

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].

**RT Scheduling in CompOSE|Q**

The real-time scheduler in CompOSE|Q attempts to provide soft real-time guarantees to actors with such requirements, while still ensuring that the normal time-sharing actors within the system do not starve. Real time actors are modeled as threads that need to execute periodically and have to be guaranteed a certain execution time within each period. Each such actor specifies its RT requirements as a 3-tuple namely a period, execution time and deadline (same as the period). The scheduler supports both the Earliest Deadline First (EDF) and the Rate Monotonic Scheduling (RMS) algorithms. The real-time scheduler performs an admission control test for every new real time base actor that requires to be scheduled by it. If the desired scheduling algorithm can accept the new actor while still ensuring that the deadlines are met, the actor is
accepted for scheduling and registered with the system. It must be noted that the Real-time scheduler runs only when real-time actors are scheduled to run. An actor without any real-time requirements runs like any other normal actor with implicit priority assignment by the native scheduler. In this sense, the real-time scheduler does not interfere with the performance of normal base actors in the absence of other base actors with soft real-time requirements. The soft real-time scheduler always runs at the highest possible fixed priority. The real-time scheduler periodically wakes up and schedules the next actor. If there is no real-time base actor waiting to run in the system, the scheduler just goes to sleep and the non-real-time base actors with dynamic priority can fairly obtain CPU by using the normal time-sharing scheduler.

**Access Control:** The idea here is provide the programmer with the mechanism to implement any kind of access control policy that would be tailored to the needs of the system. Obviously there will be some tradeoff between providing complex policies to ensure access control in the system and keeping the communication overhead to a minimum. As in any access control model, the entities that need to be protected need to be defined, as well as the necessary operations required to provide that protection. Since we are dealing with actors, the access control needs to be enforced from the point of view of incoming and outgoing messages to a particular object (actor). There are two components in this architecture, a Meta-Level Security Actor (MSA), which would be resident on each node and be responsible for carrying out the checks on each message destined to a particular actor. The information used for doing access control is quite generic at this stage, and should be flexible enough for the programmer to use capabilities, roles or a custom policy based on the needs of the system. This state information can only be referenced by the MSA; it is typically held within the directory service entry corresponding to the base-actor and cached within the local node MSA.

### 3.2 A Reflective Communication Service Architecture

Middleware abstracts communication services, allowing the development of network transparent distributed applications with QoS requirements. With advances in wireless communication, mobile computing and real time communications, dynamic customization of communication protocols will be required. Although middleware platforms provide some notion of QoS specification [ZBS97], QoS enforcement [SGHP97][WPGS97][OMG99], 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, when two or more communication services are composed in order to obtain their combined benefits, they service guarantees are not always preserved. Hence, preservation of 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. Furthermore, the integration into a middleware platform can introduce additional problems caused by the interaction of communication protocols with middleware services, e.g. providing reliable group communication in the presence of object migration.

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]. Often, the communication mechanisms are built into the architecture and dynamic installation and revocation of communication protocols on the fly to deal with such changes are cumbersome and error-prone. On the other side, research in distributing computing [AA98][S96][KLMM97][M96] and formal methods [R95][MTD99][BB98] have used the concept of reflection to provide modular and adaptable solutions to deal with the problems posed by composition of distributed communication services.

**Communication Framework:**

Meta level facilities are especially useful in an open distributed system as it allows us to specify, control and reason about the composability of multiple resource management services [MTD99]. In order to provide run-time flexibility in composition of communication services while ensuring correctness of basic middleware services in a meta level architecture for distributed resource management (e.g. garbage collection, remote creation), it is necessary to develop a communication framework that can distinguish and handle different types of messages and communication protocols among a group of objects. The framework must integrate the enforcement QoS parameters, particularly timing constraints, such as end-to-end delay, jitter and synchronization skew into the resource management and message handling processes. This requires flexibility and adaptability in the composition of communication protocols.

In order to provide correct composition of communication services to QoS-based applications in a transparent and scalable fashion, the TLAM model is extended with a composable reflective communication framework (CRCF), which customizes the base level communication services 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 purely reflective architectures, reasoning about the semantics of correct communication composition may be complicated; moreover, its implementation may be inefficient. 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.
Figure 3. CRCF model: Each base level actor has a meta level actor, called messenger and there is one meta level entity called communication manager in every node of the distributed system, which implements and controls the correct composition of communication services specified by the messenger.

4. Implementation 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. 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; migration; a basic directory service and state capture (distributed snapshot) mechanisms. We are also concurrently developing modules for QoS management such as the QoS broker, data and request management modules etc. to be integrated into the system. The distributed middleware components can execute on a single controller node or may themselves be distributed across the nodes in the system. For example the QoS broker operates on a single node, whereas the migration meta actors are distributed across nodes. The current middleware environment has been implemented using Java. Java was chosen due to its many favorable features such as its portability across a wide variety of platforms, wide user base and its support for flexible and powerful features such as introspection and run-time invocation of methods. Our approach to implementing an Actor based environment is to suitably 'constrain' an existing programming language (Java) to achieve Actor semantics; this methodology is also taken in systems such as the Actor Foundry [AA98]. Another approach would have been to define an Actor Language [K97] and then implement a compiler and runtime environment for that language.

4.1 Implementation of the Run Time Architecture

The run-time system must provide for an environment that efficiently supports the Actor paradigm. This includes operations such as defining a new Actor class, creating an Actor of that class as well as communication between the various Actors. The run-time system must also be designed in such a way as to assist the three core services in CompOSE|Q (remote creation, distributed-snapshot and directory) in achieving their tasks easily and efficiently. Keeping these goals in mind, the current run-time system consists of:

(1) A way to define new Actor classes easily by extending a basic Actor class. The Actor class provides the basic Actor semantics -- (a) a globally unique (across time and nodes) ActorId, (b) a mail queue to store incoming messages, (c) a thread to process those messages, (d) the ability to create new actors and (e) the ability to send messages to other actors.

(2) A NodeManager that manages and co-ordinates various components on a node. These tasks include:
(a) Creating and destroying actors, (b) Starting up and shutting down the various run-time modules, and (c) Communicating with the distributed middleware components.

(3) A NodeInfoManager that manages information needed by the local actors and interfaces with the directory service.

(4) A communication sub-system that handles messaging between actors.

**The Actor Class:** The Actor Class is a Java base class that supports Actor semantics. Specifying a new Actor class is as easy as deriving from this base class and adding the needed Actor state (variables) and behavior (methods) for the new class. Thus, a very primitive MathActor class would look like:

```java
public class mathActor extends Actor
{
    // state
    private static final float PI = 3.1415927;
    // behavior
    public void Add( int x, int y )
    { // Add functionality }
    public void Subtract( int x, int y )
    { // subtract functionality }
}
```

Programmatically, creating an Actor is like creating any other Java object (via the new operator). The implementation of actor creation encompasses many stages. First, a unique ActorId is generated and assigned to that actor. A MessageQManager entity allocates a message queue that serves as the mailQueue for the actor. The MessageQManager is responsible for managing all the actor mail queues on a node. A hashed methodTable is used in storing method objects corresponding to an actor and invokes the appropriate actor-method as specified in the incoming message (See Figure 4). The methodTable is an optimization to bypass constructing a method-object each time a method needs to be invoked. Generation of the methodTable is itself an expensive operation and thus is further optimized by maintaining a cache of recently constructed methodTables. Note that a methodTable is the same for all objects of an actor class. The actor then registers itself with the NodeManager and finally, starts up its internal thread which then initiates the processing of messages from the actor’s message queue.

**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. 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. To start CompOSE|Q on a node, the NodeManager has to be started first. It in turn, initiates the various other modules such as the NodeInfoManager and communication components such as the Router, Postman, and RemoteMessageReciever.

**The NodeInfoManager:** The NodeInfoManager is a repository of information as well as an interface to the main directory service in the distributed architecture. The NodeInfoManager currently implements basic
functionality to:
1) Register an actor with the directory service so that it is accessible to all other nodes
2) Search for an 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. This remote-actor cache helps alleviate the overhead of communicating with the
directory service to obtain necessary object information, e.g. current location of object.

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

The Communication Subsystem: The communication transport layer and the CRCF module, (implemented
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).

The communication transport layer consists of the following components (implemented as threads in the
runtime system): a Router, a Postman and a RemoteMessageReceiver (See Figure 4). The transport layer maintains two message queues on a node for all incoming and outgoing messages (on that node) called SendPot and ReceivePot respectively. When an actor on a node sends a message, the message is put into the node's SendPot. The Router picks up messages from the SendPot, extracts the target of the message, and then 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. If the target actor is remote, the Router sends the message to the remote node. The RemoteMessageReceiver (RMR) on the target node handles incoming messages. It extracts the message and puts it into the node's ReceivePot. The Postman then picks up the message and adds it to the target actor's message queue. Messages are currently sent using TCP sockets and as an optimization the Router maintains a cache of open-connections that it reuses while sending out messages.

The design of the CRCF module must provide for communication services to be added or removed dynamically without side effects. The communication manager is instantiated in each node during system startup. Each communication manager has a set of communication protocol actors (cpas) and a set of pool-actors (pas). When an actor is created and protocol composition services are not desired or required, its messenger is not created; the actor sends and receives raw messages using the transport layer (that is, messages without protocol services attached to it). However, when flexible communication is required or desired, an independent messenger is created for every base level actor and the entire CRCF functionality is invoked. Since an actor can potentially communicate differently (using different protocols or not using any protocol) with each of its acquaintances at any time, the overhead of the CRCF module must be minimized in the case of communications with no protocols attached. In this scenario, we tunnel raw messages through the actor's messenger directly to the transport layer. Messengers and communication protocol actors derive their basic functionality from the base Actor definition in the CompOSE/Q framework, with extensions to implement the desired behavior. On the other side, pool-actors are implemented as a pool of actors with a pool-handler. The pool-handler handles incoming requests to communication services (from messengers) and assigns them to a specific pool-actor in the pool. Pool-actors are created at startup and stored in a stack, where they sleep until the pool-handler request their services.

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. Due to lack of space, we do not describe the implementation details of the Snapshot Service, QoS Broker & Security Broker that are also currently being implemented.

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 ($\alpha$, $\sigma$, $\nu$, $\beta$). This call creates and starts an actor, $\alpha$ with the specified state, $\sigma$ on the remote node, $\nu$. The call is made on the RemoteCreationManager (RCM), which then uses the NodeManager (NM) to implement the remote creation.

Implementation-wise, the crux of remote creation is about starting an instance of a particular class on a remote node. If remote node does not necessarily have the required class, a mechanism is needed to transfer class files from one node to another and to dynamically load these classes at run-time. Two RCLs (caller and callee) communicate with each other to transfer and load the necessary classes. 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 could potentially include other checks for security and resource availability. If the handshake is successful, then the caller NM transfers the actor-state, $\sigma$ to the callee node, $\nu$ that then sets that state to a newly created instance of the actor-class.

**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 actors 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 $\alpha$, a temporary actor $\alpha_s$ 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. When the MM receives a request to migrate an actor, MigReq (\(\alpha, \nu\)), where '\(\alpha\)' is the actor to be migrated and '\(\nu\)' is the remote node, the following actions are executed by the migration mechanism.

**Initiation Phase:**
1. Create a surrogate actor $\alpha_s$ on the original node to receive the newly created actor address. The sole job of this surrogate is to receive the new address of the migrated actor.
2. Stop running the actor '\(\alpha\)' and queue all it's incoming messages.
3. The MM registers to be notified when the surrogate actor $\alpha_s$ receives a message.
4. Send a remote create message to the desired node with the description of the actor configuration to be created and the surrogate actor address to which the remote address must be sent.

**Remote Creation Phase:** The remote creation is executed and $\alpha'$ is created with the desired behavior and $\alpha'$ is sent to $\alpha_s$.

**Rerouting or Finalization Phase:** The address of the newly created component is delivered to the surrogate and this signals the migration service to complete migration. The original actor is now changed to act as a forwarder so that all messages in it's queue are forwarded to $\alpha'$. 
Directory Services in CompOSE

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). LDAP has been utilized for a variety of applications, due to its flexibility and simplicity. Various approaches for directory caching have been studied extensively in the context of LDAP [KNS00, CKS99] to reuse cached LDAP directory entries for answering LDAP queries. Other techniques, such as flexible management of lists (distribution lists, access control lists etc.) in directories have been studied by making efficient & evaluable extensions to the LDAP query language for the location and expansion of lists [JJSV98]. We utilize the OpenLDAP Group’s slapd LDAP server [OLDAP] for our implementation. Slapd is a stand-alone LDAP daemon that listens for LDAP connections on a port, responding to the LDAP operations it receives over these connections. We used the Berkeley Sleepy Cat Database (DB) [BSCAT] as the backend for slapd. An abstraction layer was implemented that presented only the necessary functionality via the Netscape Java LDAP API [NSCPE].

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 since the file system was being used heavily on the machine. This was particularly useful since the system had ample memory and this reduced the effect of the file system I/O paging out significant portions of important application heap and stack address space. 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. Write performance was improved by disabling the default behavior of synchronizing the DB after each modification. Due to our fairly small data store and frequency of updates this improves performance at the cost of a small possible risk to data consistency. To ensure strong consistency criteria, it is possible to use slave servers, which synchronize the updates in the background to ensure greater reliability of the data, as use of the DS diversifies. 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. Other minor tuning was done, such as customizing the amount of logging that the server was doing, as well as configuring the system log daemon (syslog) not to synchronize the file system with every write. As the system evolves, we intend to enhance the hardware (move the DS to a dedicated machine with a SCSI/RAID setup), which will provide further improvement in performance.

The RT Scheduler is implemented as a set of Java threads.

1) The RT Broker: The RT Broker thread is at the core of RT scheduler thread-group. It creates and initializes all the necessary data-structures and threads. Once a real-time actor is admitted for scheduling with the RT Scheduler, the broker creates a new schedule and loads it onto the dispatch table. The dispatch
table is a shared memory that can be accessed by the dispatcher for reading the schedule and dispatching the real-time actors. The RT Broker runs at the normal time-sharing priority level.

2) The Dispatcher: The dispatcher is the thread that actually dispatches the real-time actor threads according to the schedule calculated by the RT Broker. In order to schedule the real-time actors, the dispatcher must run at the highest possible fixed real-time priority. The dispatcher schedules the real-time actors by manipulating the priorities appropriately, according to the schedule written into the dispatch table. It examines each time slot periodically and yields the CPU time to the non-real-time base actors if this time slot is non-scheduled (free).

3) The Communication Threads: The RT scheduler uses two communication threads, namely the application thread and the broker thread for inter-thread access. The application thread reads the QoS requirements from the application base actor and performs the admission control before notifying the RT Broker of the newly admitted real-time actor. The broker thread communicates with the dispatcher, and keeps track of the dead threads and aids in the cleanup process.

The Application thread and the Broker communicate using a shared data structure. A similar operation is used for communication between all the threads. In order to maintain the priority of the messages of the real-time jobs the scheduler must also boost the priority of the postman and the RMR in the system, so that the real-time messages get the appropriate precedence over other messages.

![Figure 5: The Design of the RT Scheduler](image-url)
5. Preliminary Performance Results

In this section, we focus on the performance of the node-level runtime environment. The platform used to measure the initial performance of the CRCF model consisted of Sun Ultra5 workstations (333Mhz UltraSPARC IIi with 256KB external cache and 128 MB RAM) running Solaris 2.7 connected via a 10Mb/s Ethernet link. The runtime system is implemented in Java (JDK 1.2.2 with green threads) and the evaluation results are presented in the context of this environment. In the future, we expect significant portions of the runtime environment to be further optimized by using native code implementations or frameworks like JNI or NIO [JAVAIO].

Performance of the Runtime Primitives: Performance metrics of interest include timing for basic operations, the overheads of the flexible communication service layer and the directory service overheads. The execution times depicted below are average execution times over 100 iterations. All times are in microseconds. The tables show the execution times with and without JVM induced overheads.

(a) Basic Operations: The execution times of the various basic operations are summarized in Tables 1-4. Local actor creation executes in 667 microseconds. A Locality check is used to determine whether an actor is local or remote and executes in 31 microseconds (68 microseconds with JVM). A local send/dispatch operation executes in 181 microseconds. 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 microseconds, whereas a cache miss causes a remote send to take 42,630 microseconds 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 (approximately 3 seconds) and subsequent messages (50 milliseconds).

<table>
<thead>
<tr>
<th>Operation</th>
<th>Without JVM Calls Timed (in microseconds)</th>
<th>With JVM Calls timed (in microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Actor Creation</td>
<td>667</td>
<td>17,829</td>
</tr>
<tr>
<td>Locality Check</td>
<td>31</td>
<td>68</td>
</tr>
</tbody>
</table>

Table 1. Execution times of basic operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Without JVM Calls Timed (in microseconds)</th>
<th>With JVM Calls timed (in microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Message Send (total)</td>
<td>181</td>
<td>360</td>
</tr>
<tr>
<td>- Actor.sendMessage</td>
<td>103</td>
<td>103</td>
</tr>
<tr>
<td>- SendPot.getMessage</td>
<td>47</td>
<td>189</td>
</tr>
<tr>
<td>- Locality Check</td>
<td>31</td>
<td>68</td>
</tr>
</tbody>
</table>

Table 2. Local message send overhead.
Remote Message Send (total) 254 465
- Actor.sendMessage 103 103
- SendPot.sendMessage 47 189
- Locality Check (Fails) 31 68
- Remote Actor Cache Lookup 73 105
- Directory Access (in case of cache miss) 2814

Table 3. Remote message send overhead.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Without JVM Calls Timed</th>
<th>With JVM Calls timed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Receive (total)</td>
<td>141 (in microseconds)</td>
<td>318 (in microseconds)</td>
</tr>
<tr>
<td>- ReceivePot.putMessage</td>
<td>54 (in microseconds)</td>
<td>49 (in microseconds)</td>
</tr>
<tr>
<td>- Postman.getMessage</td>
<td>46 (in microseconds)</td>
<td>130 (in microseconds)</td>
</tr>
<tr>
<td>- MessageQManager.getQForActor</td>
<td>25 (in microseconds)</td>
<td>102 (in microseconds)</td>
</tr>
<tr>
<td>- MessageQ.addMessage</td>
<td>16 (in microseconds)</td>
<td>37 (in microseconds)</td>
</tr>
</tbody>
</table>

Table 4. Message receive overhead.

(b) Communication Framework: The amount of time required for the messenger and the communication manager startup is showed in Table 5. In particular, the communication manager startup is dependent of the number of protocols initially implemented and the size of the pool specified. Although expensive at startup (in terms of system resources), the pool of actors really improves the performance at runtime.

<table>
<thead>
<tr>
<th>Creation Overhead</th>
<th>Time (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication Manager</td>
<td>5216.5</td>
</tr>
<tr>
<td>Pool-actors (10)...................(3922)</td>
<td></td>
</tr>
<tr>
<td>Protocols (3)........................(928.5)</td>
<td></td>
</tr>
<tr>
<td>Other initializations.... (366)</td>
<td></td>
</tr>
<tr>
<td>Messenger</td>
<td>530</td>
</tr>
</tbody>
</table>

Table 5. Creation overhead of the CRCF module.

The overhead of the communication subsystem is showed in Tables 6-7. Since communication protocols may be required or not at any time, the time needed to send the message is measured: (a) with CRCF but no protocols attached (a raw message, tunneling the messenger) and (b) with CRCF and 2 protocols attached (reliable, in-order).

<table>
<thead>
<tr>
<th>Total overhead in message transmission (send operation)</th>
<th>Time (microseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without using CRCF (transport layer)</td>
<td>181</td>
</tr>
<tr>
<td>With CRCF module, but no protocols attached (+28.67)</td>
<td>209.67</td>
</tr>
</tbody>
</table>

All times shown do not include inlined JVM method calls
Reducing the overhead of protocol layering is a difficult task since; most of the protocols need to add additional information to the message (in the form of a header); and the execution contributes some processing overhead for every message that is sent or received. In the actual implementation, the protocol overhead varies from 7 to 20 microseconds.

(c) Directory Service Operations: As can be observed from Table 8, the overhead of directory access is high in comparison to the native execution time of basic operations. We are currently working on removing directory access out of the critical datapath for frequently executed operations. We are also studying scalability issues in directory services. Several optimizations were made to improve performance and we were quite successful in improving upon the results from the first prototype.

<table>
<thead>
<tr>
<th>Directory Operation</th>
<th>Time (microseconds)</th>
<th>Before Optimization</th>
<th>After Optimization (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adding Actor</td>
<td>137,844</td>
<td>27,463 (80)</td>
<td></td>
</tr>
<tr>
<td>Adding Actor Attribute</td>
<td>59,086</td>
<td>5875 (90)</td>
<td></td>
</tr>
<tr>
<td>Searching for Attribute</td>
<td>42,165</td>
<td>2814 (93)</td>
<td></td>
</tr>
</tbody>
</table>

Table 8. Performance of directory operations
A preliminary study was carried out on the scalability of the Directory Service by increasing the number of actors in the system and carrying out repeated queries and attribute changes. As can be seen, registration of an actor with the DS incurs significantly more overhead than the query or attribute modification operations. Performance is fairly consistent throughout, however as the number of actors approach 1000, there is an increase in access times for both adding and querying for an actor. This could be attributed to the fact that modifications to the cache sizes need to be made to accommodate the increasing load. We intend to further study directory scalability and make further changes to allow the DS to handle higher loads. Note that no changes or improvements were made as the tests progressed; e.g. cache sizes etc. were kept at the same values we determined were optimal for normal use. Obviously, the performance could be tailored depending on the requirements of the system.

(c) Migration & Remote Creation Services:

<table>
<thead>
<tr>
<th></th>
<th>Getting and Loading Remote Class (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RemoteClassLoader</td>
<td>60</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>With Remote Class Loading (milliseconds)</th>
<th>Without Remote Class Loading (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote Creation</td>
<td>108</td>
<td>48</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Pre-migration (milliseconds)</th>
<th>Migration &amp; Finalization (milliseconds)</th>
<th>Total Time (milliseconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migration</td>
<td>21</td>
<td>82</td>
<td>103</td>
</tr>
</tbody>
</table>
The above results are an average over 10 runs. The main overhead in all these operations is message passing between nodes. A round-trip between nodes takes an average of 41 ms. A Remote Creation call involves at least one round-trip message (phase 1). Thus, excluding the messaging overhead, remote creation takes only about 7 ms, which is just 1 ms more than an local actor create. In migration there are more messages being passed around, so the timings are expectedly higher (82 ms). The Migration and finalization phase includes the time for a remote creation (48 ms) plus the waiting of the surrogate actor for a message from the migrated actor.

(c) RT Scheduler:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dispatch Latency</td>
<td>113 microseconds</td>
</tr>
<tr>
<td>Dispatch Table calculation</td>
<td>66.9 milliseconds</td>
</tr>
<tr>
<td>Scheduler Response</td>
<td>16 microseconds</td>
</tr>
</tbody>
</table>

The dispatch latency is the length of time to switch from the end of one time-slot to the beginning of the next time-slot. The dispatch latency is the scheduling overhead and should be kept at the minimal value. Dispatch Table Calculation Time is the time taken by Broker to calculate the dispatch table. Scheduler Response Time represents the time elapsed between the application base actor contacting the real-time broker and getting its reply from the real-time broker.

6. Related & Future 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, GSLS, WBTK95] 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. 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 [KSCCBM98] and Broadway [S96]. Recent work [BBIT00] 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,CBC98,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]. 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 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.

In former approaches to dynamic installation of communication services, e.g. Broadway [S96], Actor Foundry [AA98], multiple protocols may be applied (installed) 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 [R95][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]. 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.

QoS enforcement has been a topic of considerable research in the multimedia and distributed object communities. 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. The EPIQ project [HSNL97] provides interfaces, mechanisms, and protocols to support QoS management of flexible applications.

We are currently implementing a number of extensions to the CompOSE|Q runtime to effectively integrate the various modules depicted in Figure 2. For instance, the runtime environment currently relies on the underlying scheduler implementation; we are in the process of integrating low overhead priority based and
constraint-based scheduling mechanisms within the runtime kernel. We are also actively working on extending CompOSE|Q to support more distributed services for security, mobility and fault-tolerance. We are 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. Modeling client interaction requires a notion of session and resources within a session. Further work is required to provide a generalized model that captures the architectural resources required in the server and network to support the session connection. We are further refining the concept of QoS Synchronizers [RVA97] and QoS-based sessions 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


[F96] Svend Frolund, Coordinating Distributed Objects: An Actor-Based Approach to Synchronization MIT
26

Press 1996


