A Role-Based Coordination Model and its Realization

Nianen Chen, Yue Yu, Shangping Ren and Mattox Beckman
Department of Computer Science,
Illinois Institute of Technology, USA
E-mail:{nchen3,yyu8,ren,beckman}@iit.edu

Keywords: coordination models, actors, roles, coordinators, open distributed and embedded systems

Received: February 1, 2008

This paper presents a framework to support Open Distributed and Embedded (ODE) application development based on the Actor-Role-Coordinator (ARC) model. The ARC model is a role-based coordination model developed to address three main concerns inherent in an ODE system: dynamicity, scalability, and stringent QoS requirements. It treats an ODE system as a composition of concurrent computation and coerced coordination. In particular, the ARC model uses concurrent objects that communicate with each other through asynchronous messages, i.e., actors, to model the concurrent computation of an ODE system, while the system's QoS requirements are mapped to coordination constraints. Coordination entities, i.e., roles and coordinators, impose coordination constraints on concurrent actors transparently through message interceptions and manipulations. In the ARC model, roles provide actor behavior abstractions for coordinators and coordinators are responsible for coordinating roles. In addition, a role also has local coordination responsibilities among actors belonging to that role. This coordination is called intra-role coordination which complements the inter-role coordination performed by the coordinators. In other words, under the ARC model, an ODE application is modeled by three orthogonal layers: computation, intra-role coordination and inter-role coordination. This separation not only improves software modularity and reusability, but also allows different levels of compositions. Our experiments show that the model scales well as the number of entities involved in the system increases, and that the performance overhead introduced by the external coordination layers is limited.

Povzetek: Opisano je ogrodje za model aktor-vloga-koordinator (ARC).

1 Introduction

Unlike most traditional software systems, open, distributed, and embedded (ODE) systems must be concerned with the environment in which they are executed. Such systems usually have rigid requirements on both the accuracy of the delivered functionality and the punctuality of its delivery. These requirements are manifested through Quality of Service (QoS) constraints, such as real-time, fault tolerance, energy consumption, and others. Another aspect of the environment is its extent. There can be many computational entities involved, and these entities are free to join or leave (intentionally or because of failures) at any time, introducing dynamicity into the system. The dynamicity and stringent QoS constraints add complexity to ODE systems, and distinguish them from traditional concurrent distributed systems.

Concurrent distributed computation models have been well studied over the past decades. CSP [21], π-calculus [33], and the actor model [1, 2] are good examples. These models are still widely used today as they provide a uniform way to model diversified applications. For instance, the Actor model treats "actors" as universal primitives: in response to a message an actor receives, the actor may make local decisions and decide how to respond to the next message received, create more actors, and/or send more messages. It is often used as a framework for modeling, understanding, and reasoning about a wide range of modern concurrent systems. For instance, Web Services with SOAP endpoints can be modeled as actors [20, 19]; an agent-based system can be modeled as an actor system, where (mobile) agents are modeled as (mobile) actors [28, 24]; and Sensor and Actor Network (SAN) is recently proposed to use the Actor model as the theoretical basis for sensor networks [26, 12, 7].

However, these models are well-defined mathematical abstractions for concurrent computation in an ideal distributed environment, in which simplifying assumptions are made to reduce the complexity of the models. For instance, communication among distributed entities is assumed to be both reliable and instantaneous. The focus of these models is on the functional behaviors of the computation. This may suffice for traditional and general purpose concurrent distributed applications, but for ODE systems, such assumptions about the run-time environment often do not hold. For example, in most embedded applications, a message that does not arrive on time is considered a fault, but traditional distributed computation models do not make

any guarantees about such QoS promises. What we need is a model to study QoS aware interaction, or coordination, among distributed computational entities in ODE systems. This model should accurately exhibit an ODE application's functional behaviors, and also precisely reflect the application's context, taking into account the dynamicity and stringent QoS requirements.

In order to conquer the complexity and dynamicity inherent to ODE systems, we may decompose these systems into different concerns. Separation of concerns as a software engineering principle is not new [18, 3]. However, how a concern is delineated plays a critical role in the quality of the delivered software models. A concern should be logically self-contained and, ideally, orthogonal and transparent to the other concerns in order to minimize the interference among them.

For instance, an open embedded real-time application, such as an environmental monitoring system, will send data from wide-area sensors to data processing entities on the Internet. The results are fed back into the physical world for actuation. In order to interact with the physical world in real time, open embedded applications must be able to fulfill a fundamental requirement, that fresh data be available at the right computation site at the right time. However, as Kang et. al [23] pointed out, current computing and communication-oriented paradigms face a huge obstacle in achieving this vision of open embedded real-time systems. Therefore, instead of interacting directly with a number of distributed data sources or actuators, it is important to have high level abstractions that federate distributed entities, coordinating them to abide by QoS requirements.

Consider the following simplified scenario as an example of the problem our research addresses. Suppose we have deployed infrared and radio wave sensors in an open space to detecting foreign objects. As shown in Figure 1, depending on the exact location of the foreign object, different groups of sensors will be active and generate data. In order for a control center to take appropriate action, data from the two types of sensors must be semantically consistent (i.e., indicating the same type of object) and they must arrive at the center within a specified time range.

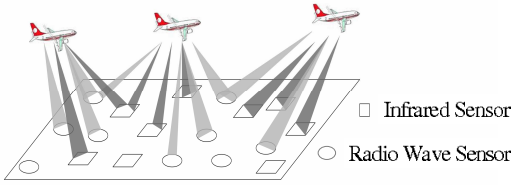

Figure 1: Open Space Surveillance

Clearly, it is a *must* that the infrared and radio wave sensors be *coordinated* in a *timed* fashion, but the nature of the problem prohibits us from statically pairing them up. The key technical challenge is that coordination is necessary, and that coordination itself is subject to QoS constraints. Furthermore, the coordinatees constitute a large and dynamically changing set. Integrating the coordination requirements into the basic computation description is not a viable solution; it only complicates an already hard problem. Unfortunately, existing research has not approached ODE applications from the coordination angle, neither have earlier coordination models addressed coordination under QoS constraints in depth. Therefore, new research is needed to support the development of ODE applications.

In this paper, we present a framework for developing ODE applications based on a role-based distributed coordination model, the Actor, Role and Coordinator (ARC) [44] model. The focus of this ARC model is to separate the QoS or non-functional requirements from the embedded applications' functional logic, and at the same time to address the dynamicity and scalability issues inherent to ODE systems. In particular, the actor layer models the concurrent computational part of an ODE system, while an independent coordination model is developed to address the federation of distributed entities to satisfy the system's QoS requirements. The coordination model contains both the coordinator layer and the role layer; the role layer provides a level of abstraction to mask the dynamicity of the actor layer from the coordinators, and each role coordinates the local group of actors that share that role. This further reduces the complexity of coordinators and improves coordination scalability. We present in detail a CORBA based implementation of ARC that provides architectural support for transparent application of QoS constraints on concurrent computations. The design criteria of the framework are performance, scalability, and flexibility.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 presents the ARC model and the composition of three autonomous entities, i.e., actors, roles, and coordinators. Section 4 presents an ARC framework and preliminary evaluation results. Finally, we conclude in Section 5.

2 Related work

Recent research has yielded significant results on coordination models and languages. In their landmark survey [41], Papadopoulos et. al. conclude that coordination models can be classified into two categories, namely data-driven and control-driven. The tuple space model (Linda) [10] represents the data-driven category, and has been extended with such systems as Lime[42], Klaim[36], and related extensions [37]. Systems such as the Ideal Worker Ideal Manager (IWIM) model [4] presents a control-driven or "exogenous" category. Recently, tuple center and ReSpecT [40, 38] provide a hybrid view.

Control-driven models, such as Abstract Behavior Types (ABT) [5], Law Governed Interaction (LGI) [34], ROAD [11], Reo [6], Orc [35], and CoLaS [13] isolate coordination by considering functional entities as black boxes. For example, the ABT model extends the IWIM model by treating both computation and coordination com-

ponents as composable Abstract Behavior Types. Like IWIM, ABT is a two-level control-driven coordination model where computation and coordination concerns are achieved in separate and independent levels. The Reo model uses a circuit-like network of composable channels to provide communication between components. Components send messages across these channels, and the geometry of the channels determine the destination or destinations of the messages. The Orc model uses "site calls" to model computation [43]. Unlike Reo channels, Orc's site calls are not expected to be persistent.

The concept of role is seen in object-oriented systems when a set of common behaviors is abstracted and can be assigned to an object [15, 25]. Roles are an important technique in a variety of computing systems. For example, in the computer security area, the Role Based Access Control (RBAC) [14] model uses roles to separate users from security policies in order to achieve scalability and flexibility. In object-oriented programming [27] and in design patterns [17], roles are used to represent solutions and experiences. There are control-driven models, such as ROAD, CoLaS, TuCSoN with Agent Coordination Contexts (ACC) [39] and Finesse [8], to name a few, that try to mitigate the scalability issues of open distributed systems by adopting role concepts. Most current role-based coordination models are based on organizational concepts, where roles abstract coordination behaviors among participants who play the roles. Cabri presents a survey of role-based coordination models in [9]. Additionally, quite a few coordination models take decentralization into account. TuCSoN [40] distributes communication abstractions (tuple centers) to Internet nodes. Every tuple center produces and maintains its own local coordination rules. CoLaS divides the whole distributed system into multiple coordination groups. Each coordination group takes care of an independent set of coordination policies. ROAD provides a recursive structure that composes fine-grained, small coordination groups into coarse-grained, large ones. LGI follows a controller metaphor and provides a controller for every object in the system, and hence implements a full-fledged decentralization.

The ARC [44] model differs from these models by separating inter-role coordination and intra-role coordination and distributing the coordination activities to coordinators and roles respectively. Roles are active entities with coordination ability instead of merely abstract interfaces. The distribution of coordination responsibility is based on the functionalities of the roles and is therefore more logical and customizable. The emphasis on active roles and the corresponding separation of inter-role and intra-role coordination distinguishes the ARC model from previous role-based coordination models.

A similar actor oriented model is advocated by Lee et al. [30, 31]. In this model, actor executions and communications are under the guidance of a "model of computation," which gives operational rules to determine when and how actors can perform their computations, update their states, or send messages to other actors. The model of computation separates the communication mechanisms and work flows of actors from their computational designs, such that reusability is possible and compositions of components are more robust.

Though the above actor-based models, like ARC, separate coordination from the functional core of a system based on concurrent actors, the focus of ARC is to address the dynamicity and scalability issues in coordinating large set of autonomous and asynchronous entities. The emphasis on role-based coordination distinguishes the ARC model from previous multi-level actor-based coordination architectures.

A set of coordination models has been proposed to address the coordination issues based on the Actor model [1, 2], such as Frølund's *Synchronizer* [16], Venkatasubramaniam's *TLAM (Two-Level-Actor-Model)* [50], and Varela's *director* [49]. One common theme of these models is the use of reflection with actors. This can be seen is systems such as ActorNet [29], and Reflective Russian Dolls (RRD) [32]. ActorNet provides a platform designed for small, heterogeneous systems. It provides a uniform environment for the actors, and makes use of `call/cc` to allow actors to migrate themselves to other nodes in the system. RRD is similar to ARC in that there are levels of coordination. Both achieve coordination by using reflection to modify the delivery of messages.

ARC, however, is a three-layer system, with functional behavior confined to the lowest level, and coordination to the upper two levels. The formal semantics of the ARC model is given in [44]. The RRD is a multi-level system; each level encapsulating the levels below it. The formal comparison between the ARC model two other coordination models, i.e. the and Reflective Russian Dolls (RRD) [47] and Reo [6], is given in [46]. Yu and et al. used Maude to further verify safety properties that can be imposed through the ARC model [51].

3 The Actor, Role, and Coordinator (ARC) model

In this section, we discuss in detail the Actor-Role-Coordinator model.

3.1 The actor model

We use active objects, i.e., actors [1, 2], to model asynchronous and distributed computations. The choice of the actor model as a foundation for the underlying computations of an ODE system is in many ways a natural one. The actor model is inherently concurrent, and systems of actors are open and distributed. However, the basic actor model does not enable the coordination of groups of actors to be specified in a modular fashion. This greatly limits their usefulness in the ODE domain. The ARC model

eliminates this impediment by introducing exogenous coordination objects, i.e., roles and coordinators.

Actors are autonomous, active entities that communicate with each other through asynchronous messages. Each actor has a unique mail address and a mailbox to receive messages. Unprocessed messages are buffered at the receiving actor's mailbox. Within each actor, there is a single thread of control that processes messages sequentially. Each actor has its own states and state dependent behaviors. The states are encapsulated and can only be changed by the actors themselves while processing messages. Different actor states may decide different behaviors that in turn affect how messages are processed. While processing a message, an actor may perform three primitive operations: send asynchronous messages to other actors, create new actors, or change its own states (become) and then become ready to retrieve the next available message in the mailbox. Figure 2 pictures the internal structure of the actors.

Here is a simple example to demonstrate the actor model. Assume an operation can be performed by a computational entity (namely, an actor) called an "executor" once and only once. Any actor that is not an executor is called a "forwarder." We distinguish an executor from a forwarder by looking at its internal state *executed*. If *executed* is false, then this actor is an executor, otherwise it is a forwarder. The behavior of an executor is as follows: when it receives a message requesting the service, it performs the service, and sets its state *executed* to be true, which triggers the actor to become a forwarder. Finally, the former executor creates another actor with the same behavior (i.e., perform the same operation) and with its state *executed* set to be false. In other words, this actor becomes a forwarder and creates a new executor. After becoming a forwarder, this actor changes its behavior. When the same message arrives at the forwarder, instead of executing the operation, it forwards the message to the executor that it created. This behavior is recursive; the "executor" to which it forwards the message may also have become a forwarder, and will in turn continue forwarding the message to successive forwarders until the current executor is located. This example explains the basic concepts of an actor and its three primitives: *send*, *create*, and *become*. All actor based computations can be implemented by these three primitives.

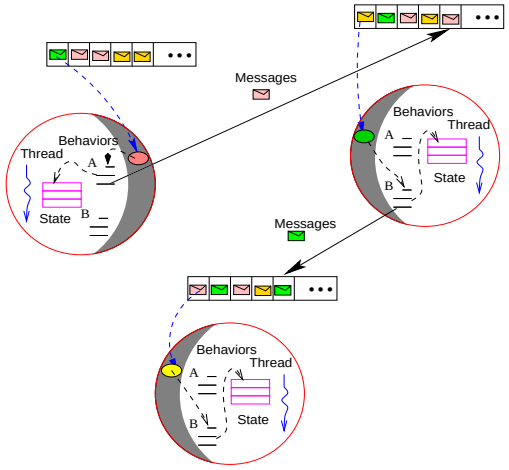

Figure 2: The Actors

3.2 The abstraction levels of ARC

In the ARC model, a role is a static abstraction for behaviors shared by a set of underlying computational actors. This abstraction decouples behaviors from their implementors and eliminates static binding among computational actors. It also shares coordination responsibilities. More specifically, there are two types of *active* coordination objects in the model: *roles* and *coordinators*. The coordination is partitioned into intra-role and inter-role coordinations and distributed among roles and coordinators, respectively. The coordinators (i.e., inter-role coordination objects) coordinate behaviors while the roles (i.e.,intra-role coordination objects) coordinate members that share the same behavior.

Coordinators constrain the coordination behavior of roles. This eventually affects a message's dispatch time and location (target) in a computation. However, computational actors and coordinators are transparent to each other. Hence, the dynamicity inherent in the computation is hidden from the coordinators. Compared to the number of actors involved in an ODE application, the number of behaviors (and therefore the number of roles) contributed by these actors is usually order(s) of magnitude smaller. Therefore, the model is not only stable, but also scalable.

Under the ARC model, the open space surveillance system introduced in Section 1 (Figure 1) can be mapped to a set of sensor *actors*, two *roles* for the infrared sensors and radio wave sensors, respectively, and a *coordinator* (Figure 4). The inter-role constraint is on the time relation of the data coming from the infrared sensor role and the radio wave sensor role. Each role can have different intra-role coordination policies. For instance, the infrared role may ensure synchrony by waiting for data from all its members, while the radio wave role only waits for data from a majority of its members.

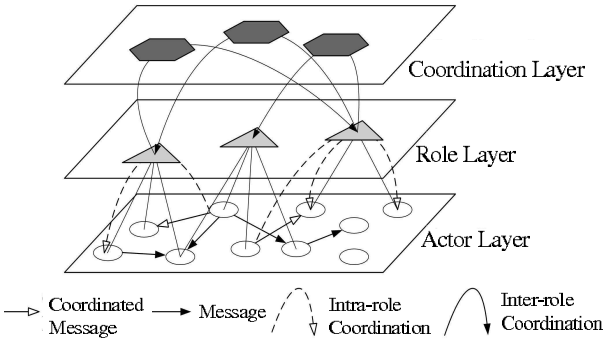

Figure 3: The ARC Model

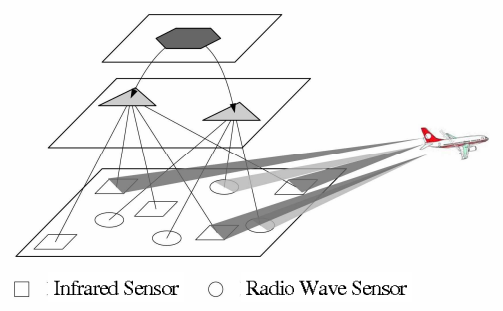

Figure 4: The ARC View of an Open Space Surveillance System

The separation of *computation*, *intra-role*, and *inter-role coordination* advocated by the ARC model is clean and orthogonal. This separation mitigates the complexity of each individual type — coordinators only concern themselves with coordinating a *small scale* of roles while roles care *only* about actors of the *same behavior*. This provides grounds for independent modeling and compositional reasoning.

Separation and transparency are the results of the following properties of the ARC model:

1. The actor layer does not depend on the coordination layer. The actors fulfill their functional behaviors independently by exchanging messages without any knowledge that the coordination entities even exist.

2. The coordination layer intercepts messages among actors and applies coordination constraints on the messages. Coordination does not require direct message interactions between actors and coordination entities.

Computation actors carry out their logical computations by reacting to messages received. As a result, if the roles or coordinators do not send any computational messages to the computation actors, the underlying computation will retain its computational properties.

The role layer bridges the actor and the coordinator layers and may therefore be viewed from two perspectives. From the perspective of a coordinator, a role enables the coordination of a set of actors that share the same static description of behaviors without requiring the coordinator to be aware of the individual actors in the set. From the perspective of an actor, the role is a coordinator that actively imposes coordination constraints on messages sent and received by the actor.

Though actors, roles, and coordinators have different responsibilities, we uniformly model their behaviors using actors. To comply with the separation of concern principle, we categorize these actors into two types: computation actors that capture system computation concerns, and coordination actors that abstract system coordination concerns. More specifically, roles and coordinators are coordination actors, whereas actors in the actor layer are computation actors. Thus, coordination actors are actors which satisfy the basic actor semantics by providing the actor operational primitives. However, they are special actors that are able to handle specific types of messages, namely, events.

In our model, actors communicate with each other via messages, which are defined as a three-tuple $< rcver - actor, op, par >$. Here $rcver - actor$ is the name of the recipient actor, op is the behavior name that the recipient actor is required to apply, and par contains the parameters that the recipient needs to perform its behavior.

Events are special messages that are atomically dispatched on coordination actors. Unlike computation messages, the recipient of an event is not an individual coordination actor, instead, events are broadcast to all coordination actors in the system. Thus, an event is defined as $< All, op, par >$ where All indicates that the event is broadcast. Though events are broadcast to all roles, we may instead use an intermediate "default" role as a mediator to receive and forward events between actors. This optimization can convert the broadcast into a two-element group-cast, reducing the communication overhead and synchronization complexity.

Another important characteristic of events is that an event is instantaneous and atomic. In other words, the generation of an event and the consumption of an event are atomic, and no actor computation messages can be processed during this period of time. This requirement guarantees that coordination constraints are applied on related messages before these messages are dispatched on computation actors.

To maintain coordination transparency and avoid interfering with the computation actors' functionalities, coordination actors are not allowed to generate or send messages to computation actors. The computation level and coordination level are connected through *events*. While messages are used between computation actors to carry out computations, events represent state changes in the system and trigger coordination related behaviors on coordination actors.

There are three events defined in an actor layer: *send* (a message is sent by an actor), *new* (creation of a new actor), and *ready* (change actor behavior if necessary and ready for next available message). All these events from computation actors are observable by roles. Upon observing the events, the roles cooperate with coordinators through *inter-role* and *intra-role* constraints to coordinate *when* and *where* messages should be dispatched among computation actors.

3.3 Roles and their responsibilities

Since an ODE system may have a large number of computational entities that are free to join or leave autonomously, the underlying actors modeling them could also be both large in number and very dynamic. Basing the stability and scalability of coordination policies on the actors themselves will be difficult. In an ODE system, however, the set of well-defined *behaviors* is limited and less dynamic. Therefore we introduce roles as a means of representing

abstractions for these system behaviors; this enables us to conceal the dynamicity and scale of the ODE environment.

In addition to representing abstractions for the properties of the system behaviors, roles also are responsible for actively coordinating their players to achieve coordination requirements. Roles serve as an abstraction by specifying membership criteria, i.e., a static specification of functional behaviors that computation actors belonging to the role must have. The role is responsible for managing the integrity of its membership. Roles also actively coordinate their member actors in order to satisfy coordination requirements. The *intra-role* coordination coerced by roles realizes and complements the *inter-role* coordination enacted by coordinators.

Membership management behavior Before a role can perform its membership management activities, the behavior abstraction, i.e. the role membership criteria, must be specified. We use logic expressions of actor states and operations to describe the criteria. More specifically, the role membership criteria are represented by a tuple $< O, A >$, where O is a set of message types (operations) that an actor must be able to process, and A is a set of attributes that actors need to display for joining the role. Any actor that is controllable by coordination rules must declare its own functional behavior, using the same tuple format.

Upon observing a *new* event or a *ready* event from a computation actor, the role acquires the newly updated behavior from the computation actor and compares it against its membership criteria. It then determines whether the actor should be added to the membership list (the actor behavior matches the role criteria), ignored (the actor was not a member and its behavior does not match the role criteria), or removed from the role (the actor was a member but its new behavior does not match the role criteria). More precisely, a role's management behavior is a mapping from a set of actor events to membership updates. Note that according to the semantics of the actor model, actors are free to reject exposing their internal states to the roles. This allows an actor to reject coordination. Such actors will belong to a "default" role that performs no coordination.

Each role has a distinct purpose. This requirement disallows overlapping criteria among roles, eliminating the possibility that conflicting constraints will be imposed on an actor by multiple roles simultaneously. This requirement has its basis in the underlying actor model: each actor has only a single thread of control and therefore may play only one role at any given time. More precisely, let $C(\gamma)$ denote the role membership criteria declared by role γ, and let $B(\alpha)$ denote the functional behaviors provided by an actor α. As we have discussed, the actor functional behaviors and the membership criteria are both represented as a comparable tuple $< O, A >$. To be added to a role, the actor functional behaviors have to match the role's membership criteria. A and Γ denote the set of actors and roles in the system, respectively, and $F : A \rightarrow \Gamma$ is a function that assigns an actor to a role. At any given time, well-defined roles and actors in a system must satisfy the following requirements:

1. Roles are exclusive: role declared behaviors do not overlap, i.e.,

$$\forall \gamma, \gamma' \in \Gamma : C(\gamma) \cap C(\gamma') = \phi$$

2. Roles are exhaustive: every actor belongs to one of the roles, i.e.,

$$(\bigcup_{i}^{n} B(\alpha_i) = \bigcup_{j}^{m} C(\gamma_j)), \text{ and}$$

$$(\forall \alpha \in A, \exists \gamma \in \Gamma : B(\alpha) = C(\gamma))$$

3. Roles are repetitive: repeated actor behaviors replicate the assignment of the actor to the same role, i.e.,

$$\forall \alpha_i, \alpha_j \in A : B(\alpha_i) = B(\alpha_j) \Rightarrow F(\alpha_i) = F(\alpha_j)$$

4. Each actor only plays one role at a given time, i.e.,

$$\forall i, j,\ j \neq i : B(\alpha) = C(\gamma_i) \Rightarrow B(\alpha) \neq C(\gamma_j)$$

Coordination Behavior As roles are abstractions of functional behaviors, it is possible that more than one actor may belong to a specific role at any given time. Actors playing the same role may need to coordinate with each other to satisfy certain QoS constraints. Such constraints are called *intra-role* coordination constraints.

A role's coordination behavior thus has two aspects: (1) it retrieves *inter-role* constraints specified by the coordinators; (2) it is responsible for enforcing both the *inter-role* and *intra-role* coordination constraints on actors. Since roles are coordination actors and are not allowed to send/receive messages to/from computation actors, message interception and manipulation is the only feasible means to apply the constraints. Furthermore, all these behaviors are triggered by observed events on computation actors. Therefore, the coordination behavior of a role can be given the following interpretation: upon observing an event from a computation actor, and based on its current states, the role may manipulate messages, generate events (which are observable by coordinators), or change its own states.

The coordination rules are enforced on actors without their awareness. The involvement of roles in the coordination process causes coordination in the ARC model to be decentralized. Active roles cause our coordination model itself to become a distributed subsystem, inheriting the full benefits that a distributed system may offer.

We can use a "Video on Demand" (VoD) application as an example to depict a role's behaviors. We assume there are multiple VoD client actors and VoD server actors in a distributed environment. Each VoD client actor can perform a *request_video* operation, while the server actor can perform *send_video* operation. However, clients

may have different requirements, which need to be met by receiving different services from servers. We therefore separate the client actors into different roles depending on their level of service attributes, i.e. a VOD client actor has a *Regular_VOD_Client Role* if its *level_of_service* attribute is set to *regular*; while a VoD client has a *VIP_VoD_Client* role if its *level_of_service* attribute is set to *very_important*. Therefore, when an actor is created or moves into the system, the roles will check its operations and attributes. For example, if a *VIP_VoD_Client* role finds that the new actor has a behavior tuple <<*request_video*>, <*level_of_service:regular*>>, which matches its role criteria, it will then help the actor join its group by performing its member management behavior.

To explain the role's coordination behavior, we assume that there are multiple VoD server actors in the environment, each of which has different resources (CPU speed, workload, memory, reserved network throughput, and so on). Based on the requests from different types of VoD clients, the *VoD_Server* role decides which server actor shall be assigned to process the current request. For example, if the request is from an actor with a *VIP_VoD_Client* role, this request will be forwarded to a *VoD_Server* actor with the highest available resources. This coordination rule is applied on the actors directly within a role, but not among roles, therefore it belongs to *intra-role* coordination.

3.4 Inter-role coordination — coordinators

In contrast to *intra-role* coordination, coordination among high-level coarse-grained roles are called *inter-role* coordination. We define another type of coordination actor, the *coordinator*, to specify *inter-role* coordination policies. These policies are written in terms of roles. A policy is a set of constraints over a set of properties. Values associated with a property are drawn from an enumerable domain. A constraint specifies a boolean relation involving a set of properties.

Similar to roles, coordinators are also active objects and impose coordination constraints based on their states. However, in our model the actor layer and the coordinator layer are mutually transparent. Coordinators do not directly apply coordination constraints on computation actors, neither do actors know of the existence of coordinators. Coordinators specify and impose policies based on abstract actor functionalities, but not on individual actors.

The role layer bridges the coordinator and actor layers. Roles propagate the events observed from the actors to the coordinators. Upon receiving such events, the coordinator locates constraints in its constraint store based on current states, and propagates the constraints to roles where these constraints are imposed on computation actors.

Consider an example in which multiple producers and multiple consumers share the same buffer. We use a producer role and a consumer role to capture the producers and consumers, respectively. The two roles must coordinate to respect the causal order (an item must be produced before it can be consumed) and buffer size. Instead of specifying the coordination among each pair of producer and consumer, we impose the coordination upon the roles which will in turn propagate the constraints to the role players.

3.5 Composition of concurrent computation and coerced coordination

Based on the ARC model, an ODE system can be specified in three steps. First, establishing the underlying functional computations (modeled by computation actors). Second, implementing the computational actors to carry out the computation. Finally, embedding the functional objects in an environment constrained by coordination actors. Here we focus on QoS constraints that can be achieved by manipulating the messages in the time and actor space dimensions. Example manipulations on the time-axis include moving messages to the beginning of the actor's mail queue, blocking them, or postponing them to later time. Manipulations on the actor space domain include taking messages sent to one particular actor and duplicating, rerouting, or broadcasting them to other actors to satisfy fault tolerance, security and other QoS requirements.

Coordination actors observe events occurring at the computation actor layer, and perform coordination behaviors accordingly. However, coordination actors are partitioned into roles and coordinators, and these two types of coordination actors also need to collaborate with each other. Their collaborations are achieved through event exchanges. The events that are observable in the ARC model are presented in Table 1.

Note that the events specified in Table 1 do not exactly follow those defined in the traditional actor model [1, 2]. In Agha's actor model, there are only three primitive events, *send*, *new*, and *ready*, where *ready* actually represents two behaviors of an actor: become a new actor and ready for next available message. After processing a message, even if an actor does not change its behavior, it still has to perform *become* to become itself. However, in the ARC model the change of behavior triggers the roles' membership management behaviors. If we perform *become* each time a message is finished processing, we will continuously trigger the member management actions in roles, which in most cases will be unnecessary. For this reason, in the ARC model we separate the *ready* event into two events, namely the *become* event and *ready* for next available message event, where *become* explicitly specifies that an actor changes its behavior and triggers membership management actions.

After a message has been sent out to a recipient computation actor, and before it can be processed, a $send(msg)$ event is broadcast and needs to be handled by the coordination actors. The argument msg is the message that has been sent. After processing the current message, the actor will enter a state in which it is ready to process the next available message in the mail queue. This will cause a $ready(msg)$ event to be broadcast to trigger coordination behaviors. Events are instantaneous; coordination actors

Location	Event	Triggered By
Actor	$send(msg)$	A computation actor performs a *send(msg)* operation.
	$new(beh)$	A computation actor performs a *create(beh)* operation.
	$become(beh)$	A computation actor performs a $become(beh)$ operation, where beh represents a behavior that is different from the actor's current behavior.
	$ready(msg)$	A new message in the actor's mailbox is dispatched at the actor.
Role	$propSend(msg)$	A $send(msg)$ event from a computation actor is observed.
	$propReady(msg)$	A $ready(msg)$ event from a computation actor is observed.
Coordinator	$tell(inter-roleconstraints)$	A $propSend()$ or $propReady()$ event is observed.

Table 1: Events Observable in the ARC Model

observe and handle events atomically. Message deliveries, on the other hand, always take time. The dispatch of a message will always happen at a later time than when the message was sent. Therefore, it is guaranteed that coordination actors can perform their coordination behaviors on messages in the recipient actors' mailboxes before those messages are processed.

The $new(beh)$ or $become(beh)$ events are triggered when a new actor is created or when an actor changes its behavior. The argument beh is the behavior of the new actor to be created, or the new behavior an actor obtains. All roles in the system are able to observe such an event and compare the behavior with their membership criteria. The role whose membership criteria matches the computation actor's behavior adds the computation actor into its group. For completeness of the roles in our system, we also introduce a *default* role. If the actor's behavior does not match any membership criteria of all the existing roles, the actor is added to the *default* role.

Upon observing the $send$ or $ready$ event from a computation actor belonging to a role group, the role propagates these events to coordinators to inquire about corresponding inter-role constraints. Unlike the original messages sent from actors, the message parameters in these events may contain extra information, such as the names of the sender and receiver actors, and their currently attached roles. This information helps the coordinator to determine what constraints need to be propagated to which role.

After observing the $propSend$ or $propReady$ event propagated from the roles, a coordinator checks its constraint store and locates the corresponding constraints, which may depend on both the message parameters and the coordinator's own states. The coordinator then enacts these constraints by sending a $tell$ event to the roles.

The formal operational semantics of the ARC model is given in [44].

4 Framework

In this section, we briefly describe several critical design issues of the framework, and then present the design in detail, along with a prototype implementation of the ARC model. Finally, we show the results of experiments demonstrating the scalability and performance overhead of the framework.

4.1 Design issues

The main design and implementation concern of the ARC framework is to provide the abstractions that implement the Actor, Role and Coordinator semantics, and at the same time provide good performance, scalability and flexibility for different applications. Based on this goal, there are several design issues we need to consider:

Implement coordination actors and events.

According to the definition of the ARC model, roles and coordinators are "coordination actors" communicating through event broadcasts. Therefore, we need to explicitly distinguish events and messages in the implementation.

As defined in [1, 2], computation actors are autonomous and active entities that communicate with each other through asynchronous messages, as are coordination actors. However, unlike computation messages that communicate among actors in a point-to-point fashion, events are broadcast to all coordination actors. Furthermore, events have a higher priority than computation messages. This ensures that messages that need to be coordinated will be manipulated by coordination actors before they are dispatched on computation actors. In both our model and implementation framework, the generation and consumption of events are treated as atomic behaviors and are enforced by using synchronization protocols.

Maintain scalability and performance as the number of entities increases.

One of the characteristics of ODE systems is that they usually have large numbers of computational entities. The introduction of active roles into the ARC model helps mit-

igate the scalability issues in coordination management by allowing coordinators to only coordinate roles, while roles only coordinator actors that share the same behaviors.

Because coordination in the ARC model is enforced transparently on the underlying actors, two problems may occur when the number of actors increases. First, every coordinated message triggers at least one event that must be handled by remote coordination actors. This may bring additional communication overhead. Second, roles and coordinators become potential bottlenecks, which may degrade performance and make systems hard to scale.

To alleviate these problems, we have developed a decentralized architecture to further distribute coordination behaviors and states to local physical nodes, thus avoiding bottlenecks and communication overhead. Because both roles and coordinators are active and stateful entities, multiple update and query operations may concurrently be applied to the states of those distributed replicas. Therefore, a synchronization protocol must be in place to ensure the consistency of the states among different nodes. If such synchronizations occur very frequently, the overhead of achieving synchronizations may exceed the benefit of distributing roles and coordinators to local platforms. Hence, tradeoffs need to be made to balance the communication and synchronization costs. Whether distributing the coordinator/role states will have performance gains is application dependent.

Avoid re-inventing the wheel to solve common problems.

Instead of developing our framework from scratch, we take advantage of existing technologies and tools to support distributed communication, i.e., distributed naming, synchronous and asynchronous communication, and locking schemes.

In the next section, we give the details of our framework's design, taking into account the above issues and providing our solutions to them.

4.2 An ARC framework

Figure 5 gives an overview use case diagram depicting the functional requirement from three categories of users in the system, i.e. the actors, roles, and coordinators. From this figure, the part within the dashed line box represent the functionality of traditional Actor system. By importing the concepts of role and coordinator, the use cases in ARC system become richer. The purpose of the framework is therefore to fulfill the functional requirements indicated in this use case diagram, while taking into account the design issues presented in Section 4.1.

The ARC framework is built on top of TAO (v1.4.1) [45], an implementation of the CORBA 3.x specification. To minimize the overhead and footprint of the ARC framework, we only use a small subset of services provided by TAO. Actors in the ARC framework are built as CORBA objects. They register themselves and locate other actors through the CORBA naming service, and communicate with each other through the TAO asynchronous message service. Figure 6 outlines the architecture of the framework. The Role Representative and Coordinator Representative objects localize the functionalities of coordination-actors to further increase scalability of the system. These concepts will be discussed in detail in a later in this section.

4.2.1 Actor platform and message manager

In the framework, an *Actor Platform* is installed on every physical node. It provides a uniform way to create actors and register actors as CORBA services. An Actor Platform is implemented as a "system actor" that creates actors, roles, and coordinators, initializes their states and behaviors, sends messages, and generates events.

With each actor creation, the Actor Platform also creates a Message Manager object for each actor (including both computation and coordination actors) to handle actor communication tasks. When an actor tries to send a message to another actor, it delegates the message to its Message Manager. For the sending actor, the Message Manager acts as a CORBA client object to send the message asynchronously to the destination actor's message manager, which acts as a CORBA server object. The receiving message manager then forwards the message to the receiving actor for processing. Thus, the CORBA middleware details are encapsulated in the implementation of the message manager and are transparent to application developers who use actors.

4.2.2 Modes

In our framework, users have the option to have logically remote coordinators and roles physically distributed to local Actor Platforms to reduce the communication overhead. Therefore, we provide three modes:

Fully Centralized Mode (FCM) In this mode, every coordination message has to go through potentially remote roles and remote coordinators. This mode is suitable for applications that require very frequent state updates in both coordinators and roles.

Partially Distributed Mode (PDM) The coordinator is distributed to the nodes where the coordinated roles are located, but roles are not distributed to the actor platforms. Therefore coordination requests from local nodes have to go through possibly remote roles, but these roles use local coordinator representatives instead of remote coordinators. Applications that do not anticipate frequent state updates in coordinators will benefit by using this mode.

Fully Distributed Mode (FDM) Both coordinator and roles are distributed to every related node. This mode brings best performance for applications with less frequent synchronization needs.

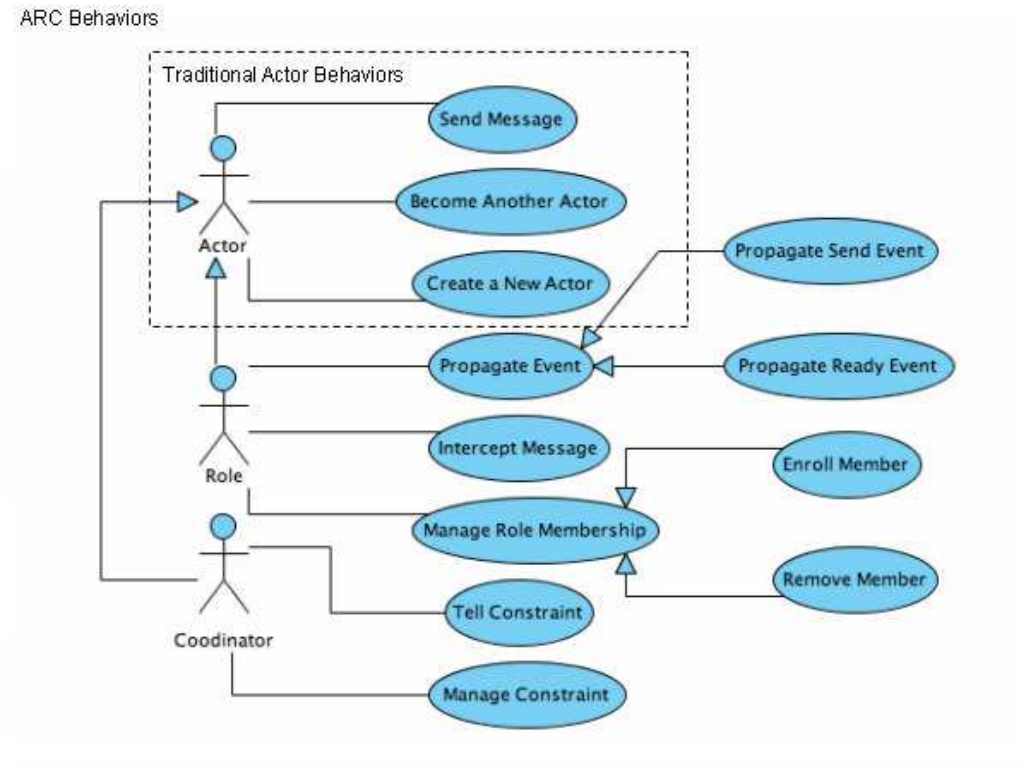

Figure 5: ARC Use Case Diagram

In the framework, we define two supporting entities: Coordinator Representative and Role Representative. As their names suggest, they represent coordinators and roles and perform coordination behaviors in local Actor Platforms. To facilitate deploying different modes, these representatives are implemented as coordination-actors. According to the definitions of coordination actors, they are able to communicate with each other through event communications. Based on the currently applied mode, different Coordinator Representative and Role Representative instances are bound to these interfaces during runtime and have different responsibilities. The relationship among Message Manager, Role, Coordinator, representative interfaces and their instances is depicted in Figure 7.

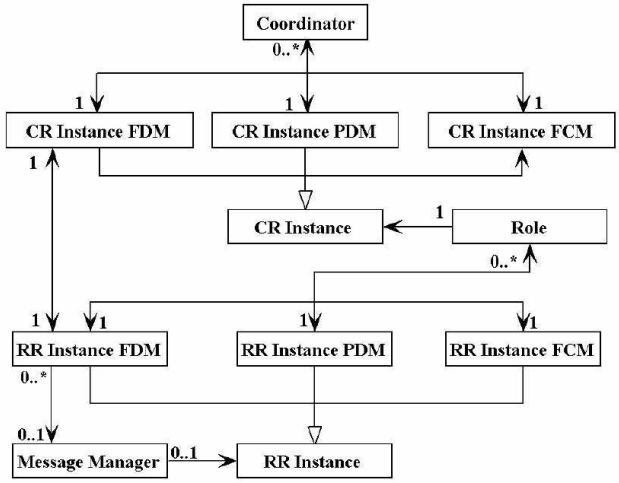

Figure 7: Multi-mode Class Diagram

4.2.3 State synchronization

In situations when synchronization is required among representatives, we apply the primary-backup and two-phase locking (2PL) protocol. The coordinators and roles are responsible for synchronizing the updates with their representatives distributed among other actor platforms. In the primary-backup protocol, these coordinators or roles act as primary objects and the representatives are backups. The Concurrency Service provided by TAO enables the primary objects to obtain and release locks in the 2PL algorithm.

4.2.4 Fully distributed mode implementation

In this paper we focus on the implementation of the Fully Distributed Mode (FDM). The implementations of the Partially Distributed Mode and Fully Centralized Mode are very similar and can be easily inferred from the current introduction.

With FDM, the local Actor Platform creates a Role Representative coordination actor for every existing role to ful-

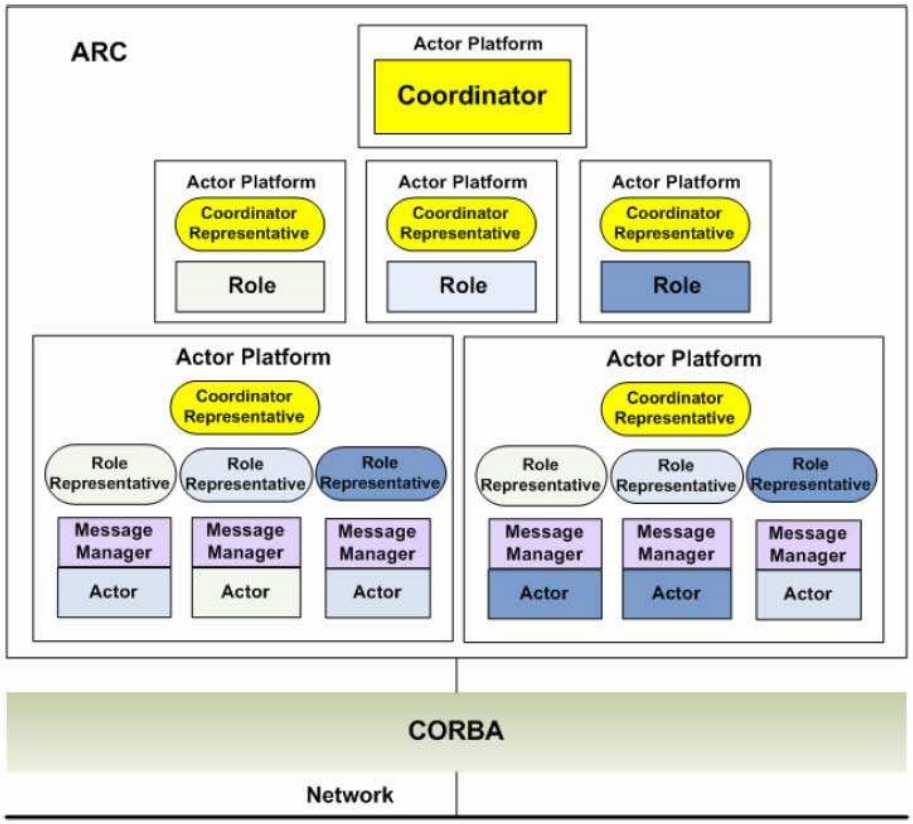

Figure 6: The Architecture of the ARC Framework

fill both its membership management behavior and coordination behavior. In the ARC model, it is the roles, but not the actors, that manage group membership. Whenever a new actor is created or an actor changes its behavior, the roles apply their *bind* and *unbind* operations to maintain the consistency of the membership. Figure 8 demonstrates the procedure of a Role Representative performing membership management and implementing the binding mechanism.

In the ARC model, coordination constraints are transparently applied to actors. This is achieved by (1) buffering the messages in receiver actors' mailboxes via Message Managers, (2) obtaining coordination constraints by forwarding events to the corresponding role representatives and coordinator representative for constraint checks, and (3) applying the coordination constraints by manipulating the messages in the mailboxes. The communication between two actors is shown in Figure 9.

If a constraint is found in its local store, the Role Representative requires the corresponding Message Manager to enact the constraint on the actor. As all these operations are performed locally and no remote communication is required, the constraint propagations do not introduce much performance overhead.

4.3 Evaluation

We have developed a prototype of the ARC framework. The experimental settings are as following: We have two Intel x86 machines. The first machine is a Pentium IV 1.7 GHz with 512MB RAM and the second is a Pentium IV 3.06GHz with 1GB RAM. Both of them are running Windows XP and connect with each other through a 100M ethernet switch. In our experiments, we developed a simple Ping-Pong application, that asks two actors, the Ping actor and the Pong actor, in different machines to continuously send and reply to a specific number of messages to each other.

Figure 10 shows the performance comparisons between the Actor Architecture (AA) framework [22] and the ARC framework. AA is an actor-based framework developed by Agha's group at UIUC. AA is implemented in Java and provides its own ad-hoc solutions to core distributed applica-

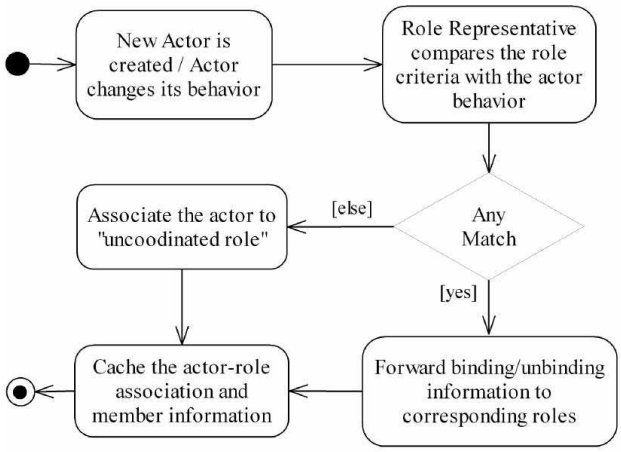

Figure 8: Actor-Role Binding Mechanism

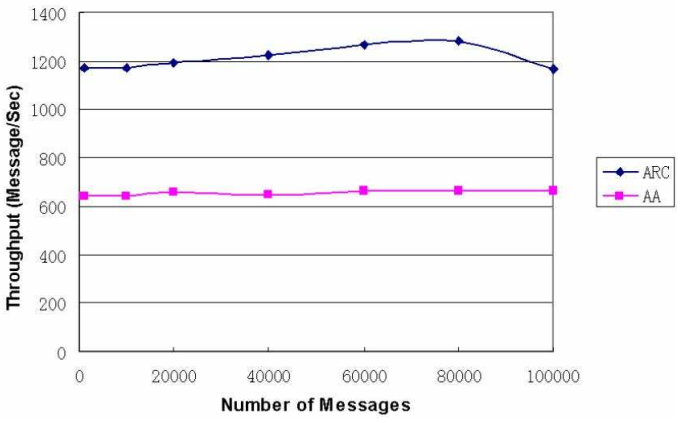

Figure 10: ARC vs. AA on actor communication

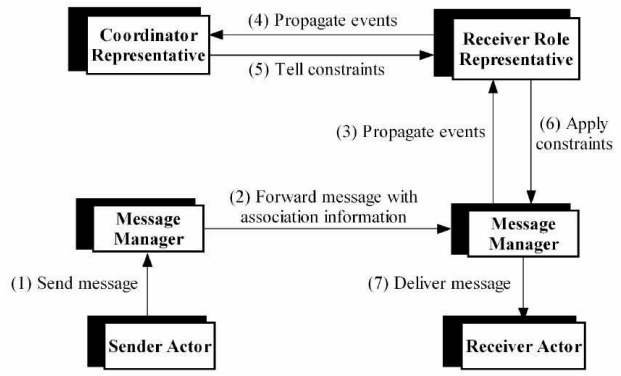

Figure 9: Communication between Coordinated Actors in FDM

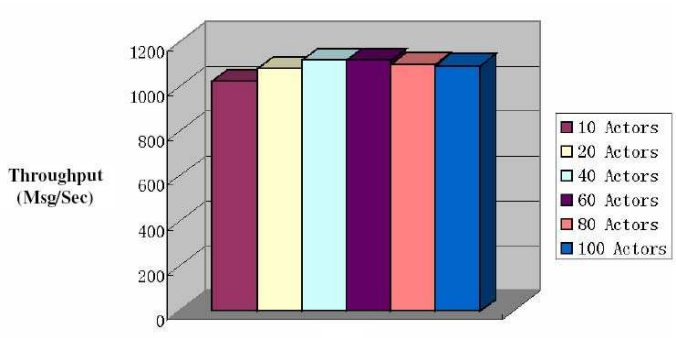

Figure 11: Performance of ARC when the number of actors increases

tion features, such as the Naming Service. The ARC framework is implemented in C++ and utilizes CORBA services.

In this test we use AA and ARC to send messages (with a size of 100 bytes) between two actors on different machines. From figure 13, we can see that ARC outperforms AA in actor communications. The average throughput by using ARC is 85% higher than using AA. This is mainly because the Java Virtual Machine brings heavy overhead to AA. In addition, the optimized naming and communication services provided by TAO also improve the ARC framework's communication performance.

Figure 11 demonstrates the situation when multiple actor pairs run and send messages concurrently. We ran up to 100 actors on each machine. These pairs of actors sent messages and replied to them concurrently. As the figure shows, increasing the number of actors had little impact on the performance of the ARC. However, the figure also shows an 'unintuitive' result: the performance of the 40 actor case is better than the 20 and 10 actor cases. This happened because the larger number of actors increases the odds that messages will share transportation connections. This reduced the overhead of opening and closing connections. Once the number of actors is greater than 40, the connections are saturated and the performance becomes stable.

In the previous experiments, we performed actor communications without considering coordination constraints. When coordination requirements are taken into account, actors need to collaborate with each other to achieve system requirements. For example, in the Ping-Pong application, we could have an extra mutual exclusion requirement that at any time only one pair of Ping and Pong actors can send messages to each other. To satisfy this requirement, the Ping and Pong actors that want to send messages have to communicate with each other to make sure that there are no other actors competing for the permission to send a message. If there is more than one actor seeking permission, a decision needs to be made about which one gets permission first. This will require communications to be sent, and some kind of election protocol to be followed.

If there are n Ping actors competing for permission to send a message, then there will be at least $2n(n-1)$ [48] communication messages to achieve synchronization before a message can be sent out. This is a typical synchronization problem for networking and distributed environ-

ments. An obvious solution is to use an explicit coordinator to synchronize the "sending message" requirements among actors. To achieve the same synchronization with an explicit coordinator requires most $3n$ [48] extra communication messages. Thus, in an ODE system or similar environment where the number of actors is large and coordination among them is frequent, an explicit coordinator can drastically reduce communication overhead and improve scalability. Figure 12 depicts the difference between the solution using a coordinator, which is represented by a star topology, and the one without an explicit coordinator, which is represented by a mesh topology.

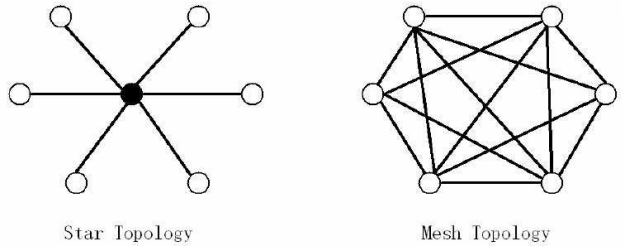

Figure 12: With or Without Coordinator - a Topology View

From the above analysis, it is clear that adding an extra coordinator layer actually increases performance when the number of actors is large and coordination among them is unavoidable. The next question to ask is if adding an extra layer (the role layer) will seriously degrade the performance of the system with a single coordinator layer. We test this by introducing role coordination entities to achieve *intra-role* coordination constraints. The current test case is under FDM and follows the procedure demonstrated in Figure 9.

In this experiment, we arranged for $10,000$ messages to be sent between two actors on different machines. We also provided both *inter-role* and *intra-role* constraints. After introducing two roles, the PingRole and the PongRole, we divided the constraints into three categories: 20% became inter-role constraints stored in a coordinator, 40% became *intra-role* constraints stored in PingRole, and the remaining 40% were *intra-role* constraints stored in PongRole. Constraint checks were simulated using simple string comparisons. Figure 13 gives the measurements.

As shown in Figure 13, when there are 100 constraints in a single coordinator, the overhead of introducing two extra role entities is about 3.5%; when there are 500 constraints, the overhead is about 2.7%. The main overhead comes from the two extra communications between the sender and receiver actors and their attached roles. This number is fixed no matter how many constraints need to be checked. The total number of constraints is the same in two situations. When there are no roles, a coordinator has to check all these constraints; in contrast, when there are two extra roles, the coordinator only handles 20% of the constraints, and rest of the constraints are handled by the two roles concurrently. As a result, the overhead actually

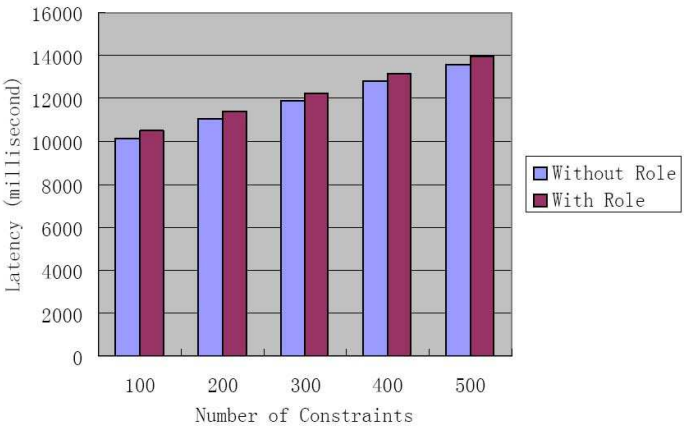

Figure 13: Overhead of introducing role layer

decreases when the number of constraints increases.

Though these tests have not conducted on FCM and PDM, we can expect by looking at their descriptions that because roles and actors live in different nodes, the communication overhead will be larger than in the current FDM test. However, in such modes, synchronization overhead will become the focus of the application and trade-offs are made to satisfy that.

Finally, we look at the modularity brought by separating the coordination layers from the underlying computational logic in the ARC framework. We demonstrate this by introducing an extra requirement for the Ping-Pong application: after a Pong actor receives a message in its mailbox from the Ping actor, it has to wait for a specific period of time $t1$ before it sends back a response. The following gives the pseudo-code that enforces this timing constraint in the Pong actor:

```
HandleMsg(String message, Int waitTime){
  Message msg = parse(message);
  String responseMsg = getResponse();
  if (msg.SenderTypeName == "Ping") {
     wait(waitTime);}
  send(Ping, responseMsg);
}
```

If we have multiple Pong actors in the system, then we will have to add this code to every Pong actor to maintain the timing constraint. Furthermore, if in the future we want to modify the time period, for example from $t1$ to $t2$, we have to update the codes for all the Pong actors, change the constraints and re-compile them. But if we have an explicit coordination actor, we can use it to specify these timing constraints.

Computational logic is separated from coordination constraints, and can be developed independently. Below shows the code with the ARC model. The `HandleMsg` is the code in a Pong actor to implement the "response" logic, and the `HandleEvt` is the code in a coordinator to specify

the timing constraint. The timing constraint is further enforced by a role that reroutes a message in the Pong actor's mailbox to a sink for a period of time `waitTime` before dispatching it for processing. This coordination operation is transparent to the actor computation, and we can modify such constraints without affecting the underlying computational logic.

```
HandleMsg(String message){
  Message msg = parse(message);
  String responseMsg = getResponse();
  send(Ping, responseMsg);
}

HandleEvt(String event, Int waitTime){
  Event evt = parse(event);
  if (evt.eventTypeName == "PropSendMsg")
   if (evt.senderRoleTypeName ==
                              "PingRole")
      tell(PongRole,
          "reroute, sink, waitTime");
}
```

5 Conclusion

In this paper, we presented a framework based on the ARC model to support the development of ODE applications. The ARC model is a role-based and decentralized coordination model. Under this model, a system's QoS requirements are treated as coordination concerns and are separated from concurrent computation logic. The coordination constraints are imposed on computations through message manipulations that are transparent to the computation itself. In addition, to address the dynamicity and the openness inherent in an ODE system, we introduced active roles that not only provide abstractions for actor functional behaviors, but also take part in the coordination activities. Hence, the coordination subsystem itself becomes distributed and thus inherits all the benefits a distributed system may offer.

The framework provides an interface to allow users to create actors, roles and coordinators. Based on detailed application requirements, the framework distributes the coordinators and roles and collocates them with local actors so that both performance and scalability can be improved. In addition, the framework also provides efficient mechanisms to support automatic and runtime role group management, and message management. Our prototyping and empirical experiments have shown that we are able to achieve role-based coordination with limited performance overhead. The experiments also indicate that the framework scales well when the number of entities involved in the system increases.

Our future work is to apply the ARC model and its realization to help mitigate the difficulties in developing practical QoS aware applications in ODE systems. Such systems may have multiple dimensions of QoS requirements such as real-time, fault tolerance, energy consumption, and security constraints, etc. To be more specific, we want to extend our framework to combine resource management, real-time features and fault tolerance mechanisms, so that multiple non-orthogonal QoS requirements can be studied and supported based on a uniform coordination model. To achieve this, we plan to use classic ODE applications, such as a simulation of a simplified Air Traffic Control (ATC) system, as cases studies to demonstrate and evaluate the advantages of the model and the framework.

Acknowledgement

This work is supported by NSF under grant CNS 0746643.

References

[1] G. Agha. *Actors: A model of concurrent computation in distributed systems*. MIT Press, 1986.

[2] G. Agha, I. A. Mason, S. F. Smith, and C. L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

[3] M. Aksit, B. Tekinerdogan, and L. Bergmans. The six concerns for separation of concerns. In *Workshop on Advanced Separation of concerns*, 2001.

[4] F. Arbab. IWIM: A communication model for cooperative systems. In *The 2nd International Conference on the Design of Cooperative Systems*, pages 567–585, 1996.

[5] F. Arbab. A foundation model for components and their composition. Technical report, CWI, Amsterdam, Netherlands, 2004.

[6] F. Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[7] Barbaran, Diaz, Esteve, Garrido, Llopis, and Rubio. A real-time component-oriented middleware for wireless sensor and actor networks. *cisis*, 00:3–10, 2007.

[8] A. Berry and S. Kaplan. Open, distributed coordination with finesse. In *The 1998 ACM Symposium on Applied Computing*, pages 178–184, 1998.

[9] G. Cabri, L. Ferrari, and L. Leonardi. Brain: a framework for flexible role-based interactions in multiagent systems. 2888:145–161, 2003.

[10] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444ÍC458, 1989.

[11] A. Colman and J. Han. Coordination systems in role-based software. 3454:63–78, 2005.

[12] Riccardo Crepaldi, Albert Harris III, Rob Kooper, Robin Kravets, Gaia Maselli, Chiara Petrioli, and Michele Zorzi. Managing heterogeneous sensors and actuators in ubiquitous computing environments. In *First ACM Workshop on Sensor Actor Networks*, 2007.

[13] J. C. Cruz. Opencolas: A coordination framework for colas dialects. 2315:231–247, 2002.

[14] D.F. Ferraiolo and D.R. Kuhn. Role based access control. In *The 15th National Computer Security Conference*, 1992.

[15] M. Fowler. Dealing with roles. In *European Conference on Pattern Language of Programs*, 1997.

[16] S. Frølund. *Coordinating Distributed Objects: An Actor-Based Approach to Synchronization*. MIT Press, 1996.

[17] E. Gamma, R. Helm, R. Johnson, and J.Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[18] W. Harsch and C. V. Lopes. Separation of concerns. Technical report, Northeastern University technical report NU-CCS-95-03, Boston, 1995.

[19] Carl Hewitt. large-scale organizational computing requires unstratified paraconsistency and reflection. In *International Conference on Autonomous Agents and Multiagent Systems*, 2007.

[20] Carl Hewitt. What is commitment? physical, organizational, and social. *LNAI*, 4386, 2007.

[21] C.A.R. Hoare. *Communicating Sequential Processes*. Computer Science. Prentice Hall International, 1985.

[22] M. W. JANG. The actor architecture manual, 2004.

[23] Woochul Kang and Sang H. Son. The design of an open data service architecture for cyber-physical systems. *AC SIGBED Review*, 5(1), 2008.

[24] Rajesh K Karmani and Gul Agha. Debugging wireless sensor networks using mobile actors. In *RTAS Poster Session*, 2008.

[25] E. A. Kendall. Role modeling for agent system analysis, design and implementation. *IEEE Concurrency*, 8(2):34–41, 2000.

[26] Ozcan Koc, Chaiporn Jaikaeo, and Chien-Chung Shen. Navigating actors in mobile sensor actor networks. In *First ACM Workshop on Sensor Actor Networks*, 2007.

[27] B. Kristensen and K. Osterbye. Roles: conceptual abstraction theory and practical language issues. *Theory and Practice of Object System*, 3(2):143–160, 1996.

[28] YoungMin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: An actor platform for wireless sensor networks. In *Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS), pages 1297-1300*, 2006.

[29] Youngmin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: An actor platform for wireless sensor networks. In *In Proc. of the 5th Intl. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*. AAMAS, 2006.

[30] E. A. LEE. What's ahead for embedded software. *IEEE Computer*, 33(9):18–26, 2000.

[31] J. LIU, J. EKER, J. W. JANNECK, X. LIU, and E. A. LEE. Actor-oriented control system design: A responsible framework perspective. *IEEE Transactions on Control System Technology*, 12(2):250–262, 2004.

[32] José Meseguer and Carolyn Talcot. Semantic models for distributed object reflection. In *European Conference on Object-Oriented Programming, ECOOP'2002, LNCS 2374*, pages 1–36, 2002.

[33] R. Milner. *Communicating and Mobile Systems: the Pi-Calculus.* Cambridge University Press, 1999.

[34] N. H. Minsky and V. Ungureanu. Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. *ACM Transactions on Software Engineering Methodology*, 9(3):273–305, 2000.

[35] Jayadev Misra and William R. Cook. Computation orchestration: A basis for wide-area computing. *Journal of Software and Systems Modeling*, May 2006.

[36] Rocco De Nicola, Gianluigi Ferrari, and Rosario Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24:315–330, 1998.

[37] R. Nicola de, J.P. Katoen, D. Latella, and M. Massink. Towards a logic for performance and mobility. 2005.

[38] A. Omicini and E. Denti. Formal respect. *Electronic Notes in Theoretical Computer Science*, 48:179–196, 2001.

[39] A. Omicini, A. Ricci, and M. Viroli. Agent coordination contexts for the formal specification and enactment of coordination and security policies. *Science of Computer Programming*, 63(1):88–107, 2006.

[40] A. Omicini and F. Zambonelli. Tuple centres for the coordination of internet agents. In *The ACM Symposium on Applied Computing*, pages 183–190, 1999.

[41] G. A. Papadopoulos and F. Arbab. Coordination models and languages. *Advances in Computers*, 46:330–401, 1998.

[42] Gian Pietro Picco, Amy L. Murphy, and Gruia catalin Roman. Lime: Linda meets mobility. In *21st Internation Conference on Software Engineering*, pages 368–377. ACM Press, 1999.

[43] José Proença and Dave Clarke. Coordination models orc and reo compared. *Electronic Notes in Theoretical Computer Science*, 194(4):57–76, 2008.

[44] S. Ren, N. Chen, Y. Yu, P.-E. Poirot, L. Shen, and K. Marth. Actors, roles and coordinators a coordination model for open distributed embedded systems. 4038:247–265, 2006.

[45] D. C. Schmidt. The design of the tao real-time object request broker. In *Computer Communications*, 1998.

[46] Carolyn Talcott, Marjan Sirjani, and Shangping Ren. Ccoordinating asynchronous and open distributed systems under semiring-based timing constraints. *Electronic Notes in Theoretical Computer Science*, 2008.

[47] Carolyn L. Talcott. Coordination models based on a formal model of distributed object reflection. *Electronic Notes in Theoretical Computer Science*, 150:143–157, 2006.

[48] A. S. Tanenbaum and M. V. Steen. *Distributed Systems - Principles and Paradigms*. Prentice Hall. Upper Saddle River, New Jersey, 2002.

[49] C. A. Varela and G. A. Agha. Towards a discipline of real-time programming. *Communications of the ACM*, 20(8):577–583, 1977.

[50] N. Venkatasubramanian, G. A. Agha, and C. Talcott. A metaobject framework for qos-based distributed resource management. In *The Third International Symposium on Computing in Object-Oriented Parallel Environments*, 1999.

[51] Yue Yu, Shangping Ren, and Carolyn Talcott. Comparing three coordination models: Reo, arc. *Electronic Notes in Theoretical Computer Science*, (16956), 2008.