The Actor Role Coordinator Implementation Developer's Manual

Nianen Chen , Li Wang
Computer Science Department
Illinois Institute of Technology, Chicago, IL 60616, USA
{nchen3, li}@iit.edu

1. Introduction

The purpose of this document is to describe the analysis, design and implementation of the Actor Role and Coordinator (ARC) framework. The ARC framework is a middleware application to specify actor, role and coordinator execution environments within which actors across distributed systems exist and operate with other actors under the constraints specified and enforced by roles and coordinators. Especially, we call an actor execution environment on a computer node an AA platform. The main features of ARC implementation are actor/role membership management, actor name management, actor, role and coordinator communication management, actor behavior management, role/coordinator constraint management and event management.

In this document, we assume that the readers of this document know the basic concept of the Actor, Role and Coordinator (ARC) model and have C++ and Corba programming knowledge. If you dont have enough background of the ARC model, we recommend you to read [2, 4]. For the design of the structure and services of ARC implementation, we recommend reading the ARC realization paper [3]. This implementation is also based on The Ace Orb (TAO), which is an implementation of Corba 3 specification. The installation, configuration and compilation of TAO and TAO based applications is beyond the scope of this guide, but will be introduced in separate documents.

2. Architecture and Software Design

2.1 Use Cases

Based on the ARC model [3, 4], the ARC framework shall focus on the following main use cases:

2.2 Design Issues

The main design and implementation concern of the ARC framework is to provide the abstractions that implements 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" dealing with constraints and communicating through event. Therefore, we need to explicitly distinguish actors from roles and coordinators, and events from messages in the implementation.

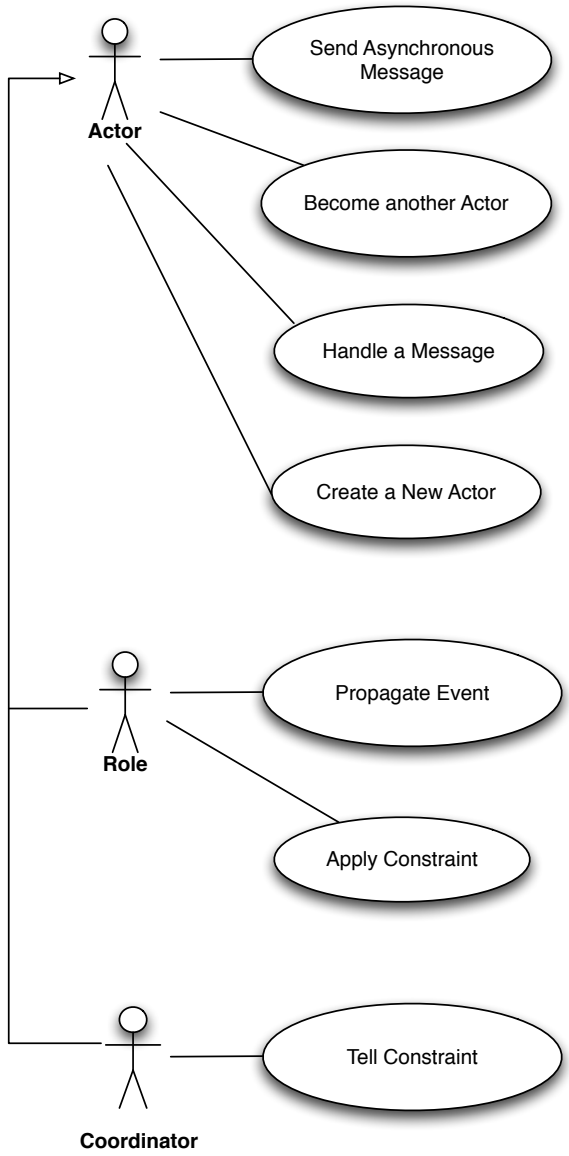

Figure 1. Use Cases

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, events have higher priorities 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 our implementation, we use "CORBA asynchronous message" to achieve the message communication between actors (including roles and coordinators); we then use "CORBA synchronous communication" to achieve event communication between roles and coordinators. When events are handled by roles or coordinators, messages are blocked, which reflects the higher priority of the events.

We describe the relationship among actor, role and coordinator in our implementation. This can be partially observed by the use cases depicted in Figure 2.2. We give more details in the following class diagram:

* It is worth noting that the method "*ask*" actually gives a confusing term. The real action the coordinator performs is "tell", which tells the roles what constraint rules shall apply on the current event. Refactory is required in the future. We just use the name at this moment and keep in mind what it really means.

Maintain scalability and performance as the number of entities increases.

One of the characteristics of a lot of modern systems is that they usually have large numbers of computational entities. The introduction of active roles into the ARC model helps mitigate 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 overheads. Second, roles and coordinators become potential bottlenecks, which may degrade performance and make systems hard to scale.

To alleviate the 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.

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 to allow user selections:

Fully Centralized Mode (FCM) In such a 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.

In our current version of software, we only have FDM implemented. To support the fully distributed architecture in FDM, we define two supporting entities: Coordinator Representative and Role Representative. As their names

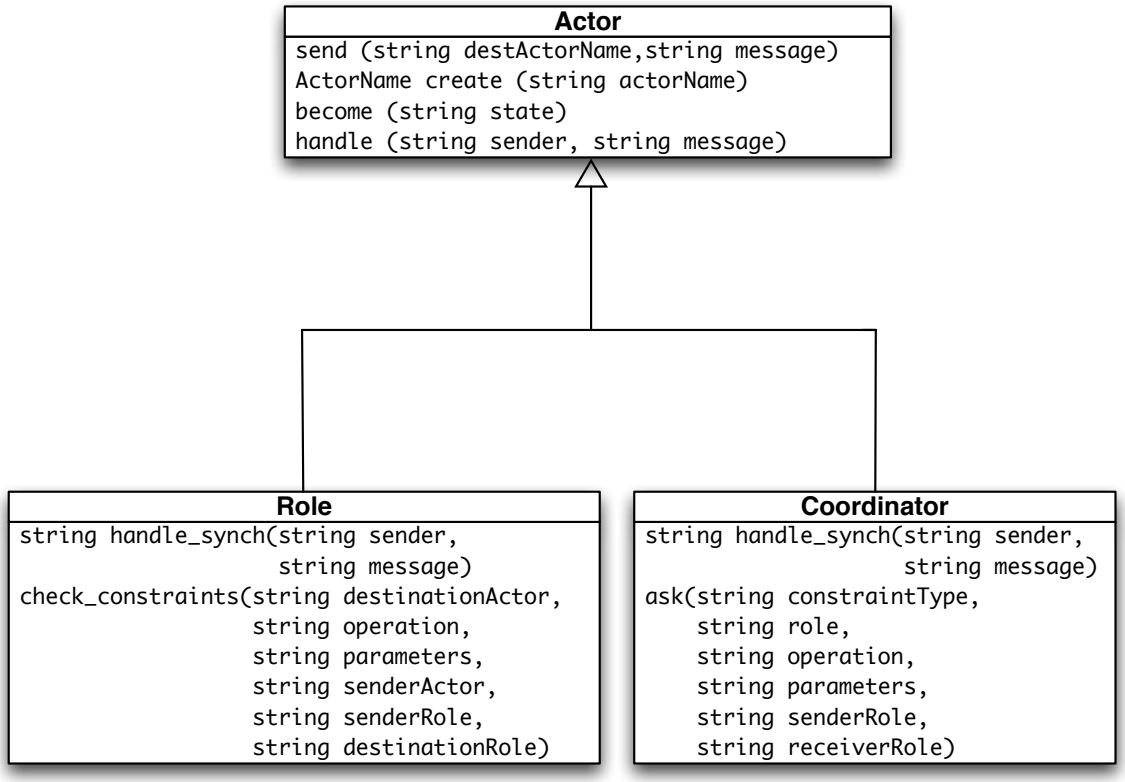

Figure 2. Actor, Role, and Coordinator Class Diagram

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 3.

We will introduce in details in next section about how to use the role representative and coordinator representative to achieve ARC communications.

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.

The ARC framework is built on top of TAO (v1.4.1) [5], an implementation of the CORBA 3.x specification. To minimize the overhead and the 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 4 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.

2.2.1 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.

We first look at the use case that an AAPlatform has to achieve in Figure 5.

In summary, AAPlatform is responsible for creating all the entities that will be used in the current system. We specify what actors, roles and coordinators we want in a text based property file. Examples of such property files will be introduced in next section. Meanwhile, as a "system actor", the AAPlatform can also send messages to other existing actors in the system. Finally, AAPlatform is also a user interface to achieve interactions with system users. In the sequence diagram in Figure 6, we further depict how a AAPlatform is involved in initialization of the ARC system when an it is started up in a physical node.

The Platform is a the class where main function locates, where the only global AAPlatform (implemented as a Singleton) is created. The first thing that the AAPlatform will do after being launched is to read a property file. Based on the content of the property file, the AAPlatform create an ActorFactory instance and use it to instantiates a bunch of actors, roles, coordinators, role representatives, and coordinator representatives. Each actor will be assigned a unique actor name (UAN), which is requested by the AAPlatform and created by a UAManager class. The format of the name is: "uan:" + actorType + ":" + platformName + ":" + actorId, where actorType is one of actor, role, coordinator, role representative, coordinator representative, platformName is specified in the property file, and actorId is generated when the actor is created. This unique UAN will be used to bind to the CORBA Naming Service than.

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

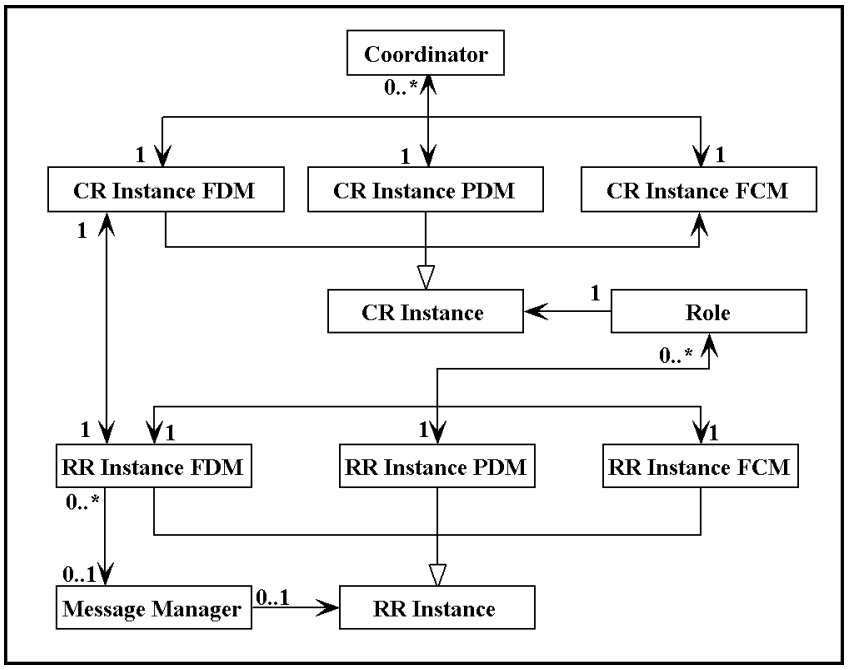

Figure 3. Multi-mode Class Diagram

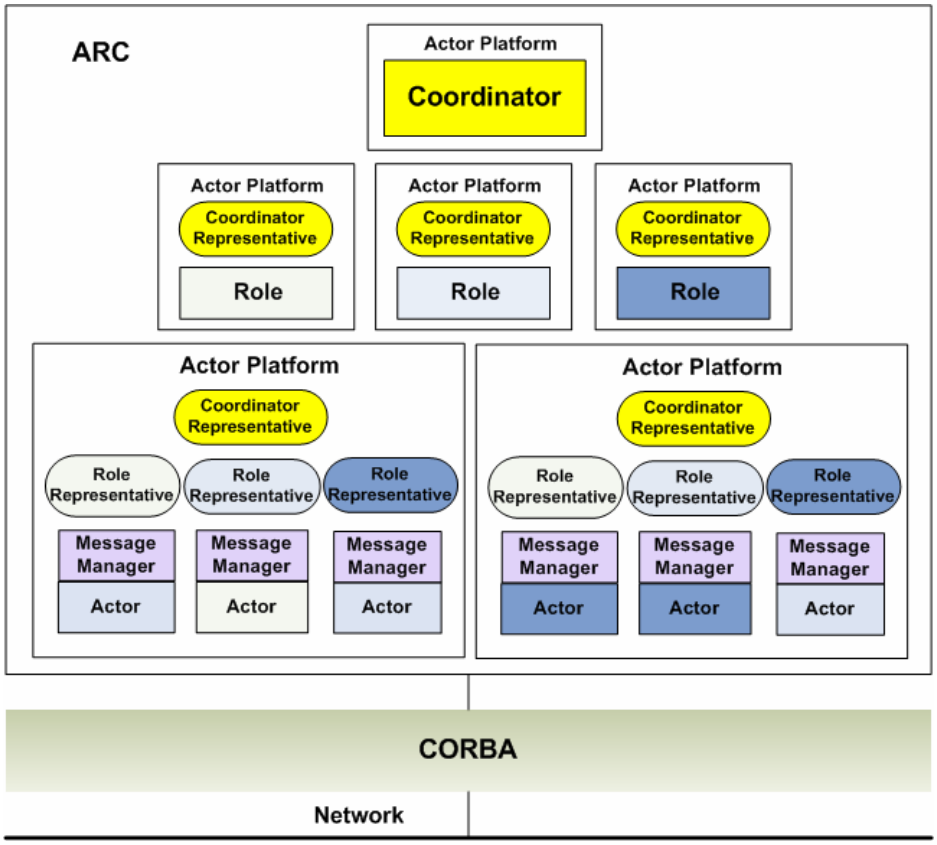

Figure 4. The Architecture of the ARC Framework

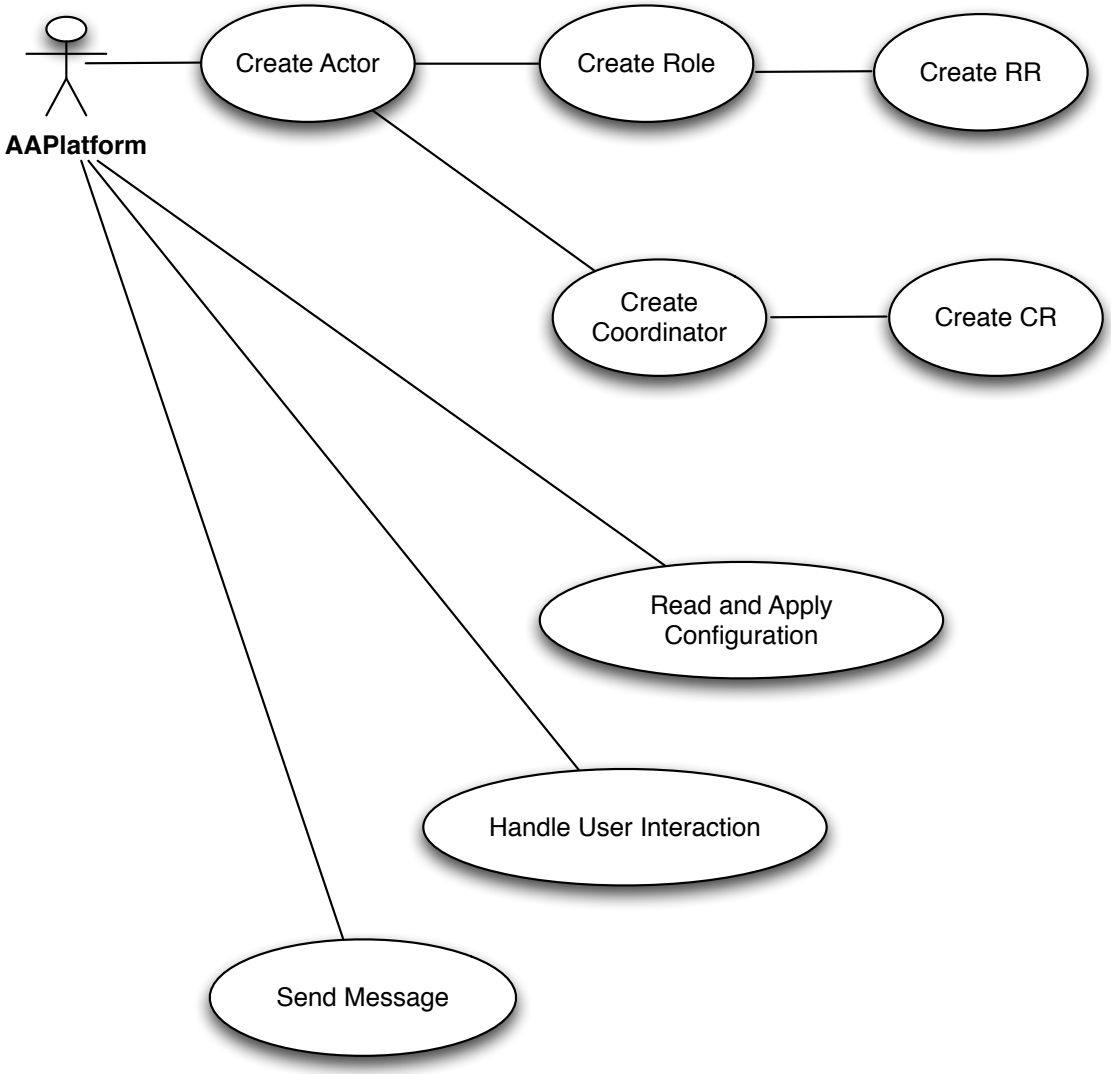

Figure 5. The AAPlatform Use Case

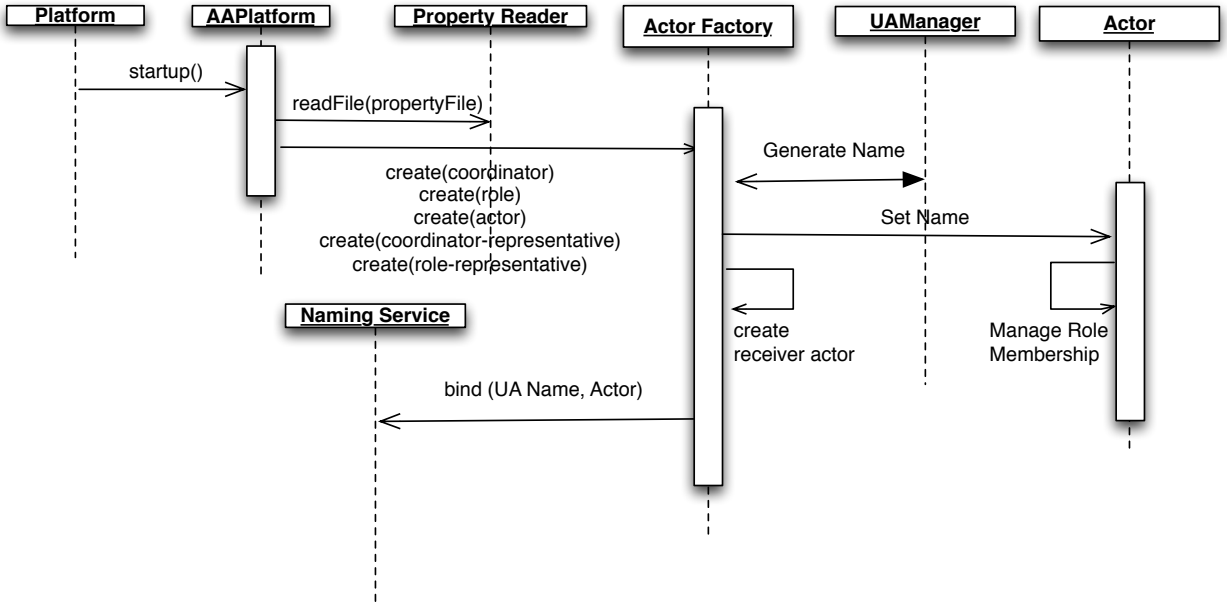

Figure 6. The ARC Startup Sequence

the message manager and are transparent to application developers who use actors. Figure 7 and Figure 8 indicates the work flow to send a message in an ARC framework.

Figure 8, using the Corba Naming service as a boundary, left part is the CORBA server side preparation. Specifically, the AAPlatform create an Actor Factory, which bind an actor's UAN to the Corba naming service, and then create an instance of the CORBA server side servant, i.e. the MessengerI class. The client side message communication will start from the Actor sending a message to the MessageManager. The MessageManager will look up the remote CORBA object location using the unique actor name (UAN) from the CORBA naming service, and then use this network address to send the message to remote actor through CORBA asynchronous communication mechanism. When receiving the message sent from the client stub, the skeleton will call the servant MessengerI, and then the MessengerI will further call the "handle" method of the real actor.

An important defect of the current implementation is that the MessengerI interface simply wait until the Corba internal scheduler to call it such that the Actor behavior can be further called. The actor itself does not have the ability to control which message shall be handled first. We depend on Corba internal scheduling to make the decision. This is fine when we want to FIFO rule on scheduling, which is implemented by Corba. However, if we want to do priority-based scheduling, this mechanism is not sufficient. We hence need to place a queue in the MessengerI, which acts as a message queue for the actor. Messages in the queue can hence be manipulated based on which scheduling algorithm we use. This shall be accomplished in our coming version.

It is worth noting that an actor is both a client and a server. In other words, actor can both send and receive messages. Therefore all the classes in Figure 8 shall be duplicated in all AAPlatforms.

2.2.2 Fully Distributed Mode Implementation

In this document we focus on the implementation of the Fully Distributed Mode (FDM) since this is so far the only mode implemented in our current version.

With FDM, the local Actor Platform creates a Role Representative coordination actor for every existing role to fulfill 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 9 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 10.

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.

3. Getting Started

In this section we use concrete examples to show how to program and run applications on ARC framework. Specifically, in section 3.1, we introduce how to build an actor-based application without involving roles and coordinators. In section 3.2, we describe how to develop an Actor, Role Coordinator application in Fully Distributed Mode.

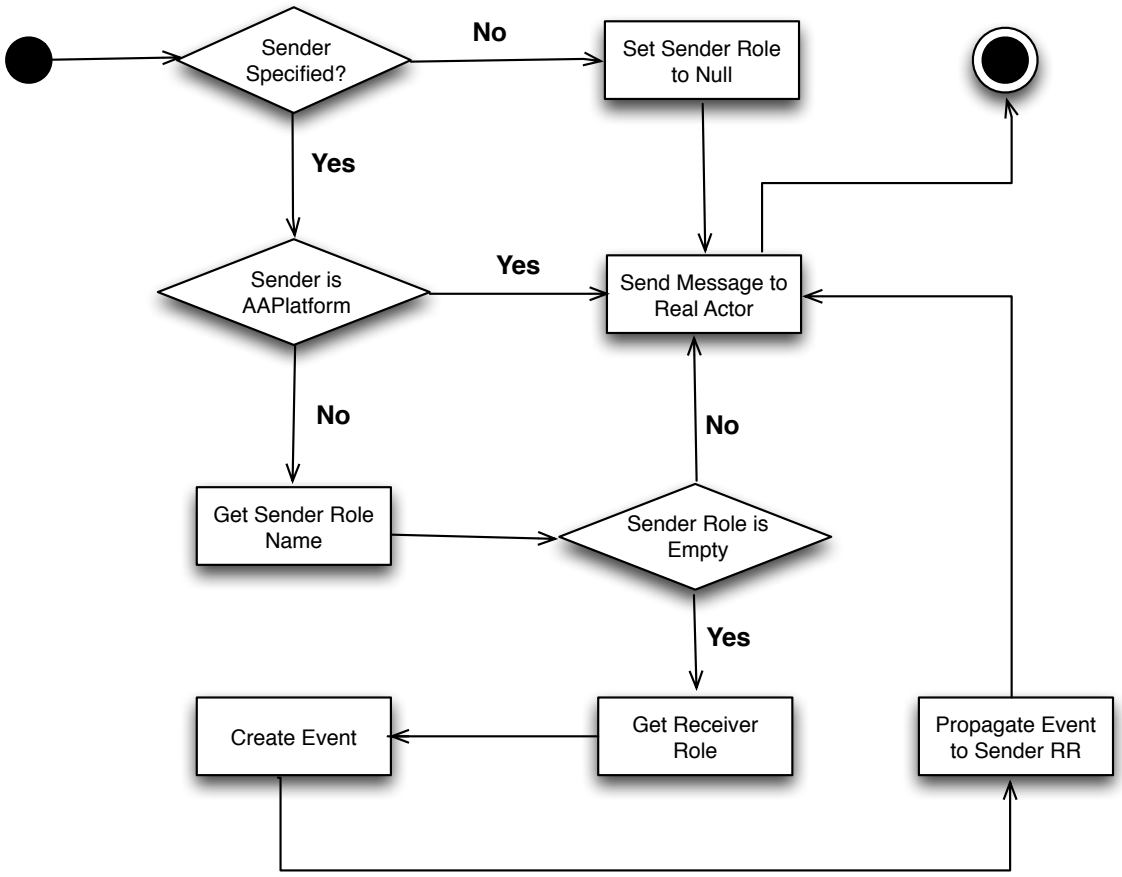

Figure 7. Work Flow for Sending a Message

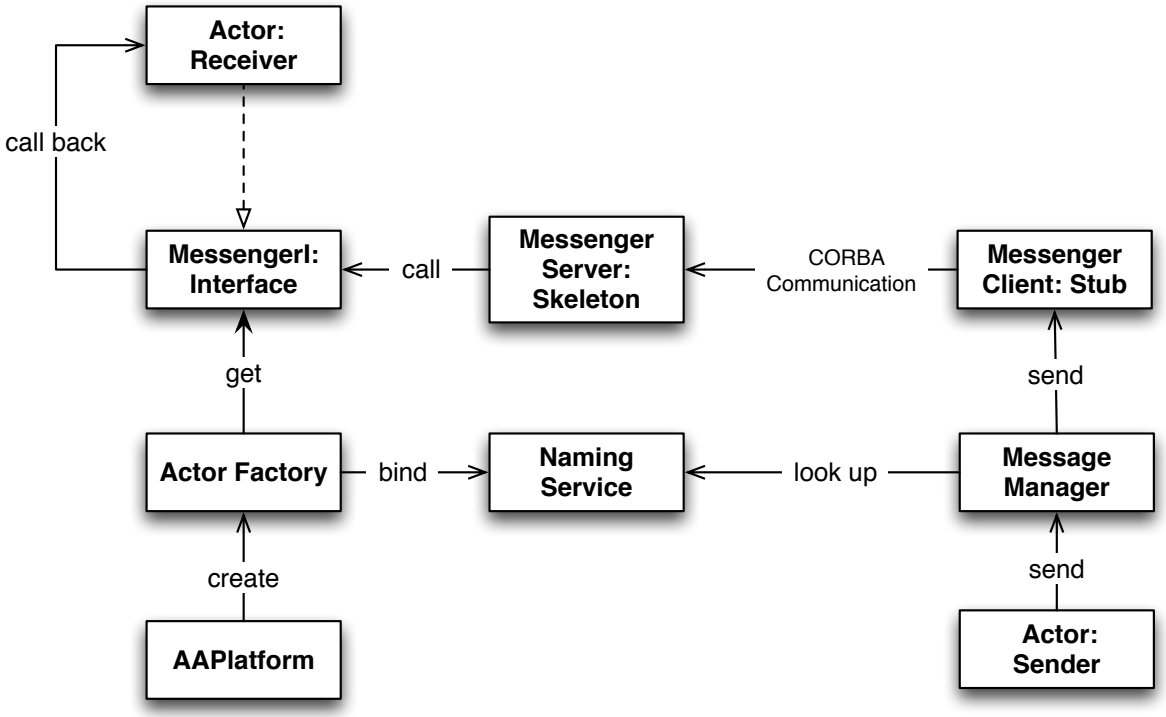

Figure 8. Message Sending Abstract Class Diagram

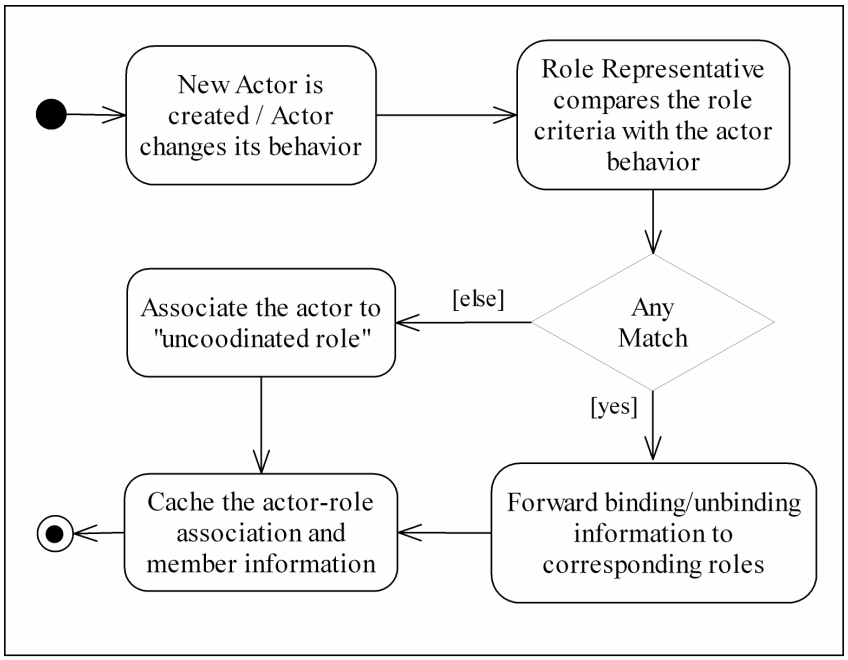

Figure 9. Actor-Role Binding Mechanism

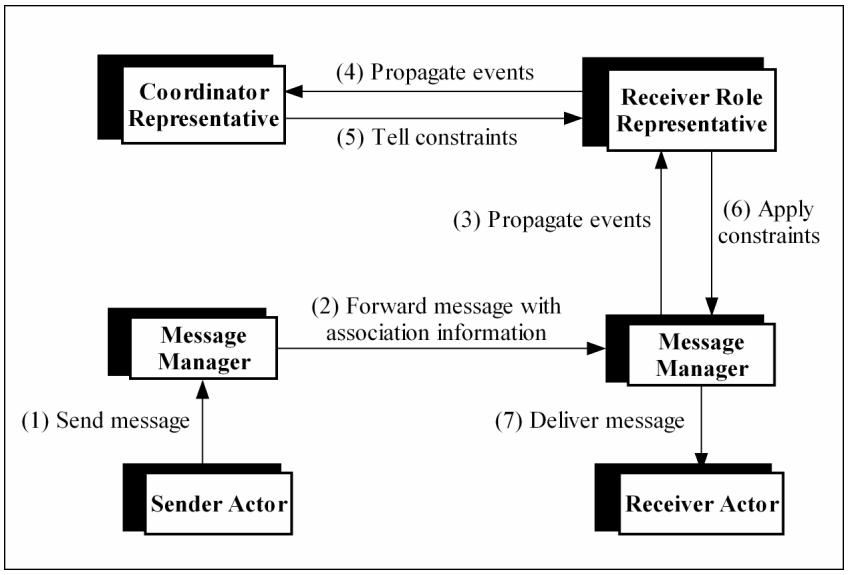

Figure 10. Communication between Coordinated Actors in FDM

3.1 Basic Ping Pong Example

In this section, we developed a simple Ping-Pong application, which asks two actors, the Ping actor and the Pong actor, in different machines to continuously send and reply a specific number of messages to each other. In this basic Ping Pong application, we don't require the actors to be bound to roles. There hence also no constraint check and state synchronization requirements among roles and coordinators.

We give step by step instructions on how to develop, deploy and run the application.

1. Create Ping and Pong Actors.

   The following Figures gives the source code for: PingActor.h, PingActor.cpp, PongActor.h, PongActor.cpp. Both Ping and Pong are inherited from an Actor base class. It is worth noting that in constructors of customized actors, we have to specify what are the behaviors of this actor, which is used during role membership management. The logics of both Ping and Pong actors are quite straightforward and self-explanotory.

```
#ifndef PINGACTOR_H_
#define PINGACTOR_H_

#include "Actor.h"

#define MESSAGE_SIZE 1000

class PingActor: public Actor
{
public:
    PingActor();
    virtual ~PingActor() {};

    void handle(string sender,string message);

private:
    string destActor;

    int max;
    int start;
    int end;

    int counter;
    char huge_msg[MESSAGE_SIZE];
};

#endif
```

Figure 11. PingActor.h

2. Add actors into ActorFactory list.

   The second step is to register these two actors to our system, such that during startup they can be initialized. As can seen from the code, we manually added lines of code to instantiate the PingActor and PongActor classes. This is due to the lack of reflection mechanism in C++.

3. Create property files.

```
PingActor::PingActor()
{
    max = 10;

    for(int i = 0; i < (MESSAGE_SIZE - 1); i++)
        huge_msg[i] = (i % 10) + 48;
    huge_msg[MESSAGE_SIZE - 1] = 0;
    //cout << huge_msg << endl;

    behavior.operations.push_back("start");
    behavior.operations.push_back("make");
    behavior.operations.push_back("default");
}

void PingActor::handle(string sender,string message)
{
    string operation = message;

    if(operation.length() < 50)
    {

        if(operation == "start")
        {
            start = GetTickCount();
            counter = 0;
            send(destActor,huge_msg);
            cout << "Ping sent 0" << endl;
        }
        else if (operation == "make")
        {
            destActor = create("PongActor").uan;
            cout << "Using sender: " << destActor << endl;
        }
        else if (operation == "default")
        {
            destActor = "uan:PongActor:localhost:";

            char buffer[20];
            itoa(name.id,buffer,10);
            destActor += buffer;
            cout << "Using sender: " << destActor << endl;
        }
        else if (operation == "cc")
        {
            destActor = "uan:PongActor:stuff:";

            char buffer[20];
            itoa(name.id,buffer,10);
            destActor += buffer;
            cout << "Using sender: " << destActor << endl;
        }
    }
    else
    {
        counter++;
        cout << "Ping received " << counter << endl;

        counter++;

        if(counter < max)
        {
            send(destActor,huge_msg);
            cout << "Ping sent " << counter << endl;
        }
        else
        {
            end = GetTickCount();
            ConcurrentPing::startTimes[name.id-1] = start;
            ConcurrentPing::endTimes[name.id-1] = end;
            int delta = end - start;
            double avg = ((double)delta) / ((double)max);
            cout << "Messages transferred: " << max << endl;
            cout << "Total time: " << delta << " ms." << endl;
            cout << "Average message time: " << avg << endl;
        }
    }
}
```

Figure 12. PingActor.cpp

```cpp
#include "PongActor.h"
#include <iostream>
#include <string>

using namespace std;

PongActor::PongActor()
{
    for(int i = 0; i < (MESSAGE_SIZE - 1); i++)
        huge_msg[i] = ((i + 5) % 10) +48;
    huge_msg[MESSAGE_SIZE - 1] = 0;
    //cout << huge_msg << endl;
}

PongActor::~PongActor()
{

}

void PongActor::handle(string sender,string message)
{
    send(sender,huge_msg);
}
```

Figure 13. PongActor.cpp

```cpp
#include "PongActor.h"
#include <iostream>
#include <string>

using namespace std;

PongActor::PongActor()
{
    for(int i = 0; i < (MESSAGE_SIZE - 1); i++)
        huge_msg[i] = ((i + 5) % 10) +48;
    huge_msg[MESSAGE_SIZE - 1] = 0;
    //cout << huge_msg << endl;
}

PongActor::~PongActor()
{

}

void PongActor::handle(string sender,string message)
{
    send(sender,huge_msg);
}
```

Figure 14. PongActor.cpp

```
ActorName ActorFactory::CreateActor(string actorName)
{
    try
    {
        Actor *actor=0;
        bool simpleName = false;
        bool bindRemotely = true;

        if (actorName == "PingActor")
        {
            cout<<"Ping Actor out of commission for now";
            actor = new PingActor();
        }
        else if (actorName == "PongActor")
        {
            cout<<"Pong Actor out of commission for now";
            actor = new PongActor();
        }
        else if (actorName == "Coordinator")
        {
            actor = new Coordinator();
            simpleName = true;
        }
        else if (actorName == "CRFDM")
        {
            actor = new CRFDM();
            simpleName = false;
            bindRemotely = true;
        }
        else if (actorName == "PingRoleRRFDM")
        {
            actor = new RRFDM("PingRole");
            simpleName = true;
            bindRemotely = !(AAPlatform::Instance()->getMode() == ARC_MODE_FDM);
        }
        else if (actorName == "PongRoleRRFDM")
        {
            actor = new RRFDM("PodngRole");
            simpleName = true;
            bindRemotely = !(AAPlatform::Instance()->getMode() == ARC_MODE_FDM);
        }
        else
        {
            ActorName failed;
            failed.id = -1;
            failed.type = "";
            failed.uan = "";
            cerr << "Unknown actor type " << actorName << endl;
            return failed;
        }

        ActorName name = UANManager::Instance()->generateName(actorName,actor,simpleName);
        actor->setName(name);

        if(bindRemotely)
        {
            Messenger_i *servant = new Messenger_i(actor);

            CORBA::Object_var objRef = poa->servant_to_reference(servant);

            CosNaming::Name objName;
            objName.length(2);
            objName[0].id = CORBA::string_dup("example");
            objName[1].id = CORBA::string_dup(name.uan.c_str());
            namingRoot1->rebind(objName,objRef.in());

            cout << name.uan << " bound in Naming Service." << endl;
        }
        else
        {
            cout << name.uan << " created locally." << endl;
        }

        return name;
    }
    catch(CORBA::Exception& ex)
    {
        cerr << "ActorFactory CORBA Exception: " << ex << endl;
    }
```

Figure 15. ActorFactory.cpp

There are two property files needed for running AAPlatforms in two physical machines. The IP addresses need to be changed to the one you are using to run the TAO naming service. As we can see, in this simple example we don't need roles and coordinators, and only one actor in each platform is created. Our next example in next section will show a more complicated property file instance.

Properties1

PLATFORM_NAME first
NAMING_SERVICE NameService=corbaloc:iiop:192.168.0.2:2809/NameService
MODE FDM
CREATE_COORDINATOR NO
ACTORS PingActor

Properties2

PLATFORM_NAME second
NAMING_SERVICE NameService=corbaloc:iiop:192.168.0.2:2809/NameService
MODE FDM
CREATE_COORDINATOR NO
ACTORS PongActor

4. Start naming service.

   In one of the machines with the IP address 192.168.0.2, run the following command to start the naming service.

   $TAO_ROOT/orbsvcs/Naming_Service/Naming_Service -ORBListenEndpoints iiop://192.168.0.2:2809

5. Start AAPlatform in physical nodes.

   In each machine, run the following command respectively.

   PlatformRun Properties1

   PlatformRun Properties2

6. Run application

   In the AAPlatform where the PingActor locates, type the following command:

   send uan:PingActor:first:1 start

   You then can see messages are exchanged between two machines just like a PingPong game.

3.2 ARC-based Ping Pong Example

To be continued.

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] N. Chen., S. Ren, Y. Yu, and M. Beckmen. A role-based coordination model and its realization. *Journal of Informatica*, 32(3):229–244, 2008.

[4] S. REN, N. CHEN, Y. YU, P.-E. POIROT, S. L., and K. MARTH. Actors, roles and coordinators a coordination model for open distributed embedded systems. 4038:247–265, 2006.
[5] D. C. SCHMIDT. The design of the tao real-time object request broker. In *Computer Communications*, 1998.