Implementation of an Enhanced Distributed Object Model using Actors

Siva Challa and Dennis Kafura

March 27, 1998

Abstract

Most common object models of distributed object systems have a limited set of object-oriented features, lacking the advanced features of 'polymorphism' (an abstraction mechanism that represents a quality or state of being able to assume different forms) and 'concurrency' (the ability to have more than one thread of execution in an object simultaneously). The lack of support for advanced features is a serious limitation because it restricts the development of new components and limits reuse of existing of components that use these advanced features. In this paper, the Interoperable Common Object Model (ICOM) centered on statically typed object-oriented languages is presented. The ICOM model is an attempt to elevate common object models (with the advanced features of polymorphism and concurrency) closer to the object models of statically typed object-oriented languages. Specific features of the ICOM object model include: remote inheritance, method overloading, parameterized types, and guard methods. This paper focuses on how the *actor* model and *reflection* techniques are used to develop a uniform implementation framework for the ICOM object model in C++ and Modula-3. Two key features of the ICOM objects are discussed, remote inheritance and atomicity, along with the distributed compilation architecture of the ICOM framework emphasizing the role of the actor model in creating a simple underlying framework is emphasized.

1 Introduction

Interoperability is the ability of the components of a software system to coexist, communicate, and cooperate with one another to achieve a common goal in a distributed and heterogeneous environment involving disparate implementation languages, operating systems, and hardware platforms. Integrating existing software components written in different programming languages is becoming a common practice in the software development process because it reduces code duplication and results in better reuse and maintenance of software.

Many common object models [2, 7, 8, 13, 14] have been proposed to support interoperability of software components developed in multiple languages. The systems based on these models try to encapsulate the communication mechanisms used for the transfer of messages between objects in a distributed environment. They provide translations of object descriptions described in Interface Definition Language (IDL), into different languages. Most of the systems provide language transparency (an application developer can act as if all application components are available in the native implementation language) and distribution transparency (an application developer can act as if all application components are available locally). Several systems have mappings supporting both object-oriented and non object-oriented languages.

Most of the common object models, including the popular industry standard models — the Common Object Request Broker Architecture (CORBA) [7] and the Component Object Model (COM) [2]— do not support certain useful and advanced features of 'polymorphism' and 'concurrency'. These models are limited in the object-oriented features supported by them. Although the features of polymorphism and concurrency are present in several object-oriented languages (see Table 1) and have been proven to be very useful in the application development process, they cannot be used for interoperability of software components because they are not part of the existing common object models. The lack of support for these features in the existing common object models is a serious limitation because, it restricts the reuse of software components developed using these advanced features.

Table 1: Statically Typed Object-Oriented Languages Versus Important Features Supported

Language	*Features*		
	Method Overloading	Generic Types	Concurrency
Ada 95	√	√	√
BETA	*weak*	√	√
C++	√	√	*library*
Eiffel	√	√	*library*
Java	√	*proposed [1, 16, 20]*	√
Modula-3	*weak*	√	√

The Interoperable Common Object Model (ICOM) presented in this paper concentrates on statically typed object-oriented languages. Static typing offers both a degree of safety to applications and is critical in defining the interfaces of remote objects and the data that is exchanged between client and server objects. The languages used in the implementation and testing are C++ [18] and Modula-3 [17]. Though these languages fall within the same family of statically typed languages, they are similar enough to yield a powerful object model, yet different enough to make the problem interesting, and widely enough used to create interesting opportunities for reuse (e.g., a Modula-3 animated user interface to a C++ digital library was implemented as a demonstration of this approach).

The focus of this paper is the ICOM framework that provides a uniform design for translation of the ICOM object model into statically typed object-oriented languages. The framework uses the *actor model* of concurrent computation and *reflection techniques*. Reflection techniques are used to map the common object model supported by the ICOM framework into an object-oriented language. Here, the behavior of the model or the language is not changed. That is, a feature which is present in the common object model but not in an implementation language is interpreted in terms of existing features of the language. It is possible that most of the existing languages support a feature, and the feature is supported in a common object model. There will be languages that do not support some of the features of a common object model. In such a case, there should at least be a provision in the common object model translation framework for one-way interoperability which allows that language to use the components from other languages developed using the same feature. For example, Modula-3 does not support method overloading. But the ICOM framework in Modula-3 allows Modula-3 code to use this feature by letting it to make invocations on the components of other languages that have overloaded methods. This is very important because application developers are not limited to reusing just the code that is developed without method overloading.

The ICOM framework differs from the other distributed object systems in the way messages are communicated and the semantics of method invocations. Specific aspects of the ICOM framework are as follows:

Autonomous Objects
: All objects that have their interfaces described in IDL are autonomous. They have their own thread of control for processing method invocations.

Asynchronous invocations
: The method invocations in the ICOM framework are asynchronous. A client object can continue after making an invocation, without waiting for a result.

Future variables
: To support synchronous invocation semantics, the ICOM framework provides future variables. Any output parameter in an operation of an interface will be mapped to a future variable. A future variable blocks an invoking process that tries to access its value when the value is not yet set. The future variable unblocks the invoking process when the variable value is set.

No return types
: There are no return types for operations of the interfaces in the ICOM framework. Any expected result must be represented as an output parameter.

Message queues
: Every autonomous object has a message queue associated with it. If an autonomous object is processing a message when a new message arrives, the new message will be entered into the message queue. After the processing of the previous method invocations is complete, the object removes the message from the queue and processes it.

Atomicity

In the ICOM framework, atomicity is guaranteed for all method invocations on autonomous objects. A method invocation is processed completely before another method invocation can start.

Implementation Inheritance

The ICOM framework supports both the interface inheritance and the implementation interitance. Interface inheritance forces the implementor of a derived interface to provide implementations for methods defined in the base interface. Using implementation inheritance, an implementor of a derived class can inherit the implementation of its base class and implement only the extended or overridden functionality in the derived class.

Method Overloading

Method overloading allows an interface to have more than one method with the same name but different signatures. When an overloaded method is invoked, the ICOM framework can identify the correct method and bind the invocation by by verifying the signatures of the methods.

Generic Types

The generic types supported in the ICOM model are 'constrained generic types'. These generic types are similar to C++ templates(unconstrained generic types), but are not as powerful as them. The generic types of the ICOM object model can have one or more type parameters. But, the interfaces of the parameters must be defined before using them as the type parameters.

Guard Methods

Guard methods are a useful feature to define run-time constraints on method invocations of an object. They are used for synchronization, preserving internal consistency of interacting objects. When a method that has a guard method associated with it is invoked, the guard method will be executed first. If the guard method returns 'true' via its output parameters, then the guarded method will be executed.

Detailed explanations of the features of the ICOM object model are discussed in [3]. This paper shows how actors and reflection are used to support two key properties of ICOM objects: implementation inheritance and atomicity.

The ICOM framework is based on a distributed actor system. Any distributed object generated as a result of IDL translation, is an *active* object in the ICOM framework. An active object is a first class entity. Every active object has a message queue. All messages intended for the object are enqueued in the message queue and processed one at a time. Any other object is a *passive* object in ICOM framework. A passive object cannot service method invocations in its own thread of execution. The ICOM framework allows the coexistence of both active and passive objects in an application. A client application treats all objects as passive in the basic ICOM framework. Some of these passive objects representing the remote objects are implemented as active objects. The framework tries to hide the differences between the active and passive objects. For an application developer there is minimal difference between implementing an active object and a passive object. The only requirement for implementing an active object is to describe the object's behavior in IDL, use an IDL translator to generate native language representations, and link the application code with the IDL translator generated code. Location transparency of the object implementations is provided in the ICOM framework.

The remainder of this paper is organized as follows. Section 2 describes the ICOM concept of a truly distributed object, an object that combines implementation inheritance and atomicity. The logical organization of the run-time structure of a truly distributed is described and the sequence of actions that support the method invocation are illustrated. Section 3 explains the role of actors and reflection in implementing the ICOM framework. The issues involved with guaranteeing the atomicity property are discussed. The features of a distributed actor system are described.

2 Truly Distributed Objects

The ICOM framework supports basic polymorphism through truly distributed objects, objects that are instances of classes whose base classes are defined across language and machine boundaries. A common object model that incorporates truly distributed objects is said to support basic polymorphism. Truly distributed objects distinguish themselves from regular objects by keeping parts of their state and method information at distributed locations. These distributed parts of the object are related by a superclass and subclass relationship. A truly distributed object allows methods in the superclass to be overridden in a subclass. When overridden methods are invoked in the parent, the overriding methods of the child are executed. A locking mechanism generates an atomic property so that the integrity of the truly distributed object is preserved in the face of simultaneous invocations.

A simplified representation of the truly distributed object is shown in Figure 1. The cracked egg shell representation is used to show the distribution of state and methods of the truly distributed object. The patterned portions represent the code that must be implemented by a developer. The method implementation that is part of the white interior in the objects, represents stub methods that delegate any invocations on them to their corresponding implementation at the other node. The combination of the patterned portion and the white portion of the cracked egg shell representation denotes a *peer proxy* at that site. So, in the ICOM framework, peer proxies represent truly distributed objects at the sites that have at least one component of the truly distributed object.

Implementation inheritance, also called *class inheritance*, allows an implementor of a derived class to inherit the implementation of its base class and implement only the extended or overridden functionality in the derived class. In contrast, existing distributed systems [2, 7, 8] support *interface inheritance* where only the specification is inherited from a parent

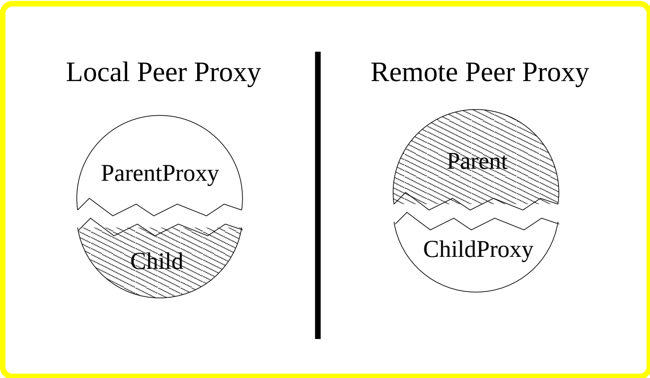

Figure 1: A Simplified Representation of a Truly Distributed Object

interface. This means that an implementation supporting an interface child that inherits from another interface parent must provide implementation for the methods in both the interfaces, child and parent. The drawback of such an approach is that if an implementation exists for the interface parent on a remote machine, the implementation of the interface child cannot reuse it on the local machine. Thus, existing systems do not support objects that are distributed across processes possibly on different machines.

Truly distributed objects differ from objects in other common object models in two ways. First, implementation inheritance is not supported in most of the existing common object models. Some distributed object systems partially support implementation inheritance using *proxies*. A proxy is a surrogate object that represents a remote object at the local site. A client using a proxy can treat it as a real object. The CORBA compliant system, ORBIX, from IONA Technologies, supports inheritance using proxies [19]. But, this works only for split objects[1]. Because, in the ORBIX system, inheritance of abstract methods is not possible across different address spaces it is fair to say that the ORBIX system supports partial implementation inheritance. Another experimental system similar to ICOM in its inheritance features is the Heron system from the Free University of Berlin [21]. Heron supports implementation inheritance following the semantics of the Eiffel language. In Heron, every participating node in the system has a configuration file that keeps track of the implementation locations of the server objects. When processed, these configuration files are translated to proxy classes that can be used by the application developers to achieve implementation inheritance. In other object models including that of Heron, it is not clear how multiple simultaneous method invocations on different components of a truly distributed object are handled. ICOM guarantees atomicity of method executions even

[1] A *split object* differs from a truly distributed object in the way the various parts of the object are distributed. The distributed parts of a split object need not be related by any superclass and subclass relationships. The distribution decisions can be arbitrary. Thus, every truly distributed object is a split object, but not every split object is a truly distributed object.

when multiple method invocations on different components of an object are involved.

2.1 Distributed Compilation

In the existing distributed object systems, an IDL translator must be invoked independently (stand alone compilation) at a client site as well as at a server site for a client and a server to work together in a distributed environment. No information exchange takes place between a client node and a server node during the translation. The main barrier to truly distributed objects in existing systems can be attributed to the fact that at a remote site that implements a superclass interface, there is no information about the existence of a subclass that can override an abstract method defined at the superclass. So, an invocation of an abstract method defined at the superclass, on an instance of the subclass, results in the incorrect invocation of the method implemented at the superclass.

In ICOM, a new concept of *distributed compilation* is introduced to support communication between distributed nodes at IDL compilation time. Distributed compilation involves executions of translators at multiple sites when a truly distributed object description is to be translated from IDL to a native language. In the ICOM framework, each IDL translator is treated as an active object that has its own thread of execution. As seen in Chapter 3, the initialization phase of the distributed object system involves initial interaction between different distributed nodes. After the communication between different nodes is established, IDL translators can be registered with each node. So, each node can access the IDL translators on all nodes. Any IDL translator can potentially send messages to any other IDL translator.

In the current prototype, because our primary focus was on the object model and its run-time support, the IDL translations are hand generated. However, all of the processes involving actor creation, registration, and communication between the objects is done automatically by the system at runtime.

Figure 2 shows the interface descriptions of two interfaces, GeneralTax and VirginiaTax. In this figure, the interface GeneralTax is a base interface and the interface VirginiaTax is a derived interface. These two classes are used to find out the total tax for the state of Virginia in the United States of America. This tax is used to decide the price of an item sold at a retail clothes store in Virginia. The GeneralTax interface defines three methods, totalTax, stateTax, and salesTax. The VirginiaTax interface inherits from the GeneralTax interface and overrides the two methods, stateTax and salesTax methods.

The distributed compilation mechanism used in ICOM is illustrated in Figure 3. In the figure, the cracked egg shell representation shows a distributed object with its state and methods split on two different nodes. The dashed arrows represent the portion of code generated by the translator. The double circles represent classes and the single circles represent objects. The patterned objects represent the code that must be implemented by a developer. All remaining objects with white interior have their code generated automatically by the IDL translator.

The working of an IDL translator can be seen with the translation of a truly distributed

```
// GeneralTax defines two methods
interface GeneralTax {
    // returns the sales tax through 'tax'
    salesTax(out double tax);
    // returns the state tax through 'tax'
    stateTax(out double tax);
    // returns the total tax for a state
    totalTax(out double tax);
}

// VirginiaTax type inherits from GeneralTax and
// overrides the two methods
interface VirginiaTax : reuse GeneralTax {
    // returns the sales tax through 'tax'
    salesTax(out double tax);
    // returns the state tax through 'tax'
    stateTax(out double tax);
  }
```

Figure 2: IDL Description of Virginia Tax Distributed Object Interface

object. Consider a truly distributed object with a superclass part GeneralTax existing on a remote node called remote, and a subclass part VirginiaTax existing on a local node called local. The nodes are separated physically via a local area network or a wide area network. Assume that an application developer at the local node intends to use an instance of the distributed object. At the local node the application developer starts the IDL translation on the IDL description of the truly distributed object that contains the interfaces of GeneralTax and VirginiaTax.

Assume that the implementation of the interface GeneralTax already exists at the remote node. For each interface in the inheritance hierarchy four classes are generated. This generation preserves the inheritance relationships at each distributed node which has the implementation of an interface in the hierarchy. The IDL translator generates four classes for the GeneralTax interface: GeneralTaxMetaClass, GeneralTaxImplementation, GeneralTaxMessageClass, and GeneralTax. The GeneralTaxMetaClass contains a method table. Each row in the method table contains a method name, a class name, and a server method object corresponding to the method name. The table contains entries for each method defined in the interface. Each server method object is an instance of a local class (called server method class) defined in the metaclass. The purpose of a server method class is to decode an incoming message. If a message arrives over a network, the method class instance whose method name matches the method name in the incoming message, decodes the arguments in the message. In the case of a local invocation that does not involve any message encoding, the decoding process simply consists of copying of the arguments from the message object. The GeneralTaxImplementation class is the actual implementation of the GeneralTax interface. It must be implemented by a developer at the remote node. All of the remaining classes are automatically generated by the IDL translator. The GeneralTaxMessage class contains

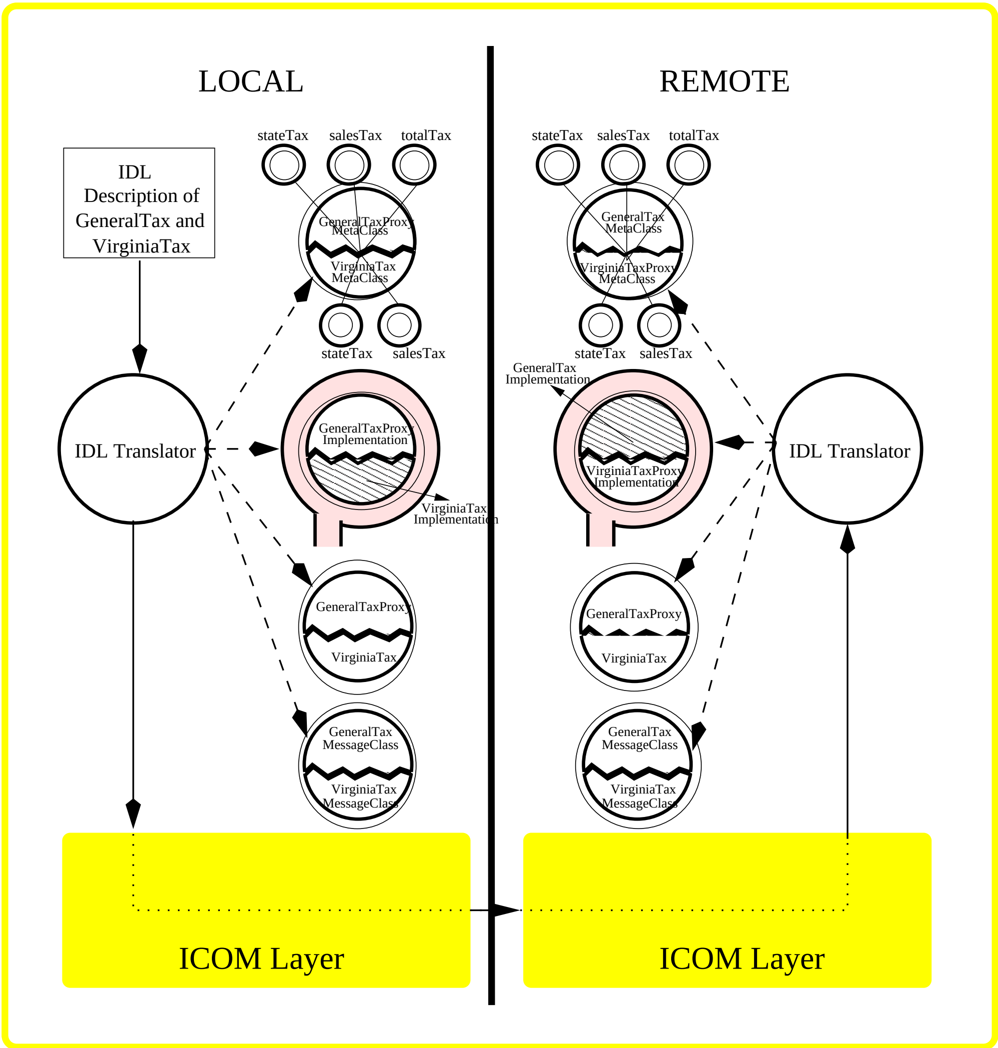

Figure 3: IDL Translation of Virginia Tax Distributed Object Interface

classes defined for each method defined in the GeneralTax interface. These method classes are called *client method classes* because their purpose is to create message objects for each method invocation on an active object. The message objects are capable of encoding themselves into a network representation format. The GeneralTax class supports the GeneralTax interface. It does not contain any state information except the address of a corresponding active object of GeneralTaxImplementation. Any application that wants to invoke methods on a GeneralTax object must use a GeneralTax class instance as a handle.

The implementation of the GeneralTax interface is made available to others by linking the IDL generated code with the GeneralTaxImplementation code by an implementor, and using a native language compiler to generate the executable. Any application must load the GeneralTaxMetaClass either statically or dynamically before trying to create and use a GeneralTax object. In the current implementation of the ICOM framework, dynamic loading is used in C++ and static loading is used in Modula-3.

In the current example, the GeneralTax is implemented at the remote node and the VirginiaTax is to be implemented at the local node. The IDL translator, when invoked on the IDL description of the truly distributed object that contains both GeneralTax and VirginiaTax interfaces, realizes that GeneralTax is implemented remotely. As in CORBA, the IDL translator can communicate with an *interface repository* which has details about all interfaces available in the distributed object system, to find out the location of the GeneralTax implementation. The translator sends a compile message to the IDL translator at the remote node by passing the interface information of the truly distributed object. It continues with its translation in parallel. The IDL translator at the remote node generates four classes representing the VirginiaTax at the remote node. The classes generated are: VirginiaTaxProxyMetaClass, VirginiaTaxProxyImplementation, VirginiaTax, and VirginiaTaxMessageClass. The VirginiaTaxProxyMetaClass inherits from the GeneralTaxMetaClass. Its functionality is the same as that of the GeneralTaxMetaClass except that it is oriented towards the VirginiaTax class. The VirginiaTaxProxyImplementation implements stub methods whose purpose is to create message objects for each method invocation of the methods in the VirginiaTax's interface and send them to the local node. The VirginiaTaxMessageClass is used to create message objects meant for VirginiaTax. The VirginiaTax class is a handle to the implementation of VirginiaTax. It supports the VirginiaTax's interface and sends all messages to it to the VirginiaTax active object at the local node. After generating the required classes, the IDL translator at the remote node invokes a native language compiler to generate executable code.

The IDL translator at the local node generates eight classes, four each for the GeneralTax and VirginiaTax interfaces. The parent classes generated are: GeneralTaxProxyMetaClass, GeneralTaxProxyImplementation, GeneralTaxMessageClass, and GeneralTax. Their functionality is similar to the VirginiaTax classes at the remote node, except that they are oriented towards the GeneralTax implementation at the remote node. They provide skeletal structures and delegate any method invocations to the remote node.

The classes generated at the local node are: VirginiaTaxMetaClass, VirginiaTaxImplementation, VirginiaTaxMessageClass, and VirginiaTax. The VirginiaTaxMetaClass inherits from GeneralTaxProxyMetaClass, the VirginiaTaxImplementation class inherits from GeneralTaxProxyImplementation class, the VirginiaTaxMessageClass inherits from GeneralTaxMessageClass, and finally the VirginiaTax class inherits from GeneralTaxClass. The derived classes' functionality is similar to the base classes' functionality at the remote node.

A developer needs to implement the VirginiaTaxImplementation class and invoke the native language compiler on the source code of the VirginiaTaxImplementation class and the IDL translator generated code to build an executable. The developer can include the VirginiaTax class in the application and use it to create and invoke messages on an instance of VirginiaTax. A client application can treat the component of the distributed object at the client site as a complete object and invoke operations on it. The underlying system executes the correct implementations of the invoked methods.

2.2 Method Invocation on a Truly Distributed Object

In the previous section, a detailed description of how the distributed translation works on a truly distributed object is given. In this section, a method invocation on an object supporting the interface VirginiaTax is described in detail.

Consider a node (machine in a network) that has class whose implementation (hereafter referred to as parent) supporting the GeneralTax interface. Assume that another node (hereafter referred to as the child node) has a class whose implementation supports the interface VirginiaTax, which inherits from the interface GeneralTax.

Assume that a client application has an instance variable handle that is a handle to the class GeneralTax, but contains an instance of the class VirginiaTax. A method invocation on the method stateTax on the handle results in a sequence of method calls. These calls are numbered in the order of their invocation as shown in Figure 4. The method invocation of stateTax is received by the VirginiaTaxProxy and is forwarded to the VirginiaTaxProxyImplementation. The VirginiaTaxProxyImplementation class forwards the method to the VirginiaTax's implementation. Eventually, results will be sent to the client application via the output parameter tax of the method stateTax. The call sequence is explained in more detail as follows.

At the parent node

A client application can make an invocation on the handle that refers to an instance of VirginiaTax class. This call is shown with the arrow labeled 1. The application uses a handle to the active object instance. This handle refers to an instance of VirginiaTax class which inherits from GeneralTax class. The handle object creates a message object for the method stateTax. The message object is forwarded (arrow labeled 2) to the instance of VirginiaTaxProxyMetaClass. This class inherits from GeneralTaxMetaClass. The message object then is forwarded to the server method object named stateTax (arrow labeled 3). Since the invocation is a local invocation at parent node, the decoding process in the server method object simply copies the arguments from the

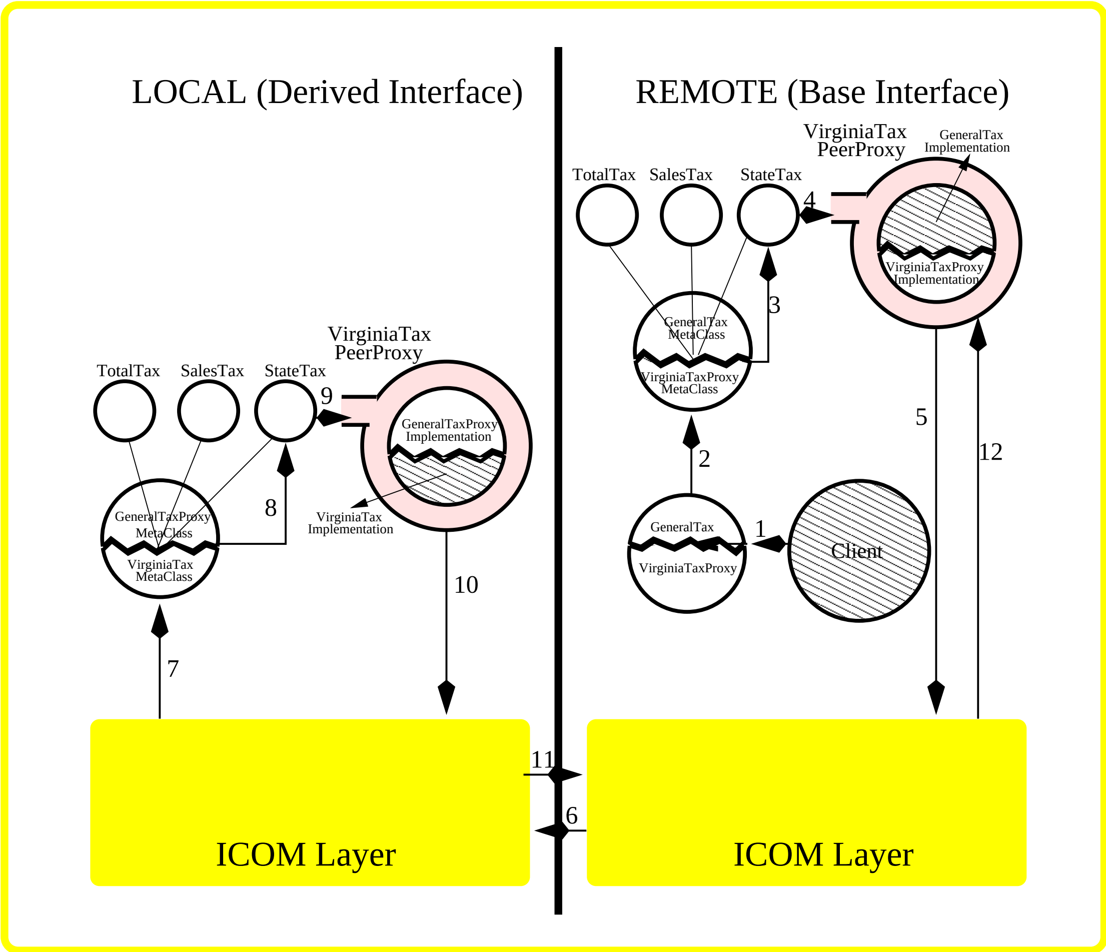

Figure 4: Dynamic Method Invocation on Virginia Tax Distributed Object

message into the method object stateTax. The server method object stateTax makes an invocation on the *peerProxy* object (arrow labeled 4). Even though a method implementation exists in the GeneralTaxImplementation class, since the method is an abstract method that is overridden in the derived class, the stateTax method of the class VirginiaTaxProxyImplementation is invoked. This proxy implementation is a stub implementation that creates a method object and forwards it to the child node (arrow labeled 5). The ICOM layer encodes the message object by calling the encode on the message object. The encoded message is then sent to the child node (arrow labeled 6).

At the child node

The child node receives the encoded message from the network. It partially decodes the message to obtain the method name and the class name of the stateTax object. It identifies the metaclass instance corresponding the name 'VirginiaTax' and forwards the partially decoded message to it (arrow labeled 7). The metaclass instance hands the message to the server method object stateTax. The message is decoded in the object stateTax (arrow labeled 8). It then sends the decoded message to the *peerProxy* (arrow labeled 9). The stateTax method of the *peerProxy* is executed. Since the message contains a *future* variable, when the variable is a value assigned to it, a message goes to the parent node (arrow labeled 10). This message is encoded and sent to the parent node through ICOM layer (arrow labeled 11).

At the parent node

The ICOM layer at the parent node receives the message, decodes it and updates the value at the client application (arrow labeled 12).

3 Implementating the ICOM Framework

To implement the framework that handles the method invocations as shown in the example described in the previous section, the ICOM framework uses reflection techniques and the actor model. The framework uses the distributed actor model to provide the underlying message passing between various nodes in the system. The framework is implemented with a minimal set of object-oriented features. The following sections describe the set of object-oriented features used for implementing the ICOM framework, reflection techniques, and the actor model in detail.

3.1 Core Set of Features

The ICOM framework is implemented using the object-oriented language features of 'object', 'class', 'single inheritance', 'method overloading', and 'concurrency'. This set of features is considered as the *core set* of object-oriented features. As mentioned in earlier, all these features are either directly or indirectly supported in statically typed object-oriented languages. The languages, BETA and Modula-3, do not support method overloading feature. But, they have other features such as 'run-time type checking' which allow method overloading feature to be realized. In all statically typed object-oriented languages concurrency is either directly supported or is supported via standard libraries. The ICOM framework can be translated to any language that supports this core set of object-oriented features.

3.2 Reflection

In a typical object-oriented system, objects perform computation about an external domain. A reflective object-oriented system is said to be *causally* connected to its domain, if one of them changes, it leads to a corresponding change in the other [12].

Reflection is defined as the ability of a system to make structures representing (aspects of) itself the subject (or domain) of its computation in a causally connected way [12].

Reification is the representation of specific aspects of a program as data structures or classes within that program [4]. These aspects include method binding, inheritance etc. The resulting objects are referred to as *metaobjects*. Invoking an operation on a metaobject is called *reflective computation*.

There are three approaches to reflective computation: *Metaclass, Metaobject*, and *Meta-Communication* [4]. If a reflective computation modifies a metaobject, then it is called a *reflective update* and the process is called *behavioral reflection*. If a reflective computation does not modify a metaobject, then that process is called *structural reflection*. The operations defined for the metaobjects constitute a *metaobject protocol* [11].

Metaclass Approach

In the *Metaclass* approach, the class of an object is considered as its metaobject. *Metaclasses* are metaobjects of classes. That is, a class is an instance of its metaclass. Metaclasses can have metaobjects for themselves, thus forming an infinite *reflective tower* [12]. This tower is normally terminated by defining a root metaclass which is a metaobject of itself [12].

A method invocation on an object can be described as follows: when an object receives a message, it is delegated to the object's metaobject, an object representing the class of the receiving object. The metaobject does a *lookup* operation for matching a method with the message and *apply* operation to execute that method. If the class itself has a metaobject, then the message is delegated to the metaobject which has its own lookup and apply operations. Normally, the root metaclass supplies default lookup and apply operations. If the behavior of a class needs to be changed, then the application programmer has to replace the default behavior of the root metaclass. This can be done by defining a specialized class from the root metaclass and assigning it as a metaclass of the class that required the behavior change. In this approach a metaclass is used for both structural reflection and behavioral reflection.

Metaobject Approach

In the Metaobject approach, an object can have a metaobject for itself as opposed to a class having a metaclass in the Metaclass approach. This metaobject is an instance of a special class, typically called the 'MetaObject'. MetaObject defines the default behavior of all objects. These metaobjects are different from the class of the receiver (base-level object). They include only the operational and the control information about their base-level objects. The class of the receiver has the structural information. Classes handle structural reflection and metaobjects handle behavioral reflection. Upon receiving a message, a receiver delegates it to its metaobject, which performs *lookup* and *apply* operations. To change the default behavior of an object, the MetaObject can be specialized and assigned to the object. Since, each object has a metaobject, the number of objects in the system grows enormously. To control the number of metaobjects in the system, a *lazy* way of metaobject creation [4] can be followed where a metaobject is created only when needed. The advantage of this approach is that individual objects can be controlled in different ways even though they belong to the same class.

MetaCommunication Approach

This approach is based on the reification of the communication mechanism. Each message is treated as an object. Messages are represented as instances of a special class, typically called 'Message', that defines a method *send* for transmitting the message to the receiver. Every message object contains the address of a sender and a receiver. It is the responsibility of a message to interpret itself. That is, a message contains enough information about its method name, class name, and the arguments for the method. This information can be used to deliver the message to the right object. The default interpretation information is provided in the 'Message' class.

A combination of the MetaClass and MetaCommunication approaches is used to map the ICOM framework into the implementation languages. Since these techniques are used to map a model into a language, but not to change the semantics of a language, they are called *reflection-like* techniques.

3.3 Actor Model

The actor model is a concurrent computation model that unifies objects and concurrency. The model provides basic building blocks, called actors, for concurrent programming. An actor is a computation agent that has the ability to process messages. It is an active entity in the sense that every actor maintains the integrity of information within itself and processes its messages at its own discretion. Any computation in a system based on actor model is a result of message processing by the actors. Thus actors are autonomous and concurrently executing objects which execute asynchronously.

An actor consists of an address which represents the actor's identity and a script which represents the actor's behavior. A message queue is implicitly maintained in an actor.

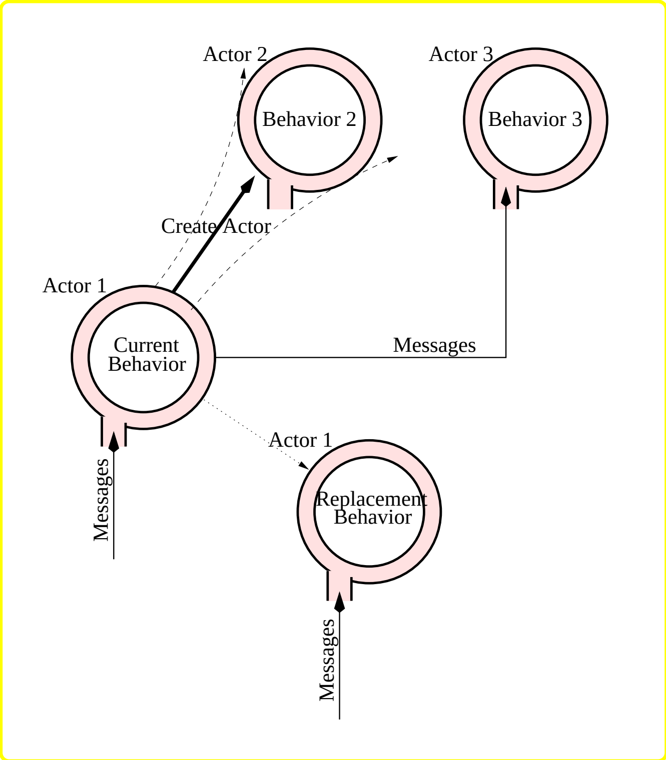

Figure 5: Actor Model

Any number of incoming messages can be received by the actor in its message queue. The behavior of an actor is responsible for processing the messages one at a time.

As a response to a message, an actor may send messages to its *acquaintances* (other actors that it knows about), create new actors, and specify a replacement behavior which processes the next message from the message queue of the actor, as shown in Figure 5. An actor can have more than one thread of execution at a time. That is, a replacement behavior can be created while a message is being executed. The new behavior can start processing another message while the previous message is being executed. This kind of behavior replacement results in concurrent execution of messages and can potentially improve the performance of a system based on the actor model. It should be noted that concurrency also results with the simultaneous message processing by different actors.

Depending on the application, an actor can choose to accept only a certain group of messages, after processing a message. For example, if an actor represents a stack behavior, the initial behavior can accept only the `push` messages. After processing a `push` message, the actor can accept both `push` and `pop` messages. Depending on what type of behavior an actor currently has, the messages in the message queue may get processed right away or stay in the queue until a suitable replacement behavior is provided by an existing behavior.

Concurrent object-oriented systems can be implemented using the actor model. Act++ [9] is a shared memory object-oriented programming environment based on actor model. It is implemented in C++. To facilitate synchronous communication, Act++ supports the concept of a 'Cbox'. A Cbox is a *future* variable. A future variable blocks the calling process until its value is set. Since messages are the only communication mechanism in the actor model, return values required in any method invocation are passed as Cbox arguments. For example, a method that is expected to return an integer can be implemented to take an 'integer Cbox' as its argument. The method implementation can set the value of its argument which is sent to the Cbox variable declared at the invoking process. The invoking process can continue its execution after the invocation. When it needs the value of the integer Cbox, it tries to access the value of the integer Cbox and is blocked if a value is not available. When the called method sets the value of its Cbox argument, a message comes back to the invoking process setting the value of the Cbox variable. At this point the Cbox value is available to the invoking process and it can continue with its computation.

3.4 Distributed Actor model

The Act++ system is extended to support distributed actors in distributed Act++ [10] programming environment. In a distributed Act++ environment every actor has a network-wide unique address. This address contains the node number which is a unique number assigned for a node in the system and a natural number that is unique for the actor at that node. A node in the distributed Act++ system is an abstract computer. An abstract computer consists of communication channels that can be used to communicate with every other abstract computer in the system. It also contains an application manager that manages the applications running in the system. A computer in a network can represent a node or multiple nodes can exist on one computer. A distributed Act++ environment provides dedicated actors for communication between the nodes in the system. Each node in the system is connected to every other node via two dedicated communication actors called MsgReceiver and MsgSender. The MsgSender actor sends messages and the MsgReceiver actor receives messages meant for that node. If there are N nodes in the system, each node has (N-1) MsgSender actors and (N-1) MsgReceiver actors in the system. So the number of dedicated actors used for communication between nodes is (N-1)*(N-1) MsgSender actors and (N-1)*(N-1) MsgReceiver actors.

A distributed actor system with two nodes on two different machines is shown in Figure 6. Each machine can support one or more nodes in the distributed actor system. One of the nodes is treated as the main node that acts as a centralized manager. Before joining the system, every other node has to establish communications with the main node. The main node helps the new node to establish communication channels with every other node in the system. The communication channels are represented in the figure as MsgSender and MsgReceiver. These channels are managed by dedicated actors whose sole purpose is to send and receive information respectively, from the nodes they represent. There is a Boot actor object that creates the AppMgr actor at any node. The AppMgr actor keeps track of all applications currently executing in the system. The initialization of the system and the communication between various actors is described below.

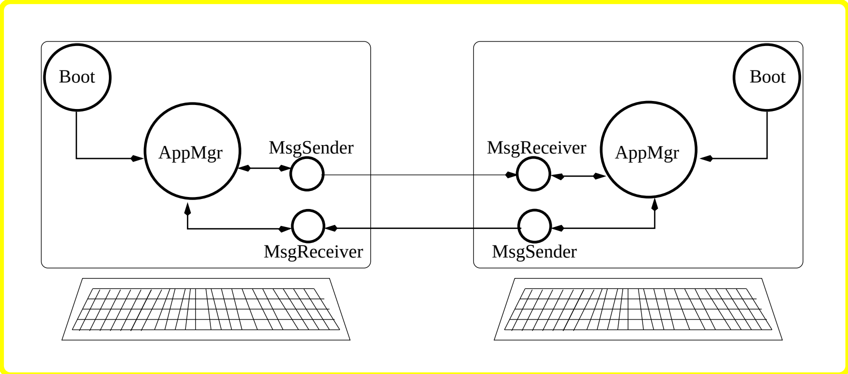

Figure 6: Distributed Act++ System with Two Nodes

The distributed Act++ environment requires an initial setup that makes the communicating nodes (on possibly distributed machines) aware of the existence of one another. This initialization establishes the communication channels among all nodes. For every node in the system, a separate channel is established with every other node in the system. On any node of a distributed system, prior to the execution of an application, the application is registered with the application manager actors on all nodes of the system. Every application manager actor maintains a table of references to application actors in the system. After the first node is created, any number of nodes can join the system by communicating with the first node. An example of the initial setup involving three nodes on three different computers in a network is shown in Figure 7.

To initiate an application at any node, a request message with an application name will be sent to any one of the application manager actors. The application manager actor loads the application actor, communicates with other application manager actors on the system about the new application and starts executing the application. The application actor can create on any node any number of actors required for its execution and communicate with them in achieving a common task.

To provide representation for data that can be transferred between machines with different hardware architectures, distributed Act++ supports a polymorphic object-oriented data specification framework (OODSF). This framework can be instantiated to support Abstract Syntax Notation 1 (ASN.1) [5] and the CORBA interface definition language (IDL) for data representation. The OODSF framework supports Basic Encoding Rules (BER) [6] and External Data Representation (XDR) [15] for transferring data between different nodes of the system.

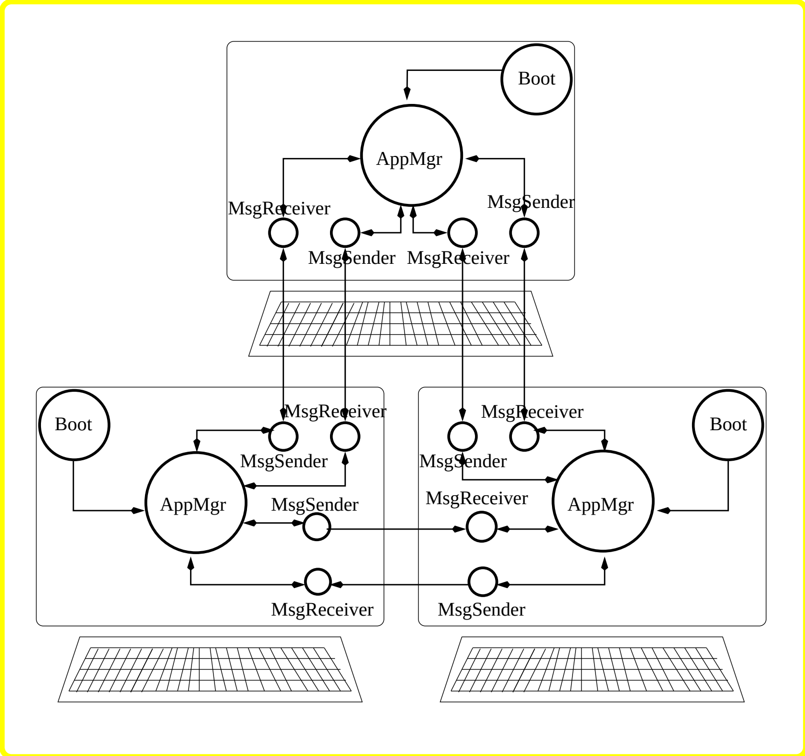

Figure 7: Distributed Act++ System with Three Nodes

3.5 Atomicity in Truly Distributed Objects

Atomic objects execute only one method at any time. That is, more than one method of the same object will not be in execution simultaneously. If there are multiple invocations on the object at the same time, only one invocation at a time will be processed by the object. There are two possible actions that can take place with the remaining invocations. In one case, all remaining invocations will be rejected so that the senders of those invocations can either attempt to invoke the same method at a later time or cancel the invocation. In the second case, the object maintains a queue for the remaining invocations and processes them one at a time. The selection can be in a First In First Out (FIFO) order or some other scheduling algorithm can be used.

In case of objects existing in a single process, a single lock per object can be maintained to ensure the atomicity property. There are two ways to use the lock. In the first way, the process trying to invoke a method of the atomic object can lock the object first by getting control over the lock prior to the invocation and releasing the lock after the completion of the invocation. One of the other processes waiting to acquire the lock gets control over the lock next, locks the object, and starts the execution of a method. Different types of locks can be used to control the locking/unlocking mechanism. Some of the mechanisms are: simple mutex variables, semaphores, and monitors. The second way to achieve the atomicity property is to let the object be responsible for managing its own lock. Before executing a message, the object sets the lock and continues with the execution. The object unsets the lock after the completion of the method execution. Any other method invocations that occur during the time the lock is set, either wait in the queue meant for the object or get rejected completely for invocation at a later time. Again, different locking/unlocking schemes can be used here.

In the case of distributed objects, it is possible that method invocations can occur at any distributed component. Two conditions must be valid with the distributed components. First, the execution sequence started by a method invocation must be completed before a second invocation starts. This sequence may involve method calls between distributed peers, or recursive method calls. Second, there should be a mechanism to share a distributed lock by the various distributed parts of the object. Any part of the object, must first acquire the lock before it can start an execution. Once an execution starts, all parts of the object must agree that they cannot start another new execution. That is, all parts of the distributed object will be in a blocked state. All of them are unblocked when the execution in progress is completed.

3.6 Actor Messages used for Communication

There are two ways to implement the communication between actors. First, dedicated actors can be created that can decide whether message is a constructor message, an invocation message or a reply message. In this approach there is only one basic message that is used for communication between actors. This approach results in many dedicated actors in the system, but the system will be uniform from the messaging point of view. The second approach extends the basic message set to include other basic messages such as the constructor message, invocation message, and a reply message. In this case, the system will have fewer actors but the messaging between actors is more complex. A sender and a receiver have to coordinate and agree upon the basic messages being sent and received. Although any approach can be followed to implement the communication, the second approach is followed in the ICOM framework implementation.

The relationship between an actor and an application object is shown in Figure 8. When the application invokes a method on an object that represents an actor (arrow labeled 1), the object first creates a message object using a message class corresponding to the method (arrow labeled 2), and then passes the message object to the acquaintance of the object (arrow labeled 3). For an application developer, this application object appears as a passive server object that responds to method invocations. But, in reality, this application object creates a message object and passes it to the actual server object implementation via the acquaintance.

There are three types of messages that are used for communication between actors in

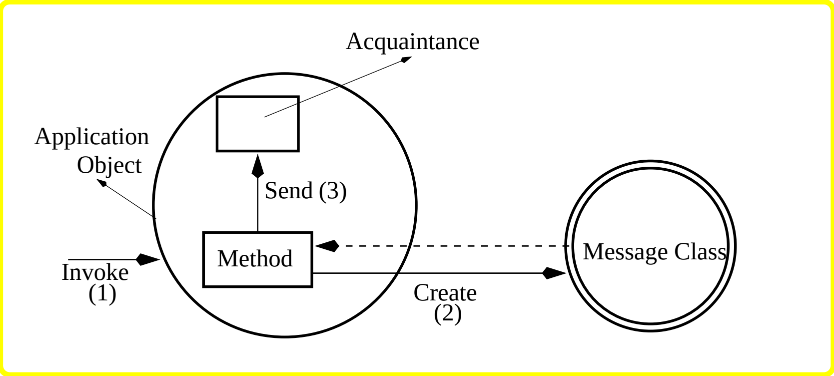

Figure 8: An Application Object that Represents an Actor

distributed Act++ system: constructor, invocation, and reply messages. The communication between actors during the creation of application objects and method invocations on these objects is transparent to the application developer. The application developer views the system as a distributed actor system.

A constructor message is an instance of a class called Create. This class contains information (the arguments needed for the class constructor method in C++ for instance) required to create an instance of an application object. An application object can be a client object or a server object. A constructor message is passed as a method argument to create an acquaintance to a local or a remote object.

If the constructor is used to create a local application object, the acquaintance to the object builds an instance of an actor by passing the constructor as an argument. Since, the actor has access to the constructor message, it can invoke a new method on the constructor object. The constructor object then creates an instance of the application object which becomes the actor's behavior. This application object is controlled by the actor that created it.

If the constructor is used to create a remote application object, the acquaintance encodes the constructor object into a common representation that is understood by all nodes in a distributed Act++ system. The encoded message contains the class name and method name of the application object. It will then be sent to the node that has the application object implementation. The invoking process blocks and waits for the address of the remote actor. When the remote node receives the encoded message it decodes it to find the class name and method name of the server object. The remote node then creates a constructor object based on the class and method information. At this point, the remote node creates an actor that is local to the remote node. The creation process is similar to the one explained in the previous paragraph.

A reply message is used to transfer addresses of actors between nodes. After a remote actor is created, its address will be sent back to the local node that originally sent a constructor message to the remote node. A reply message contains the address of an actor which is unique with regard to the distributed Act++ system. The remote node encodes and sends the reply message. The encoded message contains the class name and method name information and the address of the application actor. At the local node, the blocked process will be released after receiving the address of the remote actor.

An invocation message is an instance of a method class generated by the IDL translator. There is one method class for each operation defined in the interface of the application object. The invocation message object holds as its state information the arguments for the operation of the corresponding method of the application object. To invoke a method on an active application object, the application hands the invocation message to the acquaintance of the active object. If the invocation is on a local actor, the message object will be handed to the actor that either delegates the message to its behavior for execution or enqueues it in its message queue for later processing. If the invocation is on a remote actor, the invocation message will be encoded and sent to the remote site by the acquaintance. The encoded message contains the method and class information along with the arguments required for that method. At the receiving side, the message will be decoded and reconstructed to form a message object again. This message object will be handed to the actor of the application object. If there are any output parameters in the method, their values will be sent back to the local node using reply messages.

The framework of ICOM supports truly distributed objects by introducing two new message primitives to the existing distributed Act++ environment. The distributed Act++ environment defines three basic message primitives. These basic messages are: constructor, invocation, and reply messages. The new messages added are: peerInvoke and peerUnblock.

The new messages, peerInvoke and peerUnblock, are added to assure the atomicity of a method execution. That is, a message processing started must be completed before another message processing starts as a result of an external invocation. Various peer components of a truly distributed object must coordinate with one another to guarantee the atomicity of method execution.

In the ICOM framework, any peer proxy that starts message processing is treated as the 'owner' that has permission to process a message. The ICOM framework blocks any external invocations on all peer proxies, after an owner peer proxy starts executing a message. Any new messages will be entered into the message queue of the proxy. If a local peer proxy needs to make an internal invocation on a method of a peer proxy at a remote site, a peerInvoke message will be used. A peerInvoke message is similar to an invocation message, except that its name indicates that it is from a peer proxy of the distributed object. The actor corresponding to the remote peer proxy lets any peer invocations pass through without blocking them. This avoids deadlocks in the system that may occur because of peer invocations.

When any peer component completes an execution of a method without making any peer invocations in the method implementation, it sends peerUnblock message to the owner peer proxy that originally started the message processing. This original peer proxy is the owner of that execution. The purpose of a peerUnblock message is to unlock the peer proxies that were locked after a message processing starts. The owner peer proxy after receiving a peerUnblock either from a remote peer proxy or from itself, sends peerUnblock message to all peer proxies and waits for acknowledgments from them. When a non-owner peer proxy receives a peerUnblock message, it responds by sending another peerUnblock message back to the sender. The owner starts to execute new external messages after it receives acknowledgments from all peer proxies. The acknowledgment message is again a peerUnblock message. The owner can distinguish between the first peerUnblock message and the remaining acknowledging peerUnblock messages.

4 Summary

This paper introduced the features of the ICOM object model. The feature of implementation inheritance supported by the ICOM framework is described in detail. New concepts of truly distributed objects and distributed compilation are introduced. The paper discussed the support for remote inheritance in the distributed object system framework of the ICOM model. The implementation aspects of the ICOM framework that uses reflection techniques and the actor model are described in detail. The extensions made to the distributed Act++ system have been discussed at length. Examples are provided to explain the remote inheritance aspects of the ICOM framework.

References

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to Java language. In *Proceedings OOPSLA '97*, pages 49–65, October 1997.

[2] K. Brockschmidt. *Inside OLE 2*. Microsoft Press, Redmond, 1994.

[3] Siva Challa. *Improving Polymorphism and Concurrency in Common Object Models*. PhD thesis, Virginia Tech (VPI&SU), Blacksburg, Virginia, USA, January 1998.

[4] Jacques Ferber. Computational Reflection in Class Based Object-Oriented Languages. In *Proceedings OOPSLA '89*, pages 317–326, October 1989.

[5] International Organization for Standardization. Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Notation One(ASN.1). Technical Report ISO 8824/CCITT X.208, 1987.

[6] International Organization for Standardization. Information Processing - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One(ASN.1). Technical Report ISO 8825/CCITT X.209, 1987.

[7] Object Management Group. Common Object Request Broker: Architecture and Specification. Technical Report OMG Document Number 97.9.1, Rev. 2.1, OMG, 1997.

[8] IBM. SOMObjects Developer Toolkit Technical Overview, Version 2.0. Technical report, IBM, New York, November 1993.

[9] Dennis Kafura, Manibrata Mukherji, and Greg Lavender. ACT++: A class library for concurrent programming in C++ using actors. *Journal of Object-Oriented Programming*, pages 47–62, October 1993.

[10] Arjun Khare. ACT++ 3.0 : Implementation of the Actor Model Using POSIX Threads. Master's thesis, Virginia Tech, Blacksburg, VA, 1994.

[11] Gregor Kiczales, James des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, MA, 1991.

[12] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings OOPSLA '87*, pages 147–155, October 1987.

[13] Frank Manola and Sandra Heiler. A "RISC" Object Model for Object System Interoperation: Concepts and Applications. Technical Report TR-0231-08-93-165, GTE Laboratories, Waltham, MA, December 1993.

[14] Robert W. Mecklenburg. *Towards a Language Independent Object System*. PhD thesis, University of Utah, Salt Lake City, Utah, June 1991.

[15] Sun Microsystems. XDR: External Data Representation Standard(RFC 1014). Technical report, Network Information Center, SRI International, June 1987.

[16] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 132–145, January 1997.

[17] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[18] Bjarne Stroustrup. *The C++ Programming Language, 3rd Edition*. Addison Wesley, Reading, MA, 1997.

[19] IONA Technologies. Orbix 2 Programming Guide. Technical report, IONA, Dublin, Ireland, November 1995.

[20] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP '97*, July 1997.

[21] Thomas Wolff. Transparent Object Distribution and Remote Inheritance. In Amr Zaky and Ted Lewis, editors, *Tools and Environments for Parallel and Distributed Systems*, pages 279–303. Kluwer Academic, 1996.