Improving Concurrency in Common Object Models

Siva Challa and Dennis Kafura

March 25, 1998

Abstract

Most common object models of distributed object systems lack features related to 'concurrency' (the ability to have more than one thread of execution in an object simultaneously). The lack of support for concurrency restricts the development of new components and limits reuse of existing of components that use these advanced features. In this paper, the concurrency features of the Interoperable Common Object Model (ICOM) centered on statically typed object-oriented languages languages, is presented. The ICOM model is an attempt to elevate common object models (with the advanced features of concurrency) closer to the object models of statically typed object-oriented languages. Specific features of the ICOM object model include: atomic objects and guard methods. The *actor* model is used to develop a uniform implementation framework for the ICOM object model in C++ and Modula-3.

1 Introduction

Most common object models [1, 2, 3, 4, 6] including the popular industry standard models — the Common Object Request Broker Architecture (CORBA) [2] and the Component Object Model (COM) [1]— do not support 'concurrency'. Although language support for concurrency is present in several object-oriented languages (see Table 1) and has been proven to be very useful in the application development process, it cannot be used for interoperability of software components because concurrency control cannot be expressed directly in existing common object models but must be obtained by appeal to auxiliary services. The lack of support for concurrency in existing common object models is a serious limitation because it restricts the reuse of software components developed using or requiring this feature.

The Interoperable Common Object Model (ICOM) presented in this paper shows how to include concurrency features in a common object model targetted on statically typed object-oriented languages. Statically typed languages are chosen because they offer both a degree of safety to applications and static typing 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++ [9] and Modula-3 [7]. Though these languages fall within the same family of statically typed languages, they are similar enough to yield

Table 1: Statically Typed Object-Oriented Languages Versus Concurrency

Language	*Concurrency Support*
Ada 95	$\surd$
BETA	$\surd$
C++	*library*
Eiffel	*library*
Java	$\surd$
Modula-3	$\surd$

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 ICOM framework uses the *actor model* of concurrent computation and differs from the other distributed object systems in the following ways:

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.

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.

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.

The ICOM framework is based on a distributed actor model. 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.

In the following sections, atomic objects and synchronization of atomic objects by guard methods is discussed in detail.

2 Atomic 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.

2.1 A Simple Atomic Object

Figure 1 shows the interface, GeneralTax. This interface defines three methods, stateTax, salesTax, and totalTax. The methods salesTax and stateTax have a single output parameter, tax. The implementation of the method totalTax uses the other two methods of the interface to calculate the total tax. The methods 'stateTax' and 'salesTax' can be overridden in a derived class to provide 'inclusion polymorphism'. This type of interface can be used to incorporate general tax rules for all 'states' in the United States of America. In the 'Stores' problem, this object can be used to fix an approximate price for an item to be sold at a retail store.

When the IDL translator is invoked at a remote node that implements the interface, the set of classes that get generated are: GeneralTax, GeneralTaxMetaClass, GeneralTaxMessageClass, and GeneralTaxImplementation

The GeneralTax class is the class available to the application code that wants to use the GeneralTax interface to create instances that support the interface. A GeneralTax object simply delegates all method invocations on it to the implementation code in GeneralTaxImplementation. The GeneralTaxMetaClass is a metaclass of the class GeneralTaxImplementation. The metaclass' purpose is to intercept any messages intended for instances of GeneralTaxImplementation and process them one at a time.

To create and invoke messages on an object that supports the interface GeneralTax, a client application first creates an instance of GeneralTax class and invokes the create method on it.

If a local object is to be created, the GeneralTax class creates an instance of Gener-

```
// GeneralTax defines two methods that can be overridden
// by state tax objects
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);
}
```

Figure 1: IDL Description of General Tax Interface with Atomic Methods

alTaxMetaClass. During the creation of the instance of GeneralTaxMetaClass the method objects corresponding to each method(stateTax, salesTax, totalTax, and create are registered with the *methodTable* which is a state variable in the GeneralTaxMetaClass instance. Among the registered method objects, the object corresponding to the create method has a method called new that creates an instance of the GeneralTaxImplementation class. At this point the GeneralTax class instance has access to the address of the active object representing the GeneralTaxImplementation class. The other registered method objects have a method called invoke that invokes the corresponding actual method of a GeneralTaxImplementation instance created using the new method of the registered method object of create method.

If a remote object is to be created, the GeneralTax class creates a 'constructor' message object using the GeneralTaxMessageClass class, and hands it to the underlying communication framework. The framework encodes the message and sends it to the site where the object is to be created. At the receiving site, the message will be decoded and handed to an instance of GeneralTaxMetaClass that is automatically created. The instance of GeneralTaxMetaClass creates an instance of GeneralTaxImplementation and it passes the network address of the newly created object to the client application object at the requesting site.

An invocation of the method, totalTax is shown in Figure 3. In this figure, it is assumed that two distributed nodes, local and remote exist in a distributed system supported by the ICOM framework. A client application exists at the local node and the implementation of the GeneralTax interface is at the remote node. The client application developer must first run the IDL translator on the GeneralTax interface at the local node. The application code must be linked with the IDL translated code. The translator generates a proxy class, GeneralTax at the local node. The client application can use this proxy class to create instances of GeneralTax objects at the remote node and invoke methods on it.

To invoke a method on the instance of GeneralTaxImplementation, the application invokes the method of the GeneralTax instance. The GeneralTax instance creates a message and sends it to the active object at the remote node. If multiple invocations are involved,

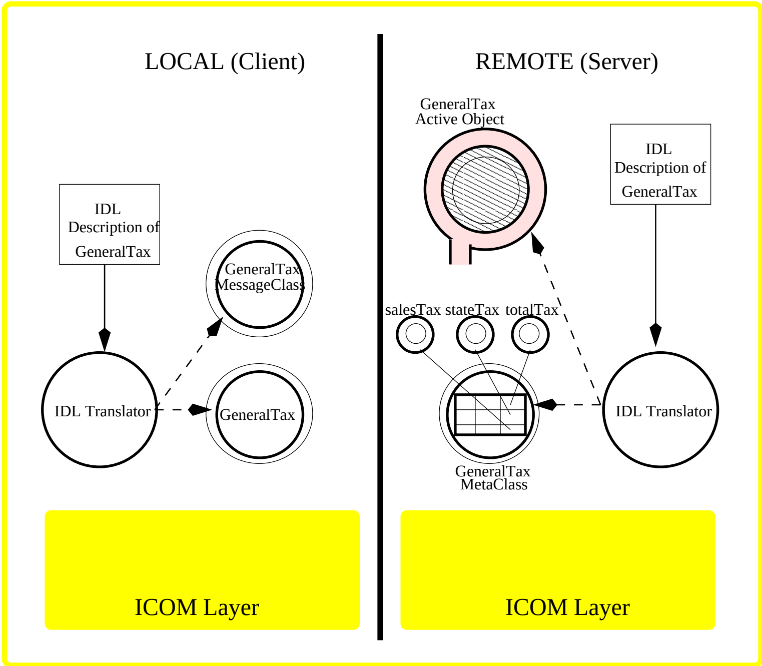

Figure 2: IDL Translation of General Tax Interface with Atomic Methods

messages get queued at the active object queue. After a method completes its execution in GeneralTaxImplementation, the active object retrieves a pending message from its queue and hands it to the GeneralTaxImplementation instance. Here the active object maintains a lock, and blocks the incoming messages and handles one message at a time as required by the atomicity property.

An invocation of the stateTax method (arrow labeled 1) and an invocation of the salesTax method (arrow labeled 'a') on the on the GeneralTax proxy instance results in a series of method calls. The sequence can be explained as follows:

At the local node

the GeneralTax proxy object communicates with the GeneralTaxMessageClass and creates two message objects corresponding to the methods, stateTax and salesTax and forwards them to the remote site one after the other (arrows labeled 2, b, 3, and c). The ICOM layer at the local node encodes the message objects before sending them over to the remote node.

At the remote node

the ICOM layer at the remote node receives the messages, partially decodes each message one at a time, to get the method name, class name, and a network address of the

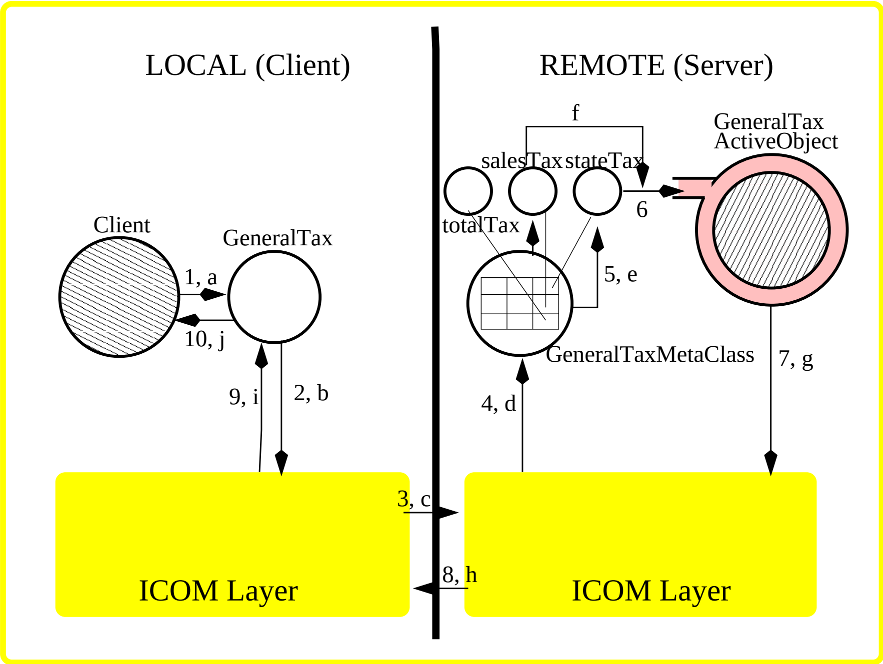

Figure 3: Dynamic Method Invocation on an Instance of General Tax Interface with Atomic Methods

active object of the GeneralTax interface. The ICOM layer then forwards the message objects to the instance of GeneralTaxMetaClass(arrow labeled 4,d). The metaclass instance looks up in its method table for a server method object, and passes the message objects to the server method objects stateTax and salesTax respectively. The method objects decode the arguments from the messages, and try to invoke the stateTax and salesTax methods of the active object of GeneralTax (arrows labeled f and 6). The active object accepts one method invocation at a time and sends the results back to the local node using the output parameters (arrows labeled 7,g, 8,h, 9, i, 10, and j).

Thus, the ICOM framework supports atomic objects that are active.

2.2 Atomic Objects involving Remote Inheritance

This sections discusses truly distributed objects that involve remote inheritance of atomic objects and provides an example demonstrating the use of inheritance in atomic objects.

2.2.1 Truly Distributed Objects

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 [1, 2, 3] support *interface inheritance* where only the specification is inherited from a parent 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.

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.

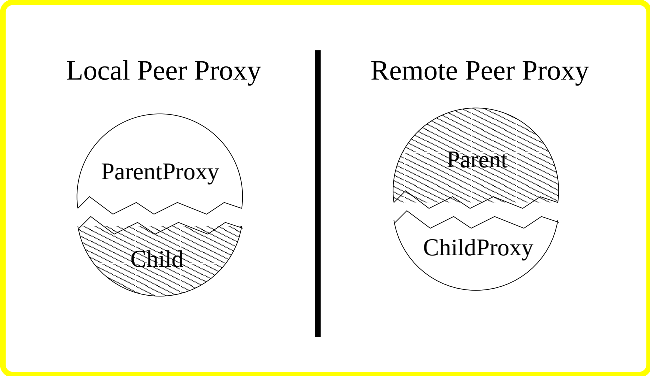

Figure 4: A Simplified Representation of a Truly Distributed Object

A simplified representation of the truly distributed object is shown in Figure 4. 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.

Truly distributed objects differ from objects in the common object models in two ways. First, implementation inheritance is not supported in most of the existing common object models[1]. Second, most of the existing distributed object systems use synchronous method invocation semantics. An application invoking a remote method must wait for a return value or an exception in these systems. There is no provision for concurrent method invocations on different components of an object. The ICOM framework supports implementation inheritance by providing location transparancy, and follows asynchronous method invocation semantics and guarantees atomicity of method executions even when multiple method invocations on different components of an object are involved.

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 a method defined at the superclass. So, an invocation of a 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. 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.

[1]There are two systems that partially or completely support implementation inheritance. The ORBIX system from IONA Technologies [11] supports partial implementation inheritance where classes in a class hierarchy can exist on multiple sites, but overriding of methods in derived classes is not allowed. The Heron system from the Free University of Berlin [12] supports implementation inheritance following the semantics of the Eiffel language.

2.2.2 Example

Inheritance involving a GeneralTax interface and a VirginiaTax interface are used to describe the working mechanism of translation and method invocation in the ICOM framework. Figure 5 shows two interfaces, GeneralTax and VirginiaTax. The GeneralTax interface defines two methods, salesTax and stateTax and totalTax. All methods have a single output parameter, tax. The VirginiaTax inherits from GeneralTax and overrides the two methods, salesTax and stateTax. The two interfaces are used to calculate the state tax, sales tax, and total tax for the 'state' of Virginia.

```
// 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 : GeneralTax {
    // returns the sales tax through 'tax'
     salesTax(out double tax);
    // returns the state tax through 'tax'
     stateTax(out double tax);
  }
```

Figure 5: IDL Description of Virginia Tax Interface with Atomic Methods involving Inheritance

In this example it is assumed that two nodes, local and remote, exist in the distributed system supported by the ICOM framework. The remote node contains the implementation of the GeneralTax interface and the local contains the implementation of VirginiaTax interface. For simplicity, it is assumed that the the GeneralTax implementation has already been done at the remote node and the VirginiaTax implementation is to be done at the local node.

An invocation of the distributed IDL translator at the local node on the IDL descriptions shown in Figure 5 results in the generation of classes at the local node. The IDL translator realizes the existence of implementation of the GeneralTax implementation at the remote node. It sends a 'compile' message to the distributed IDL translator at the remote node, by passing the IDL description. Both IDL translators can run in parallel.

The classes for GeneralTax are assumed to be available at the remote node are: GeneralTax, GeneralTaxMetaClass, GeneralTaxMessageClass, and GeneralTaxImplementation. Of these classes, the GeneralTaxImplementation class is a skeleton class that must be filled in with method implementations by a developer.

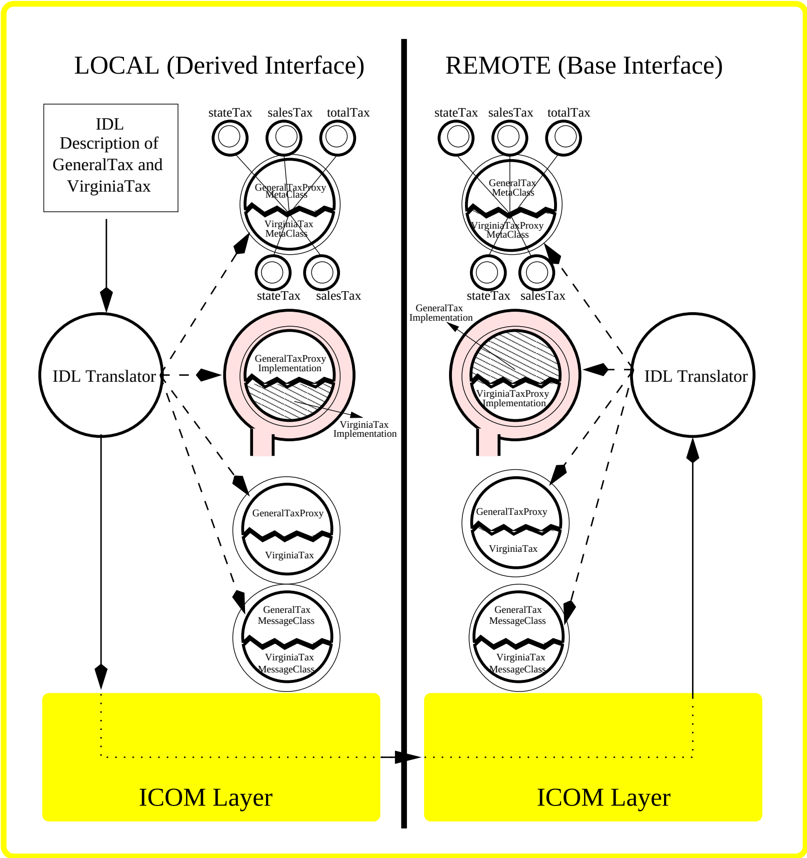

Figure 6: IDL Translation of Virginia Tax Interface with Atomic Methods involving Inheritance

The classes for VirginiaTax generated at the remote node are: VirginiaTax, VirginiaTaxProxyMetaClass, VirginiaTaxMessageClass, and VirginiaTaxProxyImplementation. The VirginiaTax class is a proxy class that inherits from GeneralTax proxy class. An instance of VirginiaTax classes at the remote node contains the network addresses of the peer proxy objects of the VirginiaTax interface at the local node and the remote node. The VirginiaTaxProxyMetaClass inherits from GeneralTaxMetaClass, the VirginiaTaxMessageClass inherits from GeneralTaxMessageClass, and the VirginiaTaxProxyImplementation class inherits from GeneralTaxImplementation. It should be noted that the implementation of all classes related to the VirginiaTax interface at the remote node are automatically generated by the IDL translator.

At the local node, the classes generated for the GeneralTax interface are: GeneralTax, GeneralTaxProxyMetaClass, GeneralTaxMessageClass, and GeneralTaxProxyImplementation. The implementation of all these classes is generated by the IDL translator.

The classes generated for VirginiaTax interface are: VirginiaTax, VirginiaTaxMetaClass, VirginiaTaxMessageClass, and VirginiaTaxImplementation

The VirginiaTax class inherits from GeneralTax class, VirginiaTaxMetaClass inherits from GeneralTaxMetaClass, VirginiaTaxMesageClass inherits from GeneralTaxMessageClass, and finally VirginiaTaxImplementation inherits from GeneralTaxImplementation. It must be noted that the VirginiaTaxImplementation class is a skeleton class and the application developer must implement the methods of this class. The implementation for the remaining classes is generated by the IDL translator.

Consider a truly distributed object from the above example, an instance of the GeneralTaxImplementation, on the remote node, and VirginiaTaxImplementation is on the local node. To guarantee the atomicity property for message processing, a shared lock can be maintained at both nodes by the active objects representing the peer proxies. Any distributed shared locking scheme can be used to maintain the locks. The algorithm used in the ICOM framework is 'Suzuki and Kasami' algorithm for sharing distributed locks. The algorithm is described in Section 6.2.4.

It is assumed that it is possible to invoke methods on both peers. Consider the method invocations salesTax and totalTax at the remote node and the local node respectively. Initially, the peer at local is the owner because we assume that the application node creates an instance of VirginiaTax at local and invokes the create() method on it. Since VirginiaTax is implemented on local, a proxy for it is created at remote. This proxy is the instance of the class VirginiaTaxProxyImplementation. Similarly, a proxy for GeneralTax class is created at local. This instance in turn creates a corresponding instance of GeneralTaxImplementation on remote by sending a create message to it.

Two invocations of the method stateTax on an instance of VirginiaTaxProxy at the remote node are shown in Figure 7(arrows labeled 1 and 'a'). The processing of the second invocation waits until the processing of the first invocation is completed. The sequence of method calls are described as follows:

At the remote node

the VirginiaTax instance creates a message object for each of the two method invocations and passes them to the metaclass instance of VirginiaTaxProxyMetaClass(arrows labeled 2 and b). The metaclass instance looks up in its method table for a method object and passes the messages to the method object stateTax (arrows labeled 3 and 'c'). The method object copies the arguments from the message and sends a message to the peer proxy on the remote node (arrow labeled 4). It repeats the process for the second message (arrow labeled d). The active object, VirginiaTax PeerProxy, starts the processing of the first message and blocks the second message. This active object sends an invocation to the peer proxy at the local node (arrows labeled 5 and 6).

At the local node

the ICOM layer receives the encoded message and partially decodes it to get the network address of the peer proxy at the local node, method name and the class

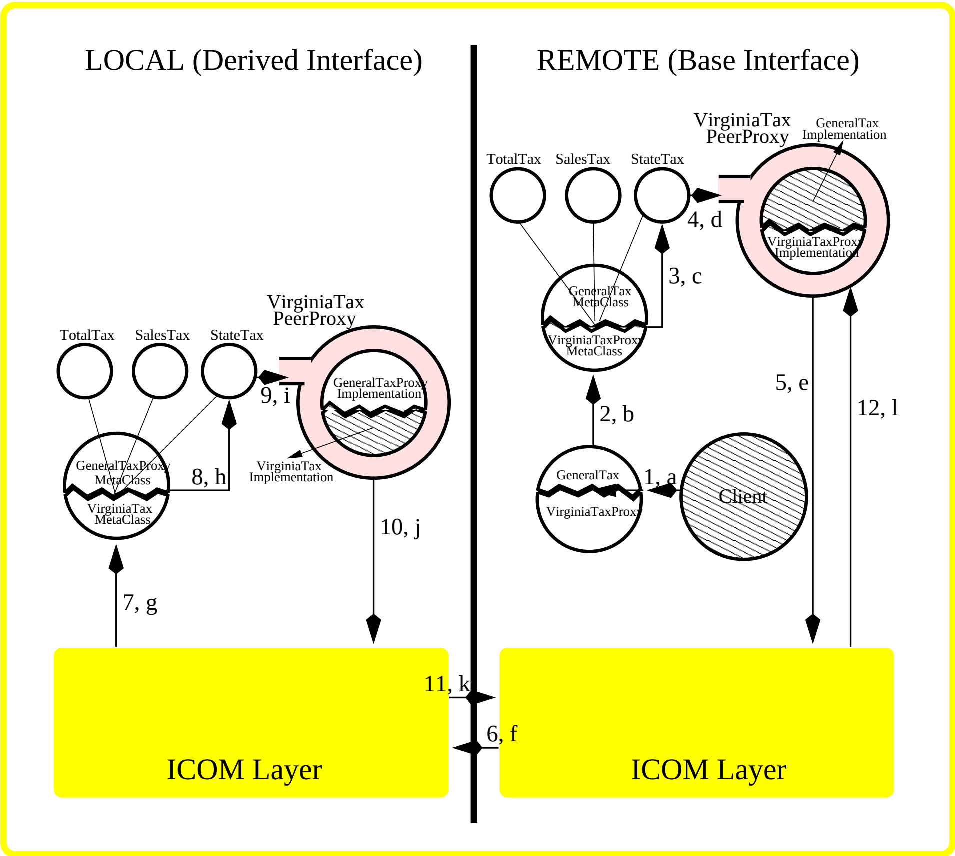

Figure 7: Dynamic Method Invocation on an Instance of Virginia Tax Interface involving Inheritance and Atomicity

name of the message. The ICOM layer then forwards the message to the metaclass instance VirginiaTaxMetaClass (arrow labeled 7). The metaclass instance looks up in the method table for a server method object and passes the message to the server method object stateTax (arrow labeled 8). The method object decodes the arguments and passes the message to the peer proxy, VirginiaTax PeerProxy (arrow labeled 9). The peer proxy executes the method and sends the result through the output parameter tax to the remote node (arrows labeled 10, 11, and 12). The peer proxy sends a *peerUnblock* (discussed in the next section) message to the peer proxy at the remote node (arrow labeled 13).

At the remote node

the ICOM layer receives the messages and passes them to the peer proxy. The peer proxy after receiving the *peerUnblock* message, resets the lock and allows the next message to be processed.

The same sequence of method calls is repeated for the second message. Thus, the ICOM framework supports concurrency that allows method invocation on any component of a distributed object. At the same time the framework assures atomicity.

Potentially, any distributed shared lock algorithm can be used for implementing a shared lock in a distributed environment. In the ICOM framework Suzuki and Kasami algorithm [10] is used for implementing atomic locks. A brief description of the algorithm is given below.

2.3 Suzuki and Kasami Algorithm

The Suzuki and Kasami algorithm was originally given to provide distributed mutual exclusion to critical sections among N nodes in a distributed network of computers. The computers (nodes in the network) are assumed to not share memory. They communicate with one another only through message passing. The mutual exclusion requirement satisfied in the algorithm is that at any time, at most one node may stay in its critical section.

The algorithm provides a data structure called 'Token' which is transferred among the nodes. Any node that receives the 'Token' can enter its critical section repeatedly until it sends the 'Token' to some other node. A node requesting the 'Token' sends a 'Request' message to all other nodes. A 'Request' message from a node j has the form Request(j,s) where j is the identifier of the requesting node and s is the 'sequence number' which indicates that the node is requesting its (n+1)st critical section invocation. Each node maintains an array, RN, of size n to record the largest sequence number ever received from each of the other nodes.

Initially node 1 has the 'Token'. If Request(j,s) is received by node i, then node i updates its RN array by RN[i] = max(RN[j],s), where max returns the maximum of the two arguments. The 'Token' is a message that has the form token(Queue, LN) where Queue is the queue of requesting nodes and LN is an array of size n such that LN(j) is the sequence number of the request of node j most recently.

When a node i completes executing its critical section, the array LN contained in the last message with 'Token' received by i, is updated by LN[i] = RN[j] to indicate that current request has been granted. Next, all node identifiers j, such that RN[i] = LN[j]+1 (i.e., node j is requesting for the 'Token'), is appended to Queue provided that j is not already in Queue. When these updates are complete and if Queue is empty then node i retains the 'Token' until a requesting node is found by the arrival of a Request message.

This algorithm is deadlock free and starvation free. Critical sections are granted in the First Come First Served manner. It requires, at most, n message exchanges per mutual exclusion invocation; (N-1) 'Request' messages and one 'Token' message or no message at all if the node having the 'Token' happens to be the node requesting.

2.4 Shared Lock Abstraction

The abstraction of the shared lock of truly distributed objects in terms of Suzuki and Kasami algorithm, is realized as follows. The nodes in the algorithm are mapped to peer components of a distributed object. A 'Token' is treated as the ownership of a peer component. Any peer component with the ownership can potentially start the execution of an external method invocation. A 'Request' message is mapped to the OwnershipRequestMessage. The 'Token' is mapped to the message OwnershipRelease that contains the addresses of peer components requesting ownership. The OwnershipRelease message also contains a second array of size N which holds the information about whether a particular peer is requesting ownership because of a guard method or not. This is required to avoid the transfer of the OwnershipRelease message among the peer components when all of the peer components are waiting on guard methods. Transfer of ownership is ensured by the message NewOwner. The NewOwner message is used to inform other nodes about the new owner and also to act as an acknowledgment between the owner peer component and other peer components.

3 Guard Methods

It is often the case that real world objects exhibit different behaviors in different situations. A classic example from the literature is that of a 'bounded buffer'. A bounded buffer cannot accept a get method when the buffer is empty. The only method accepted in this state is the put method. Similarly, the buffer cannot accept a put method when the buffer is full. It accepts only a get method at that time. But, when the buffer is not full and is not empty it accepts both put and get messages.

In the ICOM framework, guard methods [8] are used as a way of providing synchronization to objects. The guard method approach, while not eliminate the anomaly, attempts to minimize the amount of rewriting. The approach uses two types of inheritance, one for inheriting synchronization constraints and the other for normal inheritance. It is possible to get confused with the two types of inheritance [5]. In the ICOM framework only one type of inheritance scheme is considered for both synchronization and normal inheritance.

The ICOM model views guard methods as regular methods that have one boolean output parameter and any number of input parameters. By definition, they do not affect the state of an object. The restriction on the input parameters is that they must correspond to the input parameters of the method guarded by the guard method. Guard methods always evaluate to either true or false. The key word *guardedBy* is used to represent a guard method that guards another method in an interface. Consider an instance of an object supporting an interface with a method guarded by another method. When this object receives an invocation of the guarded method, the guard method will be evaluated first. If the guard method returns true via its output parameter, then the guarded method is executed. Otherwise, the original message will enter the queue and the guard method will be evaluated again at a later point of time. The evaluation takes place when the state of the object is modified by another method of the same object.

3.1 An Atomic Object with a Guard Method

Figure 8 shows the Manufacturer interface that contains a method startShipment with four input parameters, brandName, itemNumber, itemCount, and storeNumber. This method is used by a client application requiring a product to be shipped to a store. The client application and the Manufacturer implementation can exist in a distributed environment. The method startShipment is guarded by the method suppliesAvailable. The relationship between a method and its guard is represented by the keyword *guardedBy*. The guard method, suppliesAvailable takes three input parameters, brandName, itemNumber, and itemCount, and an output parameter, okay. The manufacturer cannot start shipment unless the product is available. The method suppliesAvailable will be evaluated first when an invocation on the method startShipment is received by the active object representing Manufacturer interface. The method startShipment will be executed only when there are supplies available. The availability is ensured by the guard method suppliesAvailable

```
interface Manufacturer {

  // this method is guarded by suppliesAvailable method
  startShipment(in string brandName,
                in long  itemNumber,
                in long  itemCount,
                in long storeNumber) guardedBy

    suppliesAvailable(in string brandName,
                      in long  itemNumber,
                      in long  itemCount,
                      out boolean okay);

}
```

Figure 8: IDL Description of Manufacturer Interface with a Guard Method

An IDL translation of the Manufacturer interface is shown in Figure 9. It is assumed that the implementation of the interface is to be done at a remote node and the client application at a local node. IDL translators can run independently at both nodes, because there is no remote inheritance involved in this example.

At the remote node, the IDL translator generates four classes: ManufacturerMetaClass, ManufacturerImplementation, ManufacturerMessageClass, and Manufacturer proxy class. The metaclass ManufacturerMetaClass contains a method table representing the two methods, startShipment and suppliesAvailable. Two server method objects are attached to the metaclass and they can decode the method arguments of the messages represented by them. The ManufacturerImplementation class is a skeleton class and its implementation must be written in a native language by a developer. The ManufacturerMessageClass is used to generate message objects for the two methods of the interface. The Manufacturer class is a proxy class for the Manufacturer interface. It is mainly used in the client applications, but the

ManufacturerImplementation may use it to make invocations on other Manufacturer active objects.

The implementation of the server method objects that represent guarded methods is different from other server method objects. When an invocation of the guarded method is to be made on the active object representing the Manufacturer interface, the guard method will be invoked first. If the guard method returns true, then the guarded method will be executed. Otherwise, the message enters the queue of the active object for later reevaluation of the guard method.

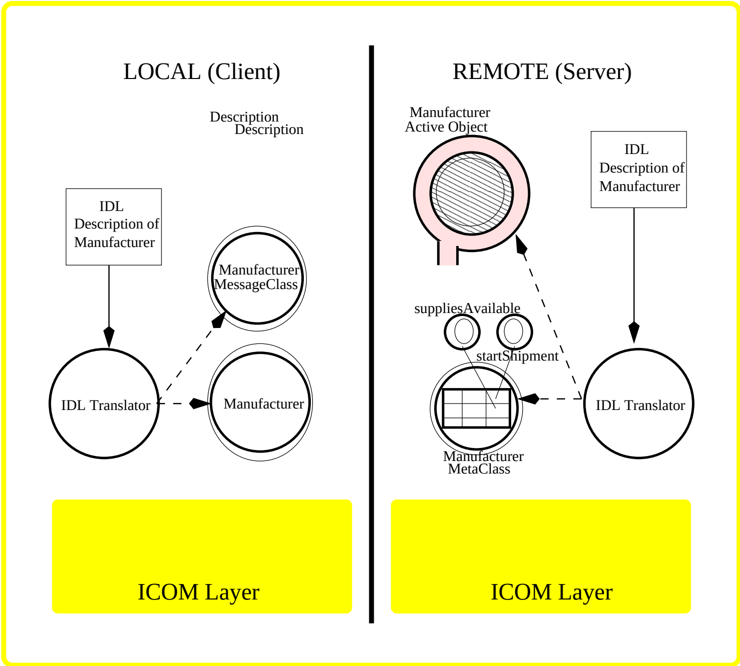

Figure 9: IDL Translation of Manufacturer Interface with a Guard Method

At the local node, where the client application is to be implemented, the IDL translator generates two classes: Manufacturer proxy class and ManufacturerMessageClass. The proxy class is used in the client application to create and invoke messages on the active object at the remote node. It uses the ManufacturerMessageClass to generate message objects that can be sent to the remote node. The client application can be a financial agency that maintains an information processing service center that represents a chain of stores. This center can request for the shipment of a product whose inventory is low at a clothes store. This is done by the invocation of the startShipment method on the manufacturer object.

Figure 10 shows a method invocation of startShipment method, on an active object representing the Manufacturer interface (arrow labeled 1). The sequence of method calls can be explained as follows:

At the local node

The instance of Manufacturer proxy class generates a message object using the instance of ManufacturerMessageClass and sends it to the remote node (arrows labeled 2 and 3).

At the remote node

the ICOM layer at the remote node receives the message and partially decodes it to get the network address of the Manufacturer active object, the method name and class name. It then passes the message object to the instance of ManufacturerMetaClass (arrow labeled 4). The metaclass instance looks up in the method table for a server method object, and passes the message object to the server method object (arrow labeled 5), startShipment. The server method object decodes the arguments and realizes that this method is guarded by the method suppliesAvailable. The server method object first tries to invoke the guard method on the active object (arrow labeled 6). If the guard method returns true via its output parameter, then it invokes the startShipment method of the active object (arrow labeled 7). If the guard method returns false via its output parameter, then the message enters the queue and the guard method will be re-evaluated when the state of the active object changes.

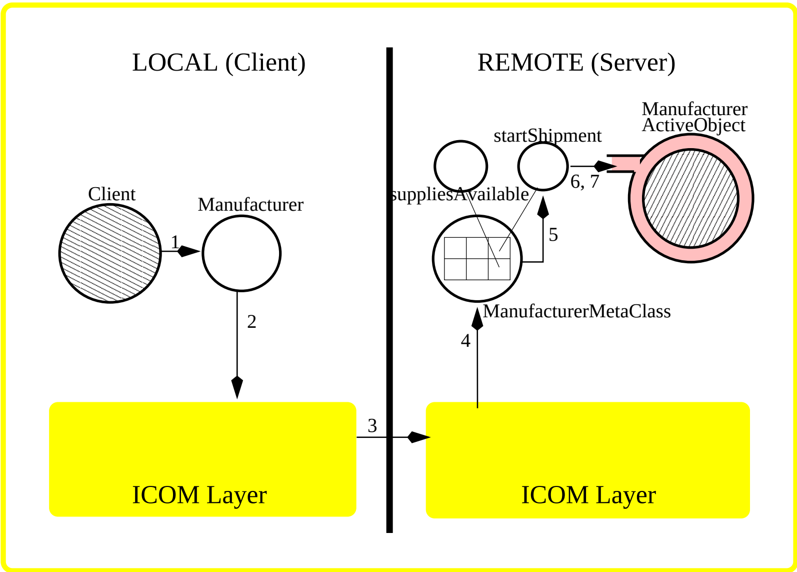

Figure 10: Dynamic Method Invocation on an Instance of Manufacturer Interface with a Guard Method

Thus, in the ICOM framework, objects can be synchronized using guard methods.

3.2 Synchronized Objects involving Inheritance

In this subsection, a simple example of a method invocation on a truly distributed object with a guarded method is described. The 'Stores' problem is used to define the interface shown in Figure 11. There are two interfaces, ClothesStore and LocalClothesStore. The second interface inherits from the first interface. The ClothesStore interface defines a method putItemOnSale which is guarded by another method isInventoryOkay. The method putItemOnSale takes an input parameter, brandName, and an output parameter, salePercentage. The method isInventoryOkay takes one input parameter, brandName and one output parameter, okay.

In this example, it can be assumed that the ClothesStore interface is implemented at the CIPS center and the LocalClothesStore is implemented at a distributed location. An application using the instance of ClothesStore can spawn a thread and send a request to the CIPS center to put an item on sale if the inventory of the item remains high over a long period of time. The CIPS center can make a decision based on the inventory status and the time of the year, and decide whether to put the item on sale or postpone the decision.

```
interface ClothesStore {
   putItemOnSale(in string brandName,
                 out long salePercentage) guardedBy

      isInventoryOkay(in string brandName,
                      out boolean okay);,
   ⋮
}

interface LocalClothesStore : ClothesStore {
   updateSales(in long itemNumber,
               in long itemCount,
               in double pricePerItem);
   ⋮
}
```

Figure 11: IDL Description of Local Clothes Store Distributed Object with a Guard Method

In this example it is assumed that a client application and the implementation of LocalClothesStore are at a local node, and the implementation of ClothesStore interface is at a remote node. For simplicity it is assumed that the implementation of the ClothesStore has already been done. When an IDL translator is invoked on the IDL description shown in Figure 11, it realizes the existence of remote inheritance and issues a 'compile' message to the IDL translator at the remote node.

The local IDL translator generates four classes corresponding to the ClothesStore interface. They are: ClothesStore, ClothesStoreProxyMetaClass, ClothesStoreProxyImplemetation, and ClothesStoreMessageClass. All these classes represent the corresponding classes at the remote node. The implementation of all these classes is generated by the IDL translator. The classes generated for the LocalClothesStore interface are: LocalClothesStore LocalClothesStoreMetaClass LocalClothesStoreImplemetation, and LocalClothesStoreMessageClass. The LocalClothesStore class inherits from ClothesStore class. The LocalClothesStoreMetaClass inherits from ClothesStoreProxyMetaClass. The LocalClothesStoreImplemetation class inherits from ClothesStoreProxyImplemetation class. It should be noted that the implementation generated for the class LocalClothesStoreImplemetation is a skeleton implementation and the actual coding for the methods must be done by the application developer. The LocalClothesStoreMessageClass inherits from the class ClothesStoreMessageClass.

At the remote node, the IDL translator generates four classes corresponding to the LocalClothesStore interface. They are: LocalClothesStore, LocalClothesStoreProxyMetaClass, LocalClothesStoreProxyImplemetation, and LocalClothesStoreMessageClass. The LocalClothesStore classes inherit from the corresponding classes of the ClothesStore interface that were already generated when the ClothesStore is implemented earlier.

In this example the stores and central information processing service are geographically distributed. A local store client application can spawn a thread and invoke the method putItemOnSale of the store object. The thread blocks and waits for a response in the salePercentage output parameter. The invocation (arrow labeled 1) is shown in Figure 13. The sequence of method calls is explained as follows:

At the local node
: the instance of LocalClothesStore proxy class generates a message object for the method putItemOnSale and passes it to the metaclass instance of LocalClotheStoreMetaClass (arrow labeled 2). The metaclass instance looks up in the method table for a server method object and passes the message object to the server method object of putItemOnSale (arrow labeled 3). The server method object copies the arguments and invokes the method on the active object implementation of LocalClothesStoreImplementation. Since the actual implementation of the method exists at the remote node, the active object sends an invocation to the remote node (arrows labeled 5 and 6). The encoded message is sent to the ICOM layer at the remote node.

At the remote node
: the ICOM layer at the remote node receives the message and partially decodes it to get the network address of the active object of ClothesStoreImplementation, method name and class name of the message object . The ICOM layer then passes the message object to the ClothesStoreProxyMetaClass instance (arrow labeled 7). The metaclass instance does a look up in its method table for a server method object and passes the message to the server method object, putItemOnSale. The method object decodes the message arguments. Before invoking it realizes the existence of the guard method, isInventoryOkay, and invokes it first (arrow labeled 9). If it evaluates to true, then the method object invokes the putItemOnSale method on the active object (arrow labeled 10). The result is sent to the local node with the output parameter, salePercentage

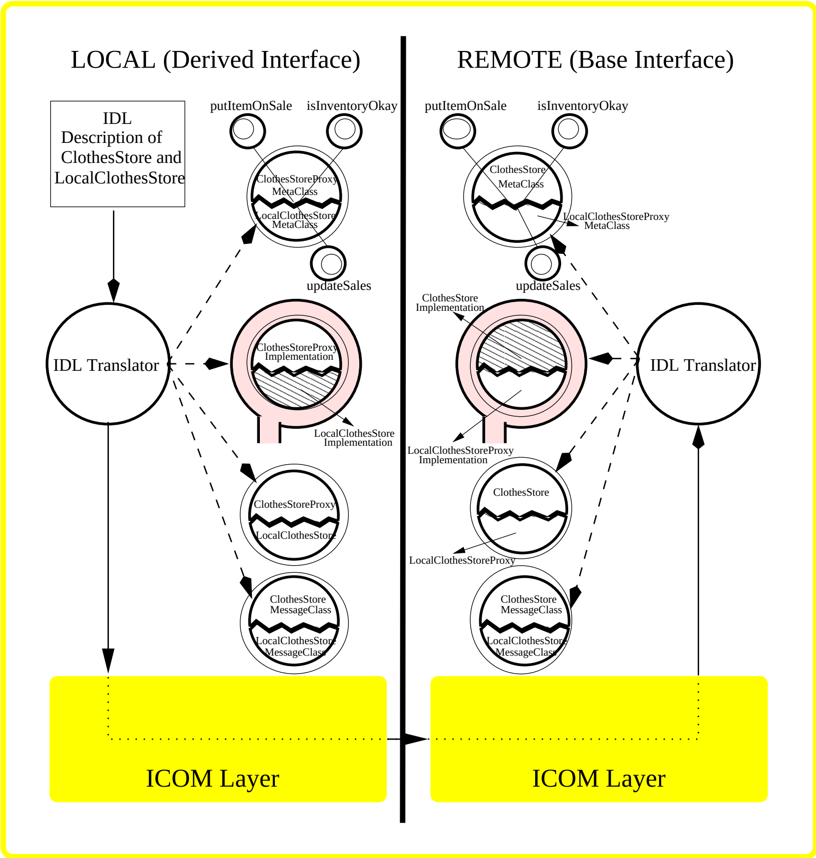

Figure 12: IDL Translation of the Local Clothes Store Distributed Object with a Guard Method

(arrows labeled 11,12 and 13).

Thus, in the ICOM framework, guard methods can be used for synchronizing method invocations on object. The guard methods can be inherited and overridden just like any other methods.

3.3 General Limitations

The current implementation framework of ICOM supports atomic object only. It does not allow a full set or subset of methods in an interface to be concurrent. To provide the flexibility of having some methods as atomic and some methods as synchronized methods, method level locks can be used. The subset of methods requiring atomicity among themselves can have a lock associated with their group. The inheritance of a class with some of its methods

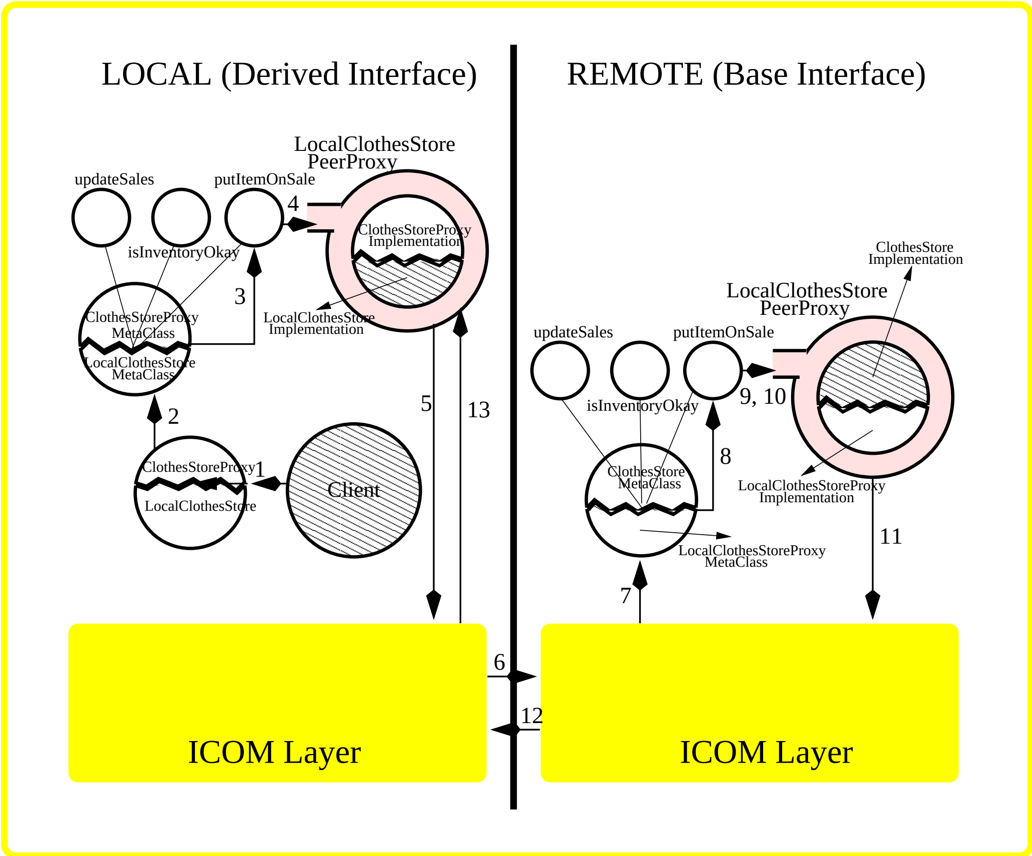

Figure 13: Dynamic Method Invocation on the Local Clothes Store Distributed Object with a Guard Method

concurrent needs to be investigated further.

4 Summary

This paper introduced the features of the ICOM object model. 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 atomicity property and the feature of guard methods are discussed in detail. Examples are provided to explain the features of the ICOM framework.

References

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

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

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

[4] 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.

[5] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. The MIT Press, Cambridge, MA, 1993.

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

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

[8] Etsuya Shibayama. Reuse of Concurrent Object Descriptions. In *Proceedings of TOOLS 3, Sydney, Australiz*, pages 254–266, November 1990.

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

[10] Ichiro Suzuki and Tadao Kasami. A Distributed Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, November 1985.

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

[12] 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.