Improving Polymorphism in Common Object Models

Siva Challa and Dennis Kafura

March 25, 1998

Abstract

Most common object models of distributed object systems lack support for 'polymorphism' (an abstraction mechanism that represents a quality or state of being able to assume different forms). This lack of support 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), which focuses on a subset of object-oriented languages, specifically statically typed languages, is presented. The ICOM model is an attempt to elevate common object models closer to the object models of statically typed object-oriented languages by including support for polymorphism. Specific features of the ICOM object model include: remote inheritance, method overloading, and parameterized types.

1 Introduction

While many common object models [2, 3, 4, 5, 7] have been proposed to support interoperability of software components developed in multiple languages, most of the models, including the popular industry standard models — the Common Object Request Broker Architecture (CORBA) [3] and the Component Object Model (COM) [2]— do not support advanced features of 'polymorphism'. Although the features of polymorphism 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.

The Interoperable Common Object Model (ICOM) presented in this paper shows how to include advanced features of polymorphism in a common object model limited to statically typed object-oriented languages even if these languages do not support the features or if the languages supports only a variation of the features. 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++ [11]

Table 1: Statically Typed Object-Oriented Languages Versus Polymorphism

Language	*Polymorphism Features*		
	Method Overriding	Method Overloading	Generic Types
Ada 95	$\surd$	$\surd$	$\surd$
BETA	$\surd$	*weak*	$\surd$
C++	$\surd$	$\surd$	$\surd$
Eiffel	$\surd$	$\surd$	$\surd$
Java	$\surd$	$\surd$	*proposed [1, 8, 13]*
Modula-3	$\surd$	*weak*	$\surd$

and Modula-3 [10]. 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).

Two general categories of polymorphism have been identified - ad hoc polymorphism and universal polymorphism [14]. Ad hoc polymorphism represents 'type coercions', 'operator overloading', and 'method overloading'. Type coercions and operator overloading are not discussed any further because they involve language specific details which could not be addressed at a common object model level[1]. The advanced features of universal polymorphism are 'inclusion polymorphism' that involves inheritance of abstract methods (or overriding of methods), and 'parameterized types'. The ICOM model supports these aspects of polymorphism in the following ways:

Method Overriding

The ICOM framework supports both the interface inheritance and the method overriding. The interface inheritance forces the implementor of a derived interface to provide implementation for the base interface too. Using method overriding, 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

[1] This is because one of the design considerations of this research is that language semantics and compilers should not be changed

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.

Other of the ICOM object model are discussed in [6]. The following sections describe the method overriding, method overloading, and parameterized types in ICOM object model.

2 Polymorphism via Method Overriding

Polymorphism provides higher level abstraction mechanisms in object-oriented programming languages. It enhances domain understanding by representing in an abstract entity the common characteristics of domain-specific entities. Polymorphism represents the quality or state of being able to assume different forms. A class is polymorphic if a method invocation of the class results in the execution of a method defined in a subclass of the class. A type is polymorphic if a variable of the type can change its type during the life-time of a program. Similarly, a polymorphic method can accept different types of arguments and a polymorphic operator can operate on multiple types of operands.

The following sub-sections describe polymorphism that is achieved through inheritance which may involve classes implemented at multiple locations.

Method overriding, 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, 3, 4] 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.

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 [12]. 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.

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 abstract methods in the superclass to be overridden in a subclass. When overridden abstract 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.

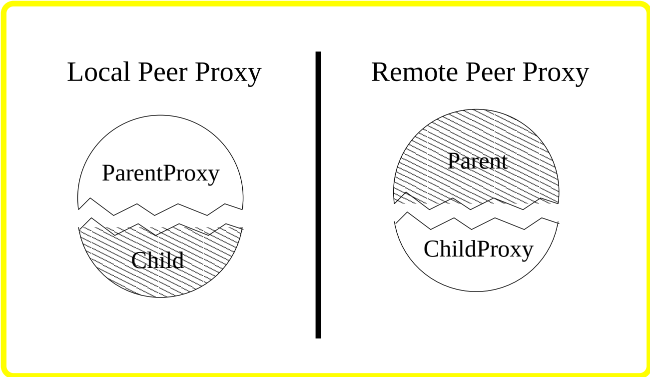

Figure 1: A Simplified Representation of a Truly Distributed Object

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.

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[2]. 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

[2]There are two systems that partially or completely support implementation inheritance. The ORBIX system from IONA Technologies [12] 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 [15] supports implementation inheritance following the semantics of the Eiffel language.

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 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. 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.

3 Method Overloading

Method overloading is an aspect of ad hoc polymorphism. An interface with overloaded methods defines more than one method with the same name but different signatures. That is, the types and/or count of the parameters is different in each overloaded method. In most statically typed object-oriented languages, method overloading is achieved by defining multiple methods with the same name but different signatures. An alternative way of achieving method overloading is to inherit from a base class and define methods (in the derived class) whose names match with those in the base class. But, the signatures of the newly defined overloaded methods in the derived class must be different.

Method overloading is different from method overriding. In overriding, a base class method signature is same as the overriding method in the derived class. An overriding method hides the overridden method. An explicit super class syntax is needed to invoke the method defined in the superclass. In method overloading, this the base class method is not hidden. All methods are visible to a client. The method binding for a method invocation of an overloaded method can be done at compile time.

The ICOM framework supports method overloading feature in its distributed object model. Overloaded methods can be described in an IDL interface. The IDL translator of ICOM generates metaclasses that are capable of identifying the correct method to invoke, based on the decoded arguments information from a message.

A major design criterion in the ICOM framework is that a distributed object model should appear as close to the native language as possible. Some languages such as BETA and Modula-3 do not have the method overloading feature in their object models. But they have other features using which method overloading can be added to the language, but at the expense of 'name mangling' schemes being visible to the application developers. Also, 'name mangling' looks unnatural to a developer. So, even though it is possible to add the method overloading feature to these two languages, the ICOM framework chooses not to translate the feature in these languages. Instead, ICOM provides a way for software components in these languages can access software components in other languages that support the method overloading feature. So, an application developer can use the feature from other languages, but cannot write code using this feature. Thus, the ICOM model provides one-way interoperability of the common object model features in languages that do not support the common object model features.

3.1 Method Overloading in a Localized Object

Figure 2 shows a ClothesStore interface that contains three methods. The first two methods have the same name, priceOfItem, but different signatures. The first method takes an itemNumber as its first argument, whereas the second method takes an itemLabel as its first arguments. These two parameters differ in their types. The method priceOfItem is used by a client to find out the price of an item. The third method of the interface is putItemOnSale. It is used to find the sale percentage on an item whose inventory remains the same for a long period of time, so that the item can be sold quickly.

```
interface ClothesStore {

    // returns price of an item through 'price'
    priceOfItem(in long itemNumber,
                out double price);
    // returns price of an item through 'price'
    priceOfItem(in string itemLabel,
                out double price);
    // returns the sale percentage through 'salePercentage'
    putItemOnSale(in long itemNumber,
                  out double salePercentage);
}
```

Figure 2: IDL Description of Clothes Store Interface Containing Overloaded Methods

During the IDL translation, each overloaded method will be numbered with a natural number as shown in Figure 3. If there are three methods with the same method name in an interface, they will be numbered as methods 1, 2 and 3. Both the client side and the server side agree upon the number assigned to each method. The IDL translator allocates method numbers in the order the methods appear in the IDL description.

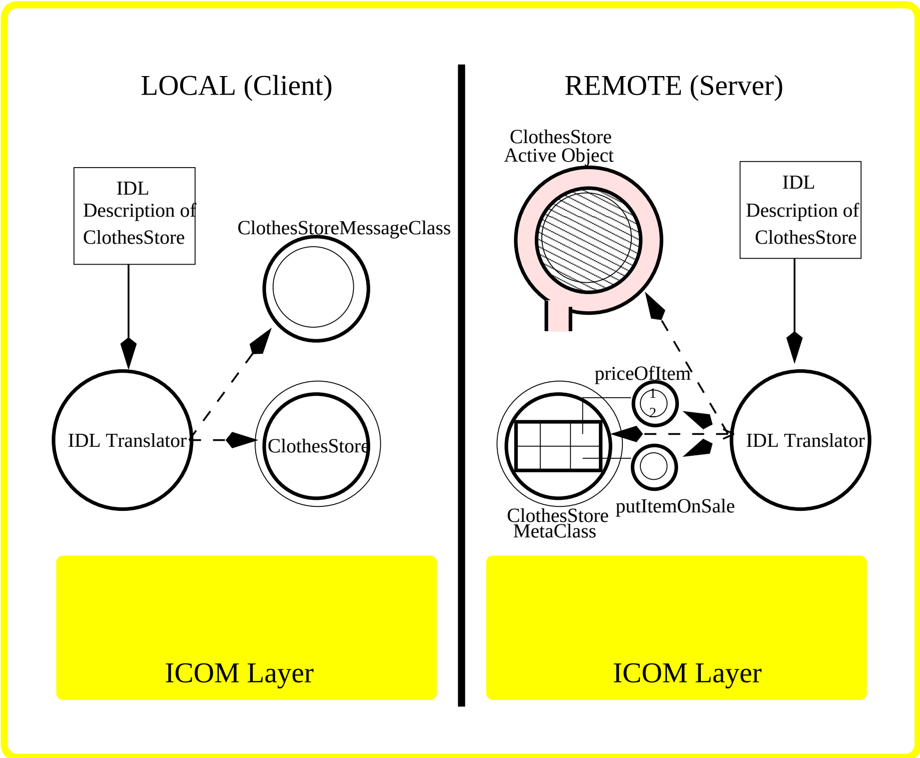

Figure 3: IDL Translation of Clothes Store Interface Containing Overloaded Methods

At the receiving side (say remote receiving side) the metaclass instance looks up in its method table, with the method name. Then the metaclass instance passes the argument list to the method object. The method object decodes the first integer argument which is the method number. With this information the method object knows which signature of the overloaded method to use. Using its own state information, the method object then decodes the rest of the arguments from the message object. The method object then invokes the correct method on the active object.

The IDL translator generates two classes on the client side: ClothesStore proxy class and ClothesStoreMessageClass. ClothesStoreMessageClass is different from a normal message class that is generated for non-overloaded methods in an interface. To create a message object for the method priceOfItem, the translator adds an extra argument to the argument

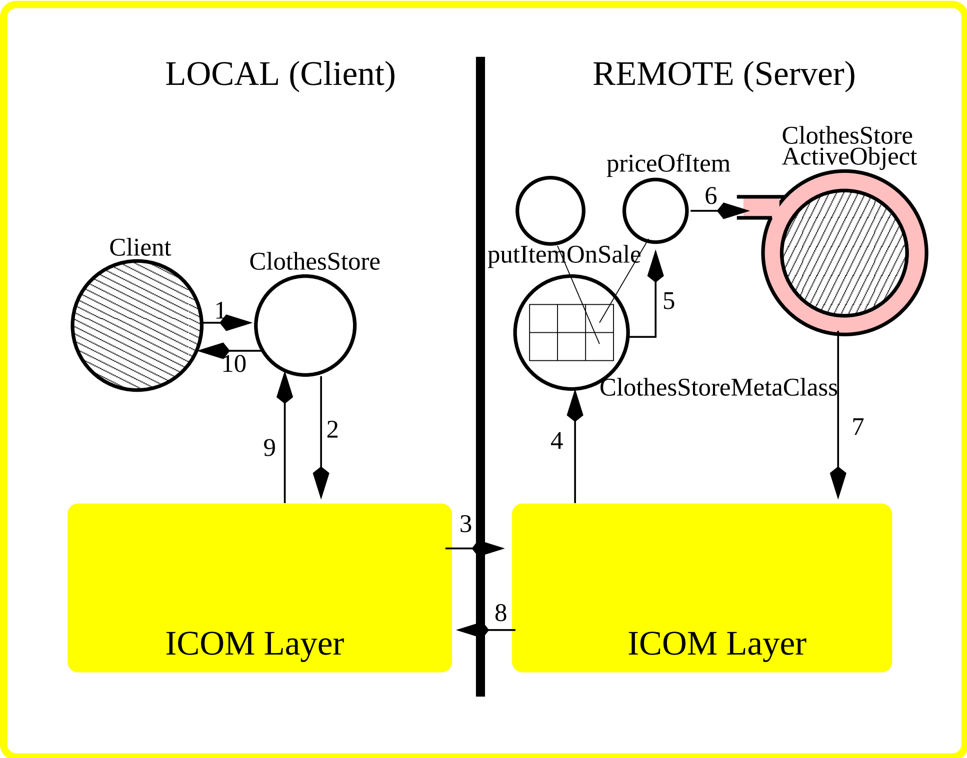

Figure 4: Dynamic Method Binding on an Instance of Clothes Store Interface Containing Overloaded Methods

list. In the case of an invocation of method priceOfItem(itemNumber, price), the translator generates the extra argument that has value 1 representing the first method in the interface. In the second case, it generates the argument with value 2, representing the second method.

When an invocation is made on an instance of ClothesStore, with the parameters of an item number, 'itemNumber' and a future variable, 'price' (arrow labeled 1), the sequence of method calls are explained as follows:

At the local node

an invocation message is created by the ClothesStore instance using the ClothesStoreMessageClass whose class name and method name are set to ClothesStore and priceOfItem respectively. It takes the arguments list generated by the message creator class when invoked for message priceOfItem. The arguments list contains the method number as the first argument. In the example it is 1. This invocation message is encoded into a network representation(XDR) representation and sent to the server side (arrows labeled 2 and 3).

At the remote node

at the server side, the encoded message is received. It is partially decoded to get the network address of the ClothesStore ActiveObject, and the method name and the class name for the method, priceOfItem. The ICOM layer uses the 'metaclass table' to get the instance of the metaclass, ClothesStoreMetaClass. It then passes the partially decoded message to the metaclass instance (arrow labeled 4). The metaclass instance, using its method table, looks up for a method object, and passes the message object to the server method object, priceOfItem (arrow numbered 5). This method object decodes the first argument to get the number of the overloaded method to be invoked. It realizes based on the decoded value that it is the first method of the interface ClothesStore. The method object then decodes the remaining arguments, and invokes the correct method of the ClothesStore implementation (arrow numbered 6).

During the method execution, the output parameter of the method priceOfItem will be filled with a value at the ClothesStore Active Object implementation. Since it is an output parameter, internally it is represented as a future variable. So, the future variable at the client application receives the value which is set at the server side (arrows numbered 7, 8, 9, and 10). The client application thread blocked on the future variable can continue its execution.

3.2 Method Overloading in a Distributed Object

In the previous subsection, method overloading in a localized object is explained in detail. Another way of achieving method overloading in an object is by inheritance. A method defined in a base class can be re-defined with the same name but different signature in a derived class. Both methods visible for a client of a subclass instance.

Two interfaces that have a method each with the same name are shown in Figure 5. The interface ClothesBase can be implemented at the CIPS center and the interface ClothesDerived can be implemented at a remote location, for instance, Virginia. The interface ClothesBase defines a method priceOfItem that takes an input parameter, itemNumber, and has an output parameter price. The derived interface ClothesDerived defines one method. This method has the same name as the method defined in the base interface ClothesBase. But the signatures of the method are different. The method in the derived interface takes an input parameter, itemName, and an output parameter, price. So, the first arguments of both methods with the name priceOfItem differ in their type. In this example, the price of an item can be determined with the item number by using the method implemented in the base interface. It can also be determined by using an item label using the derived interface implementation. This is a useful mechanism because a designer of the derived interface need not invent new names for the methods providing the same functionality.

The IDL translation scheme used for the above example is shown in Figure 6. As shown in the figure, the translation at the local site with the derived interface involves invocation of the translator at the remote site.

```
interface ClothesBase {
    // returns price of an item through 'price'
    priceOfItem(in long itemNumber,
                out double price);
    ⋮
}

interface ClothesDerived {
    // returns price of an item through 'price'
    priceOfItem(in string itemLabel,
                out double price);
    ⋮
}
```

Figure 5: IDL Description of Clothes Store Distributed Object Containing Overloaded Methods

Assume that the base interface has already been implemented at the remote node. When an IDL translator is invoked at the local node, the classes generated relevant to the ClothesBase interface are: ClothesBaseProxyMetaClass, ClothesBaseProxyImplementation, ClothesBaseProxy, and ClothesBaseMessageClass. The classes generated relevant to the ClothesDerived interface are : ClothesDerivedMetaClass, ClothesDerivedImplementation, ClothesDerived, and ClothesDerivedMessageClass. The classes relevant to the derived interface, inherit from their corresponding counterparts in the base interface. The representation is shown using a cracked egg shell notation. The patterned portions in the figure represent the code that must be implemented by the developers. The ClothesDerivedMetaClass contains two entries for the method priceOfItem in its method table. The first entry contains the method name, 'priceOfItem', the class name, 'ClothesBase', and a method object that can decode the arguments of the method, priceOfItem, defined in the ClothesBase interface. The second entry contains the method name, 'priceOfItem', the class name, 'ClothesDerived', and a method object that can decode the arguments of the method, priceOfItem, defined in the ClothesDerived interface. Even though, these two methods are overloaded at a logical level, they can be identified separately because each message object contains both the method name and the class name for the invoked method.

The IDL translator at the local node, realizing that the inheritance involves a remote interface, sends a 'compile' message to the remote IDL translator. The compile message contains details about the inheritance hierarchy involved using the base and derived interfaces. At the remote node, the translator generates four classes corresponding to the derived interface ClothesDerived. These classes are: ClothesDerivedProxyMetaClass, ClothesDerivedProxyImplementation, ClothesDerivedProxy, and ClothesDerivedMessageClass. The translator reuses the existing translator-generated classes for the remote interface ClothesBase. The

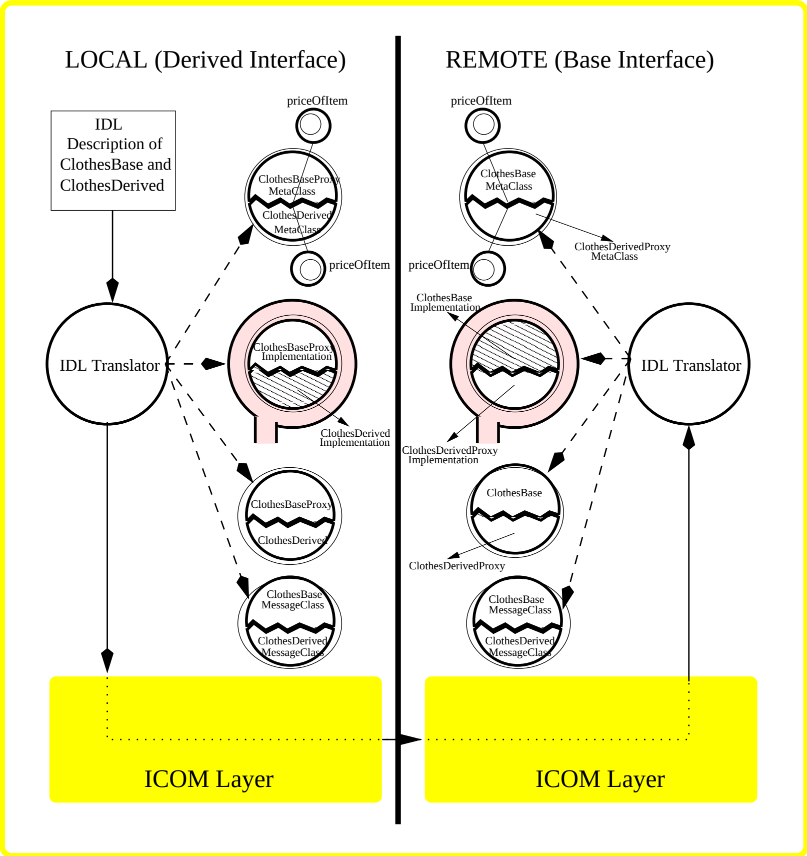

Figure 6: IDL Translation of the Clothes Store Distributed Object Containing Overloaded Methods

newly generated classes at the remote node inherit from their counterparts of the base interface. For example, ClothesDerivedImplementation class inherits from ClothesBaseProxy-Implementation class. The translator then compiles the generated classes and links the base interface implementation code to generate an executable for the 'ClothesStore PeerProxy' at the remote node.

The application developer at the local node must provide implementation for the derived interface, compile it with the native language compiler and link the implementation code with the translator generated code to create an executable. The ClothesDerived class can be included in any application at the local node and used to create its instances and invoke methods on them.

A method invocation of the method priceOfItem with the first argument as an item number, on an instance of ClothesStoreDerived class, results in a series of method invocations at the local as well as remote nodes. This series of method invocations is shown in Figure 7.

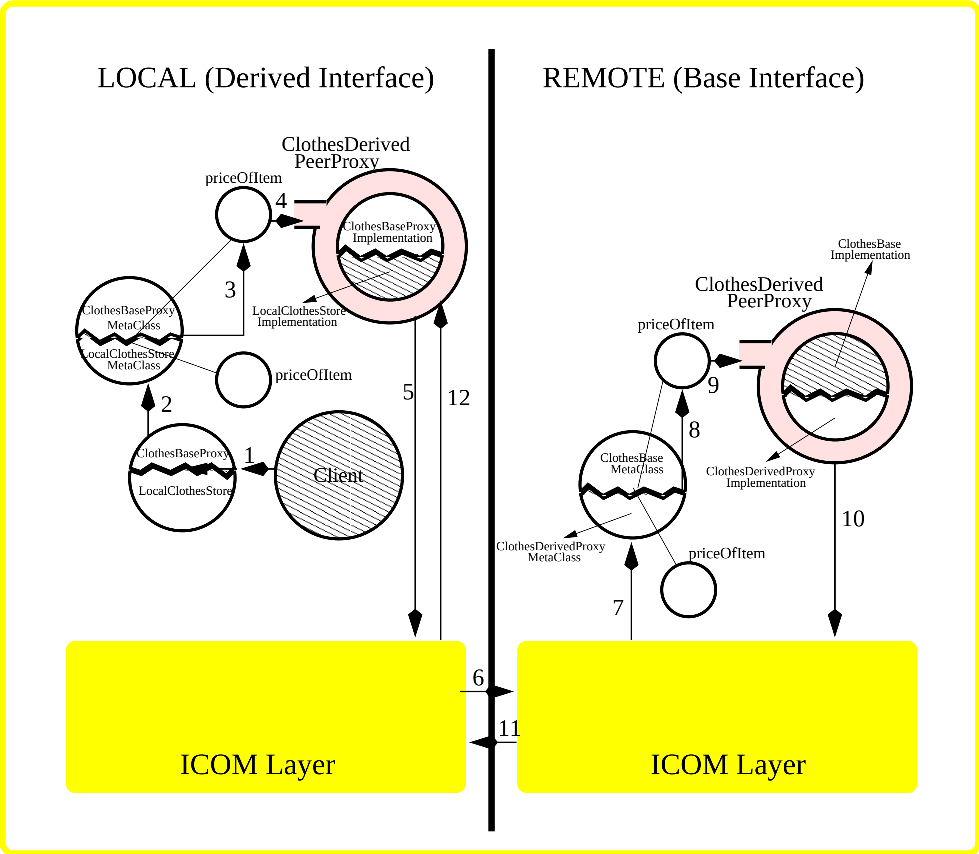

Figure 7: Dynamic Method Binding on a Clothes Store Distributed Object Containing Overloaded Methods

In this example, the client application and the implementation of the interface ClothesDerived are assumed to be on a local node. The implementation of the interface ClothesBase is assumed to be on a remote node. After the translation phase is over, the proxies for the distributed object of ClothesDerived are created at both nodes. When the client application invokes the method priceOfItem with the first argument, 'item number', and the second argument, 'price' (arrow numbered 1), the sequence of method calls is as follows:

At the Client Site

the instance of ClothesDerived creates a message object with the method name, 'priceOfItem', and class name, 'ClothesBase', along with the arguments. This message is delivered to the metaclass instance , ClothesDerivedMetaClass (arrow labeled 2). The metaclass instance passes the message to the method object instance of priceOfItem for the first entry of the method table (arrow labeled 3). The method name and class name for the first entry are 'priceOfItem' and 'ClothesBase' respectively. The method object copies the arguments from the message object, and hands the message to the ClothesDerivedPeerProxy instance. The peer proxy has a stub method for the corresponding method implemented at the remote node. The message is encoded and sent to the remote node (arrows labeled 5,6, and 7).

At the Server Site

the ICOM layer at the remote node partially decodes the message to extract the network address of the peer proxy, and the method name and the class name contained in the message. The peer proxy does a lookup operation in the 'metaclass table' to locate the instance of the metaclass, ClothesDerivedProxyMetaClass(arrow labeled 7). The metaclass instance, using the method name and the class name, looks up in the method table for a server method object. The metaclass instance then passes the message object to the server method object to decode the arguments(arrow labeled 8). The method object decodes the arguments and invokes the priceOfItem method of the ClothesDerivedPeerProxy (arrow labeled 9). The implementation of the method at the remote node executes, setting the value of the output parameter price. Since, every output parameter is a *future* variable, a message is returned to the local node setting the value of the corresponding future variable in the client application (arrows labeled 11 and 12). At this point, the client can proceed with its execution if it has been waiting for the future variable value.

Thus, the ICOM framework supports the method overloading feature of the ICOM object model. In languages that do not support method overloading, the framework supports one-way interoperability. The one-way interoperability allows these languages to use software components of other languages that use method overloading.

4 Parameterized Types

Type parameterization is used to allow a generic type to be instantiated with one or more types. For example we may define a class to implement a generic stack data type. This type can be instantiated with other types such as the basic types, integers and characters, to produce stacks of integers and stacks of characters. The same generic type can be instantiated with user defined types such as a database record to produce stacks of database records. Support for parameterized types exists in several popular languages: templates in C++, generics in Ada 95, Modula-3, and Eiffel. Experience with the use of type parameterization in software developed shows that the ability to define and instantiate a general concept is useful in the design and construction of reusable software components. The prevalence of type parameterization in several popular languages, and recently the efforts to add parameterized types to the language Java indicates the general applicability of parameterized types.

Parameterized types can play an important role in the development of class hierarchies. A popular example is the C++ standard template library.

The generic type scheme, *constrained genericity*, is supported in the ICOM framework. Constrained genericity requires the interface of a parameter to be defined while defining a generic type that uses the parameter. It is not as flexible as a C++ template scheme (called *unconstrained genericity*) where the behavior of a template parameter is not required while defining a template. A C++ compiler can parse the source code of the template(generic type) implementation to determine the interface of the template parameter. The reason for choosing constrained genericity in the ICOM framework is because of the design criterion of the ICOM object model that language compilers should not be changed. If unconstrained genericity were to be supported, the source code of a native language that implements the generic type must be parsed to extract the interface of the parameter, which is undesirable. Also, some of the statically typed object-oriented languages do not support unconstrained genericity. In ICOM, a parameter interface must be present for a generic type to be defined with the parameter. In other words, the interface of the parameter must be known at the compilation time. But, the generic types in ICOM are powerful in the sense that a generic type can inherit from its parameter. The C++ language supports inheritance of template parameter type, but other languages such as Modula-3 do not support it.

Parametric types can be implemented in two ways. A separate implementation can be generated for each type instantiation, as in C++ [11], or a single implementation can be used for all type instantiations, as in ML [9], depending on how the language semantics are defined. For example, 'static' variables play different roles in these two types of implementations. In the first way, each type instantiation will have a separate static variable where as in the second way, only one static variable will be used for all type instantiations. The ICOM framework uses the first method of implementation, separate implementation for each type instantiation.

4.1 Parameterized Types without Inheritance

In this section, the parameterized types of the ICOM object model are explained with an example. This example does not use any inheritance relationships. Figure 8 shows a representation of a 'Department Stores' problem. This figure assumes a system distributed across four states in the United States of America. The central information processing center(CIPS) is assumed to be in New York. A grocery store is in Kentucky and a clothes store is in Virginia. A manufacturer is assumed to be in Georgia. In the figure, parameterized types are shown as smaller circles. The overlapping circle is the parameter. The interface of the parameter type must be present along with the IDL description of the parameterized type StoreTemplate. The cracked egg shell fragments shown represent instantiation of the parameterized type with a truly distributed object. The GroceryBase, Tax, and ClothesBase interfaces are implemented at the New York site. The GroceryDerived and KentuckyTax are implemented at the Kentucky site. The VirginiaTax, and ClothesDerived interfaces are implemented at the Virginia site. The Manufacturer interface is implemented at the Georgia site. The interfaces GroceryDerived and KentuckyTax inherit from GroceryBase and Tax interfaces, respectively. The interfaces VirginiaTax and ClothesDerived inherit from Tax and ClothesBase respectively.

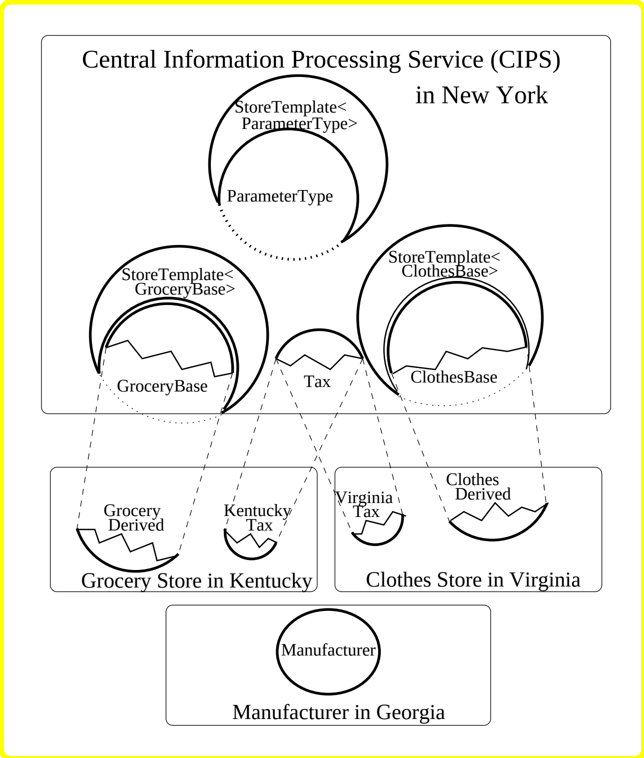

Figure 8: Parameterized Types

The central information processing center(CIPS) can define a default interface for a grocery store, clothes store, and a tax interfaces at the New York site. Through inheritance, these interfaces can be specialized for local stores in Kentucky and Virginia. From the perspective of the financiers of the chain stores, all stores exhibit the same behavior; all stores have the same interface containing the net profits and gross sales. So, the individual stores can be abstracted into a StoreTemplate generic type that has a general store behavior.

Figure 9 shows an instantiation of the generic type StoreTemplate with the interface ClothesStore, and three interfaces, StoreType, StoreTemplate, and ClothesStore. The StoreType is a parameter to the generic type, StoreTemplate. It supports two methods, monthlySales and quarterlySales. The monthlySales method takes two input parameters, monthNumber and yearNumber, and an output parameter, amount. The quarterlySales method takes two input parameters, quarterNumber and yearNumber, and an output parameter, amount. The StoreTemplate generic type defines two methods, netProfit and grossSales. The netProfit method takes an input parameter, yearNumber and an output parameter, amount. Similarly, the grossSales method takes an input parameter, yearNumber and an output parameter, amount. The interface ClothesStore can be used to instantiate the generic type because it supports the methods defined in the interface, StoreType. In addition to the two methods of the StoreType interface, the ClothesStore interface supports two other methods, priceOfItem and putItemOnSale. The first method is used to find out the price of an item given an item number and the second method is used to determine whether to keep an item on sale or not.

```
interface StoreType {
  // returns sales in a month,  of a store through 'amount'
  monthlySales(in int monthNumber, in int yearNumber,
               out double amount);
   // returns sales in a quarter,  of a store through 'amount'
   quarterlySales(in int quarterNumber, in int yearNumber,
                  out double amount);
}

genericInterface StoreTemplate ⟨StoreType⟩ {
   // returns net profit through 'amount'
   netProfit(in int yearNumber, out double amount);
   // returns gross sales through 'amount'
   grossSales(in int yearNumber, out double amount);
}

interface ClothesStore {
   // returns sales in a month,  of a store through 'amount'
   monthlySales(in int monthNumber, in int yearNumber,
                out double amount);
   // returns sales in a quarter,  of a store through 'amount'
   quarterlySales(in int quarterNumber, in int yearNumber,
                  out double amount);
   // returns price of an item through 'price'
   priceOfItem(in long itemNumber, out double price);
}

 // an instantiation of the generic type
 StoreTemplate ⟨ClothesStore⟩ clothesStoreA;
```

Figure 9: IDL Description of the Clothes Store Generic Interface

In this example two distributed nodes are assumed, local and remote. At the remote node, an implementation of the StoreTemplate is provided. Since the parameterized types are constrained, the types of the parameters must be known while defining a parameterized type. That is, the StoreType interface must be defined before defining the StoreTemplate. So, at the remote node the StoreTemplate definition and implementation are available along with StoreType interface.

The IDL translation of the StoreTemplate ⟨ClothesStore⟩ instantiation is shown in Figure 10. The local node defines the ClothesStore interface that conforms to the StoreType

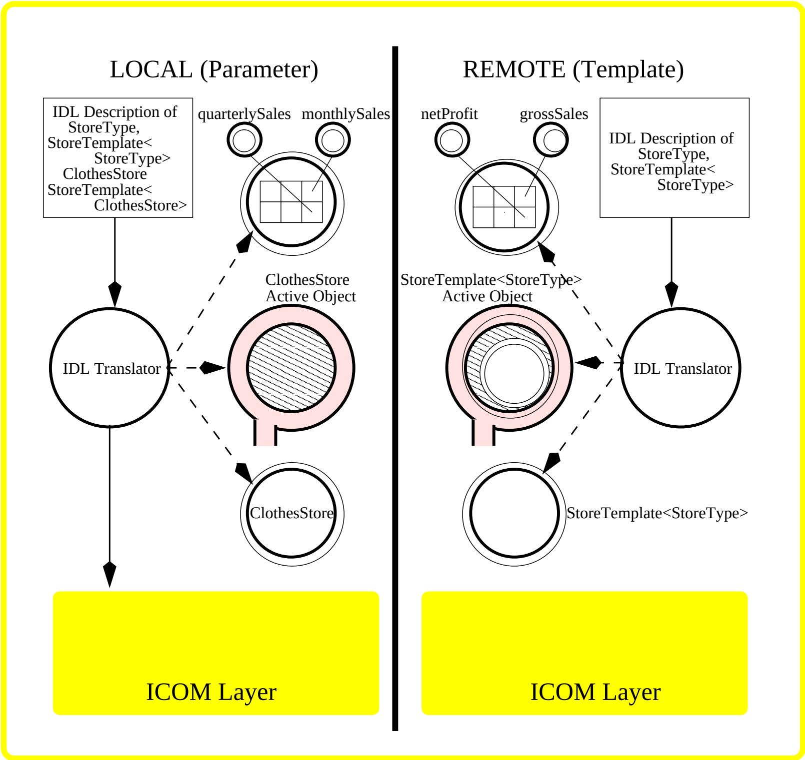

Figure 10: IDL Translation of Clothes Store Generic Interface

interface. A client application also resides at the local node. To instantiate the StoreTemplate at the remote node using the ClothesStore available at local node, a client developer first declares the instantiation in IDL. The description of StoreTemplate is also available in IDL to the client developer.

The IDL translator generates a proxy class for the StoreTemplate. The client developer can use the proxy class to create an instance of the class which is an instantiation of the StoreTemplate with StoreType. The constructor for the proxy class takes the name and location of the ClothesStore interface and the node where it is implemented. The ICOM framework receives a constructor message that has the state information with name value pairs methodName: create, className:StoreTemplate, parameterName:ClothesStore, parameterLocation:local, and other constructor arguments for the StoreTemplate if any. This message is encoded into a common network notation (XDR) and transferred to the remote node. At the remote node, the message is partially decoded. An instance of StoreTemplate is created. This instance contains the information about the ClothesStore and its location. The StoreTemplate instance creates an instance of StoreType which acts as a handle to ClothesStore. The instance of StoreType in turn creates an instance of ClothesStore at the local node. Any method invocations on this handle will be sent to the ClothesStore implementation at the local node.

An invocation of the method grossSales on an instance of the parameterized type, StoreTemplate ⟨ClothesStore⟩ (arrow labeled 1) is shown in Figure 11. The patterned portions of the figure represent the code that must be implemented by the application developers. The method sequence is explained as follows:

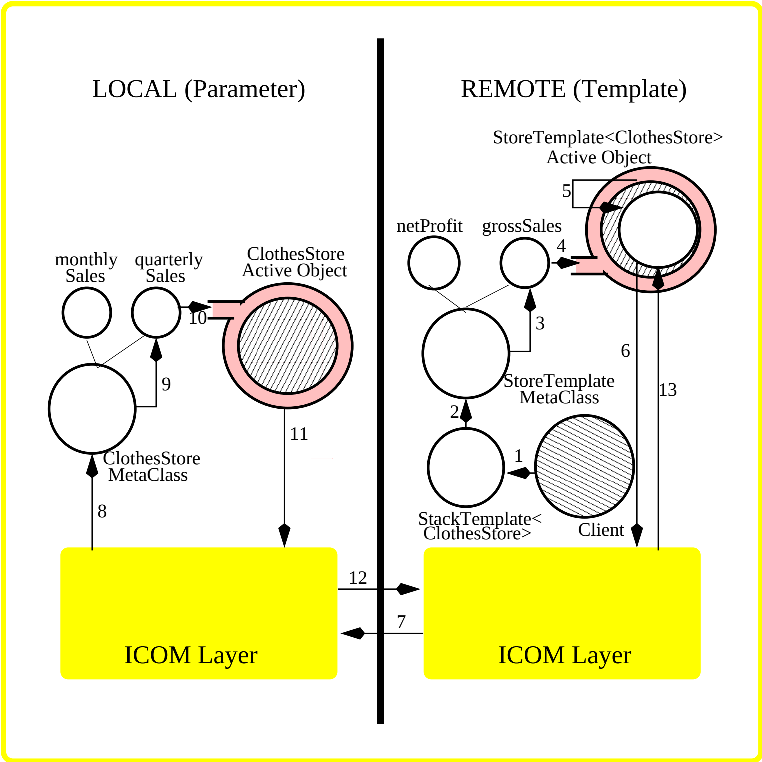

Figure 11: Dynamic Method Binding on an Instance of Clothes Store Generic Interface

At the remote node

the handle object which is an instance of StoreTemplate ⟨ClothesStore⟩ creates a message object for the method grossSales and sends it to the instance of StoreTemplateMetaClass(arrow labeled 2). The metaclass instance looks up in its method table for a method object, and passes the message object to the server method object for decoding of the method arguments (arrow labeled 3). The grossSales method object copies the method arguments (because the invocation is a local invocation) and invokes the grossSales method of the StoreTemplate ⟨ClothesStore⟩ instance(arrow labeled 4). The implementation of the method at the remote node, calls the quarterlySales method of the StoreType interface instance four times to get the gross sales for a particular year (arrows labeled 6,7).

At the local node

the ICOM layer receives the encoded message object for the method quarterlySales. This layer partially decodes the message to obtain the network address of the ClothesStore instance that is used as an argument to the generic type instantiation at the remote node. The message is then passed to the instance of ClothesStoreMetaClass (arrow labeled 8). The metaclass instance does a look up in its method table for a method object and passes the message object to the method object quarterlySales (arrow labeled 9). The method object decodes the message arguments and invokes the quarterlySales method on the instance of ClothesStoreImplementation class (arrow labeled 10). The output argument amount is assigned a value which is forwarded to the StoreTemplateImplementation instance at the remote node (arrows labeled 11,12, and 13). The client application eventually receives the result via the output parameter amount that was passed in the method call of grossSales.

Thus, generic types defined at a remote node can be used to instantiate a type that is compatible with the parameter of the generic type.

4.2 Parameterized Types with Inheritance

In this subsection, a simple example of a parameterized type that uses inheritance is explained. The same example used in the previous section is modified to describe inheritance involving generic types.

Figure 12 shows four interfaces and an instantiation of a generic interface. The interface StoreTemplateBase defines three methods, grossSales, netProfit, and calculate. The methods grossSales and netProfit use the method calculate to finalize the gross sales and net profit at a store. The interface StoreTemplateBase is an example of a situation where the future development of a software is not known when the software is developed. When a single store grows into a national chain, abstraction of one store should be elevated to a chain of stores while reusing the existing code. In this case, the StoreTemplateBase interface can be inherited by a generic type StoreTemplate ⟨StoreType⟩ and the StoreType interface can be defined as a parameter type interface. The figure also shows three other interfaces, StoreType, StoreTemplate ⟨StoreType⟩, and ClothesStore. The generic type StoreTemplate ⟨StoreType⟩ inherits from the interface, StoreTemplateBase. This derived generic interface redefines the two methods of its base interface, grossSales and netProfit. The rest of the interfaces are similar to the example discussed in the previous section.

In this example, it is assumed that the interfaces, StoreTemplateBase, StoreTemplate ⟨StoreType⟩, and StoreType are implemented at the remote node. A client application also exists at the remote node. The interface, ClothesStore is assumed to be implemented at the local node. IDL translators can be independently invoked at both nodes. Since, there are no distributed objects in this example, no exchange of information takes place between the

```
interface StoreTemplateBase {
    // returns net profit through 'amount'
    netProfit(in int yearNumber, out double amount);
    // returns gross sales through 'amount'
    grossSales(in int yearNumber, out double amount);
    // helper routine used by 'netProfit' and 'grossSales'
    calculate(in int yearNumber, in double inAmount, out double outAmount);
}

interface StoreType {
    // returns sales in a month,  of a store through 'amount'
    monthlySales(in int monthNumber, in int yearNumber,
                 out double amount);
    // returns sales in a quarter,  of a store through 'amount'
    quarterlySales(in int quarterNumber, in int yearNumber,
                   out double amount);
}
genericInterface StoreTemplate ⟨StoreType⟩ : StoreTemplateBase {
    // returns net profit through 'amount'
    netProfit(in int yearNumber, out double amount);
    // returns gross sales through 'amount'
    grossSales(in int yearNumber, out double amount);
}
interface ClothesStore {
    // returns sales in a month,  of a store through 'amount'
    monthlySales(in int monthNumber, in int yearNumber,
                 out double amount);
    // returns sales in a quarter,  of a store through 'amount'
    quarterlySales(in int quarterNumber, in int yearNumber,
                   out double amount);
    // returns price of an item through 'price'
    priceOfItem(in long itemNumber, out double price);
}
 // an instantiation of the generic type
 StoreTemplate ⟨ClothesStore⟩ clothesStoreA;
```

Figure 12: IDL Description of Clothes Store Generic Interface involving Inheritance

translators at compilation time. Assume that the interface ClothesStore has been implemented after translating the IDL description of that interface into a native language at the local node.

The IDL translator at the remote node generates the classes: StoreTemplateBaseMetaClass, StoreTemplateBaseImplementation, StoreTemplateBase, and StoreTemplateBaseMessageClass for the interface StoreTemplateBase. The translator also generates the classes: StoreTemplateMetaClass, StoreTemplateImplementation, StoreTemplate, and StoreTemplateMessageClass for the generic interface StoreTemplate ⟨StoreType⟩. These classes inherit from the corresponding classes of the interface StoreTemplateBase. The translator generates, StoreTypeImplementation, and StoreTypeMessage classes for the parameter type StoreType.

The last declaration in the IDL description creates an instance of the generic type StoreTemplate ⟨StoreType⟩. It is instantiated with the type ClothesStore. The IDL translator generates a class StoreTemplateClothesStore that represents the instantiation of the generic type.

A statement in a client application that creates an instance of the class StoreTemplate-ClothesStore generates an instance of the generic type StoreTemplate that is associated with an instance of ClothesStore.

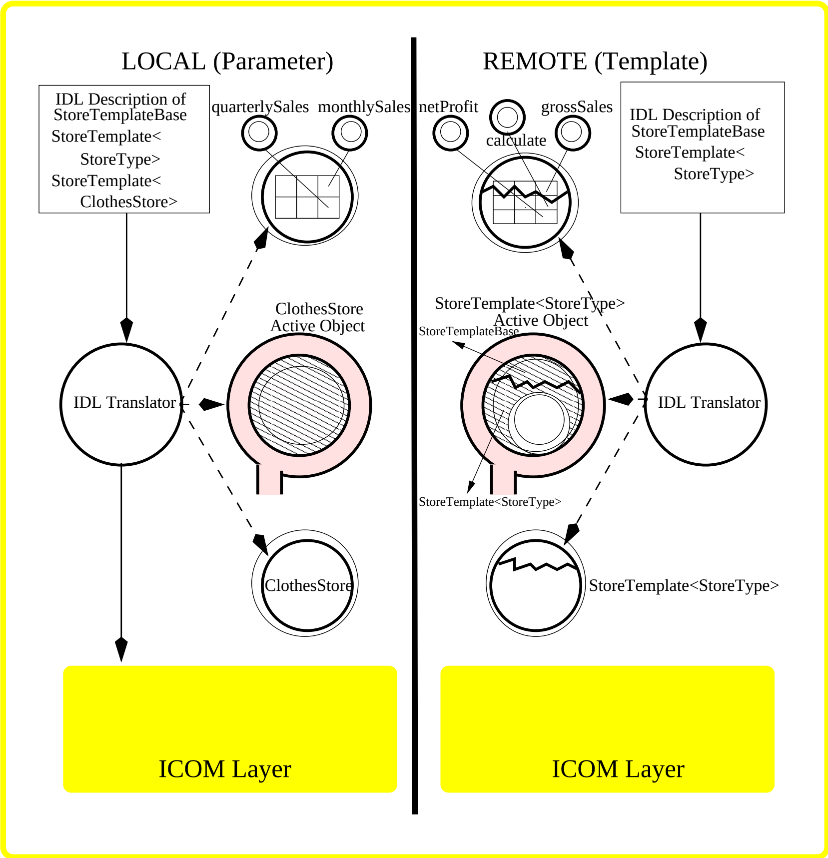

Figure 13: IDL Translation of Clothes Store Generic Interface Involving Inheritance

Figure 14 shows an example invocation of the method grossSales on the instance of StoreTemplate ⟨ClothesStore⟩(arrow numbered 1). The sequence of calls can be explained as follows:

At the remote node

the instance of StoreTemplate ⟨ClothesStore⟩ creates a method object and passes it to the metaclass instance, StoreTemplateMetaClass(arrow numbered 2). The metaclass instance looks up in its method table and passes the message to the method

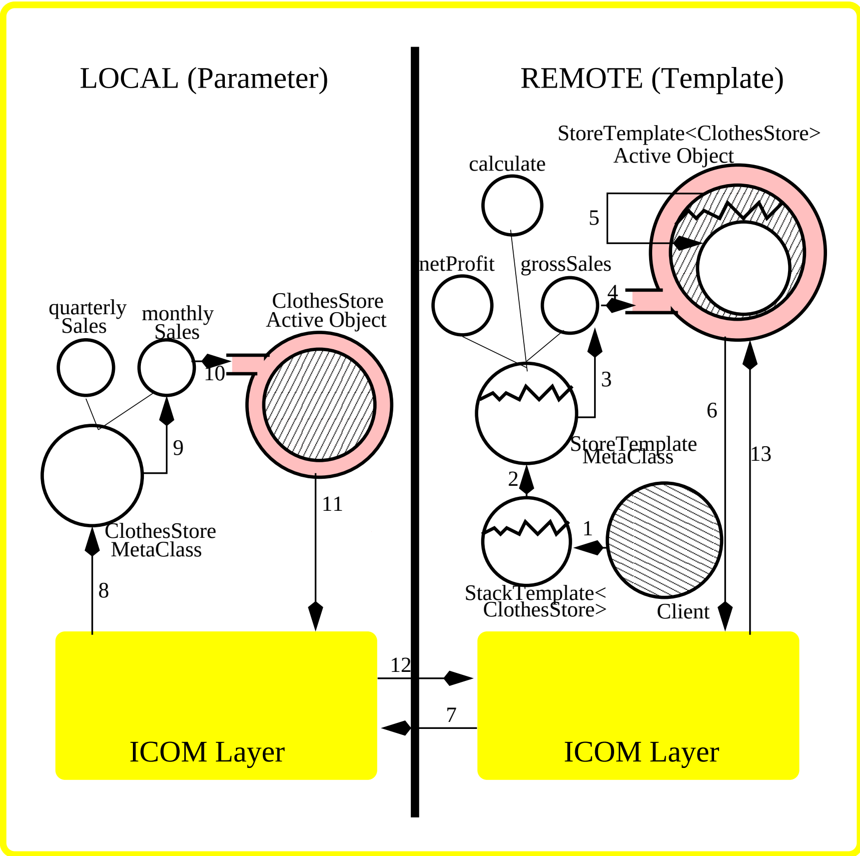

Figure 14: Dynamic Method Invocation on an Instance of Clothes Store Generic Interface Involving Inheritance

object instance, grossSales (arrow numbered 3). The method object copies the argument from the message object(since the method invocation is local) and invokes the method on the generic type implementation(arrow numbered 4). The StoreTemplateImplementation in turn makes an invocation on the method quarterlySales of the StoreType instance four times to calculate the gross sales for a year (arrow numbered 5). The instance of StoreType acts like a proxy for the instance of ClothesStore at the local node. It forwards the method call to the local node (arrows numbered 6 and 7).

At the local node

the ICOM layer at the local node receives the encoded message and partially decodes it to get the network address of the active object of ClothesStoreImplementation. The partially decoded message is forwarded to the meta class instance of ClothesStoreMetaClass (arrow numbered 8). The metaclass does a look up in its method table for a method object and passes the message to the method object, quarterlySales (arrow numbered 9). The method object decodes the arguments and invokes the quarterlySales method on the active object of ClothesStoreImplementation (arrow numbered 10). The output parameters are represented as future variables similar to the previous example. These values are returned to the remote node (arrows numbered 11, 12, and 13). The gross sales for the required year are returned to the client application from the StoreTemplate ⟨ClothesStore⟩ active object. The client can proceed with the execution if it was waiting on the results of the method call.

Thus, generic types that inherit from other types can be used to instantiate with a type that is compatible with the parameter type of the generic type.

5 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 features of method overloading and parameterized types are discussed in detail. Examples are provided to explain the features 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] Object Management Group. Common Object Request Broker: Architecture and Specification. Technical Report OMG Document Number 97.9.1, Rev. 2.1, OMG, 1997.

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

[5] 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.

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

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

[8] 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.

[9] Colin Myers, Chris Clack, and Ellen Poon. *Programming with Standard ML*. Prentice Hall, 1993.

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

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

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

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

[14] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings OOPSLA '87*, pages 168–182, October 1987.

[15] 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.