Automated Generation Of Wrappers For Interoperability

Ngom Cheng
cheng@cs.nps.navy.mil

Valdis Berzins
berzins@cs.nps.navy.mil

Luqi
luqi@cs.nps.navy.mil

Swapan Bhattacharya
swapan@cs.nps.navy.mil

Department of Computer Science
Naval Postgraduate School
833 Dyer Road
Monterey, CA. 93943 USA

Abstract

The major hurdle in developing distributed systems is the implementing the interoperability between the systems. Currently, most of the interoperability techniques require that the data or services to be tightly coupled to a particular server. Furthermore, as most programmers are trained in designing stand-alone application, developing distributed system proves to be time-consuming and difficult. This paper address the issues by creating an interface wrapper model that allows developers the features of treating distributed objects as local objects. A tool was developed to generate Java interface wrapper from a specification language called the Prototyping System Description Language.

1. Introduction

1.1 Background

Interoperability between software systems is the ability to exchange services from one system to another. In order to exchange services, data and commands are relayed from the service providers to the requesters. Current business and military systems are typically 2-tier or 3-tier systems involving clients and servers, each running on different machines in the same or different locations. Current approaches for n-tier systems have no standardization of protocol, data representation, invocation techniques etc. Other problems with interoperability are the implementation of distributed systems and the use of services from heterogeneous operating environments. These include issues concerning sharing of information amongst various operating systems, and the necessity for evolution of standards for using data of various types, sizes and byte ordering, in order to make them suitable for interoperation. These problems make interoperable applications difficult to construct and manage.

1.2 Current State-of-the-art solutions

Presently, the solutions attempting to address these interoperability problems range from low-level sockets and messaging techniques to more sophisticated middleware technology like object resource brokers (CORBA, DCOM). Middleware technology uses higher abstraction than messaging, and can simplify the construction of interoperable applications. It provides a bridge between the service provider and requester by providing standardized mechanisms that handle communication, data exchange and type marshalling. The implementation details of the middleware are generally not important to developers building the systems. Instead, developers are concerned with service interface details. This form of information hiding enhances system maintainability by encapsulating the communication mechanisms from the developers and providing a stable interface services for the developers. However, developers still need to perform significant work in incorporating the middleware's services into their systems. Furthermore, they must have a good knowledge of how to deploy the middleware services to fully exploit the features provided.

Current middleware approaches have another major limitation in the design - the data and services are tightly coupled to the servers. Any attempt to parallelize or distribute a computation across several machines therefore encounters complicated issues due to this tight control of the server process on the data.

1.3 Motivation

Distributed data structures provide an entirely different paradigm. Here, data is no longer coupled to any particular process. Methods and services that work on the data are also uncoupled from any particular process. Processes can now work on different pieces of data at the same time. So far, building distributed data structures together with their requisite interface has proved to be more daunting than other conventional interoperability middleware techniques. The arrival of JavaSpace has changed the scenario to some extent. It allows easy creation and access of distributed objects. However, issues concerning data getting lost in the network, duplicated data items, out-dated data, external exception handling and handshaking of communication between the data owner and data users are still open. The developers have to devise ways to solve those problems and standardize them between applications.

1.4 Proposal

The situation concerning interoperability would greatly improve if a developer working on some particular application were provided with the features capable of treating distributed objects as local objects within the application. The developers could then modify the distributed object as if it is local within the process. The changes may, however, still need to be reflected on other applications using that distributed object without creating any problems related to inconsistency. The current research aims at attaining this objective by creating a model of an interface wrapper that can be used for a variety of distributed objects. In addition, by automating the process of generating the interface wrapper directly from the interface specification of the requirement, developers' productivity is greatly improved.

The tools, named as Automated Interface Codes Generator (AICG), has been developed to generate the interface wrapper codes for interoperability, from a specification language called the Prototype System Description Language (PSDL) [LUQ88]. The tool uses the principle of distributed data structure and JavaSpace Technology to encapsulate transaction control, synchronization, and notification together with lifetime control to provide an environment that treats distributed objects as if there were local within the concerned applications.

2. Review of Existing Work*s*

2.1 ORB Approaches

A basic idea for enhancing interoperability is to make the network transparent to the application developers. The existing approaches [1] include 1)Building blocks for interoperability, 2) Architectures for unified, systematic interoperability and 3) Packaging for encapsulating interoperability services. These approaches have been assessed using the Kiviat graphs by Berzins [1] with various weight factors. The Kiviat graphs give a good summary of the strong and weak points of various approaches. ORB and Jini are currently the more promising technologies for interoperability.

There are however, some concerns with the ORB models. Sullivan [13] provides a more in-depth analysis of the DCOM model, highlighting the architecture conflicts between Dynamic Interface Negotiation (how a process queries a COM services and interface) and Aggregation (component composition mechanism). The interface negotiation does not function properly within the aggregated boundaries. This problem arises because components share an interface. An interface is shared if the constructor or QueryInterface functions of several components can return a pointer to it. QueryInterface rules state that a holder of a shared interface should be able to obtain interfaces of all types appearing on both the inner and outer components. However, an aggregator can refuse to provide interfaces of some types appearing on an inner component by hiding the inner component. Thus, QueryInterface fails to work properly with respect to delegation to the inner interface.

Hence, for the ORB approaches, detailed understanding of the techniques is required to design a truly reliable interoperable system. Programmers however, are train mostly on standalone programming techniques. Adding specialized network programming models increases the learning as well as development time, with occasional slippage of target deadlines. Furthermore, bugs in the distributed programs are harder to detect and consequences of failure are more catastrophic. An abnormal program may cause other programs to go astray in a connected distributed environment [9], [12].

2.2 Prototyping

The demand for large, high quality systems has increased to the point where a quantum change in software technology is needed [9]. Rapid prototyping is one of the most promising solutions to this problem. Completely automated generation of prototype from a very high-level language is feasible and in-fact generation of skeleton programming structures is very common in the computer world. One major advantage of the automatic generation of codes is that it frees the developers from the implementation details by executing specification via reusable components [9].

In this perspective, an integrated software development environment, named Computer Aided Prototyping System (CAPS) has been developed at the Naval Postgraduate School, for rapid prototyping of hard real-time embedded software systems, such as missile guidance systems, space shuttle avionics systems, and military Command, Control, Communication and Intelligence (C3I) systems [11]. Rapid prototyping uses rapidly constructed prototypes to help both the developers and their customers visualize the proposed system and assess its properties in an iterative process. The heart of CAPS is the Prototyping System Description Language (PSDL). It serves as an executable prototyping language at a specification or design level and has special features for real-time system design. Building on the success of computer aided rapid prototyping system (CAPS) [11], the AICG model also uses the PSDL for the specification and automates the generation of interface codes with the objective of making the network transparent from the developer's point of view.

2.3 Transaction Handling

Building a networked application is entirely different from building a stand-alone system in the sense that many additional issues need to be taken care of for smooth functioning of a networked application. The networked systems are also susceptible to partial failures of computation, which can leave the system in an inconsistent state.

Proper transaction handling is essential to control and maintain concurrency and consistency within the system. Yang [16], examined the limitation of hard-wiring concurrency control (CC) into either the client or the server. He found that the scalability and flexibility of these configurations is greatly limited. Hence, he presented a middleware approach: an external transaction server, which carries out the concurrency control policies in the process of obtaining the data. Advantages of this approach are 1) transaction server can be easily tailored to apply the desired CC policies of specific client applications. 2) The approach does not require any changes to the servers or clients in order to support the standard transaction model. 3) Coordination among the clients that share data but have different CC policies is possible if all of the clients use the same transaction server.
The AICG model uses the same approach, by deploying an external transaction manager provided by SUN in the JINI model. All transactions used by the clients and servers are created and overseen by the manager.

3. The Basic Model

The AICG model is based on the concepts of encapsulating some of the features of the JavaSpace and Jini to provide a simplified ways of developing distributed applications. Section 3.1 examines the principles of JavaSpace and section 3.2 discusses some of the features of AICG model.

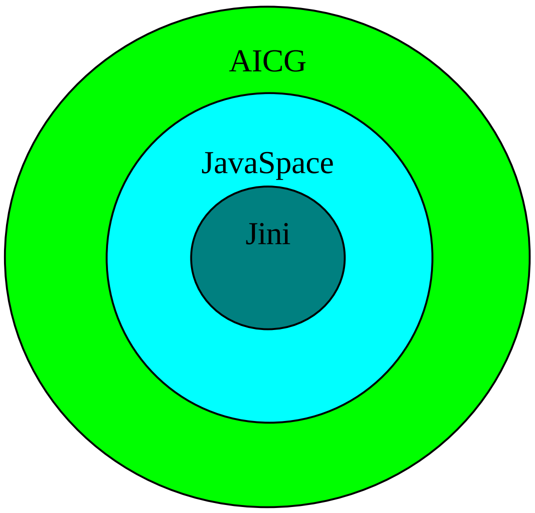

Figure 1, AICG Model

3.1 The JavaSpace Model

JavaSpace model is a high-level coordination tool for gluing processes together in a distributed environment. It departs from conventional distribution techniques using message passing between processes or invoking methods on remote objects. The technology provides a fundamentally different programming model that view an application as a collection of processes cooperating via the flow of freshly copied objects into and out of one or more spaces. This space-based model of distributed computing has its roots in the Linda coordination language [3] developed by Dr. David Gelernter at Yale University.

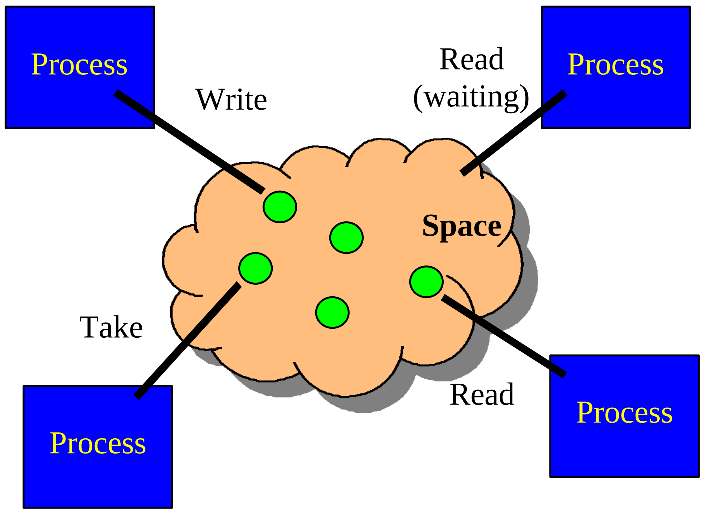

Figure 2, JavaSpace operations

A space is a shared, network-accessible repository for objects. Processes use the repository as a persistent object storage and exchange mechanism. As shown in figure 2, processes perform simple operations to write new objects into space, take objects from space, or read (make a copy of) objects in a space. When taking or reading objects, processes use a simple value-matching lookup to find the objects that matter to them. If a matching object is not found immediately, then a process can wait until one arrives. Unlike conventional object stores, processes do not modify objects in the space or invoke their methods directly. To modify an object, a process must explicitly remove it, update it, and reinsert it into the space. During the period of updating, other processes requesting for the object will wait until the process write the object back to the space.

Key Features of JavaSpace:

- Spaces are persistent: Spaces provide reliable storage for objects. Once stored in the space, an object will remain there until a process explicitly removes it.
- Spaces are transactionally secure: The Space technology provides a transaction model that ensures that an operation on a space is atomic. Transactions are supported for single operations on a single space, as well as multiple operations over one or more spaces.
- Spaces allow exchange of executable content: While in the space, objects are just passive data, however, when we read or take an object from a space, a local copy of the object is created. Like any other local object, we can modify its public fields as well as invoke its methods.

3.2 The AICG Model

The AICG interoperability approach proposes a tool for building distributed applications. The tool is designed to generate interface wrappers for data structures or objects that need to be shared, and are particularly useful for applications that can model as flows of objects through one or more servers. Build on top of JavaSpace, the AICG model hides the space and its implementation details entirely from the application. The interface wrapper allows applications to treat distributed data structures or objects as local within the application space. This enhanced interoperability by making the network transparent to the application developers.

The interface wrappers are generated from an extension of a prototype description language called Prototyping System Description Language (PSDL). The extended Description language (PSDL-ext) expands property definitions that are specific only to AICG model. Some of the salient features of the AICG model are:

- Distributed objects are treated as local objects within the application process. The application code needs not depend on how the object is distributed, since the local object copy is always synchronous with the distributed copy. (see section 5)
- Synchronization with various applications is automatically handled. Since the AICG model is based on the space transaction secure model, all operations are atomic. Deadlock is prevented automatically within the interface by having only a single distributed copy, and through transaction control. (see section 6, 8)

- Any type of object can be shared as long as the object is serializable. Any data structure and object can be distributed as long as it obeys and implements the java serializable feature (see section 10.2).
- Every distributed object has a lifetime. The distributed object lifetime is a period of time guaranteed by the AICG model for storage and distribution of the object. The time can be set by developer (see section 7).
- All write operations are transaction secure by default. AICG transactions are based on the Atomicity, Consistency, Isolation, and Durability (ACID) properties (see section 8).
- Clients can be informed of changes to the distributed object through the AICG event model (see section 9). A client application can subscribe for change notification, and when the distributed object is modified, a separate thread is spawned to execute the callback method defined by the developer.
- The wrapper codes are generated from high-level descriptive languages; hence, they are more manageable and more maintainable.

4. Developing Distributed Application with the AICG Tool

This section describes the steps for developing distributed applications using the AICG model. An example of a C4ISR application is introduced in section 4.2 to aid the explanation of the process. The same example will be used throughout this paper.

4.1 Development Process

The developer starts the development process by defining shared objects using the Prototyping System Description Language (PSDL). The PSDL is processed through a code generator (PSDLtoSpace) to produce a set of interface wrapper codes (figure 3). The interface wrapper contains the necessary codes for interaction between application and the space without the need for the developers to be concerned with the writing and removing of objects in the space. The developers can treat shared or distributed objects as local objects, where synchronization and distribution is automatically handled by the interface codes. The complete cycle for generating the interface codes is shown in figure 4.

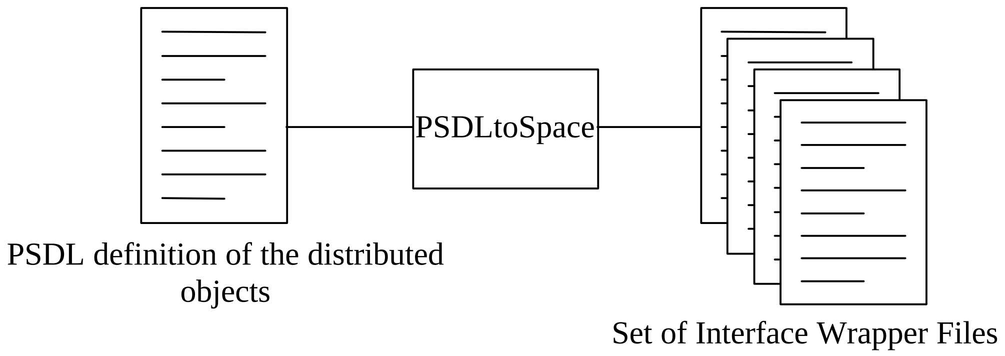
PSDLtoSpace
PSDL definition of the distributed objects
Set of Interface Wrapper Files

Figure 3, PSDL to Space

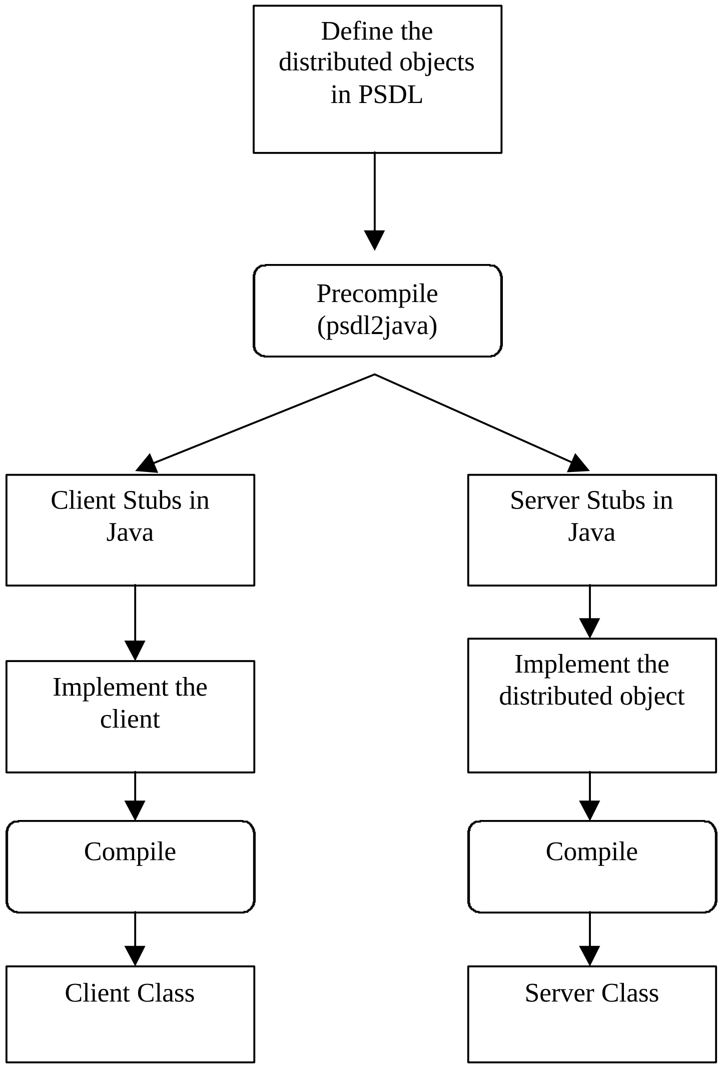

Figure 4, Generating the interface

4.2 Input definition to the Code generator

The following example demonstrates the development of one of the many distributed objects in the C4ISR system. Airplane positions picked up from the sensors are processed to produce track objects. These objects are distributed over a large network and used by several clients' stations for displaying the positions of planes. Each track or plane is identified by track number. The tracks are 'owned' by a group of track servers, and only the track servers can update the track positions and its attributes. The clients only have read access on the track data. Figure 5 shows the PSDL codes for the track object and its methods. Figure 6 shows the PSDL codes for the **Track_list** object and its methods.

```
Type track
 SPECIFICATION
  tracknumber: integer
 END

  OPERATOR track
   SPECIFICATION
    INPUT x:integer
   END
   IMPLEMENTATION
    SPACE
     PROPERTY  SPACEMODE=
CONSTRUCTOR
   END
  END

  OPERATOR getID
   SPECIFICATION
    OUTPUT x:integer
   END
   IMPLEMENTATION
    SPACE
     PROPERTY  SPACEMODE= READ
    END
  END

  OPERATOR setCallsign
   SPECIFICATION
    INPUT sign: string
   END
   IMPLEMENTATION
    SPACE
     PROPERTY  SPACEMODE= WRITE
     PROPERTY TRANSACTIONTIME = 300
   END
  END

  OPERATOR getCallsign
   SPECIFICATION
    OUTPUT sign: string
   END
```

```
IMPLEMENTATION
    SPACE
      PROPERTY  SPACEMODE= READ

    END
  END

  OPERATOR setPosition
   SPECIFICATION
    INPUT post : position_type
   END

   IMPLEMENTATION
    SPACE
      PROPERTY  SPACEMODE = WRITE
      PROPERTY TRANSACTIONTIME = 2000

 END
  END

  OPERATOR getPosition
   SPECIFICATION
    OUTPUT post : position_type
   END
   IMPLEMENTATION
    SPACE
      PROPERTY  SPACEMODE = READ

   END

 IMPLEMENTATION
  SPACE
   PROPERTY SPACENAME= DODSpaces
   PROPERTY OWNERSHIP = YES
   PROPERTY SECURITY = SERVER
   PROPERTY LEASE  = 12000
   PROPERTY CLONE = MANY
   PROPERTY NOTIFY = NO
  PROPERTY RETRY = 10
 END
```

Figure 5, Track example in PSDL

```
TYPE  track_list
  SPECIFICATIO
  END

    OPERATOR track_list
      SPECIFICATION
      END
      IMPLEMENTATION
        SPACE
          PROPERTY  SPACEMODE=
CONSTRUCTOR
      END
    END

    OPERATOR getID
      SPECIFICATION
        INPUT index: integer
        OUTPUT x:integer
      END
      IMPLEMENTATION
      SPACE
        PROPERTY  SPACEMODE= READ

      END
    END

    OPERATOR setNewID
      SPECIFICATION
        INPUT id: integer
      END
```

```
      IMPLEMENTATION
        SPACE
          PROPERTY  SPACEMODE= WRITE
           END
    END

    OPERATOR removeID
      SPECIFICATION
        INPUT id: integer
      END
      IMPLEMENTATION
        SPACE
          PROPERTY  SPACEMODE= WRITE
          PROPERTY TRANSACTIONTIME = 2000
      END
    END
  END

IMPLEMENTATION
    SPACE
      PROPERTY SPACENAME= DODSpaces
      PROPERTY OWNERSHIP = YES
      PROPERTY SECURITY = SERVER
      PROPERTY LEASE  = 0
      PROPERTY CLONE = ONE
      PROPERTY NOTIFY = YES
     PROPERTY RETRY = 5
    END
```

Figure 6, Track list example in PSDL

The PSDL grammar used for the AICG is an extended version of the original PSDL grammar (Appendix A). PSDL model is very extensive and can be used to model an entire distributed system. However, the AICG only used a portion of the PSDL to describe the interface between systems. In another word, interactions between applications are defined using the PSDL but not the application itself. Because of this, slight modifications on the PSDL grammar were needed. The complete listing of the changes in the grammar statements can also be found in Appendix A.

The track PSDL starts with the definition of a *type* called **track**. It has only one identification field **tracknumber**. Of course, the **track** objects can have more than one field, but only one field is in this case is used to uniquely identify any particular **track** object. The type **track_list** shown in figure 5, on the other hand, does not need an identification field since there is only one **track_list** object in the whole system. **Track_list** is used to keep a list of all the active tracks **tracknumber** in the system at that moment in time.

All the operators (methods) of the *type* are defined immediately after the specification. Each method has a list of *input* and *output* parameters that define the arguments of the method.

The most important portion in the method declaration is the *implementation*. The developer must be able to define the type of operation the method supposed to perform. The operations are *constructor* (used to initialize the class), *read* (no modification to any field in the class) and *write* (modification is done to one or more fields in the class). These are necessary, as the code generated will encapsulate the synchronization of the distributed objects.

The other field in the implementation portion of the method, is *transactiontime*. *transactiontime* defines the upper limit in milliseconds within which the operation must be completed. The transaction property is discussed in detail in Section 8.

Upon running the example on figure 5 through the generator tool, a set of Java interface wrapper files are produced. Developers can ignore most of the generated files except the following:

- Track.java: this file contains the skeleton of the fields and the methods of the track class. The user is supposed to fill the body of the methods.
- TrackExtClient.java: this is the wrapper class that the client initialized and used instead of the track class.
- TrackExtServer.java: this is the wrapper class that the server initialized and used in replace for the track class.
- NotifyAICG.java : this class must be extended or implemented by the application if event-notification and call-back are needed.

The methods found in the trackExtClient and trackExtServer have the same method names and signatures of the track class. In fact, the track class methods are been called within trackExtClient or trackExtServer.

5. Distributed Data Structure and Loosely Coupled Programming

Conceptually a distributed data structure is one that can be accessed and manipulated by multiple processes at the same time without regard for which machine is executing those processes. In most distributed computing models, distributed data structures are hard to achieve. Message passing and remote method invocation systems provide a good example of the difficulty. Most of the systems tend to keep data structure behind one central manager process, and processes that want to perform work on the data structure must "wait in line" to ask the manager process to access or alter a piece of data on their behalf. Attempts to parallelize or distribute a computation across more than one machine face bottlenecks since data are tightly coupled by the one manager process. True concurrent access is rarely achievable.

Distributed data structures provide an entirely different approach where we uncouple the data from any particular process. Instead of hiding data structure behind a manager process, we represent data structures as collections of objects that can be independently and concurrently accessed and altered by remote processes. Distributed data structures allow processes to work on the data without having to wait in line if there are no serialization issues.

The distributed protocol for modification ensures synchronization by enforcing that a process wishing to modify the object has to physically remove it from the space, alter it and write it back to the space. There can be no way for more than one process to modify an object at the same time. However, this does not prevent other processes from overwriting the corrected data. For example, in the normal JavaSpace, process A instead of performing a "take" follow by a "write operation, the programmer wrote a "read" operation, followed by a "write" operation. This results in 2 copies of the object in the Space. The AICG model prevents this by encapsulating the 3 basic commands from the developers. All modification on the object are automatically translated to "take", followed by "write" and all operations that access the fields of the distributed object are translated to "read". These ensure that local data are up-to-date and serialization is maintained.

Loosely-coupled programming has it pitfalls also. Distributed objects may be lost if a process removes it from the space and subsequently crashes or is cut off from the network. Similarly, the system may enter in a deadlock state if processes request more than one distributed object while, at the same time, holding on to distributed objects required by other processes. In cases like this, the AICG model groups multiple operations into a transaction to ensure that either all operations complete or none occur, thereby maintaining the integrity of the application. With transaction control, deadlock is prevented if the process did not complete the operation within a certain permitted time. The application can retry the operation immediately or wait for a random time before performing the operation again

6. Synchronization

Synchronization plays a crucial role in any design of distributed application. Inevitably, processes in a distributed system need to coordinate with one another and avoid bringing the system into an unstable state such as deadlock. Creating distributed applications with AICG can significantly ease the burden of process synchronization since synchronization is already built into the AICG operations. Multiple processes can read an object in a space at any time, but when a process wants to update an object, it has to remove it from the space and thereby gain exclusive access to it first. Hence, coordinated access to objects is enforced by the AICG interface doing *read*, *take* and *write* operations.

More advanced and complex synchronization schemes can be easily build upon from the basic atomic features of the AIGC operations. An example is semaphores. Semaphores, a synchronization construct that was first used to solve concurrency problems in operating systems, are commonly found in multithreaded programming languages, but are more difficult to achieve in distributed systems. Semaphores are typically implemented as integer counters that require special language or hardware support to ensure the atomic properties of the *UP* (signal) and *DOWN* (wait) operations. Using AIGC space model, we could easily implement a semaphore as a shared variable that holds an integer counter. By assigning a distributed variable or object as a semaphore, groups of distributed objects can be synchronized. Hence, the AIGC model permits the developers to develop more complicated distributed applications without being concerned about synchronization and deadlock. Furthermore, all operations within the AICG model can impose transaction control with timeout monitoring. After the timeout period, the transaction would rollback the application to a stable state.

7. Object Life Time (Leases/Timeout)

Leasing provides a methodology for controlling the life span of the distributed objects in the AICG space. This allows resources to be freed after a fixed period. This model is beneficial in the distributed environment, where partial failure can cause holders of resources to fail thereby disconnecting them from the resources before they can explicitly free them. In the absence of a leasing model, resources could grow without bound.

There are other constructive ways to harness the benefit of the leasing model besides using it as a garbage collector. As for example, in a real-time system, the value of the information regarding some distributed objects becomes useless after certain deadlines. Accessing obsolete information can be more damaging in this case. By setting the lease on the distributed object, the AICG model automatically removes the object once the lease expires or the deadline is reached.

Java Spaces allocate resources that are tied to leases. When a distributed object is written into a space, it is granted a lease that specifies a period for which the space guarantees its storage. The holder of the lease may renew or cancel the lease before it expires. If the leaseholder does neither, the lease simply expires, and the space removes the entry from its store.

The AICG model simplified the Java Space lease model into two configurations. These are

1. Generally, the distributed object lasts forever as long as the space exists, even if the leaseholder (the process that creates the object) has died. This configuration is enabled by setting the *SPACE lease* property in the Implementation to 0.
2. In real-time environment, the distributed object lasts for a fixed duration of x ms specified by the object designer. To keep the object alive, a write operation must be performed on the object before the lease expires. This configuration is set through the *SPACE lease* property in the Implementation to the time in ms required.

Hence, the developer must provide due consideration towards leasing while developing the application. If an object has a life time, it must be renewed before it expires. In the AICG model, renewal is done by calling any method that modifies the object. If no modification is required, the developer can consider defining a dummy method with the spacemode set to “write”. Invoking that method will automatically renew the lease.

8. Transactions

The AICG model uses the Jini Transaction model, which provides generic services concerning transaction processing in distributed computing environment.

8.1 Jini Transaction model:

All transactions are overseen by a transaction manager. When a distributed application needs operations to occur in a transaction secure manner, the process asks the transaction manager to create a transaction. Once a transaction has been created, one or more processes can perform operations under the transaction. A transaction can complete in two ways. If a transaction commits successfully, then all operations performed under it are complete. However, if problems arise, then the transaction is aborted and none of the operations occurs. These semantics are provided by a two-phase commit protocol that is performed by the transaction manager as it interacts with the transaction participants.

8.2 AICG Transaction model

AICG model encapsulates and manages the transaction procedures. All operations on the distributed object can be either with transaction control or without. Transaction control operations are controlled with a default lease of 2 sec. This default value of leasing time may, however, be overriden by the user. This is kept by the transaction manager as a leased resource, and when the lease expires before the operation committed, the transaction manager aborts the transaction.

Transactions have the following desirable effect on the semantics of the AICG operations. When a distributed object is created, the object is not seen or accessible outside of the transaction until the transaction commits. However, when a distributed object is updated or read under transaction, it can come from new object created within the transaction or objects in the space.

The AICG model by default, enable all transaction for *write* operations and the transaction lease time is two seconds. The developer can modify the lease time through the PSDL SPACE *transactiontime* property.

PROPERTY
transactiontime= 0: Disable transaction for that method
                 /n: Set the lease time to n ms.

All the read operations in the AICG model do not have transactions enabled. However, the user can enable it by using the property transactiontime with the upper limit in transaction time for the read operation. To used the same transaction for more than one operation, the following property must be set.

PROPERTY
transactionID = 99 : An ID number that are the same for more than one method.

9. AICG Event Notification

In the distributed and loosely-coupled programming environment, it is desirable for an application to react to changes or arrival of newly distributed objects instead of "busy waiting" for it through polling. AICG provides this feature by introducing a callback mechanism that invokes user-defined methods when certain conditions are met.

Java provides a simple but powerful event model based on event sources, event listeners and event objects. An event source is any object that "fires" an event, usually based on some internal state change in the object. In this case, writing an object into space would generate an event. An event listener is an object that listens for events fired by an event source. Typically, an event source provides a method whereby listeners can request to be added to a list of listeners. Whenever an event source fires an event, it notifies each of its registered listeners by calling a method on the listener object and passing it an event object.

Within a Java Virtual machine (JVM), an application is guaranteed that it will not miss an event fired from within. Distributed events on the other hand, had to travel either, from one JVM to another JVM within a machine or between machines networked together. Events traveling from one JVM to another may be lost in transit, or may never reach their event listener. Likewise, an event may reach its listener more than once.

Space-based distributed events are built on top of the Jini Distributed Event model, and the AICG event model further extends it. When using the AICG event model, the space is an event source that fires events when entries are written into the space matching a certain template an application is interested in. When the event fires, the space sends a remote event object to the listener. The event listener codes are found in one of the generated AICG interface wrapper files. Upon receiving an event, the listener would spawn a new thread to process the event and invoke the application callback method. This allows the application codes to be executed without involving the developer in the process of event-management.

There are a few steps for setting up AICG event for a particular application. Firstly, the distributed objects must have the SPACE properties for *Notification* set to yes. One of the application classes must *implement* (java term for inherit) the notifyAICG abstract class. The notifyAICG class has only one method, which is the callback method. The user class must override this method with the codes that need to be executed when an event fires.

10. AICG Design

This section explains the design of the AICG and the codes that are generated from psdl2java program. The codes used in this section to explain the AICG and the development processes are generated from the track PSDL of section 4.2.

10.1 AICG Architecture

The AICG architecture consists of four main modules. They are the Interface modules, the Event modules, Transaction modules and the Exception module. The interface modules implement the distributed object methods and communicate directly with the application. In reference to the example in section 4, the interface modules are entryAICG, track, trackExt, trackExtClient, trackExtServer. Instead of creating the actual object (track), the application should instantiate the interface object either the trackExtClient or trackExtServer. Event modules (eventAICGID, evenAICGHandler, notifyAICG) handle external events generated from the JavaSpace that are of interest to the application. Transaction modules (transactionAICG, transactionManagerAICG) support the interface module with transaction services. Lastly, the exception module (exceptionAICG) defines the possible types of exceptions that can be raised and need to be catch by the application. Figure 7 below shows the architecture of the generated interface wrapper and the interaction with the other modules and application.

Each time the application instantiate a track class by creating a new trackExtServer, the following events take place in the Interface:

1. An Entry object is created together with the track object by the trackExtServer. The tack object is placed into the Entry object and stored in the space.
2. Transaction Manager is enabled.
3. The reference pointer to trackExtServer is returned to the application.

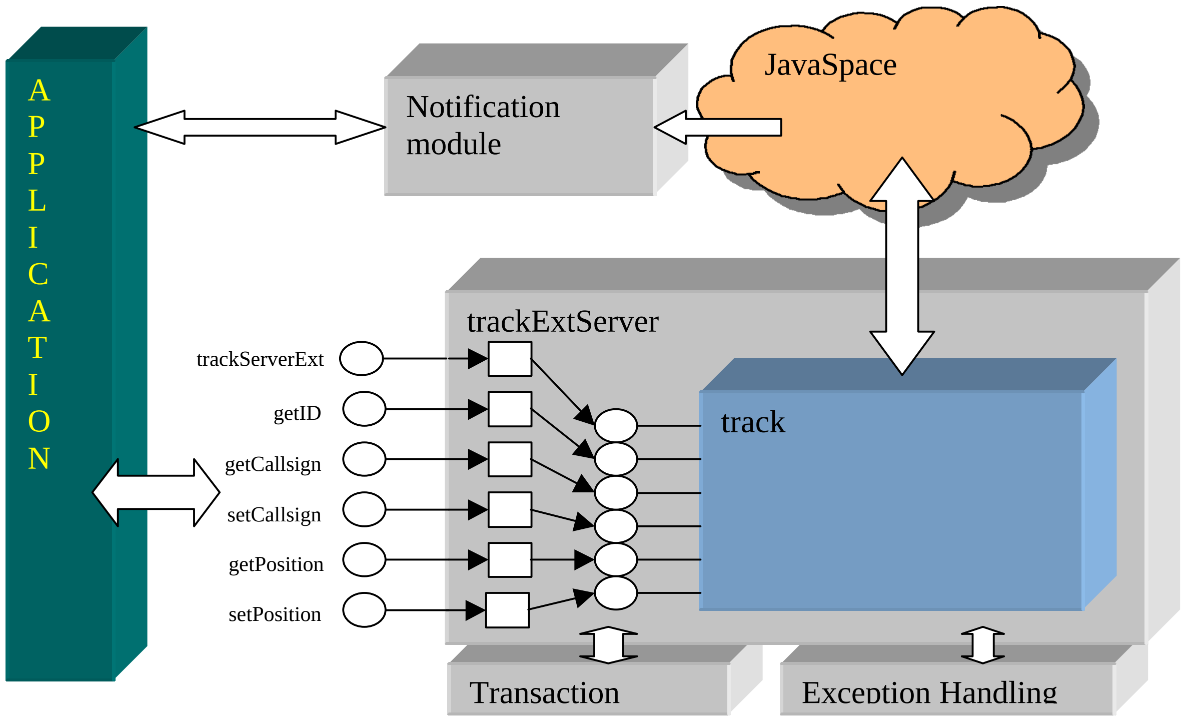

Figure 7, Architecture of the generated interface wrapper and the interaction with the other modules and application

Each time a method (getID, getCallsign, getPosition) that does not modify the contents of the object is invoked, the following events take place in the Interface:

1. When the application invokes the method through the Interface (trackExtServer/trackExtClient).

2. The Interface performs a Space “get” operation to update the local copy.
3. The method is then executed on the updated copy of the object to return the value back to the application.

Each time a method (setCallsign, setPosition), which does modify the contents of the object is invoked, the following events take place in the Interface:

1. When the application invokes the method through the Interface
2. The interface performs a Space “take” operation, which retrieves the object from the space.
3. The actual object method is then invoked to perform the modification.
4. Upon completion of the modification, the object is returned to the space by the interface using a “write” operation.

10.2 Interface Modules

The interface modules consist of the following modules; an entry (entryAICG) that are stored in space, the actual object (trackExt)that are shared and the object wrapper (trackExt, trackExtClient, trackExtServe.).

10.2.1 Entry

A space stores *entries*. An entry is a collection of typed objects that implements the Entry interface. The base class of the AICG distributed object:

```
public abstract class entryAICG implements Entry
   {
      // main identifcation number
     public Integer entryID;
      // required by JavaSpace //default constructor
     public entryAICG( )
     { }
     public entryAICG(int id){
        entryID = new Integer(id);
     }
     // return the object stored in   //the entry
   public abstract Object
      getObject( );
}
```

The Entry interface is empty; it has no methods that have to be implemented. Empty interfaces are often referred to as “marker” interfaces because they are used to mark a class as suitable for some role. That is exactly what the Entry interface is used for, to mark a class appropriate for use within a space.

All entries in the AICG extend from this base class. It has one main public attribute, an identifier and an abstract method that returns the object. Any type of object can be stored in the entry. The only limitation is that the object must be serializable. Serializable allows the java virtual machine to pass the entire object by value instead of by reference. Here is an example “track” entry codes generated by the AICG from the PSDL file in figure 4. The interface contains the object track in one of the field and an ID.

```
public abstract class trackEntry
   extends entryAICG
{
// id is required if there are more
// than one similar object in
// the space
   public Integer id;
   // track object
   public track data;
   // default Constructor
    public trackEntry(){ }
   // Constructor with information
   //extracted from the track PSDL
   // file.
   public trackEntry(int aid, Integer
      inID, track inData){
         super(aid);
         data = inData;
         id = inID;
   }
   public Object getObject(){
      return data;
   }
}
```

All Entry attributes are declared as publicly accessible. Although it not typical of fields to be defined in public in object-oriented programming style, the associative lookup is the way the space-based programs locate entries in the space. To locate an object in space, a template is specified that matches the contents of the fields. By declaring entry fields public, it allows the space to compare and locate the object. AICG encourage object-oriented programming style by encapsulating the actual data object into the entry. The object attributes can then be declared as private and made accessible only through clearly defined public methods of the object.

10.2.2 Serialization

Each distributed interface object is a local object that acts as a proxy to the remote space object. It is not a reference to a remote object but instead a connection passes all operations and value through the proxy to the remote space. All the objects must be serializable in order to meet this objective. The Serializable interface is “marker” interface that contains no methods and serves only to mark a class as appropriate for serialization. Here is the Serializable interface:

```
public abstract interface Serializable
{
   // this interface is empty
}
```

In that case, the *track* class of the example needs to implement the interface Serializable.

```
public class track implements
   Serializable {
     . .
   // since Serializable is a marker
   // interface no methods need to be
   //override.
}
```

10.2.3 The Actual Object

We now look at the actual objects that are shared between the servers and clients. The psdl2java generates a skeleton version of the actual class with the methods names and its arguments. The body of the methods and its fields need to be filled by the developers. The track class generated is shown below:

```
public class track implements
java.io.Serializable
{
   private Integer trackNumber;

   public track(int inID){
   // insert the body here
   }
   public int getID(){
    // insert the body here
   }

   public void setPosition
      (position_type post){
    // insert the body here
   }

   public position_type getPosition(){
   // insert the body here
   }

   public String getCallsign(){
   // insert the body here

   }
   public void setCallsign(String
         sign){
   // insert the body here
   }

   // automatically generated do
   // not delete!!
   public Integer autoGetID1(){
      return trackNumber ;

   }
}
```

10.2.4 Object Wrapper

Wrapping is an approach to protecting legacy software systems and commercial off-the-shelf (COTS) software products that require no modification of those products [1]. It consists of two parts, an adapter that provides some additional functionality for an application program at key external interfaces, and an encapsulation mechanism that binds the adapter to the application and protects the combined components [1].

In this context, the software being protected contains the actual distributed objects, and the AICG model has no way of knowing the behaviors of the distributed object other than the type of operations of the methods. The adapter intercepts all invocations to provide additional functionalities such as synchronization between the local and distributed object, transaction control, events monitoring and exceptions handling. The encapsulation mechanism has been explained in the earlier section (AICG Architecture). Instead of instantiation of the actual object, the respective interface wrapper is instantiated. Instantiating the interface wrapper would indirectly instantiate the actual object as well as storing the object in the space.

Three classes generated for every distributed object. There are named with the object name appended with the following Ext, ExtClient, and ExtServer.

10.3 Event Modules

The event modules consist of the event callback template (notifyAICG), the event handler (evenAICGHandler) and the event identification object (eventAICGID).

10.3.1 Event Identification object

The event identification object is used to distinguish one event from others. When an event of interest is registered, an event identification object is created to store the identification and event source. Together these two properties act to uniquely identify the event registration.

The object has only two methods, an ‘equals’ method that check if two event identification objects are the same and a ‘to string’ method which is used by the event handler for searching the right event objects from the hash table.

10.3.2 Event Handler

Event Handler is the main body of the event operation in the AICG model. It handles registration of new events, deletion of old events, listening for event and invoking the right callback for that event. Inside the event handler are in fact, three inner classes to perform the above functions. Events are stored in a hash table with the event identification object as the key to the hash table. This allows fast retriever of the event object and the callback methods.

The event handler listens for new events from the space or other sources. When an object is written to the space, an event is created by the space and captured by the all the listeners. The event handler would immediately spawn a new thread and check whether the event is of interest to the application.

```
// call when an external event is
// “fired”.
public void run() {
    Object source = event.getSource();
    long id = event.getID();
    long seqN =
       event.getSequenceNumber();
    // create a new event identifcation
    //object
    eventAICGID keyID= new
       eventAICGID(id,source);
    registerAICG tempReg;

    String key = new
       String(keyID.toString());
    // check if the key exist in the
    // hash table (storage)
    if ((tempReg = (registerAICG)
       storage.get(key)) !=null)
```

```
      {
      // check if the event is an old or
      // duplicate event
         if (seqN > tempReg.seqNum) {
            tempReg.seqNum = seqN;
            src.listenerAICGEvents
               (tempReg.anyObj);
         } else {
         // old events ignored
            return;
         }
      }
   }
}// end of notifyHandler
```

10.3.3 The Callback Template

The callback template is a simple interface class with an abstract method listenerAICGEvents. Its main function is to allow the AICG model to invoke the application program when certain events of interest is “fired”. As explain earlier, the template need to be implemented by the application that wishes to have notification.

```
   public interface notifyAICG
   {
      public abstract void
      listenerAICGEvents(Object obj);

   }
```

10.4 The Transaction Modules

The transaction modules consist of transaction interface (transactionAICG) and the transaction factory (transactionManagerAICG).

The transaction interface is a group of static methods that are used for obtaining reference to the transaction manager server somewhere on the network. It uses the Java RMI registry or the look-up server to locate the transaction server.

The transaction factory uses the transaction interface to obtain the reference to the server, which is then used to create the default transaction or user-define transaction. In short the transaction factory can perform the following:

1. Invoke the transaction interface to obtain a transaction manager.
2. Create a default transaction with lease time of 5 seconds.
3. Create a transaction with a user define lease time.

10.5 The Exception Module

The exception module defines all the exception code that is return to the application when certain unexpected conditions occur in the AICG model. The exception include

- "NotDefinedExceptionCode"; unknown error occur.
- "SystemExceptionCode"; system level exceptions, such disk failure, network failure.
- "ObjectNotFoundException"; the space does not contain the object.

- "TransactionException"; transaction server not found, transaction expire before commit.
- "LeaseExpireException"; object lease has expired.
- "CommunicationException"; space communication errors.
- "UnusableObjectException"; object corrupted.
- "ObjectExistException"; there another object with the same key in the space.
- "NotificationException"; events notification errors.

11. Conclusion

This paper demonstrates the ease of sharing distributed objects and automates the generation of generic interface wrappers directly from the Prototype System Description Languages. However, the design has a performance price penalty. Every read operation requires the interface to synchronize the local object with the distributed object before the value is returned. Every write operation requires two Space operations. Adding the overhead for transactions, event monitoring and control, reading operations are in the range of a hundred milliseconds and writing is in the range of a few hundred of milliseconds. The high overhead lies within the Java Virtual Machine (JVM), the JavaSpace Model and the network latency. Current versions of JVM and JavaSpace are in a premature state in terms of performance. Even so, the performances are still suitable for most applications that are not time critical. Similar implementations of distributed systems with the above features of AICG interface in CORBA and Java would not perform any better. Hence, the AICG model is still a viable option in developing interface wrapper for distributed system.

12. References

[1] Naval Postgraduate School Report NPSCS-00-001, *Interoperability Technology Assessment for Joint C4ISR Systems*, by Valdis Berzins, Luqi, Bruce Shultes, Jiang Guo, Jim Allen, Ngom Cheng, Karen Gee, Tom Nguyen, and Eric Stierna , September 1999.

[2] Nicholas Carriero, and David Gelernter, "*How to Write Parallel Programs: A Guide to the Perplexed*" ACM Computing Surveys, September 1989, pp.102-122.

[3] David Gelernter, "*Generative Communication in Linda*", ACM Transaction on Programming Languages and Systems, Vol. 7, No. 1, January 1985, pp. 80-112.

[4] Bill Joy, *The Jini Specification*, Addison Wesley, Inc., 1999.

[5] Edward Keith, *Core Jini*, Prentice Hall, PTR, 1999.

[6] Eun-Gyung Kim, "*A Study on Developing a Distributed Problem Solving System*" IEEE Software, January 1995, pp. 122-127.

[7] Fred Kuhns, Carlos O'Ryan, Douglas Schmidt, Ossama Othman, and Jeff Parsons, "*The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware*", IFIP 6th International Workshop on Protocols For High-Speed Networks (PfHSN' 99), August 25-27, 1999.

[8] David Levein, Sergio Flores-Gaitan, and Douglas Schmidt, "*An Empirical Evaulation of OD Endsystem Support for Real-time CORBA Object Request Brokers*", Multimedia Computing and Network 2000, January 2000.

[9] Luqi, and Valdis Berzins, "*Rapidly Prototyping Real-Time Systems*", IEEE Software, September 1988, pp. 25-35.

[10] Naval Postgraduate School, *A Proposed Design for a Rapid Prototyping Language*, by Luqi, Valdis Berzins, Bernd Kraemer, and Laura White, March 1989

[11] Luqi, Mantak Shing, "*CAPS – A Tool for Real-Time System Devleopment and Acquisition*", Naval Research Review, Vol 1, 1992, pp.12-16.

[12] Luqi, Valdis Berzins, and Raymond Yeh, "*A Prototyping Language for Real-Time Software*", IEEE Software, October 98, pp.1409-1423.

[13] Kevin Sullivan, Mark Marchukov, and John Socha, "*Analysis of a Conflict Between Aggregation and Interface Negotiation in Microsoft's Component Object Model*" IEEE Transactions on Software Engineering, Vol. 25, No. 4, July/August 1999, pp. 584-599.

[14] Antoni Wolski, "*LINDA: A System for Loosely Integrated DataBases*", IEEE Software, January 1989, pp. 66-73.

[15] Andrew Xu, and Barbara Liskov, "*A Design of a Fault-Tolerant, Distributed Implementation of Linda*", IEEE Software January 1989, pp. 199-206.

[16] Jingshuang Yang, and Gail Kaiser, "*JPernLite: Extensible Transaction Services for the WWW*", IEEE Transactions on Knowledge and Data Engineering, Vol. 11, No. 4, July/August 1999, pp. 639-657.

Appendix A: PSDL Grammar

A.1 Extension to PSDL for AICG model

<u>Changes:</u>

17. type_impl = "implementation ada" id "end"
| "implementation" type_name {"operator" id operator_impl} "end"
| "implemetation space" <space_impl>

18 <operator_impl> ::= implementation <id>
<id> end | implementation <psdl_impl> end
| implementation
<class_impl> end

* Those in bold is the additional grammars added.

<u>Additional:</u>

52. **<space_impl> ::= space {<property>} end**