Empirical Software Engineering, 4, 217–240 (1999)

Collecting Metrics for CORBA-Based Distributed Systems

JOHN D. MCGREGOR johnmc@cs.clemson.edu
Dept. of Computer Science, Clemson University

IL-HYUNG CHO ihcho@cs.clemson.edu
Dept. of Computer Science, Clemson University

BRIAN A. MALLOY mallory@cs.clemson.edu
Dept. of Computer Science, Clemson University

E. LOWRY CURRY ecurry@cs.clemson.edu
Dept. of Computer Science, Clemson University

CHANIKA HOBATR puh@cs.clemson.edu
Dept. of Computer Science, Clemson University

Abstract. The Common Object Request Broker Architecture (CORBA) supports the creation of distributed systems that cross processor, language and paradigm boundaries. These systems can be large and complex entities that consume considerable resources in their creation and execution. Measurements of characteristics of software systems is an important area of study in general and of particular interest for distributed systems. In this paper, we present a specific technique for instrumenting components in a distributed system. The technique constructs a wrapper around the component being measured. The wrapper monitors interactions with the ORB (Object Request Broker) and other components. Each wrapper mimics the interface of the component that it is wrapping so that the remaining objects in the system do not need modification. Two approaches to wrapping the component are presented and contrasted. The result is an efficient and modular technique that can quickly be applied to a component.[1]

1. Introduction

Distributed systems constructed around the Common Object Request Broker Architecture (CORBA) are intended to achieve a wide range of objectives. These include opportunities for the integration of programs written in different languages, the integration of processes running on dissimilar processors, and the interoperation of components designed using different paradigms. The architecture is being widely used as the basic structure for distributed systems that use object-oriented design techniques. The work presented in this paper is a technique for measuring attributes of systems designed around CORBA.

The use of CORBA makes an application developer's job easier by simplifying the traditionally complex task of communication between various elements of a distributed system. CORBA provides software designers with an object-oriented paradigm for building network applications. These CORBA-based objects can reside anywhere on the network, and an access to another process becomes nothing more than method invocations on local objects acting as proxies for objects in other processes and possibly on other processors. The use of

CORBA has made the integration of components into network applications easier, yet the consequence of such a simple interface has produced applications that populate the network with objects distributed with no regard to the performance impact on the overall system. Heavier than necessary network traffic, poor throughput, and other factors can often make a CORBA application's overall performance unsatisfactory.

This paper explores the need for "performance-aware" CORBA objects. A CORBA-based object should be able to monitor its own overall health, measure network and CPU utilization, and realize when its own activity is adversely affecting the performance of the entire distributed system. One way of adding this functionality to CORBA objects is through a wrapping technique described in this paper. A generic statistical wrapper can be placed around most CORBA servers or clients, with minimal or no changes to existing code. With this approach, a server object can provide a user interface showing vital statistics—feedback that can help a user or administrator understand implications the CORBA object has on the system so that corrective measures can be taken.

Wrapping is a commonly used technique that enables a component or a complete application to be used in different environments by modifying the interface or the behavior of the component. A wrapper can be used for diverse reasons:

1. An application is written in a procedural language, like C, and we want to make it work in distributed fashion in an ORB.

2. The program we want to wrap is written in an object oriented language, like C++ or Java. The application either

   - does not implement an IDL (Interfact Definition Language). That is, the application is not CORBA-ready, and we want to make it work in distributed fashion in an ORB. Or,
   - already implements an IDL. That is, the application is in CORBA-ready form, and we want to add additional features, for example, codes for measuring metrics.

The contribution of this work is in the area of measurement of distributed systems. In particular, we have developed an ORB independent technique for capturing information that characterizes the interaction between components and the ORB. In this paper we focus on conceptual issues relating to the mechanics of monitoring the traffic between components.

In section 2, we present the background needed to understand the techniques. We also describe the simple system we will use as an example to illustrate each of the techniques in detail. Section 3 reviews the GQM (Goal/Question/Metric) measurement framework. In section 3.2 the technique is described in detail and it is contrasted with alternatives. Finally, in section 7 we describe the feature course of our work.

2. Background

The technique being presented operates within the context of the CORBA approach to distributed systems. The technique gathers measurements that are used to characterize the various components of the distributed system. For illustrative purposes, the game of Nim

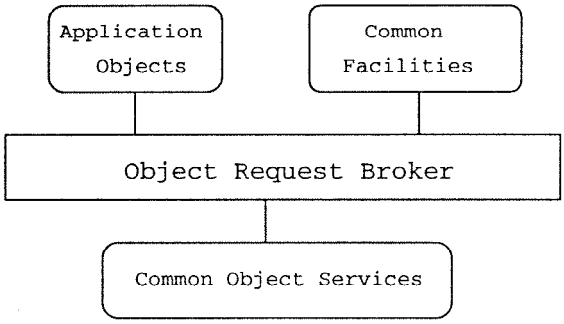


Figure 1. Object management architecture.

has been chosen as an example application. A brief background is provided for each of these areas.

2.1. CORBA

The Object Measured Group (OMG) defined CORBA to facilitate the interoperation of programs written by multiple vendors using different languages and different design paradigms (Object Management Group, 1995). The architecture of a CORBA-based system includes an Object Request Broker (ORB). The ORB encapsulates the services required to take a message from one application running on a specific processor and delivers it to another process running on the same or different processor. The architecture is illustrated in Figure 1. These services include the translation of data representations from their processor of origin to a representation appropriate for the destination processor. CORBA was intended to support the interoperation of components that are dissimilar.

CORBA achieves its goals by connecting components through *interfaces*. A standard Interface Definition Language (IDL) has been defined by the OMG. Each component developer who wishes to provide services to other components defines an interface for the component using IDL. A compiler translates the IDL implementation for the server's interface into the languages needed by the other components, termed clients. The result is that each component can "speak" to a proxy, termed a stub, for the server. The proxy provides an interface to the server written in the same language as the client. Thus the flexibility of CORBA comes at a cost of the interaction of several layers. The intermediate layers can be customized to improve performance. Our work provides a means of measuring the effectiveness of this customization.

2.2. Structure of the Nim Game

Nim is a two player game. The game consists of a number of rows of objects. The objects can be anything you can imagine—matches, toothpicks, oranges, etc. During a turn, the

```
// code for referee

void main()
{
  int board[10];
  configBoard(board);
  int noWinner = 1;
  int turn = 0;
  while (noWinner) {
    if (turn==0)
      player0(board);
    else player1(board);
    if (moveOK())
      adjustBoard();
    else {
      // invalid move
      winner != turn;
      noWinner = false;
    }
    if (winnerFound()) {
      winner = turn;
      noWinner = false;
    }
    else turn != turn;
  }
}
```

```
// code for player0
void player0 (int board[])
{
// code for finding best move
// according to its strategy
}
```

```
// code for player1
void player1 (int board[])
{
// code for finding best move
// according to its strategy
}
```

Figure 2. Pseudo code for the nim game.

player picks any number of the objects from a single row. Whoever picks the last object wins. (Or whoever picks the last object loses.)

The rows of objects are implemented in a single dimension array of integers. We call it a board. Each array cell contains an integer that is greater than zero. The integer depicts the number of objects in each row of the game. The game uses three modules: a referee and two players, each in separate files. The referee first configures the board to play and invokes each player to take a turn to play the game. In its turn, each player figures out the best move according to its strategy. The referee then checks for the validity of the move and decides if the game is over and who the winner is. The pseudo C code in Figure 2 illustrates the structure of the game. (Our actual implementation is written in C++).

2.3. Distributing the Nim Game

Using CORBA, only a few changes are required to implement the three processes as distributed objects. An IDL interface is used to define interfaces to objects in a distributed system, providing all the information that clients need to access the object across a network. A single IDL interface can be used to define both player0 and player1. The relation of the IDL interface and the implementation class is shown in Figure 3.

The IDL interface code is compiled via an IDL compiler, which generates a stub and a skeleton class, and a header file for both classes. The header file defines the class nimGame, while the stub and skeleton provide the client and server with the low level communication and parameter marshaling necessary in a distributed application. A client program, in this case the referee, will obtain an object reference of the type class nimGame from the ORB and will use it to access the remote object. The programmer is responsible

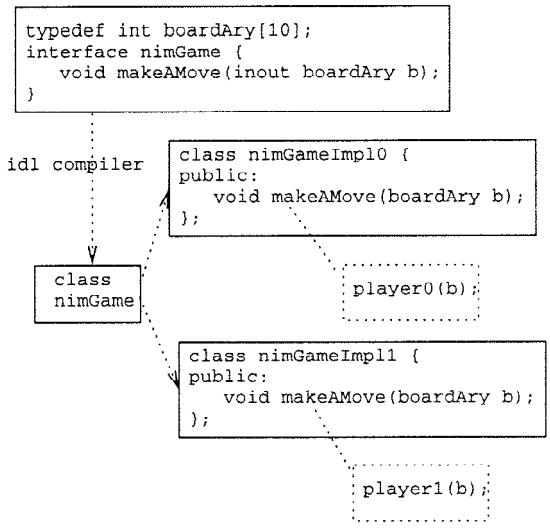


Figure 3. The implementation class of the IDL interface.

for implementing the code for the methods declared in the IDL interface. In this example, nimGameImpl0 and nimGameImpl1 are the programmer-defined classes that implement the interface. The method makeAMove invokes the procedure containing the implementation for the appropriate players move. Note that the implementation class wraps the C code (i.e., working as a wrapper) and makes it work on the ORB.

2.4. Basic CORBA Client Approach

The client program (the referee) first needs to obtain a reference for each implementation, and then sends the message makeAMove to each player. The distributed version of the referee code is shown in Figure 4.

Once the ORB is initialized, the client program requests a remote object reference to the server of choice and then invokes operations on the object reference. To accomplish this, the client program first obtains a reference to the ORB and then requests a reference to the Naming Service from the ORB. The ORB returns an object reference of type CORBA::Object that needs to be narrowed to a more specific type. Once obtaining a reference to the Naming Service, the client program can request the reference for each of the players. An object reference is instantiated from an IDL interface class. For an interface A, `A_var` denotes the object reference variable type of the interface. `A_ptr` denotes the object reference pointer type. The term "object reference" is used to refer to either an object reference variable or an object reference pointer. For further details of the differences of the two types, refer to (Object Management Group, 1995).

```
// code for referee
void main(int argc, char *argv[])
{
  int board[10];
  nimGame_var player0, player1;

  // initialize ORB
  ...
  // obtain a ref for player0 and 1
  ...
  configBoard(board);
  while (noWinner) {
    if (turn == 0) player0.makeAMove(board);
    else player1.makeAMove(board);
    if (moveOK()) adjustBoard();
    else { // invalid move
      winner = !turn;
      noWinner = false;
    }
    else turn = !turn;
  } // while
  displayWinner();
}
```

Figure 4. Referee (client) program.

2.5. Basic CORBA Server Approach

A server needs to initialize the ORB and BOA (Basic Object Adapter), and register the programmer-defined implementation object with the ORB. The programmer supplies implementation classes for an IDL interface. In the Nim example these classes were called nimGameImpl0 and nimGameImpl1. The CORBA standard does not explicitly specify how the programmer-defined implementation class becomes associated with the interface class (nimGame) to allow the implementation object to be registered with the ORB. Thus different ORBs implement the object registration in different ways. This is why porting server programs to a different ORB is a difficult task. (Porting client programs are relatively easy.) Orbix supports two approaches to allow a programmer to implement an IDL interface, referred to as TIE and BOAImpl.

2.5.1. BOAImpl Approach

For each interface, the Orbix IDL compiler automatically generates a skeleton class with the same name as the interface and appended with BOAImpl. For the interface nimGame, the class nimGame-BOAImpl would be generated. The BOAImpl class inherits from the corresponding IDL C++ class, and is used as a base class for the implementation class. Figure 5 shows the structure for the nimGame.

Each box represents a class. An arrow shows an inheritance relationship. The nimGameImpl class implements operations defined in the IDL interface. Figure 6 shows how the server program is initialized.

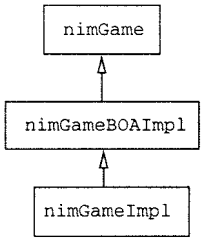


Figure 5. Nim game structure.

The constructor of the nimGameBOAImpl class registers the implementation object with the BOA and it is hidden from the implementation class. The BOAImpl class is produced if the −B switch is specified to the IDL compiler.

2.5.2. TIE Approach

In the TIE approach, the programmer implements the IDL operations in a class that does not inherit from the BOAImpl class. Instead, the programmer provides an implementation class and tells Orbix that the class implements the IDL interface. This is done through the TIE macro provided by Orbix. The main program is shown in Figure 7.

The DEF_TIE macro generates a TIE class TIE_nimGame(nimGameImpl0) that records a particular interface (nimGame) that is implemented by a particular implementation class (nimGameImpl0). The macro is of the form DEF_TIE_*interface name(implementation class name)*. Figure 8 shows the relationship.

Note that the object reference to be accessed by a client program is of type nimGameImpl0, but what is actually returned is the TIE object, the instantiation of the class TIE_nimGame-(nimGameImpl0). The TIE object forwards all the invocations to itself to the implementation object. One distinctive difference from the BOAImpl approach is that, in the TIE approach, the implementation class is not derived from the IDL interface class.

```
// main program for player0 - BOAImpl approach
int main(int argc, char *argv[]) {

  // initialize the ORB and BOA
  CORBA::ORB_var orb = CORBA::ORB_init(argc,argv,0);
  CORBA::BOA_var boa = orb->boa_init(argc,argv,0);

  nimGameImpl0 nimgameImpl;  // create an impl object
  // tell Orbix that the server init is completed
  boa->impl_is_ready("nimPlayer0");
  return 0;
}
```

Figure 6. The main program for server—BOAImpl Approach.

```
// main program for player0 - TIE approach

DEF_TIE_nimGame(nimGameImpl0); // tells Otbix that
// nimGameImpl0 implements the nimGame IDL interface

int main(int argc, char* argv[]) {
  // create an impl object and tie together
  nimGame_ptr nimgameImpl =
     new TIE_nimGame(nimGameImpl0)(new nimGameImpl0);
  // initialize the ORB and BOA
  ...
  boa->impl_is_ready("nimPlayer0");
  return 0;
}
```

Figure 7. The main program for server—TIE Approach.

3. Measuring CORBA-Based Code

3.1. Deriving the Required Metrics

Following the GQM (Goal/Question/Metric) approach (Basili, 1992), metrics are characterized by the goals that they attempt to quantify. For distributed systems two major categories of goals are measures of design quality and measures of performance. The former provides guidance during development about the appropriateness of the design choices being made. The latter provides guidance during implementation and deployment about the arrangement of processes and the speed with which they are able to interact. In this work we focus only on measures of performance. Although our focus was mainly on the use of existing software, the infrastructure being built to integrate the components provided an opportunity to investigate measurements that supported some design decisions.

Each goal is the source of a series of questions that elicit the information required to determine whether the stated goals have been achieved. A question addresses a portion of a particular goal, or, in some cases, cuts across several goals. For example, in the area of performance, one question might be how much of the time required for an operation is due to the actual computation versus the time required to deliver messages between nodes.

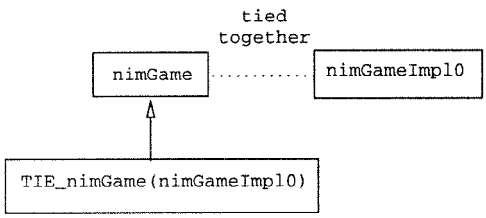


Figure 8. Class structure for TIE.

Finally, metrics are defined to collect the data needed to answer the questions. Simple measurements, such as counting the number of nodes involved in a computation or the number of packets required to communicate a message, can be aggregated to answer specific questions about the level of effort or time required for a specific type of computation.

Using the GQM approach results in a set of metrics whose utility is clearly justified. Commercially available metrics tools offer a spectrum of measures since the company seeks to attract the widest possible audience. This often results in project personnel being told which questions can be answered based on what measures are available. Our intent is to provide a framework in which a project using CORBA-based components can quickly develop the code necessary to measure specific attributes about component interactions that will answer their questions.

The problem for which this research was initiated was the integration of several large components into a transportation simulation system that will ultimately include macro views of entire fleets and micro views that will completely simulate the interaction of driver and machine. A number of large-scale software components were already available, such as symbolic math systems and discrete event simulators. These had not been implemented to work with any other systems much less each other. The separate applications were to be integrated into a single system using a CORBA-based approach.

The basic goals are;

G1. To create a working system from these existing components and to configure it to provide the performance necessary for realistic simulations.

G2. To provide a flexible system that could be configured with a variety of components to meet many different simulation needs.

These goals led to the formulation of a number of questions. A sample of these is included below.

Q1. What is the difference in total time for each required operation to be performed? This question reflects a tradeoff. In one scenario, two processes are loaded on the processor and must compete for processor time. In the alternative, each process is given its own processor but network overhead is now added to the total computation time.

Q2. Where does the system spend most of its computational time? If different combinations of operations result in different answers to question 1, then the developer needs to be able to determine the most effective and efficient combination of methods. Establishing an operational profile (Basili, 1992) would allow the developer of the system to either choose an explicit configuration or to allow a range of possibilities.

Q3. How much of the total execution time is spent on network travel and how much is true computational time? The ability to measure each of these independently gives the developer who configures the system the information necessary to judge the impact on the system of using a faster, but probably more expensive, communication route between the components.

A number of measurements were incorporated into the metrics systems to answer the questions posed above.

- total number of messages
- total bytes sent

- total service time
- total time taken from end to end for a message
- total interval time (network latency)
- arrival rate of messages (frequency of method invocation)
- average service time
- server's utilization

From these measurements we can provide information that answers the end user's questions, such as, worst case, average case, and best case service time of each method, and frequency of method invocation. Also some information can be provided for a system administrator, for example, utilization of the server, network latency, and number of bytes sent per second. Those performance metrics have been measured and reported by a number of authors (Harrison et al., 1997; Chatterjeet et al., 1997; Datametrics Systems Corporation; Bond and Hine, 1991; Schmidt et al., 1995; Corwin and Braddock, 1992; Braddock et al., 1992). There are some more interesting measurements that can be gathered related to what other people have done. If the system we are interested in is a real-time system, schedule bound, the maximum resource utilization possible without deadlines being missed, can be measured (Harrison et al., 1997). Also, event channel overhead can be captured when the system uses the CORBA Event Service (Harrison et al., 1997). The resulting impact of system design, presented in (Chatterjeet et al., 1997), can be applied if we want to contrast performance properties of design approaches, TIE and BOAImpl approach in Orbix.

3.2. Making Measurements

The traditional use of wrappers in distributed object programming is to provide a CORBA layer around existing, non-distributed applications. It is often the case that legacy code provides valuable functionality locally, but because it is not CORBA-enabled the functions are not available to remote clients on the network. Using the concept of object wrappers, existing non-distributed implementations can be made available to the network. To take this concept one step further, another layer (statWrapper) can be placed around the CORBA layer, similar to the way the legacy code was wrapped. This provides a technique for defining a generic wrapper with the specific job of collecting metrics on objects in a distributed system. The diagram in Figure 9 illustrates this idea.

The statWrapper becomes a channel through which all method invocations flow, allowing the activity of the object to be measured. All implementation pertaining to metrics gathering is confined to the statWrapper layer; thus the wrapper can be added or removed without effecting the overall system. The specific measurements taken by the statWrapper and degree of granularity of the measurements are left to the designer of the statWrapper layer.

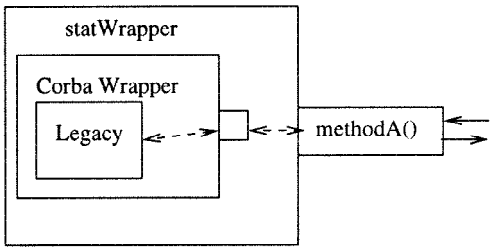


Figure 9. Object Wrappers can extend the functionality of an existing distributed system.

There are two common design techniques for implementing the above concepts, and successfully adding an extra layer of functionality to an object. The first technique is through the use of inheritance to capture the interface of the class being wrapped. The second technique is to use object composition where the object being wrapped is instantiated within the wrapper and the interface for the wrapper comes from the IDL interface. We described two approaches—BOAImpl and TIE—that are specific to the Orbix ORB in section 2.5, and illustrated the two methods on the wrapper structures of the nim game described in section 2.2. In the next sections we consider specific techniques for adding functionality to the wrapper to measure characteristics of the wrapped component.

4. Wrapper Structure for IDL Defined Applications—Adding Private Operations

Sometimes we want to add additional features to an IDL-implemented class. This can be done either by subclassing or composing the original class. The wrapper is only used with server side applications. A client application will not know that it is interacting with the wrapper object as opposed to the original object. In this section we present the final piece needed to understand the metrics wrapper.

We want to measure the performance of the Nim game: time taken on the makeAMove operation and the interprocess communication time between the client (referee) and the server program (each player). The time measuring code will be added to the wrapper class, and the wrapper class will provide a routine (either local or remote) that returns the measurement results. If the routine is local we can still use the original IDL interface, but if it is remote, the routine has to be specified in the IDL interface so that other clients, interested in collecting metrics, may be able to invoke the routine. The former case is simpler, and we consider it first.

4.1. Subclassing

Wrapping using subclassing is possible for both BOAImpl and TIE approaches. We can simply have the wrapper class inherit from the original implementation class. The diagram in Figure 10 illustrates the case.

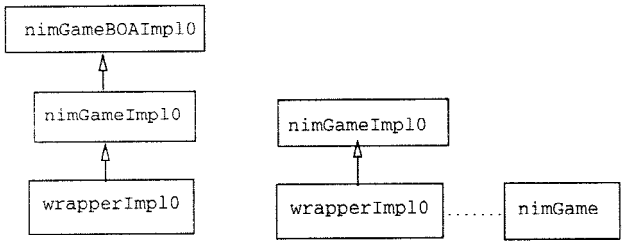


Figure 10. Subclassing for BOAImpl and TIE approach.

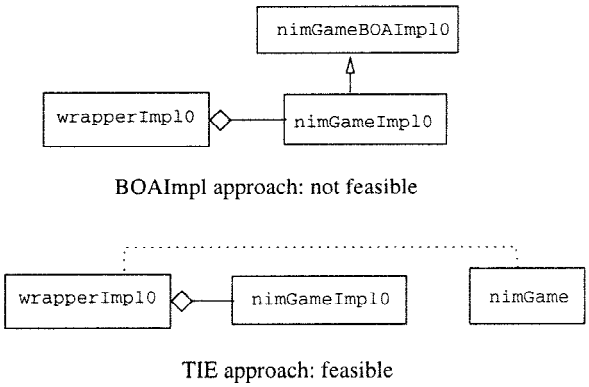


Figure 11. Object composition for BOAImpl and TIE approach.

In the BOAImpl approach, the wrapper class is the one that is created as an implementation object. Since it is a subtype of the BOAImpl class, the wrapper class is capable of registering itself with the BOA and the client can invoke operations on the wrapper class via the object reference of the wrapper. In the TIE approach, the wrapper class is the one that is tied to the IDL interface class.

4.2. Object Composition

Possible class structures of object composition for both BOAImpl and TIE approaches are shown in Figure 11. We mentioned in section 2.5 that the constructor of the BOAImpl class registers an implementation object with the BOA. The client program should get an object reference to that implementation object to invoke operations remotely. However, the nimGameImpl0 object (or its subclass object) is the one that can be registered with the BOA, not the wrapperImpl0 object. The wrapper is responsible for getting messages from the client program but the client program cannot obtain an object reference for the wrapper

Figure 12. Wrapper for TIE with object composition.

object. Therefore, the object composition approach is not compatible with the BOAImpl approach.

As shown in Figure 11, we can directly tie the wrapper class with the IDL C++ class nimGame. (Previously, the wrapped class nimGameImpl0 was tied to the nimGame class.) The macro Def_TIE_nimGame (wrapperImpl0) tells Orbix that the wrapper Impl0 class implements the IDL interface, and ties the implementation object to the interface object. The nimGameImpl0 class does not change. The wrapper class and the main program are shown in Figure 12.

Note that the client program does not need to be aware of the changes. It can still ask for an object reference type nimGame_var (or nimGame_ptr) with the service name "nimPlayer0" (or "nimPlayer1") and invoke remote operations on the wrapper class because the wrapper class is tied together with the IDL C++ class nimGame. Also note that the main program for player0 is the same as for the TIE subclassing mechanism. We have shown that the TIE approach works with both the subclassing and object composition, but the BOAImpl approach works with only the subclassing approach.

5. A Wrapper Structure for IDL Defined Applications—Adding Public Operations

If a method needs to be added to the class and accessed by other objects, the method must be declared public in the implementation class, and an operation with the same name as the method, must be provided by an IDL interface. Now, a new remote client program can

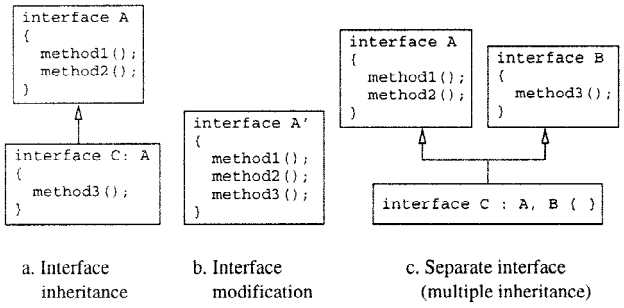


Figure 13. Wrapper structure for new IDL operation.

invoke an operation on the new interface method. The object that implements the method would be the same as the implementation object described in the previous section.

There are several possible design possibilities: interface inheritance, interface modification, and separate interface. The three diagrams in Figure 13 show the structure of each approach.

We assume that the interface A is the original interface, method1() and method2() are the methods defined in the existing interface, and method3() is the new method. The interface C in Figure 13(a) inherits the interface A and defines the new method in it. The original client program will still get an object reference of type A, but a new client that invokes the new operation will get an object reference of type C. However, revealing methods (method1 and method2) to the new client is not good design since the new client does not need to know about them.

In Figure 13(b), the original interface A is modified to A′ by defining the new method. Then, both old and new clients will see the same interface even though they need only a part of it. In Figure 13(a), unnecessary methods are exposed to the new client, but in Figure 13(b), both clients suffer the problem. The interface C multiply inherits from the interface A and B in Figure 13(c). The interface B is a new interface and defines the new method. Now each client sees its own portion of the interface, and the previous problem is resolved.

So far, we have discussed the client side of interface use. Figure 13(c) shows a conceptually cleaner design than the other two since it does not expose unnecessary methods. The following subsections show the BOAImpl and TIE approach with subclassing and object composition techniques.

5.1. Interface Modification Approach

We first consider the approach depicted in Figure 13(b) since it provides simpler implementation than the other two approaches. We use the same interface name as the one we used before, but the interface now has one additional method showMeasure(). This new method is for a new GUI client that asks for measured data and displays it in the GUI environment.

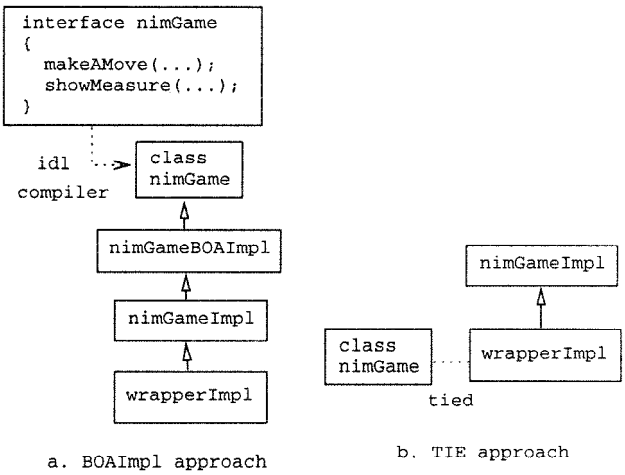


Figure 14. Wrapper structure for interface modification—subclassing.

Figure 14(a) is for subclassing on BOAImpl. The showMeasure() routine will need an `out` parameter for measured data. The nimGameImpl class is the implementation class defined before the new method was added to the interface. Thus, it defines only the makeAMove() method. In the main server program, we can create a wrapperImpl object, but not a nimGameImpl object since the class nimGameImpl implements only a part of the methods declared in the nimGame interface. The wrapperImpl class defines the additional method, showMeasure(), and for the makeAMove() method, it simply forwards the message to its superclass.

The TIE approach with inheritance is similar to the BOAImpl approach. The wrapperImpl class defines the new method, and is tied to the class nimGame.

Figure 15 shows the structure of the object composition technique. Note that the object composition does not work for the BOAImpl approach with the same reason as in 4.2.

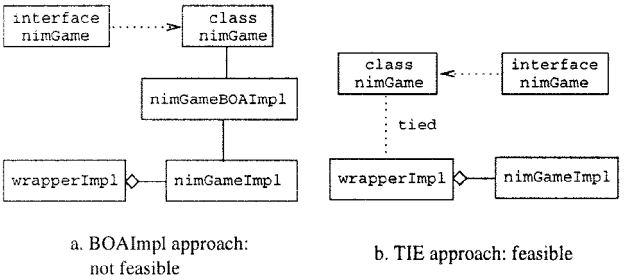


Figure 15. Wrapper structure for interface modification—object composition.

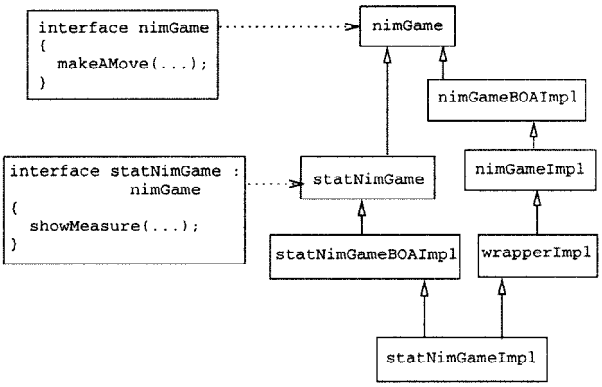


Figure 16. Wrapper structure for interface inheritance—BOAImpl.

In the TIE approach, the wrapperImpl object forwards the makeAMove() message to the nimGameImpl object, but handles the showMeasure() message by itself.

5.2. Interface Inheritance

Figure 16 shows the class structure of the BOAImpl approach. The statNimGameImpl class works as a wrapper, not the wrapperImpl class that is the wrapper for a nimGameImpl object. As illustrated in section 4, the wrapperImpl class measures time before and after it forwards the makeAMove() message to the nimGameImpl class. Note that the nimGameImpl and wrapperImpl class need no change. The referee client will still invoke operations with the reference type nimGame_ptr (or nimGame_var), and the GUI client will invoke operations with the reference type statNimGame_ptr (or statNimGame_var), but there will be only one implementation object of class statNimGameImpl. (If the wrapperImpl object is created in the main server program, we now have two separate implementation objects, and the data measured in the wrapperImpl object will not be accessible to the statNimGameImpl object which simply up-calls to wrapperImpl class to get the data.)

The statNimGameImpl class wraps the wrapperImpl class. Upon receiving the GUI clients request, the statNimGameImpl object will invoke the local method of the wrapperImpl class (show_measure()) to get the measured data. The referee's call will also be directed to the statNimGameImpl object, but the call will be handled by its parent class wrapperImpl.

The structure of the TIE approach is relatively simple. Both the subclassing and object composition techniques are illustrated in Figure 17. The client code would be the same. In the server program, only the statNimGameImpl implementation object needs to be tied to its IDL C++ class.

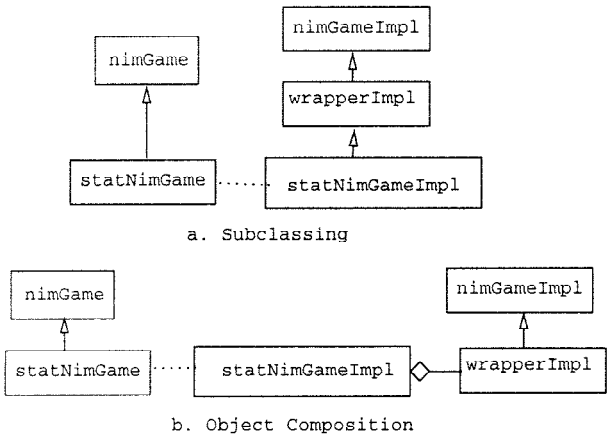


Figure 17. Wrapper structure for interface inheritance—TIE approach with subclassing and object composition.

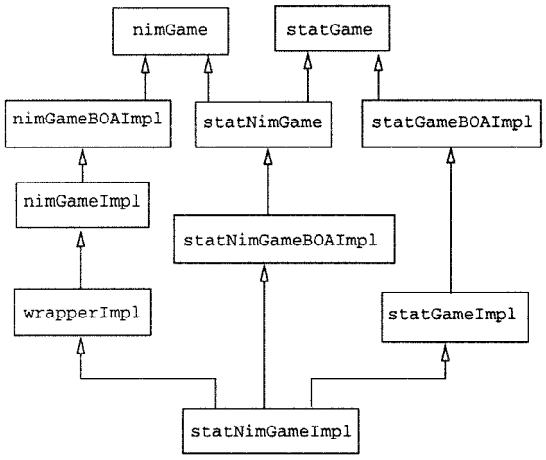


Figure 18. Wrapper structure for separate inheritance—BOAImpl approach with subclassing.

5.3. Separate Interface (Multiple Inheritance)

As shown in Figure 18, the class structure of the BOAImpl approach is somewhat complicated. The statNimGameImpl class works as a wrapper for the same reason described in section 5.1. The referee will ask for an object reference type nimGame_var, and the GUI client will ask for an object reference type statGame_var, but in the server, only the implementation object of the IDL interface statNimGame should be created. The referee program does not need to change. The object composition is not feasible.

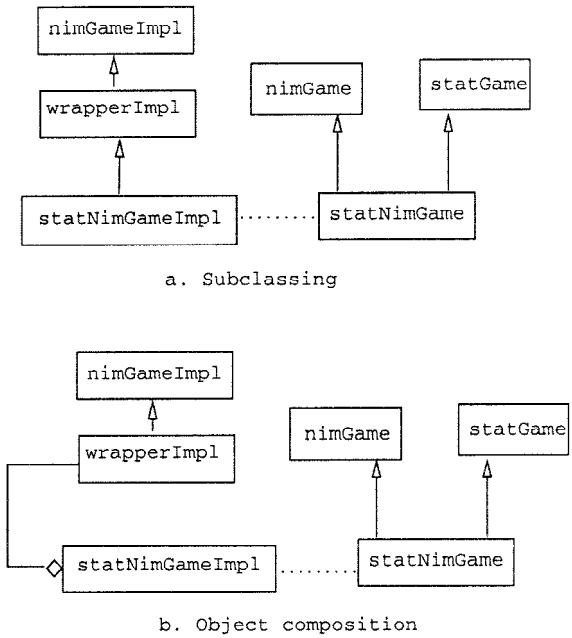


Figure 19. Wrapper structure for separate inheritance—TIE approach with subclassing and object composition.

The last to be mentioned is the TIE approach for the separate interface. The client program would be the same as in the BOAImpl approach. The main server program has to tie the statNimGame type with its implementation object. Figure 19 shows the structure of the TIE approach with subclassing and object composition.

6. Results

In this section we provide examples of the specific metrics for the Nim game and discuss how they were measured.

6.1. Measuring Communication and Service Times

In this section, we describe how we measure the interprocess communication time and service time of the Nim game. Figure 20 shows the overall structure of the system. As stated earlier, we assume that we do not have access to the source code of the programs, and we will not measure time from the client (the referee) perspective. The referee and two players are implemented in C++, and the Orbix ORB is used to distribute the objects. The referee and each of the players are run on separate machines. (The machines used are SUN SPARCstation-4s connected by a LAN.) OrbixWeb is used to implement GUI client

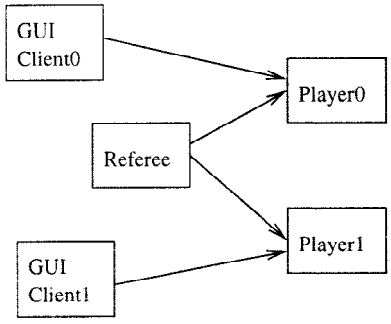


Figure 20. Nim game structure with GUI clients.

in Java. The GUI client periodically invokes the showMeasure() method on the player to get the measured data and display them to the user. Since we are restricted to measuring time on the server side, we can only measure the service time accurately, not the message transmission time. However, if the referee spends minimal computing time for its task and the communication time taken for both players is about the same, we can compute an approximation to the message transition time.

Upon receiving the makeAMove() message, the wrapper of the player will record the time and then forward the message to the player routine which would service the message. When the service is completed, the wrapper will record the time again before returning the result to the referee. The difference of these times is the service time for each makeAMove() request. The sum of these times is the overall time taken for each player routine to finish the game. The followings are the results of one game.

Player0:	Number of messages:	46	
	Total service time:	396.6	ms
Player1:	Number of messages:	45	
	Total service time	28330.2	ms
Total		29830.4	ms

Player0 uses an optimal algorithm (i.e., perfect strategy), while Player1 uses a suboptimal ad hoc algorithm and the measurements support this. The board size was 20. (That is, the number of rows of the Nim game is 20.) Note that the total time taken is from the start to the end of the game. The total time (TT) includes the service time of Player0 (ST0), the message transmission time between the referee and Player0 (MT0), the service time for Player1 (ST1), and the message transmission time between the referee and Player1 (MT1). The objective of the game is to pickup the last object. Player0 has one more message than Player1 because it won the game.

If we assume that the time taken for the referee is negligible, MT0 is approximated by

$$\begin{aligned} TT - ST0 - ST1 - MT1 &= \\ 29830.4 - 396.6 - 28330.2 - MT1 &= 1103.6 - MT1 \end{aligned}$$

If we further assume that $MT0 \simeq MT1$, then

$$MT0 \simeq MT1 = 1103.6 \div 2 = 551.8$$

Average communication time per message is

$$551.8 \div 46 = 12.0 \text{ ms}$$

6.2. Validation of the Technique

In order to validate the data collected using the wrapping approach, we created a version of the referee in which we modified the code to measure actual communication times. In this section, we compare the data from the wrapping approach to the actual communication time measured from the referee.

Within the referee code we measure the time taken for both players. The time taken for Player0 (TP0) is 791.1 ms and for Player1 (TP1) is 28785.8 ms. Therefore,

$$MT0 = TP0 - ST0 = 791.1 - 396.6 = 394.5$$

The average communication per message for Player0 is

$$394.5 \div 46 = 8.58 \text{ ms}$$

For Player1,

$$MT1 = TP1 - ST1 = 28785.8 - 28330.2 = 455.6$$

The average communication per message for Player1 is

$$455.6 \div 45 = 10.12 \text{ ms}$$

The difference between the approximated value (12.0) and the actual values (8.58 and 10.12) is due to the processing time taken for the referee which is not counted in obtaining the approximated value. This error is directly related to the service time of the referee. If we had executed the referee in a faster machine we would have obtained a more accurate approximated communication time with the servers. Our technique allows direct measurement of servers but only approximations for clients. Since the clients are often the application code written last to take advantage of the capabilities of existing servers, it is more likely that we will have access to the actual source code. A future direction of our research will consider how to instrument these clients as cleanly as possible.

6.3. A Second Scenario

The Nim game has a fairly simple communication pattern. If the application we measure has a more complicated communication pattern, it is not always easy to obtain an accurate communication time. Consider Figure 21.

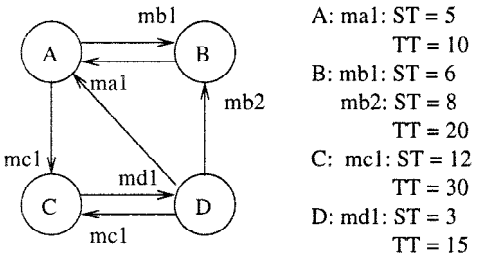


Figure 21. An application with complicated communication pattern.

A, B, C, and D denote tasks running on different machines. We assume that there is one object per task. ma1 is the method defined in the object running on A, and etc. A, C, and D have one method and B has two methods. B and D invoke ma1 on A, and etc. ST is for service time and TT is for total time. The total time for each task is different from others. Each task runs concurrently and some of the tasks may or may not depend on others.

For this type of setting, it is not feasible to calculate communication time taken between tasks. Since we do not check for the source of the message, we cannot distinguish between which tasks the communication occurred. For example, even though we could tell that 10 units of time is taken for the message transmission of mc1, we cannot tell how much is taken between AC and DC. To measure communication time between tasks, we have to have control over the source code and clock time before and after each message invocation.

7. Conclusions

In this paper we have presented a technique for obtaining measurements for components that have been integrated into a CORBA-based system. The technique is non-invasive, requiring no modification of the components' implementations. The technique can be used to directly measure certain attributes of the system and supports the estimation of other attributes. The measurements provide feedback to system designers who are integrating existing applications into a CORBA-based system. The designer may collect information that guides the customization of the stubs and skeletons that provide a layer of indirection in the system. Information can also be collected that guide the distribution of objects.

We intend to extend this work in several directions. First, we will formulate additional metrics that guide the deployment of distributed systems. Second, we will continue to elaborate the instrumentation of the individual components. We have only alluded in this paper to the GUI that is used to present the measurements to the user. This interface will be made more modular so that each object can easily have a tailored interface. Our goal is to provide the maximum in information with the minimum of modification to the system.

Notes

1. This work was partially supported by a grant from TACOM-AMSTA-AQ-SCC, Warren Michigan contract No: DAAE07-97-C-X130.

References

Basili, V. 1992. Software modeling and measurement: The Goal/Question/Metric paradigm. Technical Report CS-TR-2956, Dept of Computer Science, University of Maryland.

Harrison, T., Levine, D., and Schmidt, D. C. The design and performance of a real-time CORBA event service. *OOPSLA 97*, 184–200.

Chatterjeet, S., Bradley, K., Madriz, J., Colquist, J., and Strosnider, J. 1997. SEW: A toolset for design and analysis of distributed real-time systems. *Proceedings of the Third IEEE Real-time Technology and Applications Symposium (RTAS97)*.

Datametrics Systems Corporation and Zitel Corporation. The seven challenges to distributed performance management.

Bond, A., and Hine, J. 1991. DRUMS: A distributed performance information service. *Proceeding of the 14th Australian Computer Science Conference*, Sydney.

Schmidt, D. C., Harrison, T., and Al-Shaer, E. 1995. Object-oriented components for high-speed network programming. *Proceeding of the USENIX Conference on Object-Oriented Technologies*, Monterey, CA.

Corwin, B., and Braddock, R. 1992. Operational performance metrics in a distributed system, Part I: Strategy. *ACM*.

Braddock, R., Claunch, M., Rainbolt, J., and Corwin, B. 1992. Operational performance metrics in a distributed system, Part II: Metrics and interpretation. *ACM*.

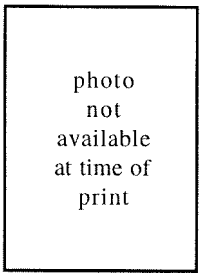
Object Management Group. 1995. *The Common Object Request Broker: Architecture and Specification*. 2.0 ed.

Dr. John D. McGregor is an associate professor of computer science at Clemson University and a senior partner in Software Architects, a software design consulting firm, specializing in object-oriented design techniques. Dr. McGregor has conducted funded research for organizations such as the National Science Foundation, DARPA, IBM and AT&T. Dr. McGregor has developed testing techniques for object-oriented software and developed custom testing processes for a variety of companies. Dr. McGregor is co-author of "Object-oriented Software Development: Engineering Software for Reuse" published by Van Nostrand Reinhold. Dr. McGregor is also co-author of A Practical Guide to Testing Object-Oriented Software to be published by Addison-Wesley. He has published numerous articles on software development focusing on design and quality issues. Dr. McGregor's research interests include software engineering specifically in the areas of design quality, testing and measurement.

Il-Hyung Cho received the B.E. degree in electronics and computer engineering in 1987 from Yonsei University, Seoul, Korea, and the M.S. degree in 1990 from Bowling Green State University, Ohio. He is currently a PhD candidate at Clemson University. His research interests are object oriented software engineering, distributed object systems, and component specification and testing.

Dr. Brian A. Malloy is an associate professor in the department of Computer Science at Clemson University. Dr. Malloy's research extends to several areas of computer science. He has developed techniques for compiler optimization including instruction scheduling, parallelization and cache optimization. He has developed program representations to facilitate analysis and optimization; these representations include program dependence graphs for imperative and object-oriented softare. He has developed several languages for process-oriented simulation including SimCal, for Pascal, and SimPol, for C++. In addition, he has developed techniques for parallelizing discrete event simulation.

Current projects include the construction of a class hierarchy graph for object-oriented programs that maps the structure of the program to intermediate and assembly code representations.



Lowry Curry is a Customer Engineer at Iona Technologies. In 1994 he received a B.A. English from Wofford College, Spartanburg SC. In 1998 he received an MS Computer Science from Clemson University, Clemson SC. While at Clemson he served as a lab instructor for object-oriented design labs. From May 1997–May 1998 he worked for Advanced Automation in Greenville, SC where he helped develop real-time control software for

automated machinery using CORBA. At Clemson University he developed a method to allow monitoring and testing of CORBA applications.

Chanika Hobatr is a PhD student at Clemson University. Her current interests lie in compiling and debugging of object-oriented programs. She received a BS in computer science from Thammasat University in Thailand and an MS in computer science from Vanderbilt University.