WIZCOM: A TOOL FOR SUPPORTING DISTRIBUTED OBJECT ORIENTED PROGRAMMING

Amit S. Kale, Rushikesh K. Joshi
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai - 400 076, India
email: {askii,rkj}@cse.iitb.ernet.in

ABSTRACT

This paper presents the design and implementation of WizCom, a wizard tool for supporting object oriented distributed programming in C++ without the need to use language extensions or special libraries. WizCom generates client-side and server-side code interactively. Server programmer is required to specify the server interface prototypes to the tool and subsequently plug in the method definitions in the server-side code generated by WizCom. Communication code is automatically generated. The WizCom model supports anticipatory message which makes it possible for the clients to invoke messages ahead of server activations. The generated server code can be mixed with other code or modified conveniently.

1 INTRODUCTION

WizCom is a wizard based tool for supporting object oriented distributed programming in C++ [9]. The C++ programming language does not provide the remote invocation mechanism. Hence, a distributed program in C++ needs to use a suitable communication paradigm such as sockets or message queues and provide an encoding scheme for remote method invocation.

Solutions to this problem are available through some of the existing concurrent and distributed programming paradigms. An approach is to use languages that are tailored for concurrent or distributed computing such as Actors [1], Hybrid [8], SR [2], DOWL [3] and COOL [5]. For example, in Hybrid, all objects are active entities and *activity*s are used for concurrent execution. Another approach is to provide a library for supporting activation and communication. Examples of this approach are the *Concurrency* class provided for Eiffel [7], and the

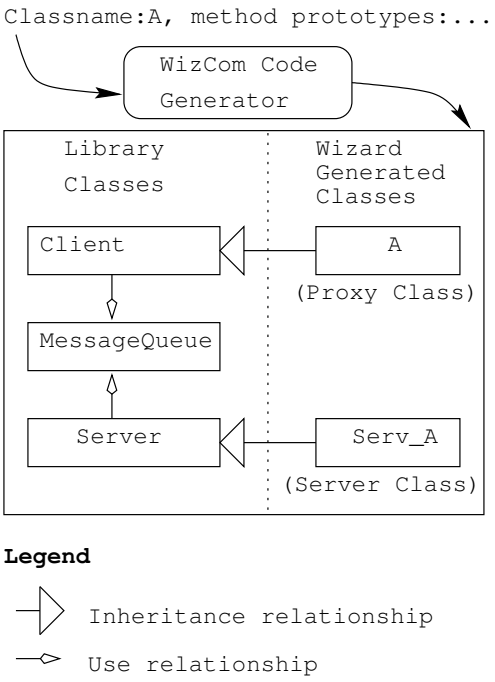

Figure 1: WizCom Interactive Code Generator

Replica class of the C++ based ShadowObjects model [6]. However, the library approach leaves the responsibility of method encoding and method dispatch to the programmer.

The WizCom approach relieves the C++ programmer from the use of language extensions or the difficulties faced in the library approach. WizCom interactively accepts the class name and its method prototypes and generates the skeletons that perform method encoding and method dispatch. For the rest of the responsibilities, it uses its libraries. A server code that is generated by WizCom can be subsequently stuffed with actual method bodies. Other objects required to implement the server functionalities can be conveniently implemented within the generated server code. It is also possible to iteratively evolve the existing server code by addition of new member functions to an existing server implementation.

Rest of the paper is organized as follows. Section 2 provides the design of WizCom. Implementation details are provided in Section 3 followed by a brief discussion in the last section.

2 THE DESIGN OF WIZCOM

The tool contains two programs *genclass* and *addmethod*. The *genclass* program is used for generating initial class template. The *addmethod* program is used to add a method to be exported by the server class. It is possible to evolve a server implementation code over time since

the *addmethod* program is designed to iteratively modify the generated code.

The tool uses its library of two classes called *Client* and *Server* as shown in Figure 1. The interfaces of these library classes are shown in Figure 2. The class *Client* is used for sending method requests and receiving the results, whereas, the class *Server* provides request retrieval and result dispatch. The tool generates two classes, one serves as the *implementation* and the other serves as a *proxy* [4] for an instance of server implementation. For example, in Figure 1, two classes *A* and *Serv_A* are generated by WizCom. The other classes shown in the figure are the library classes. The proxy class *A* uses class *Client* through inheritance and similarly, the implementation class *Serv_A* uses class *Server*.

When an object of class A is instantiated, it can work as a proxy for an actual implementation. WizCom supports two types of binding methods for binding a proxy with an implementation. A proxy can bind with an existing active server implementation by calling the member function *attach*. This binding can be terminated through a call to *detach*. This mode is suitable for client programs. In the other mode of binding, a server implementation is activated first, followed by the binding. This mode is used by server creator programs through a single member function called *activate*. The server creator program can subsequently terminate the server through a *deactivate* member function call.

The *Client* and the *Server* class use the class *MessageQueue* for actual communication of messages and results. Methods that are invoked by clients are deposited in a server message store. Servers asynchronously pick up messages from the server message store and subsequently deposit replies in the client message store. This method of messaging makes it possible for clients to invoke methods on servers ahead of server activation in anticipation. The interface for the MessageQueue class is shown in Figure 2.

Summarily, programmers are required to call only four member functions called *activate*, *deactivate*, *attach* and *detach* on the proxy class in order to initiate or terminate distributed computing mode. Once the distributed computing mode is setup by server and client programs, any further inter-object communication follows the conventional call syntax, thus providing location transparency for server objects and eliminating explicit handling of message encoding or message dispatch by the client and server respectively.

3 IMPLEMENTATION OF WIZCOM

WizCom generates a proxy class and an implementation class for every server class specified. The proxy class is given the same name as the specified name. The implementation class has to be later stuffed with

```
Client
public:
  activate : Creates a server object and attaches to it.
  attach: Attaches to a server object identified by
    given ID.
  detach: Detaches the client from the server
  deactivate: Deactivates the server.
protected:
  remote_invoke: Calls a given server object method
    with given parameters.
Server
protected:
  get_request: Returns a request from queue of method
    call requests.
  send_result: sends result of a method call to
    the caller proxy.
  scheduler: Virtual method that runs a loop which
    fetches requests and calls appropriate methods.
MessageQueue
public:
  send: Adds given message to the message queue
  receive: Receives a message of given type
    from the message queue
```

Figure 2: Methods in library classes

```
#include "Client.h"
class A: public Client {
...
public:
        //@@marker for exported functions
        int m(int i,int j);
};
```

Figure 3: File A.h Client class interface

the actual method implementations by the programmer. As far as the usage of the server object is concerned, it is always through the proxy object. The interfaces and implementations of the proxy and implementation classes are provided in Figures 3, 4, 5 and 6. Code denoted by normal text in these figures is generated at the time of file creation whereas code denoted by bold text is generated by adding a method *m*. Implementations of these classes make calls to member functions inherited from their respective superclasses.

As soon as a server class is activated through an *activate* member call, a *fork* operation is performed to start an active server implementation. The child process assumes the role of the server implementation by starting the method dispatcher called *scheduler* as soon as it comes into existence. The scheduler waits on the server message queue to receive requests from clients. The client requests are targeted to servers identified by their unique names or *uids*. A server picks up a request and deposits the result in the client message queue. The result is identified by the process name or *pid* of the requesting client. A client subsequently picks up its result from its message queue.

```
#include "A.h"
#include "serv_A.h"
...
int A::m(int i,int j) {
struct args_m { int i; int j; } packet;
packet.i=i;packet.j=j;
int ret;
remote_invoke("m",sizeof(packet),
&packet.sizeof(ret),&ret);
return ret;
}
```

Figure 4: File A.C Client class implementation

```
#include "MessageQueue.h"
#include "Server.h"
class Serv_A: public Server {
...
private:
   //@@ marker for exported methods
   int m(int i,int j);
};
```

Figure 5: File Serv_A.h Server class interface

```
#include <string.h>
#include "Serv_A.h"
...
void Serv_A::scheduler(void) {
do {
        ...
        get_request(sizeof(buffer),(void*)buffer);
        if(!strcmp(buffer,"attach")) {
        ...
        }
        if(!strcmp(buffer,"deactivate")) {
        ...
        }
        //@@ marker for exported methods
        if(!strcmp(buffer,"m")) {
        struct args_m { int i;int j; }
        &packet=*(args_m*)data;
        int ret=m(packet.i,packet.j);
        send_result(sizeof(ret),&ret);
        continue;
        }
}
...
}
int conc_A::m(int i,int j){
}
```

Figure 6: File Serv_A.C Server class implementation

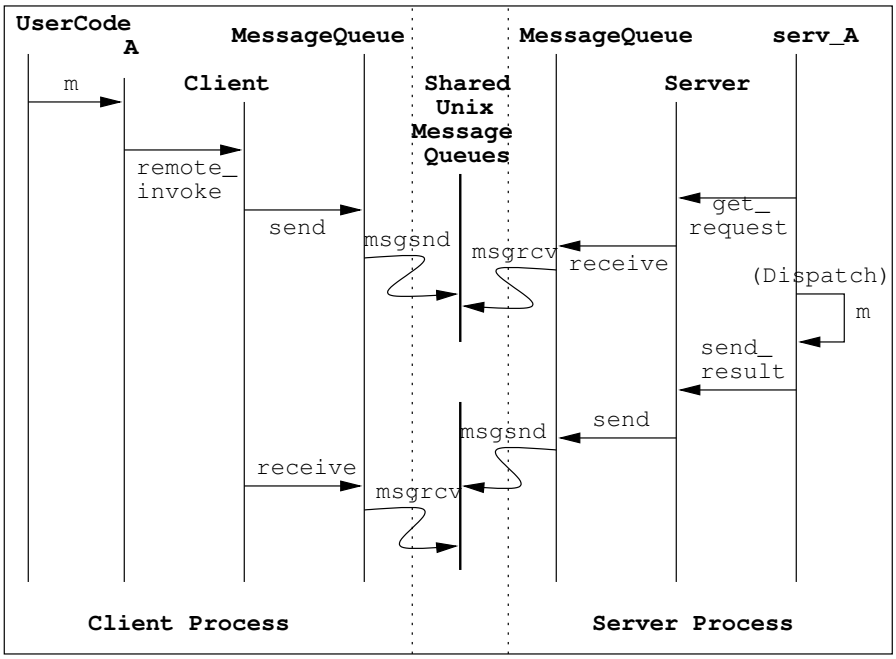

Figure 7: Event Trace of call to A::m

The present prototype implementation of WizCom runs on a single machine and can handle objects distributed in different address spaces on the same machine. It can however be extended to handle objects that are distributed across address spaces on different machines by modifying some of the library classes without having to change the user interfaces.

Figure 7 shows event trace of a successful call to method m in proxy class A . When a method from a client object is called, it further invokes the method *remote_invoke* from the base class. This method invokes the method *send* on class *MessageQueue* that acts as a wrapper to Unix message queues. This call deposits the request into a Unix message queue. The type field in this request message identifies the destination *uid* for this message. The request message also carries an identifier *pid* for the requesting client. It can be noted that the wizard does not support nested objects as arguments to method invocations. The server subsequently receives the request and dispatches it to an actual method implementation. The result is placed in message queue by class *Server*. Client subsequently pick up the result through the member function *receive* in class *Client*.

This mechanism allows the clients to send *anticipatory messages*. Method requests can be placed in the message queue before the server is ready to receive them. The server need not be activated at the time of a client call. Server can eventually read the requests and process it.

The *genclass* program generates four new files for the mentioned class as shown in Figures 3, 4, 5 and 6. Special markers are placed in the

client and server class header files, and also in the server implementation file. The markers are subsequently used by the *addmethod* program for finding place for inserting new methods into the class. Once the markers are placed by the *genclass* program, they remain unchanged. A server implementor can use the implementation file to stuff appropriate implementation code.

4 SUMMARY AND FUTURE WORK

The WizCom tool provides a support for *easy to use* distributed computing in C++. The main advantage of the WizCom approach is that it eliminates the use of extended language features or special libraries required for handling distribution. Standard C++ compilers can be used to compile the generated code. The classes generated by WizCom can be easily integrated with other code. It is possible to evolve an existing class without disturbing its interface. Communication through message queues allows asynchronous operations inclusive of asynchronous activations that leads to the anticipatory message invocations. Some of the classes in the tool can be evolved to handle distribution of objects across address spaces on multiple machines. Currently, clients have to wait to receive their replies causing a busy wait situation. Asynchronous result receive primitives can be provided to solve this problem.

The code for the tool is available on request.

REFERENCES

1. Agha G.H. (1986), "ACTORS: a model of concurrent computation in distributed systems", MIT Press, Cambridge, Mass.

2. Andrews G.R. et al. (1988), "An overview of the SR language and implementation", ACM TOPLAS, Vol. 10.

3. Caromel D. (1993), "Towards a method of object oriented concurrent programming", Communications of ACM, Vol. 36/9.

4. Gamma E. et al. (1994), "Design Patterns", Addison-Wesley.

5. Lea R., Jacquemot C., Pellevesse E. (1993), "COOL: System Support For Distributed Programming", Communications of ACM, Vol. 36/9.

6. Joshi R. K., Ongole R., Janaki Ram D. (1998), "A programming model for control replication in object oriented distributed systems", Communicated for publication.

7. Karaorman M. (1992), "Introducing concurrency to a sequential language", Communications of ACM, Vol 36/9.

8. Neirstrasz O.M. (1987), "Active Objects in Hybrid", ACM SIGPLAN Notices, Vol 22.

9. Stroustrup B. (1995), "C++ programming language, 3rd Ed.", Addison-Wesley.