ShadowObjects: A Programming Model for Service Replication in Distributed Object Systems

Rushikesh K. Joshi
Department of Computer Sc. & Engg.
Indian Institute of Technology
Bombay-400076, India

O. Ramakrishna, D. Janaki Ram
Department of Computer Sc. & Engg.
Indian Institute of Technology
Madras-600036, India

E-mail: rkj@cse.iitb.ernet.in, oram@lotus.iitm.ernet.in, djram@lotus.iitm.ernet.in

Abstract

With the growing emphasis on distributed applications, sophisticated programming models for handling replication become important. Control replication as an issue has so far been considered in the larger context of distributed systems such as ISIS and Arjuna. We present a programming model called ShadowObjects for programming replicated services to cater to the needs of control replication in distributed object oriented systems. The ShadowObjects model provides primitives for building highly available and redundant services. Replication in ShadowObjects is encapsulated and a replica access control mechanism is provided. Messages accessing the services exported by a server object can be captured on-the-fly, and can be appropriately scheduled on the replicas. The ShadowObjects model can be used to develop applications for distributed systems.

Keywords: control replication, distributed object systems, highly available services, redundant services, replica access control

1 Introduction

Replication has been widely recognized for its ability to provide high availability, parallelism and fault tolerance in distributed systems. Control replication deals with replication of control flow. An example of control replication is the replicated workers model in distributed systems [1]. Another example of control replication is a replicated stateless service. This paper deals with mechanisms to program control replication that are

used to build replicated services. The paper does not address the issues of data replication. The replicas in ShadowObjects are independent servers and hence do not maintain consistency among their local states.

Various existing approaches to distributed programming systems such as ISIS [2], Arjuna [10] and Charm++ [7] provide support for replication. In these systems, replication is provided as part of larger systems that are built for applications such as database systems or parallel computations. Whereas, ShadowObjects is a highly specialized programming model tailored only for the needs of control replication. ShadowObjects treats the replicas as servers exporting services and not as data objects that are replicated for availability.

While replicated data items in ISIS, active and passive replicas of Arjuna, and design patterns such as Field Partitioning [15] provide data-centric approaches to replication, Charm++ [7] and Interface Groups of ANSA [11] provide a group communication model for replicated objects that can be used for control replication. In Charm++, a branched chare forms a single handler for the replicas, whereas interface groups provide one interface handle for multiple replicated interfaces. While Charm++ is targeted for a distributed programming environment, the ShadowObjects model discusses the issues relating to replication of independent servers in distributed systems. Horus[3] is another example of a group communication protocol which supports groups of processes. In contrast to *group-oriented* replication, where a group addressing abstraction is primary, the ShadowObjects model provides a *service-oriented* model for replication. Replicated services can not only be used in a group to obtain redundant services, but they can also be converted into independent servers.

Novel features of ShadowObjects include the access control mechanism and on-the-fly message filtering. The other features provided by ShadowObjects are vote-based N-modular-redundant services and highly available remote calls. These mechanisms of ShadowObjects are intended for building replicated services. It is possible to adapt the mechanisms of ShadowObjects for a more general distributed programming language.

Most of the features of ShadowObjects are provided using a library approach. This approach makes ShadowObjects a general purpose utility for building distributed services. ShadowObjects was implemented on a network of Sun workstations for the C++ programming language [13]. This paper describes examples and some interesting features

of the implementation at appropriate places.

2 The Mechanisms of ShadowObjects

In the ShadowObjects model, an object that replicates is called a *replicate agent*. A replicate agent obtains the replication behavior via inheritance, and replicates itself by making a member function call. Replicas provide the same interface as provided by the replicate agent. Replicas can be created on remote machines. Remote client objects may access services of the replicas. Replicas are asynchronous and independent. This section describes various features of the ShadowObjects model.

Encapsulation of Replication

Replication may be an object's internal decision hidden behind its public interface. For example, a server object might replicate itself internally to cater to an additional load without the clients having to know about the replication.

Replica Access Control Mechanism

A replicate agent may hide the replicas as its internal implementation. In such a case, the access to replicas is governed by the replicate agent itself. A construct called *Capture* is provided to filter messages to the replicate agent and to route them to its replicas. Replicate agents provide their own capture specifications.

A direct access to a replica may also be made by remote clients. To provide this open distributed access, a replicate agent performs an *expose* operation on its replica. During run time, it is possible to switch between direct access and access through the replicate agent. As an example, a resource manager controlling a pool of replicated services may be modeled as a replicate agent making the replicate-agent's address accessible only to privileged users. If the inflow of the privileged requests is low, the replicate-agent may make available some of its replicas to requests from a wider domain.

Support for N-Modular-Redundant Programming

This feature provides redundancy to the services of the replicate agent. Using this mechanism, it is possible for a replicate agent to simultaneously execute multiple invocations of a service on its many replicas. One of the available results can be voted to be the

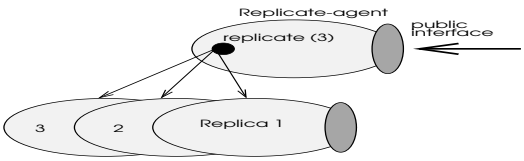

Figure 1: The Basic Replication Model

final result. The voting mechanism is not only useful for redundant services but also for applications that return varying results. As an example, a service that reports the load on its host machine can be modeled using NMR services. The service can be made to return the most lightly loaded machine's address for use by an independent parallel application such as a PVM [14] task. The voting routine has a specific semantics and the routine is supplied by an application.

Highly Available Remote Services

Access to the highly available remote services in ShadowObjects is based on the dynamic binding of remote calls to servers. When a client accesses a highly available service through a specially designed interface, the destination of the invocation is left unspecified. At the time of actual invocation, the system chooses one of the available replicated services for the actual execution of the service requested. As an example, a news-reader service may be replicated at multiple locations and late binding can be applied to select the actual reader.

3 The Basic Replication Mechanism

A *replicate agent class* inherits a standard superclass called *Replica*. This approach is similar to earlier library approaches for extended functionalities such as for providing concurrency in Eiffel [8], and atomic transactions and persistence in Arjuna [10].

An instance of class *Replica* may create its own replicas by making an internal call to a function called *replicate()* that is defined in the superclass *Replica*. Figure 1 shows the basic replication mechanism. The member *replicate()* takes as its arguments, the number of replicas and a list of machines on which the replicas are to be created.

The superclass exports an interface that consists of mechanisms for controlling the access

```
class Eval : public Replica {
private:
        Eval (int);
        serv_add ();
        serv_mult ();
public:
        float add (float, float);
        float mult (float, float);
}
```

```
Eval :: Eval (int copies) {
        MachineList ML;
        // build the list by invoking calls on ML
        ...
        bindService (0, serv_add);
        bindService (1, serv_mult);
        replicate (copies, ML);
}
main () {
        Eval E (4);
        ....
}
```

Figure 2: An Example of Server Replication

to replicas and mechanisms for redundancy. When an object replicates, a replica receives the state of the object just prior to replication. The control in the creator of the replica resumes, whereas the replica is now a shadow object of the creator ready to provide services. Since replication is achieved by making a function call, replication can be dynamic.

A replicate-agent has to define *message service routines* to be invoked by the replicas for processing the messages received. A service is initiated by making a call to the member function *bindService()*. This call binds each message service routine to an identifier. Messages to replicas use this identifier to access the services.

An example program in Figure 2 shows a C++ based implementation of replication of a server. The server object *E* belonging to class *Eval* is modeled as a replicate agent. It exports two functions and is replicated to obtain four replicas. The replicate-agent defines message service routines *serv_add()* and *serv_mult()*, and binds them as services of the replicas before making a call to *replicate()*.

A replica becomes an active object immediately after its creation. Once a replica becomes active, it enters the service loop where it waits for messages from the replicate-agent and invokes the service routines bound earlier by the replicate-agent.

In Figure 2, the member function *replicate()* receives an argument which specifies the number of replicas to be created. Another attribute is the placement specification for replicas specifying the nodes on which the replicas are to be created.

3.1 Creation of remote replicas

By default, the replicas of an object are named by numbers starting from 0 and onwards, and the place for the replicas is the same machine as that of the replicate-agent object. Remote replicas are created by explicitly stating the set of remote machines.

The ShadowObjects model implements two mechanisms for creation of replicas on remote machines. In the first approach, a replica is created on the same machine and migrated to its destination using process migration. After the migration, communication links are established. This approach has the advantage that it does not need a translation stage. However, process migration fails when heterogeneous architectures are involved.

The second approach is the *RIB (Remote Instruction Block)* code migration approach as implemented in Object-Based Subcontracting [5]. In this approach, the blocks that need to be activated on a remote machine need to be built at compile time from the source program and assembled as an RIB. The RIBs may be kept in executable formats for homogeneous computing. Source code migration with remote compilation may be used for heterogeneous architectures.

Process migration is costlier than the RIB approach in terms of execution time. However, the RIB approach has an additional overhead of a translator stage for building the RIBs. The current implementation of ShadowObjects supports a low level mechanism that accepts prewritten RIBs in a standard RIB format.

3.2 Writing message service routines

The message service routines are executed when messages are delivered to a replica. A service routine performs three tasks. It receives arguments from the caller, calls the desired service using these arguments, and subsequently saves the results for future use. Figure 3 shows the definitions of the two service routines as declared in Figure 2. The functions in ShadowObjects need to use predefined messages for receiving and sending the arguments. The following functions are used for writing the message service routines for use by the shadow objects:

1. *rcvMsgFromCaller (void *buffer, int sz)*

The replica receives the message of *sz* bytes into the specified *buffer* space.

```
void Eval :: serv_add () {
float x, y, z; // locals
        rcvMsgFromCaller (&x, sizeof x);
        rcvMsgFromCaller (&y, sizeof y);
        z = add (x, y);
        saveResult (&z, sizeof z);
};
```

```
void Eval :: serv_mult () {
float x, y, z; // locals
        rcvMsgFromCaller (&x, sizeof x);
        rcvMsgFromCaller (&y, sizeof y);
        z = mult (x, y);
        saveResult (&z, sizeof z);
};
```

Figure 3: Defining Message Service Routines

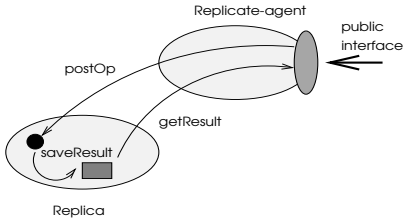

Figure 4: Accessing the Services of a Replica

2. *saveResult (void *result, int sz)*

The result of the current computation, as identified by the buffer space *result* of *sz* bytes is saved into an internally managed result space. This makes the calls to service members from the replicate-agent non-blocking. A result can be pulled out of the result space at a later time by the replicate-agent by using a predefined function *getResult()*.

3.3 Calling the services of a replica

Figure 4 depicts the mechanism for accessing the services of a replica. The services are called in a non-blocking fashion. The replica stores the results until the results are fetched by the replicate-agent. A particular service in a replica can be called with a call to *postOp()*, which posts an *operation request* to the replica. It makes a non-blocking call to a service routine of the replica. The *postOp()* function has the form:

int postOp (Name replica_name, int service_id, arglist..),

Where *replica_name* is the name of the replica to which the message is being sent, and *service_id* is the identifier for the service desired. The *arglist* has the form $size1, \&arg1, size2, \&arg2, ..., T$. It specifies a list of pointers to the arguments to be

```
main ()
{
int rid1, rid2;
float a=..., b=...;
float res1, res2;
      Eval E (2); // create 2 replicas
      rid1 = E.postOp (0, 0, sizeof a, &a, sizeof b, &b, T); // call service 0 of replica 0
      rid2 = E.postOp (1, 1, sizeof a, &a, sizeof b, &b, T); // call service 1 of replica 1
      E.getResult (rid1, sizeof res1, &res1);
      E.getResult (rid2, sizeof res2, &res2);
      cout << res1 << "\n" << res2 << "\n";
      E.destroy (ALL);
}
```

Figure 5: Function main() using Replication

passed to a service along with their sizes. A system defined terminator T specifies the end of the list. The call returns immediately with a *request identifier*. This identifier can be used later to obtain the results of the execution. This mechanism is similar to the promises of Argus [9] used for asynchronous calls.

3.4 Obtaining the results

The results of a particular operation can be collected by making a call to *getResult()*:

*getResult (int request-id, int sz, void *result)*

The first argument specifies the *id* of an earlier posting. Argument *sz* specifies the size of the expected result. The result is deposited at the buffer space specified by the *result* pointer.

Figure 5 shows the *main()* function in which replicas are created and their services called using the features discussed until now.

4 N-modular-redundant programming

This mechanism provides language support for N-modular-redundant programming [12]. An application can request multiple invocations of a particular service and subsequently select one of the available results through a voting mechanism. A variant of the function *postOp()* called *NMRpostOp()* is provided:

int NMRpostOp (int redundancy_number, int service_id, arglist..),

This member function takes the redundancy number as an argument instead of *replica_name* as in *postOp()* function. It returns an *id* on which the voted result can be collected.

The superclass *Replica* defines a virtual member called *NMRvote()*. A derived replicate-agent class that uses the NMR facility must define this virtual member. *NMRvote()* is called automatically by the *NMRgetres()* function, which obtains the result of an earlier NMR posting. *NMRgetres()* internally calls *NMRvote()* whenever a new result becomes available. The results form the arguments for successive *NMRvote()* invocations. When the status of all executions is available, an additional last call to *NMRvote()* is made with an argument *VOTE_NOW*. At this invocation, *NMRvote()* must return the agreed-upon value. This value is returned as the return value of the *NMRgetres()* call. It is also possible to return the final result after at least one replica returns the result successfully. In this case, results from other invocations are discarded. A final return value is distinguished from a pending return value by using a special return value called *PENDING_RESULT*. Applications must provide appropriate voting routines that follow these semantics.

5 The Replica Access Control Mechanism

This mechanism makes it possible to provide centralized or distributed access to the replicas. The replicate-agent regulates access to the replicas. By default, the replicas are hidden behind the public interface of the replicate-agent. The *Capture* construct can be used to capture the messages to replicate-agent on-the-fly and to route them appropriately to its replicas.

A replicate-agent can expose its replicas leading to distributed access. In the case of exposure, outside objects in the system can communicate with the replicas bypassing the replicate-agent after they undergo a *link* operation.

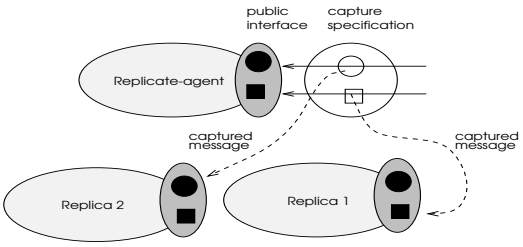

Figure 6: Call Capturing

```
class Eval : public Replica {
captures;
private:
        void serv_mult ();
        // ... members for round robin
        // scheduling algorithm
public:
        Eval (int);
        float mult (float, float);
};
Eval :: Eval (int copies ) {
        ...
        bindService (0, serv_mult);
        replicate (copies) ;
};
...
```

```
Eval :: Capture {
float mult(float x, float y)
@{
        int rid, location;
        float res;
        location = RoundRobin ();
        rid = postOp (location, 1, sizeof (x),
                        &x, sizeof (y), &y, T);
        getResult (rid, sizeof res, &res);
        return (res);
}@
};
```

Figure 7: A Capture Specification

5.1 Centralized access through *hide*

Centralized access is the default choice made by a replicate-agent. In the case of hiding, other objects in the system know only of the replicate-agent and send messages to the replicate-agent alone. Thus the replicate-agent provides a single interface to all the replicas and internally handles the scheduling of requests to them. The *Capture* code captures the messages sent to an object before they are actually delivered to it. It is a static form of the more general filter objects [6] that manipulate messages on-the-fly. The *Capture* code is used for capturing messages sent to a replicate-agent and for scheduling them on the replicas instead of processing them at the replicate-agent itself. Figure 6 shows a replicate-agent that performs call capturing and scheduling. A call to any public member function of the replicate-agent can be captured and routed to one of the replicas.

An example *capture specification* is shown in Figure 7. A capture code can access the

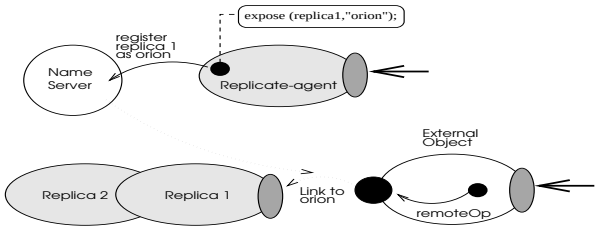

Figure 8: The Expose Mechanism

arguments to the member function that it captures. Local variables can be created within a capture specification. A capture code is enclosed within capture code markers as shown in the figure.

The capture construct has an additional overhead that it requires a translation stage. Alternatively, a capture code could be directly introduced into the code of the function that it captures, thereby eliminating the translation stage. However, this would merge the functionalities of the message scheduler and the actual message processing code. Capture provides a better abstraction for separating message control from message processing.

5.2 Distributed access through *expose*

Distributed access to replicas can be achieved by exposing them to objects on other nodes. After the exposure, the replicas can be directly contacted by other objects. The expose mode makes the replicas independent and allows them to be accessible from a wider domain. It also removes the bottleneck of the replicate-agent since the replicas are independently contacted in expose mode.

Figure 8 depicts the expose mechanism of the ShadowObjects model. The function *expose()*, is called by the replicate-agent to expose its replica. A new system-wide name is given to it for further referencing by other objects in the network. A name server maintains the system-wide names. An exposure can be withdrawn with a call to function *unexpose()*. If an *unexpose()* call is made to a replica when it is in the midst of servicing a request, the requested service is completed first. No more direct requests are accepted after an *unexpose()* call, and the exposure is withdrawn.

```
class Interface : public Link { ...
public:
        float remote_add (float, float);
        float local_mult (float, float) {...};
};
int Interface :: remote_add (float a, float b) {
int rid;
float result;
        rid = remoteOp (0, sizeof (float),
                        &a, sizeof(float),&b,T);
        getRemoteResult (rid, sizeof (float),
                        &result);
        return (result);
};
```

```
...
main () {
Interface i_obj;
        i_obj.link ("orion");
        cout << i_obj.remote_add (...);
        // a remote operation
        cout << i_obj.local_mult (...);
        // a local operation
        i_obj.unlink ();
}
```

Figure 9: Linking an Exposed Replica

5.3 Linking to remote replicas

To communicate with a replica, a remote program creates an *interface object*. This object is created by inheriting a system defined superclass called *Link*. The *Link* class defines members to build interface objects that import remote services.

Figure 9 shows a program specifying an interface object which links and communicates with a remote replica. In the figure, class *Interface* defines the interface object by inheriting class *Link*. A *remote_add()* function has been defined for calling a service of a replica. The *local_mult()* function executes locally. When the object *i_obj*, an instance of the *Interface* class, invokes the inherited *link()* member function, the specified replica *orion* is linked with *i_obj*. It enables *main()* to send a message *remote_add()* to *i_obj*. At the end of the program, an unlink is performed.

The public members of the *Link* class are described below.

1. *int link (char *name)*

A call to *link()* links the named object to a replica. The replica can be specified by argument *name*. A positive integer return value specifies that the linking is successful.

2. *int remoteOp (int operation-id, arglist..., T)*

The function sends a message to the replica that has been linked. The arguments are similar to those in *postOp()* member function provided for hidden replicas. The call returns with a request identifier.

3. *getRemoteResult (int rid, int sz, void *result)*

It blocks the caller for obtaining the result of a previous operation invoked on a replica. The identifier of the operation and the space for holding the result is specified. By checking the return value of *getRemoteResult()*, a failure due to a crash or an unsuccessful operation due to an *unexpose* can be detected.

4. *unlink ()*

The interface object can unlink itself from the replica by a call to *unlink()*. The call takes no arguments. A previously linked interface object cannot link itself to a new object until it unlinks by an *unlink()* call.

The differences between proxies [4] and interface objects are noteworthy. Proxies directly correspond to their actual objects, whereas, interface objects are only links to remote servers. Interface objects can be linked dynamically to objects of different types unlike proxies which are meant to handle requests only for specific types. Remote method invocation through interface objects is performed by two independent calls, one for making the member function invocation and the other for obtaining the result subsequently. In between these two activities, the client may perform another useful computation. The interface object shown in Figure 9 implements an RPC-like synchronous interaction.

6 Highly Available Distributed Services

ShadowObjects provides a mechanism to exploit the highly available replicated services. Special types of links called *highly available links* are supported to enable clients to take advantage of high availability. A client of a highly available service does not bind to a particular replica, instead it relies on late binding.

Figure 10 depicts this late binding mechanism. If the linking of an interface object is performed with a member function *HA_link ()*, the link is not bound immediately, unlike the ordinary link() call. The *HA_link ()* call binds the link to a family of replicas and not to a specific replica. When a *remoteOp ()* call is made, the specific link binding is performed. Out of the available replicas, the system chooses a suitable replica and executes the requested service on it. This mechanism makes it possible to exploit the high availablily of distributed services. Since late binding is performed, unavailability

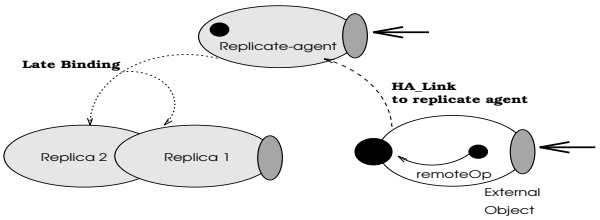

Figure 10: Late Binding in ShadowObjects

due to failures can be tolerated. Moreover, it is possible to equally distribute the load among the replicas, if highly available services are used.

7 Conclusions

We presented ShadowObjects, a programming model for building replicated services in the context of control replication in distributed systems. The mechanisms of ShadowObjects constitute encapsulated dynamic replication, replica access control, N-modular-redundant services and highly available services. Both distributed and centralized access to replicas are supported. A capture construct is introduced to capture on-the-fly the messages sent to a replicate-agent, and to schedule them on the replicas. The mechanisms of ShadowObjects are implemented in C++ on a network of Sun workstations.

Acknowledgement

We thank the anonymous reviewers for their efforts in providing valuable suggestions and comments, which have resulted improvements in this work.

References

[1] G. R. Andrews, Paradigms for Process Interaction in Distributed Programs, *ACM Computing Surveys*, **23**(1): 49-90; March 1991.

[2] Kenneth P. Birman, The Process Group Approach to Reliable Distributed Computing, *Communications of the ACM*, **36**(12): 37-53; December 1993.

[3] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis, Horus: A Flexible Communication System, *Communications of the ACM*, **39**(4): 76-83; April 1996.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Reading, Addison Wesley, 1995.

[5] Rushikesh K. Joshi, D. Janaki Ram, Object-Based Subcontracting for Parallel Programming on Loosely-Coupled Distributed Systems, *Journal of Programming Languages*, 4 (1996), 169-183.

[6] Rushikesh K. Joshi, Vivekananda N., D. Janaki Ram, Message Filters for Object Oriented Systems, *Software-Practice and Experience*, Vol. 26(6), June 1997, 667-699.

[7] Laxmikant Kale and S. Krishnan, Charm++: A Portable Concurrent Object Oriented System Based On C++, *In 8th Annual Conf. on Object-oriented Programming Systems, Languages and Applications, ACM Sigplan Notices* **28**(10): 91-108, Oct. 1993.

[8] Murat Karaorman & John Bruno, Introducing Concurrency to a Sequential Language,*Communications of the ACM*, **36**(9); Sept. 1993, 103-115.

[9] Liskov B. and Shrira L., Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems, Programming Language Design and Implementation, SIGPLAN'88 Conference, 260-267.

[10] Mark C. Little, Santhosh K. Shrivastava, Object Replication in Arjuna, ESPRIT Basic Research Project 6360, University of Newcastle, UK.

[11] Ed Oskiewicz, Nigel Edwards, A Model for Interface Groups, APM.1002.01, ANSA, Cambridge, U.K., May 1994.

[12] S.K. Shrivastava, Replicated Distributed Processing, In Proceedings of the conference on Networking in Open Systems, LNCS-248, 1987.

[13] B. Stroustrup, The C++ Programming Language, 2nd Ed., Addison-Wesley, 1991.

[14] V.S. Sunderam, PVM: A Framework for Parallel Distributed Computing, *Concurrency: Practice and Experience*, **2**(4): 315-339; December 1990.

[15] C. Weir, Using Replication for Distribution, *Object Designers Ltd.*, August 1995.