Replication in Spring: A New Subcontract

Joshua S. Spiewak

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-95-16
May 1995

Replication in Spring:
A New Subcontract

Joshua S. Spiewak

Department of Computer Science
Brown University

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University

May 1995

Professor Thomas W. Doeppner
Advisor

1 Introduction

It is impossible to jump into the meat of a paper without understanding its title. And so, by way of introduction, I will explain what meaning Replication in Spring: A New Subcontract holds. First I will give some background as to what Spring is, other than a season. Next I describe what a subcontract is and give an example. Finally, I will begin to describe replication, and its connection to Spring and subcontracts. My description of replication will continue through this paper as the issues of how to implement it, and the details of the project are discussed.

1.1 Background

Spring is an experimental development system consisting of an object-oriented, distributed operating system and tools to build distributed applications within this environment. Applications generally follow the client-server model where servers provide functional objects for client use. In order to express the client interface to these server provided objects, Spring provides its own interface definition language (IDL) and a set of tools for generating both the shell of the server implementation, and remote procedure call (RPC) stubs for both the client and the server. These tools are **contoimpl** and **contocc** respectively. The RPC stubs generated are generic, that is, they do not use a particular RPC mechanism. Instead, these stubs make use of another layer called the subcontract. This name derives from the interface being called the contract, i.e. the contract between the client and the server. It is the subcontract that implements the invocation portion of RPC mechanisms. The goal of separating the subcontract from the stubs was to allow different RPC mechanisms to be easily added to Spring, and for application authors to be able to make use of different subcontracts relatively easily.

Every subcontract must provide a standard interface, of which the generated stub code makes use. The interface consists of a subcontract server class, a local class, and a client class. The server class is instantiated with every server side object. It provides handling for invocations by the client. A subcontract's local class, at a minimum, provides a conversion routine that returns a fat pointer, also called an object reference, to the implementation of an object. What actually occurs is, of course, dependent on the subcontract. And lastly, the client class provides marshalling routines, one to copy an object and the other to consume it, as well as unmarshalling and invoke routines[1]. Again, the information actually marshalled and unmarshalled depends on the subcontract, but generally speaking some mechanism to allow the client to invoke functions on the implementation is sent.

The most basic subcontract is aptly named simpleton. It meets all of the requirements of subcontracts and no more. The common subcontract, used by the majority of objects in Spring, is the singleton subcontract. It improves on the simpleton subcontract by expanding the role of the subcontract local class. Rather than blindly returning a fat pointer that uses the client class, as does simpleton, the conversion routine returns a fat pointer using the local class. The benefit is that any invocations on the fat pointer while it is still in server space do not need to travel via the normal invocation route of marshalling the arguments and placing the call through a door. Instead,

1. The marshalling and unmarshalling routines are for object references only. Built-in types are not marshalled by subcontracts.

a fast local call can be made. Most of the other, more complicated, subcontracts also make use of this idea. Figure 1 depicts the three distinct layers of functionality on both the client and server sides of a distributed Spring application.

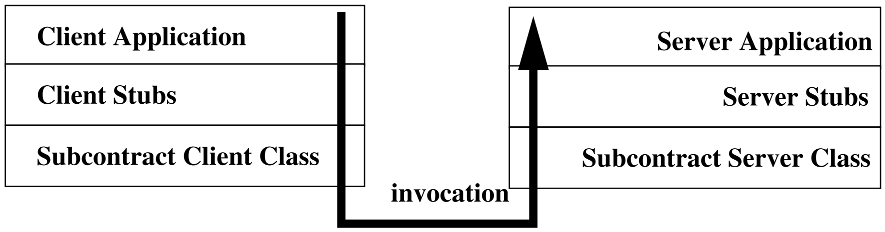

Figure 1: Components of client and server, showing a method invocation

1.2 Overview of Replication

Clearly, once applications moved into the client-server model, replication became a very desirable feature. If there is only one server for multiple clients, and it crashes, all the clients are unable to access their information. Further suppose part of the network breaks down. Without replication, clients are again without their information. Therefore we have multiple servers. Other reasons for replication's desirability are load balancing and fast communication between particular machines. Thus it seemed natural that there should be some form of object replication in Spring, since distributed programming comes so easily. Further, this idea of replication fits in well with the notion of subcontracts. In fact, a very simple replicative subcontract, called replicon, was written by the creators of Spring. However, it was not kept update and was eventually completely discarded. The final motivation for creating a subcontract that allows replication was to discover how easy or difficult it truly is to create new subcontracts in Spring and make use of them.

There were very few requirements of the replication subcontract (named replcon, as opposed to replicon), and while they are presented here, their full description and rationale are left for the next sections. First, objects were divided into two categories: master and slave. A master object is the writable copy of an object, while a slave object is a read-only copy. So there is a server that contains the master replica of an object, and other servers that have slave replicas. Further, each client object reference, also referred to as a fat pointer, must have associated with it information to allow the subcontract to contact those servers that have a copy of the implementation of the object. Lastly, the subcontract needs some means of distinguishing between operations that change the state of the object and those operations which do not alter the state of the object. This is the notion of read and write operations. Knowing what type of operation is being invoked will allow the subcontract to decide whether it should contact the master replica of the object or whether any replica will do. Given this server information and knowledge of which operations need to contact the master object and which may contact any replica, a mechanism for invocations and maintenance of the server list must be provided. This mechanism is known as management objects, and described in full in Section 3. Given these minimal requirements for the subcontract, the actual responsibility for the replication and updating of objects is left to the server. The subcontract needs only to provide the ability for server objects to identify themselves, and correctly marshal and unmarshal this information so the client subcontract may use this for invocations.

2 Design

Over the course of designing this system of replication, many questions were raised and decisions made. What follows is first a brief overview of the design followed by a chronicle of the design process and a discussion of the rationales behind the design decisions.

2.1 Overview

Earlier, the replcon subcontract was described as having information that allowed it to contact servers in order to invoke methods on objects. This information consists of three things: an object identifier, a list of servers, and a list of write operations. The subcontract is given this information when the object is created. The replication system distinguishes between the master replica and slave replicas of objects. There is only one master replica, and all write operations are directed to it, thus the need for knowing which are the write operations.

In addition to the replcon subcontract, there are two other pieces of the replication system: the management object and the server. Figure 2 shows the general structure and relationship of the replication system. This figure will be used later with more detail to explain the internals of the system and invocations.

The management object is a representation of a replicative server. It is an object interface to the server that allows clients of the management object to invoke methods on the server. The functionality provided by the management object is used both by the subcontract and servers other than the one represented by the management object. Its interface allows servers to replicate and update objects to other servers. These methods transmit an object's state and an object identifier. The other part of the management object's functionality allows the subcontract to get a handle to a particular replica of an object as well as update the subcontract's information on the current set of servers for an object. In both methods, the subcontract must make use of an object identifier. The handle is used to invoke methods on the object.

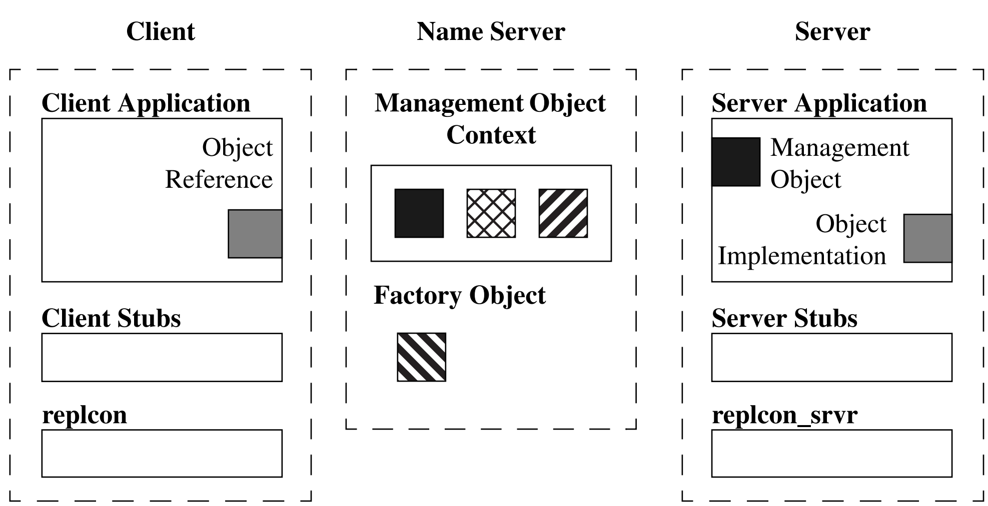

Figure 2: Structure of replication system

Each server is given a unique name. Using this unique name, servers create and bind their own management object into the namespace. This makes it possible to think of a set of servers as a list of paths in the namespace. This idea is used both by the server as it performs replication of its objects and by the subcontract as its information about the servers on which an object can be found. Also using its unique name, servers create an identifier for each object created on the server. This identifier is given to the subcontract for use when contacting servers.

2.2 Issues

There were several questions and problems that led to the creation of what are called management objects. First, there was a need for a mechanism to actually replicate the objects. That is, some means of transmitting the state of objects from one server to another. Also needed was a means of determining the current set of running servers. This was necessary in order to discover new servers to send replica objects. The second problem could be solved by maintaining a list of active servers which was composed of path names. But paths to what? The answer was to create management objects to provide the replication mechanism and to use paths to their location in the namespace to maintain the list of active servers. This led to requiring that each server be identified by a unique name, so that when the management object was bound into the namespace there would not be conflicts.

One of the very first questions posed was whether the system would be replicating objects or entire servers, and exactly what would be replicated in either case. If replication is on the scale of the entire server, should every object in a server be replicated wholesale? And if so, should a server be able to specify if it can be replicated? If, on the other hand, objects were to be replicated on an individual basis, was *every* object to be replicated? The initial plan was to have a single master server and multiple slave servers. Objects would be replicated from the master server to the slave servers. All operations that altered the state of the object would have to use the copy of the object on the master server. These operations are called write operations, while operations not affecting the object's state are read operations. This model made certain things simple because only the master server had work to do, but the model suffered because the servers would have to have different implementations. Instead, it seemed better to have generic servers and distinguish between master and slave replicas of the object itself.

In order for objects to be updated and so the replcon subcontract may refer to all replicas of an object by the same means, there must be some way of referring to objects by a handle. The servers may then store their objects and use the handle as a key to look them up when necessary. Because the object will exist on several servers, and the subcontract will refer to the object by the same handle on each server, obviously the handle must be unique not just to a particular server, but across all servers. Each object is therefore referred to by an identifier constructed from the name of the server on which it was created and a numeric tag assigned by the creator server. The combination of the unique server name and tag number that increases with each object created gives each object its own unique name.

When a server crashes and the master objects it contained are no longer available, it might be necessary for some replica of those objects to become the master. In this case, it seemed natural that the name of the object would then be changed. Since originally it contained the name of the server on which it was created, i.e. the location of the master replica, the base of the object's new name would be the new location of the master replica. However, this approach caused problems in updating any bindings of the object in the namespace as well as any clients holding references to the object. Any time the master object changed servers, it would change names and the binding in the namespace would have to be rebound. More difficult to deal with is updating clients. Due to the fact that servers have no knowledge of what clients it's objects have, it would be impossible to inform all clients of an object that the object's identifier had changed. The solution was to maintain a separate notion of object's name and the list of servers where it exists. The name of the server that forms the base of the name of the object has no bearing whatsoever on which server contains the master replica.

So the server on which an object was created contains the master copy. But how do objects get created in the first place? Objects can be divided into two categories, they are either factory objects or functional objects. A factory object is one that has the ability to create other objects, whereas a functional object does not. Typically a server will create a factory object and bind it into the namespace, with the expectation that clients will resolve the factory and use it to create functional objects. The question is whether the create method is a write operation that needs to contact the master replica of the factory, or a read operation that may contact any of the replicas of the factory. Clearly if the creation method is a write operation, then whichever server creates the factory, that is to say the server containing the master replica, will also contain the master replica of every object the factory creates! This defeats the entire purpose of having the master replica of objects located on different machines. The new master object could be migrated to a random server, however the creation method of factories is most easily treated as a read operation that may contact any of the servers which has a replica of the factory. Figure 3 depicts an example of the interaction between factory creations and replication of both factories and functional objects. We begin with three servers and the first one creates a factory, it is presumably bound into the namespace. In Figure 3b, the factory has been replicated from Server 1 to Servers 2 & 3. Two clients each resolve the factory object from the namespace. The clients then invoke the create method on the factory. Since create is a read operation, the invocation may go to any of the servers, which we see in Figure 3c. Finally, the newly created functional objects are replicated.

In the vein of discussion of read and write operations is the decision of what paradigm to use for write operations. This decision has implications as far as how to propagate updates of objects as well as keeping replicas of objects consistent across multiple servers. Three distinct methods for write operations exist, each with their pluses and minuses: write to any, write to all, and write to one. Writing to any server is perhaps the easiest to implement from the client's perspective, there is no distinction between read and write operations and the need for master copies of objects is eliminated. However keeping the all of the replicas consistent becomes very hard, if not impossible! Sending the write operation to all servers was another option. This eliminates the need for the servers to do update work since the client is carrying out its operation on each replica, and again there is no need to distinguish between master and slave replicas of an object. Both of these methods meant more work for the subcontract and less flexibility for the implementor of the server, as well as difficulties with consistency of the replicas and update propagation. Therefore the method chosen was to send write operations to the single server containing the master copy of the object. It is then the server's responsibility to update the replicas of the object.

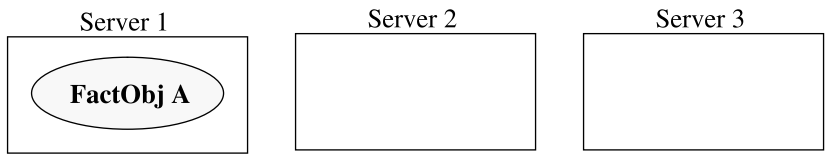

Figure 3a: Startup with three servers, Server 1 creates a factory

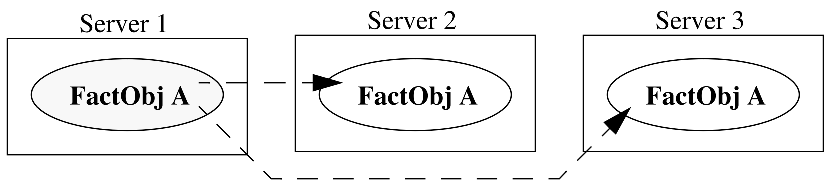

Figure 3b: Replication of initial factory object

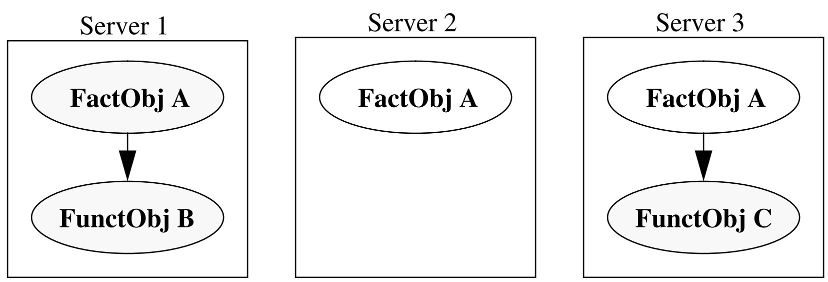

Figure 3c: Two clients contact the servers and create two functional objects

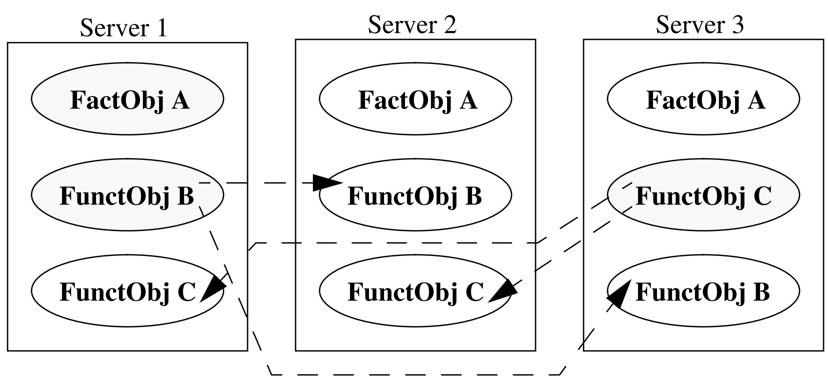

Figure 3d: The functional objects are then replicated

It has been mentioned a few times that each object has a list of servers associated with it, or rather a list of paths to management objects to the servers containing the replicas of the object. This solution was originally thought of as a first pass to get the project working. The major competitor to this method is that used by the replicon subcontract, the original replication subcontract. Replicon maintained a list of door identifiers to the copies of objects on various servers. This method is simple, but there are actually a number of benefits to using a list of paths over a list of door identifiers. First, the management objects were already necessary for communication between servers and for figuring out what servers were available to replicate to. Thus we were already dealing with paths. Second, it is cleaner for the server code which is responsible for replication and updating to manipulate lists of paths rather than low-level door identifiers. Lastly, if a server crashes and is restarted, having a list of paths gives clients the opportunity to not notice any change. When the server is restarted, many of the objects originally on the server may be returned, thus clients that happen to not contact the server while it is crashed might be able to continue with no knowledge of the crash. On the other hand, if a list of doors were maintained, as soon as the server crashed all the doors to its objects become invalid.

Regardless of whether lists of door identifiers or lists of paths to management objects are used, there is the issue of making sure that the client has the most up-to-date version of the list. This is important because as servers crash and new servers are started, the set of servers holding replicas of an object is fluid. In fact, during the lifetime of an object the set of servers may completely change, that is no server to which the object was initially replicated holds a replica in the end. Thus it is necessary to keep the client up-to-date, otherwise the client may find itself unable to locate a replica of the object to use. There are two choices in this matter: either the client may contact the subcontract of one of the servers in the object's list, or it may contract any server directly via the server's management object. In order to implement the first method a door identifier to the object on a particular server would have to be kept around. And while using the door would be very quick, it may not be desirable to keep a door identifier lying around for similar reasons as those mentioned in the previous paragraph. In addition, it would mean that every time a change to the list of servers for an object is made, the subcontract of each of the copies of the object would have to be informed. The alternative method contacts the server directly, and since the client side of the subcontract is already contacting the server to retrieve a door identifier to the object for invocations, this additional call is inconsequential. The next question is when should this request for an updated list be made? Certainly not so infrequently that there is a high probability that all the servers in the current list have become invalid. And so rather than creating an intricate solution to a tangential problem, the request is made after every invocation on the object by the client.

It is interesting to note at this point what structure the list of servers takes. Typically an object will have a master replica. Somehow the location of the master replica should be distinct in the list of servers. At times, however, the server containing the master replica may crash, and temporarily there may be no replacement master replica. So whatever form the list of servers takes, and however the updating of the list occurs, it must include the ability to identify one of the servers as containing the master replica and be able to change the state of whether there even *is* a master replica. The answer is a list that is simply an array of strings, the first entry representing the server containing the master replica of the object. When the server containing the master replica has crashed, either the client or the server realized the condition first. If the client side of the subcontract recognizes this state, it eliminates the first entry and marks the object as not having a master replica. If the servers recognize the situation first, then when the server list is updated a flag is returned to indicate that there is no master for the object. At point later in time, when this situation is resolved, the updated list will return a flag indicating that the first entry in the list is indeed the location of the master replica.

The next major design issue was how to make invocations work. The client side of the subcontract had an object identifier and a list of servers. It could also get a handle to the management object for any of the servers. But how would it actually get the method identifier and arguments from the client to an appropriate server? The first idea was simply to use the management object and call an invoke function with the object identifier, an identifier for the method, and the method arguments. This would have required marshalling the arguments to be sent, and somehow dealing with the return arguments from the method. Essentially this idea would require re-implementing the built-in mechanisms that Spring already provided! What was really desired was to take the object identifier and management object and obtain a reference to the object on a particular server and then to make a normal invocation. But how would the management object method be able to return an appropriate object reference when there might be many different types of replicated objects? This problem seemed similar to that of contexts. Context contain many different types of object references, and in order to bind and resolve one must make use of a templated helper class ***naming***. So the next idea was to create a similar helper class for management objects. There would be a *getObjRef* method that would take a management object and object identifier. The class would be templated on the object reference. Again, there was a problem with the approach. The object reference was still using the replication subcontract, and so an invocation on it would simply recurse. To fix this it would have been necessary to create to types of object references for every type of replicated object: one using the replicated subcontract and the other using the singleton subcontract. At this point a major simplification became clear: rather than getting an object reference to a particular copy of the replicated object, retrieve from the server chosen a door identifier to the object on the server. Thus the need for a helper class was eliminated, and the method of the management object was simply *getObjDoor* with a single parameter of the object identifier.

One of the initial design questions was how to transmit objects to other servers to be replicated and updated. What was desired was some method of capturing the state of an object in a form that could be sent via a call to a management object. Writing a simple wrapper around an array of bytes and the number of bytes seemed to be a starting point. This method is known in Spring as *pickling an object*. Fortunately, it turned out that Spring came with its own pickling and unpickling routines for a class called ***pickle_jar***. Once one created an instance of a ***pickle_jar***, one could add basic data structures such as integers and strings to it. Thus all that was necessary was to create pickling and unpickling routines for each replicated object and make sure that the unpickling removed the information in the same order in was inserted.

The last major design issue is why so much of the functionality of this replicative system resides in the server application rather than the replication subcontract. The main rationale for giving so much responsibility to the server is to allow the server as much flexibility as possible. The author of servers should be able to implement replication in whatever manner they choose, making use of the provided tools. Thus the subcontract is left with little responsibility, and little functionality. What would have been ideal is to create a generic server that implements as much of the server requirement as possible, giving the author of a server the option of inheriting existing functionality or implementing everything themselves. Unfortunately, the design of the server did not quite reach that stage.

3 The Project

The pieces for this replicative system fall into four areas: the replication subcontract, the management object interface, replicated objects, and server requirements. Much of subcontract has already been discussed at a high level, but now will be detailed. The management object has been described in conjunction with the subcontract, but also is pivotal for the replication and updating of objects, as well as for administrative tasks. Replicated objects should all have similar interfaces and basic functionality. Lastly, a server for replicated objects needs to have certain functionality to make the whole system work. Figure 4 shows the structure of the system, detailing what information the subcontract holds.

3.1 Replication Subcontract

Perhaps what lies at the heart of this project is the question of what it takes to create a new subcontract, and in particular, a subcontract that handles replicated objects. Earlier it was mentioned that the subcontract provides the ability for server objects to identify themselves. This ability is implemented through an API of four calls to the server class of the subcontract:

- *void set_id(string object_id)*
- *void set_list(array_string server_list)*
- *void set_write_ops(array_int write_ops_list)*
- *door_identifier get_did()*

The first method, *set_id*, which gives the subcontract the object's unique name. In order to attach the list of servers to the subcontract's notion of the object, there is the *set_list* call. The array of strings is not truly of server names, but of paths to each server's management object in the namespace. For identification of the write operations, *set_write_ops* is given an array of method identifiers. The author of the server only puts in the array those identifiers generated by **contocc** that correspond to write operations. The identifiers are of the form of *<module>_<interface>_codes::f_<method>* where *<module>_<interface>_codes* is a class and *f_<method>* is part of an enum contained in the class[2]. Lastly, *get_did()* returns a door identifier

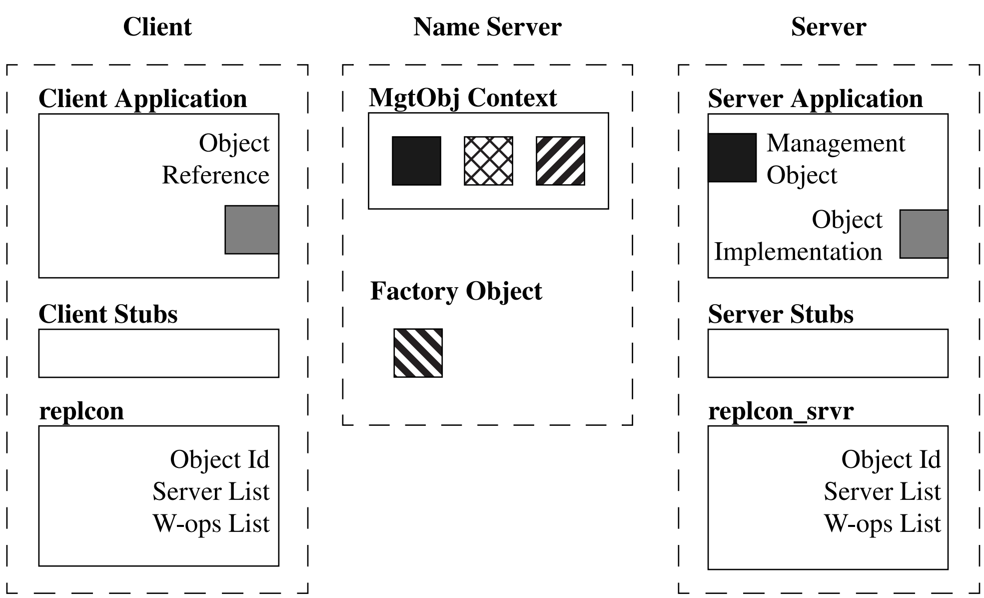

Figure 4: Structure of replication system

to the object. This last call is used in conjunction with the management object for invocations.

Underneath the interface to the object implementations is the functionality of the subcontract. Rather than attempting to create the subcontract from scratch, the replication subcontract was modified from the source of the singleton subcontract (refer to Section 1 for the description of singleton), thus it provides the subcontract local class and the ability for fast local calls on the server. In fact, the only parts of the singleton subcontract needing to be changed were those dealing with marshalling, unmarshalling, and invocation. Specifically, the *marshal_consume* and *marshal_copy* routines in both the subcontract local and client classes need to marshal the object's identification, list of servers, and list of write operations. Similarly, the subcontract client class *unmarshal* routine needs to unmarshal the same information. The most complicated addition to make replication work was the client class's *invoke* method. Based on the method being a read or write operation, *invoke* chooses an appropriate server from the object's server list. If the method is a write operation, and the server list does not contain a server for the master object, an error code is returned. And of course, if there are no servers left in the list, an error code is returned[3]. Once a server has been chosen, its management object is resolved from the namespace and is used to obtain a door identifier to the object in question. If either the resolution of or the invocation on the management object fails, the server is eliminated from the list and another is chosen. Once a door identifier to the object is obtained, the invocation is actually made. If the invocation fails, another server is chosen and the process repeats[4]. After the invocation returns, another call is made on the management object used to get an updated version of the server list for the object. Using Figure 4 as a model, Figure 5 depicts the flow of a typical invocation.

Aside from the minimal modifications to singleton to make replication work, there were two somewhat intricate issues. First, the replication subcontract needed a type identifier. Normally a *type_id* is generated for an interface by **contocc**, but the subcontract does not have an interface. The solution was to create, by hand, a **typemgr** file to generate the appropriate *type_id*. The guideline used for this file was the entry in ***/sys/interfaces/spring/types.tm*** for the singleton subcontract. The second issue regards learning of unreferenced objects. In the standard subcontracts, when the client handle to an object "goes away," the reference count on the server's door identifier for the object is decreased. When the count hits one, the server subcontract class is informed that the server's reference is the last remaining. The author of the object has the option of handling this by overriding the *_unreferenced* method. In order to be able to clean up after the objects properly, a door must still be marshalled with the object even though it is not actually used for any other purpose. This is not an atrocious solution, but neither is it entirely satisfactory since it may not work in all circumstances.

---

2. Here, <*module*> refers to the name of the module defined in the ***.idl*** file, and <*interface*> refers to an interface, defined in the module, that is inherited from the replicated object interface. See Appendix E for an example of an ***.idl*** file.
3. The correct reaction to this is to search the namespace for management objects, contacting each in turn, and asking if that server has a replica of the object in question. Only if all of the servers answer in the negative should the *invoke* method return an error code.
4. Instead of simply checking the fail condition, it should determine if it is a contract fault of some form, or a higher level exception that should actually be returned to the user rather than attempting the same call on another copy of the object.

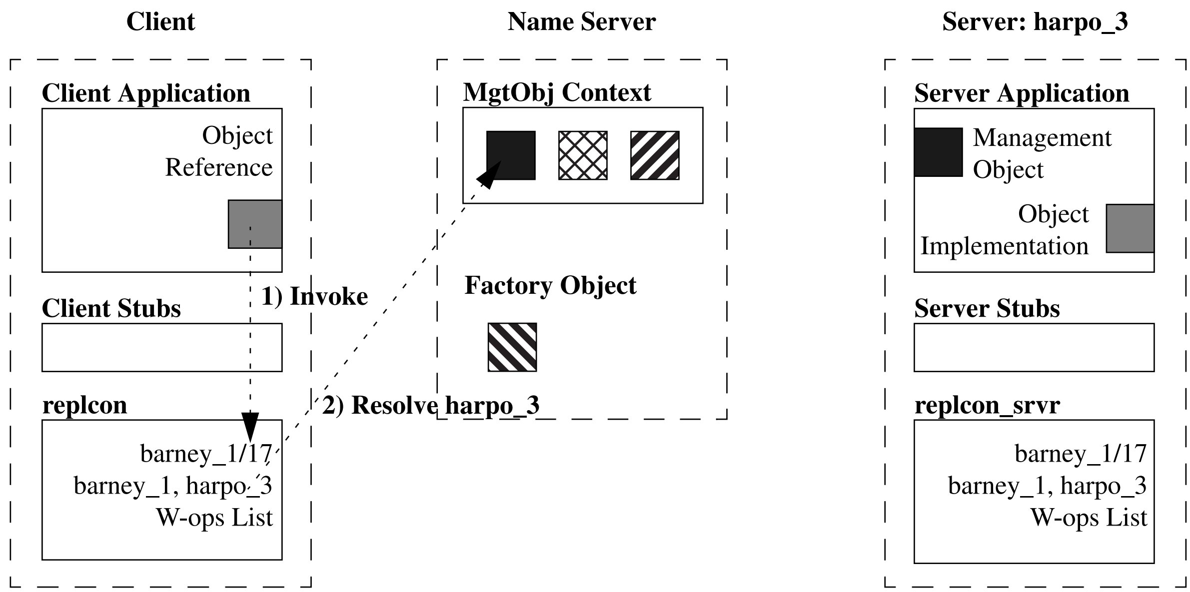

Figure 5a: Diagram of an invocation. Step 1, the client calls a method on an object reference it holds. Step 2, the replcon client class chooses harpo_3 and resolves it management object from the namespace.

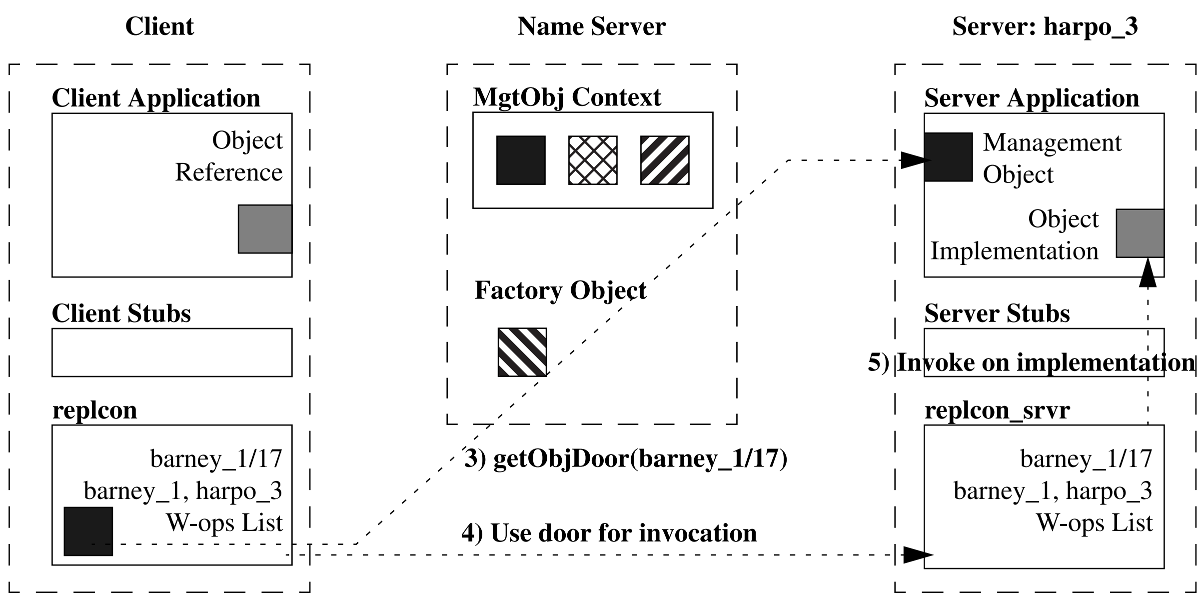

Figure 5b: Step 3, the replcon client class uses the management object to contact server harpo_3. It sends the object identifier, barney_1/17, and is returned a door identifier to the replica on harpo_3. Step 4, using the door identifier, the subcontract client class contacts the server class to perform the invocation. Step 5, the subcontract server class invokes the method onthe implementation of the object.

3.2 Management Object

Aside from the interface required for the subcontract client class to work properly, there are two other types of communication to servers that are necessary: server to server and administration to server. In the first case, clearly servers need a communications channel to each other for the replication and updating of objects. In the second case, administrative methods for informing servers of other, crashed servers, and shutting servers down cleanly were desired[5]. Coupled with the need mentioned earlier for a mechanism for invocations and updating the server list of client objects, the interface for management objects was created.

As mentioned earlier in Section 2.1, management objects are object interfaces to servers. The idea is that every server for replicated objects would create a management object to represent it. Then, as a part of replicating and updating objects, each server would invoke operations on other servers' management objects. In addition, subcontract client classes would be clients of server's management objects, as would the administration program. The implementation of the management object was divided into two parts: behind the scenes work encapsulated in a library, and an API that each server that creates a management object must provide. The reason this is necessary is to separate the error checking and data manipulations from the true work that the server must perform. To aid in the error checking, a function was added to the API the servers must provide to check that the server contains a replica of an object. The client interface to the management object is in ***mgtobj.idl*** in Appendix A, and a description of the server API is part of ***mgtobj_impl.h***, the relevant section located in Appendix B.

The two major implementation issues of management objects centered around the types one is able to use in ***.idl*** files. It was discussed earlier why pickling and unpickling were the methods chosen for transmitting objects for replication and updating, and therefore Spring's *pickle_jar* type was used as the data structure. However, *pickle_jar* does not have an interface defined, nor the functions necessary to allow it to be used as a type in an ***.idl*** file. The easiest solution was to construct a *raw_data* structure from the contents of a *pickle_jar*, and since *raw_data* was a type usable by interfaces, transmit the information in that form. In order to accomplish this, it was necessary to alter the ***pickle.h*** file to make conversion routines to and from *raw_data* public instead of private. The other difficulty encountered with types in ***.idl*** files had to do with returning a door identifier to an object. Again, *door_identifier* is not a type that can be used in an interface file. Worse still, there are separate areas of the marshal buffer for general data and door descriptors. This meant that it was not possible to disguise the *door_identifier* as another data type, since it would not be returned in the proper section of the marshal buffer[6]. The solution in this case was to modify the stub code generated by **contocc** for the management object. It was necessary for the server side of the *mgtobj_MgtObj::getObjDoor(string)* call to marshal the

5. In fact, there is no other mechanism for cleanly shutting down servers. Because there is no signal handling, it is not possible for a server to clean up after **nuke** is run on its process, or Control-C is hit. With this method of shutting down, it is necessary for the server providing the management object to allow the shutdown call to return before it actually exits. The alternative is that the server process exits, causing a contract fault to be returned on the shutdown call.
6. A *door_identifier* is simply an index into the process's door table. If the *door_identifier* were not returned in the correct section, the kernel would not translate the door from the server's door table to the client's door table. In this case, the client would have a useless index.

door_identifier returned by *srvr_getObjDoor(string)*, and for the client side to correspondingly unmarshal the *door_identifier*. In turn, the client side would then simply return the value contained by the *door_identifier*. Excerpts from ***mgtobj.cc*** that include these stub code modifications can be found in Appendix F.

3.3 Replicated Objects

The requirements of a replicated object can be seen from two different perspectives: the client application's and the server application's. There are certain public routines that a client should be able to invoke on all replicated objects, such as setting how frequently updates should be made to slave copies of the object. This contrasts with the private routines that servers would like to require of all replicated objects, such as the ability to pickle an object, create an object from a pickled state, and updating an object from information in a pickled state. Two types of inheritance are called for here: interface and implementation. The public interface, that to be used by clients of the object, is encapsulated into ***replobj.idl***, found in Appendix C. This interface is interface inherited by every replicated object in its ***.idl*** file. The private server implementation interface is defined in ***ReplObj.h***, found in Appendix D, and should be inherited by each object's C++ class generated by **contoimpl**.

3.4 Server Requirements

At first glance it may seem that there is little required of servers, only that they provide management objects. Yet, as discussed earlier, most of the work of replication is left to the servers, and in fact a great deal is expected of them. The requirements come in three flavors: startup, runtime, and miscellanies.

It was already mentioned that servers need unique names in order to identify their management objects and to aid in uniquely identifying the objects created on that server. In addition to a unique name, it is required that there be only one initial server. An initial server is a server that binds objects (other than management objects) into the namespace at startup typically the objects are factory objects. Limiting the initial servers to one is necessary to simplify startups. If every started server attempted to create and bind a factory to the same location, chaos would ensue and many objects would be unnecessarily replicated. In line with the issue of an initial server is the replication of the objects the initial server creates and binds. Clearly, if the object is bound before being replicated, any client resolving the object will know only about the initial server to begin with. If the initial server ever crashes, the client will have problems. Servers must address this issue in some way. Finally, upon startup each server, whether initial or not, must create and bind into the namespace a management object.

Whenever a new object is created it must be given a name unique to all objects across all servers and stored such that it can be found with the name as a key. As mentioned several times previously, the server must provide an implementation for management object methods (see Appendix B). And finally, the server must have a thread to check for new servers, replicate objects as necessary, and update those objects needing to be updated[7].

---

7. Actually, two threads might be even better. Checking for new servers and replication could run in one thread, and updates could be made in the other.

The last set of requirements have to do with replicated objects and administering the servers. First, replicated objects must make calls to the replcon subcontract upon creation in order to set their object identification, server list and list of write operations. Second, some form of administration program must be written. One of the hardest questions in distributed computing is determining when another machine has crashed or is just slow to respond. Rather than attempting to address this question, clients are informed when it might be the case that a server has crashed. It is then left to the administrator to confirm this. If a machine has crashed, the administrative program should provide the ability to inform the other servers. In addition to being able to inform servers of crashes, the administrative program should at a minimum allow for shutting down servers.

4 The Prototype

The prototype used the ***dbex*** example from the Spring course, with some alterations. ***Dbex*** is a simple database program. It implements a factory which creates collections of name-value pairs. In addition, collections may create iterators to iterate through their contents. The interface for ***dbex*** is in Appendix E. This program was chosen to be the prototype server for three reasons: it is conceptually simple, it has a straightforward implementation, and it provides objects that one would want to replicate. In addition to the implementation of the functionality of the ***dbex*** interface, the prototype met the requirements of servers of replicated objects. The rest of this section is in two parts. First we discuss what the prototype does correctly. Following that, we bring to light a number of things it doesn't do quite right.

4.1 Highlights

The prototype consists of three components: the client, server and administration programs. The client program does nothing any differently than client programs using singleton objects; the replication is completely hidden. The administration program does a little more than the minimum of contacting servers to shutdown or concerning crashed servers. In addition to these requirements, the administrator is able to remove management objects from the namespace, remove the context that contains the management objects, and remove the initial factory that is bound into the namespace. Lastly, the server program is where all the pieces come together.

When the server starts up, it takes two required arguments and one optional argument. The first two arguments are a unique server name and a number of times to replicate objects created on the server. The optional third argument is a flag to signify that this server is the initial server. Given this unique name, the server makes sure that a context for the management objects exists in the namespace[8]. If the context already contains a management object with the same name given on the command line, the user who started the server (typically the administrator) is asked if they would like to replace the management object[9]. Once the management object is correctly placed into the namespace, the server creates a factory object if it is designated an initial server. The last thing the server does is to become a spring server with the *spring_lib::become_server* call.

8. The context was created in ***village/services/dbex_mgt*** in order that all servers could access it. It is possible that with replicated contexts a more interesting method for this might be found.

During its execution, the server makes use of its name to give objects it creates a unique name. The server concatenates its name and a numeric tag to create an identifier for each object. The numeric tag is made persistent by storing it in a file with the server's name. Thus when a server returns after having crashed it does not reuse object names which might result in conflicts with objects that still exist on other servers. Every replicated object, either factory or collection, is then stored in a table, keyed by the object identifier.

4.2 Lowlights

There are some unfortunate shortcomings of the ***dbex*** prototype. First of all, it was not actually tested over several machines and with network failures. Spring was only installed on one machine, the other intended machine having unsupported hardware. Presumably the system would have handled this situation well, since it is the administrators responsibility to determine whether a machine has actually crashed or not. Another major drawback is that ***dbex*** as a system does not have clients making heavy use of the same replicated object. This situation is interesting in order to observe consistency in write operations among many clients and read operations among many servers.

The last two lowlights of the ***dbex*** prototype have to do with the implementation. In order to keep things simple, the *setUpdate* method for replicated objects does not actually have any effect for the replicated factory and collection objects. All replicated objects are updated as needed, and are checked every fifteen seconds. This is for the sake of being able to demonstrate the prototype. It would be trivial to keep track of both whether an object needs an update and when it is scheduled to be updated. And the final issue concerns replication of objects bound into the namespace. As mentioned earlier, servers must address this somehow. The ***dbex*** prototype solution was to ask the administrator if they would like to wait for the initial factory object to be replicated before binding into the namespace. If yes, then the server waits a little while and checks if the factory has been replicated. If it has, the object is bound, otherwise the administrator is asked again. The drawback to this solution is that the set of servers that have replicas of an object changes as servers go down and come up. And at some point, even if the initial factory object has been replicated, the list of servers bound in the namespace may no longer contain any of the actual servers. A better solution would be to rebind initial objects each time their list of servers changes. In addition, each replica of the object would have to know that the object is bound in the namespace and where in case that server is called upon to become the repository for the master replica of the object.

5 Conclusion

Despite the drawbacks mentioned in the previous section, this system is a solid prototype that

9. The first attempt at this tried to *bind* into the namespace and handle the "already bound" exception and then *rebind* if appropriate. While this seemed to work from the server's perspective, when the management object was resolved, the calls on the object failed. This was because fat pointers may not be used in the server once they have been bound. Unfortunately this also includes binding attempts that fail. The solution was to first check if the management object was already in the namespace, and remove if it if desired. Only then is the new management object created and bound into the namespace.

demonstrates working replication. The design of the system is useful because it provides functionality to server writers, allowing them to decide many details of how they would like to do the replication.

As a prototype system, there are still many areas of work to improve this system beyond replication specific areas such as load balancing and determination of quick machines. One of the major areas to be explored is security. It has not been worked out what access control lists should be associated with various objects in order to make the system secure. Additionally, the issue of capabilities should be further explored.

Another area to improve the system would be to cache door identifiers in the subcontract client class instead of making a call to the server for every invocation. Management objects might also be cached so as not to have to resolve from the namespace quite as much. Coupled with these speedups, it would be informative to attempt to get some statistics on how fast replicated objects work as compared to the same object using the singleton subcontract. It might also be interesting to provide **tabulator** information to get a glimpse of what is happening internally.

A few other ideas include creating replicated contexts, supporting write to any or write to all paradigms, or having multiple initial servers. One of the hardest ideas to work cleanly into Spring's object model is to allow the client to have access to the most up-to-date copy of a replicated object. Clearly it is a desirable piece of functionality, but the subcontract is supposed to be hidden completely from the client's view!

Appendix A: mgtobj.idl

```
import raw_data;
import array_string;

module mgtobj {
   enum status_t {
      SUCCESS,
      FAILURE
   };

   exception MgtObjErr {
      string msg;
   };

   typedef string                obj_id_t;
   typedef raw_data::raw_data    pickle_jar_wrapper;
   typedef long                  door_id_value;

   interface MgtObj {
      door_id_value getObjDoor(copy obj_id_t obj_id) raises(MgtObjErr);
      boolean updateSrvrList(copy obj_id_t obj_id,
                             produce array_string::array_string srvr_list)
                             raises(MgtObjErr);
      status_t replicate(copy pickle_jar_wrapper pickled_obj);
      status_t update(copy obj_id_t obj_id, copy pickle_jar_wrapper
                      pickled_obj) raises(MgtObjErr);
      status_t master(copy string crashed_server);
      status_t shutdown();
   };
};
```

Appendix B: Excerpt from mgtobj_impl.h

```
// As a server for a mgtobj, you must provide the following functions:

    // Returns true if server has object, called before getObjDoor,
    // updateSrvrList, and update.
extern bool             srvr_checkObj(mgtobj_obj_id_t obj_id);

    // Return a door to the object, call _rep.get_did on the object
    // impl to contact the subcontract for a door
extern door_identifier  srvr_getObjDoor(mgtobj_obj_id_t obj_id);

    // Return the current list of servers the object is replicated to
extern bool             srvr_updateSrvrList(mgtobj_obj_id_t obj_id,
                                            array_string *& list);

    // Replicate the object to this server, return true on success
extern bool             srvr_replicate(pickle_jar * jar);

    // Update the object on this server, return true on success
extern bool             srvr_update(mgtobj_obj_id_t obj_id,
                                    pickle_jar * jar);

    // Take over all objects owned by the crashed server
extern bool             srvr_master(string crashed_srvr);

    // Shut the server down
extern bool             srvr_shutdown();
```

Appendix C: replobj.idl

```
module replobj {
   enum update_freq_t {
      IMMEDIATE,
      HIGH,
      MEDIUM,
      LOW
   };

   interface ReplObj {
      void setUpdate(copy update_freq_t freq);
   };
};
```

Appendix D: ReplObj.h

```
class ReplObj {
public:
    ReplObj() {}

      // Note: in order to force the writer of an object server
      // to override this function, it could be neither a
      // constructor nor a static function.  The writer will have
      // to have a null-object or figure another way to make use of this.
    virtual void *        createObj(pickle_jar * jar) = 0;

    virtual pickle_jar *  pickleObj() = 0;
    virtual void          updateObj(pickle_jar * jar) = 0;
};
```

Appendix E: dbex.idl

```
import replobj;

module dbex {
   struct NVpair {
      string name;
      string value;
   };

   interface collIter {
      NVpair next();
   };

   interface NVcollection : replobj::ReplObj {
      long size();
      string query(copy string name);
      boolean add(copy string name, copy string value);
      boolean remove(copy string name);
      collIter getIter();
   };

   interface NVcollection_factory : replobj::ReplObj {
      NVcollection create();
   };
};
```

Appendix F: Excerpt from mgtobj.cc

```
mgtobj_door_id_value
mgtobj_MgtObj_methods::getObjDoor(any_obj *obj, mgtobj_obj_id_t arg_obj_id)
throw (mgtobj_MgtObjErr, contract_fault)
{
     // Because I actually need to transmit a door here, I couldn't
     // use the generated stub byte code structure.  There was no
     // way to fake it into dealing with a door because it puts
     // doors into a different area of the marshal buffer.  I used
     // contocc with -fno-byte-code-stubs to generate client side
     // code (I already had the server side marshalling code).  I
     // cut the code out for getObjDoor and pasted it into the code
     // generated with -fbyte-code-stubs, and then modified it to
     // unmarshal the door identifier.

    mgtobj_door_id_value r;
    door_identifier did;
    DEFINE_MARSHAL_BUFFER(m_buf);
    obj->sc->invoke_preamble(obj, m_buf);
    m_buf.put_string(arg_obj_id);
    int s = obj->sc->invoke(obj, m_buf, mgtobj_MgtObj_codes::f_getObjDoor);
    switch (s) {
        case isoh::ok:
            did = m_buf.get_door_identifier();
            r = did.value();
            break;
        case mgtobj_MgtObj_codes::e_mgtobj_MgtObjErr:
            unmarshal_and_raise_mgtobj_MgtObjErr(m_buf);
        default:
            throw contract_fault(s);
    }
    return r;
}

int mgtobj_MgtObj_srvr::_std_invoke(mgtobj_MgtObj_srvr *p,
                                   MARSHAL_BUFFER &m_buf, int f) throw ()
{
    int rc = isoh::ok;
    try {
        switch (f) {
            case mgtobj_MgtObj_codes::f_getObjDoor: {
                mgtobj_obj_id_t a_obj_id;
                a_obj_id = m_buf.get_string();
                m_buf.cleanup_and_reinitialize();
                mgtobj_door_id_value r = p->getObjDoor(a_obj_id);
               door_identifier did(r);
                    // Need to actually marshal the door_identifier so
                    // it gets translated from server table into
                    // client table.
                m_buf.put_door_identifier(did);
                break;
            }
```

```
            case mgtobj_MgtObj_codes::f_updateSrvrList: {
                mgtobj_obj_id_t a_obj_id;
                array_string *a_srvr_list;
                a_obj_id = m_buf.get_string();
                m_buf.cleanup_and_reinitialize();
                bool r = p->updateSrvrList(a_obj_id, a_srvr_list);
                array_string::_marshal_consume(a_srvr_list, m_buf);
                m_buf.put_bool(r);
                break;
            }
              .
              .
              .
            default:
                rc = p->_bad_method(m_buf, f);
        }
    } catch(mgtobj_MgtObjErr &e) {
        m_buf.cleanup_and_reinitialize();
        m_buf.put_string(e.msg);
        rc = mgtobj_MgtObj_codes::e_mgtobj_MgtObjErr;
    } catch(contract_fault &e) {
        m_buf.cleanup_and_reinitialize();
        rc = isoh::e_server_contract_failure;
    } catch(copy_fault &e) {
        m_buf.cleanup_and_reinitialize();
        rc = isoh::e_copy_fault;
    }
    return rc;
}
```