LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

UCRL-TR-220292

How to Implement a Protocol for Babel RMI

G. Kumfert, J. Leek

March 31, 2006

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

How to Implement a Protocol for Babel RMI

Everything you need to know, and more
30 March 2006

UCRL-TR-??????

Gary Kumfert and James Leek

Table of Contents

I. Goals....4
II. The Way Things Work (Generally)....4
  A. Client Side Interactions....4
    1. Protocols and URLs....5
    2. Client Side Creation & Connection....6
    3. Client Side Method Invocation....6
    4. Argument Passing....7
    5. Client Side Deletion or Disconnect....8
    6. Client Side Cast....9
    7. Nonblocking....9
  B. Server Side Interactions....10
    1. Starting a BOS....10
    2. The Instance Registry....11
    3. BOS Functionality....11
    4. Server Side Deletion & Disconnect....12
    5. Nonblocking....12
III. Implementation details....13
  A. Client Side Interactions....13
    1. Interface Connection....13
    2. Reference Counting....13
    3. RMI Cast....14
    3. Nonblocking....15
  B. Server Side Interactions....15
    1. InstanceRegistry (again)....15
      i) Adding Objects....15
      ii) Reference Counting and the InstanceRegistry....16
    2. Passing Object References....16
      i) Identifying Local Objects....16
      ii) Reference Counting....17
      i.In Object Arguments....17
      ii.Out Object Arguments and Return Values....18
      iii. InOut Object Arguments....18
    3. Passing Objects By Copy/Serialization....18
    4. Exceptions....19
    5. Connecting and Casting from the IOR....20
III. Protocol Implementor's Guide....20

A. Overview.....20
1. Serializer.....21
1. Deserializer.....21
B. Client Side.....22
1. InstanceHandle.....22
2. Invocation.....22
3. Response.....22
4. Nonblocking Implementation.....23
B. Server Side.....23
1. ServerInfo.....23
2. Call.....24
3. Return.....24
4. Basic BOS Calls.....24
V. Babel RMI Reference Manual.....25
A. RMI Builtin Methods.....25
I) User-Exposed Static Methods.....25
1. _create[Remote](in string url).....25
2. _connect(in string url).....25
II) User-Exposed Virtual Methods.....26
1. _getURL().....26
2. _isRemote/_isLocal().....26
2. _exec(in string methodName, in sidl.rmi.Call inArgs, in sidl.rmi.Return outArgs).....26
III) Internal Static Methods.....27
1. _IHConnect(in InstanceHandle instance).....27
2. _connectI(in string url , in bool ar).....27
3. _fconnect/_fcast.....27
II) Internal Virtual Methods.....28
1. _raddRef().....28
B. Interfaces Added to sidl.sidl.....28
1. sidl.io.Serializer interface.....28
2. sidl.io.Deserializer interface.....29
3. sidl.io.Serializable interface.....30
4. sidl.rmi.InstanceHandle interface.....30
5. sidl.rmi.Invocation interface.....31
6. sidl.rmi.Response interface.....31
7. sidl.rmi.BaseServer interface.....32
8. sidl.rmi.ServerInfo interface.....32
C. Singleton Classes Added to the Babel Runtime System.....32
1. sidl.rmi.InstanceRegistry (singleton).....32

2. sidl.rmi.ProtocolFactory class (singleton)....................33
3. sidl.rmi.ConnectRegistry class (singleton)....................33
4. sidl.rmi.ServerRegistry class (singleton)....................33
D. Babel RMI Exception Hierarchy....................34

I. Goals

RMI support in Babel has two main goals: transparency & flexibility. Transparency meaning that the new RMI features are entirely transparent to existing Babelized code; flexibility meaning the RMI capability should also be flexible enough to support a variety of RMI transport implementations.

Babel RMI is a big success in both areas. Babel RMI is completely transparent to already Babelized implementation code, allowing painless upgrade, and only very minor setup changes are required in client code to take advantage of RMI.

The Babel RMI transport mechanism is also extremely flexible. Any protocol that implements Babel's minimal, but complete, interface may be used as a Babel RMI protocol. The Babel RMI API allows users to select the best protocol and connection model for their application, whether that means a WebSerivices-like client-server model for use over a WAP, or a faster binary peer-to-peer protocol for use on different nodes in a leadership-class supercomputer. Users can even change protocols without recompiling their code.

The goal of this paper is to give network researchers and protocol implementors the information they need to develop new protocols for Babel RMI. This paper will cover both the high-level interfaces in the Babel RMI API, and the low level details about how Babel RMI handles RMI objects.

II. The Way Things Work (Generally)

Despite the simplicity of the interfaces that the protocol writer must implement, RMI protocol implementations (which are by nature generic libraries) have a fairly complex interaction with the Babel code (which is generated). This section will attempt to is sketch out roughly how this works. For a more complete view of the user's interactions with Babel RMI, one should consult the Remote Method Invocation chapter of the Babel User's Guide. The deep implementation details will be covered later in this paper.

For the purposes of this discussion, we assume the following example:

```
package pkg version 1.0 {
    class cls {
        bool mthd ( in double idbl, out float oflt,
                    in string istr) throws Exception;
  }
}
```

A. Client Side Interactions

First a connection must be established between a stub and a live object elsewhere. We

will assume implementations (such as Proteus) may change the actual underlying protocol dynamically. Babel does not care about the specific protocol across the wire, only that an implementation (SIDL class) that implements the desired SIDL APIs exits (and that Babel is able to load it).

1. Protocols and URLs

The first thing any user of Babel RMI has to do is choose a Babel RMI protocol. They then bootstrap the protocol by adding it to the sidl.rmi.ProtocolFactory with the addProtocol() call.

```
static bool addProtocol( in string prefix, in string typeName );
```

The prefix is the short name of the protocol used in front of a URL. The typeName is the full sidl declared type name of the protocol's implementation of sidl.rmi.InstanceHandle. For example:

```
success = addProtocol("simhandle", "sidlx.rmi.SimHandle")    ;
```

This call adds the "Simple Protocol" to the protocolFactory. The shortname used here **has** to match the name returned by sidl.rmi.ServerInfo.getServerURL(). Otherwise Babel will have trouble correctly loading the server from automatically generated URLs. Every protocol is free to define it's own standard for URLs. The only thing Babel requires is that Babel can find the protocol short name before the first non-alphanumeric character. For example, the "Simple Protocol" URL looks like this:

```
simhandle://host.com:port/<objectID>
```

but one can also imagine urls of the form:

```
weird://05:16:5B:BD:E1:73+<objectid>
```

for the weird protocol, or:

weird+SSL://05:16:5B:BD:E1:73+<objectid>

for running the weird protocol over SSL. Babel RMI itself does not attempt to parse anything past the first non-alphanumeric character, so most of the URL is entirely protocol dependent.

2. Client Side Creation & Connection

Client side creation or connection occurs when users use the static builtin methods "_create[Remote]()" or " _connect()" on a Babel object. Both methods create a stub on the local machine that holds a connection to an implementation instance on a remote server. The former creates a new remote instance (and therefore only applies to SIDL classes), of the specific type on the named server using the named RMI implementation. The latter connects to an existing implementation instance on the named server (and can therefore be any interface or class the implementation class extends). Users may connect explicitly, but connections will probably more often occur implicitly as objects are passed by reference in argument lists.

Regardless of the language binding, while creating the local stub is done inside of Babel generated code, the creation of the remote reference (the InstanceHandle) is delegated to the sidl.rmi.ProtocolFactory class implemented in the sidl runtime library.

```
sidl_rmi_InstanceHandle ih =
sidl_rmi_ProtocolFactory_createInstance( url, typeName );
```

The ProtocolFactory will get the protocol short name prefixed on the URL, identify an implementation associated with the protocol portion of the URL, use sidl.Loader to instantiate it, and finally initialize it with a call to InstanceHandle.initCreate() or initConnect() respectively. The RMI library will implement the sidl.rmi.InstanceHandle interface. Remote Babel stubs store a sidl.rmi.InstanceHandle in their IOR in order to maintain their connection to the remote object.

The URLs for _create[Remote]() and _connect() are slightly different. The URL for _connect() must, in one way or another, include the objectID used to uniquely identify the object on the remote BOS. _remote[Create]() does not need this. Here is an example of how this works with the "Simple Protocol" URLs:

```
_remoteCreate("simhandle://host.com:port/");
_connect("simhandle://host.com:port/myObject1234");
```

Notice that the remote[Create]() URL uniquely identifies the BOS, but the _connect() URL uniquely identifies both the BOS and the desired object on that BOS.

3. Client Side Method Invocation

When Stubs to remote objects are created, their EPV is specially initialized for RMI dispatch. The special remote functions that the remote EPV is initialized with must do some special packing and unpacking of arguments. Those Babel generated functions will always be in C, and look like this:

```
sidl_rmi_InstanceHandle = (sidl_rmi_InstanceHandle) self->d_data;
```

```
sidl_rmi_Invocation i =
 sidl_rmi_InstanceHandle_createInvocation( c, "mthd" _ex );
 SIDL_CHECK(*_ex);

sidl_rmi_Invocation_packDouble( i, "idbl" , 2.0, _ex );
SIDL_CHECK(*_ex);
sidl_rmi_Invocation_packString( i, "istr", "Hello",  _ex );
SIDL_CHECK(*_ex);

sidl_rmi_Response r = sidl_rmi_Invocation_invokeMethod( i, _ex);
SIDL_CHECK(*_ex);
netex = sidl_rmi_Response_getExceptionThrown(_rsvp, _ex);
if(netex != NULL) {
  sidl_BaseInterface throwaway_exception = NULL;
  *_ex = (sidl_BaseInterface) sidl_BaseInterface__rmicast(netex,
          &throwaway_exception);
  return _retval;
}
sidl_rmi_Response_unpackBool( r, "_retval", &_retval _ex );
SIDL_CHECK(*_ex);
sidl_rmi_Response_unpackFloat( r, "oflt", &oflt _ex );
SIDL_CHECK(*_ex);
```

Where the RMI libraries implement the following SIDL interfaces: sidl.rmi.InstanceHandle, sidl.rmi.Invocation, and sidl.rmi.Response.

It should also be noted that two different kinds of exceptions may be returned from getException(). Both exceptions thrown from the remote method and network communication exceptions may be returned. Runtime/Network exceptions are not always thrown via getException(), only sometimes. In some cases the exception will be thrown from the invokeMethod() method call.

4. Argument Passing

In general, Babel RMI objects are passed by value. Both basic types and arrays are passed exclusively by value/copy. This is even true for inout arguments where the user passes in a pointer to the argument, and the value of the pointer may be changed. In the underlying RMI, a value was actually passed to the server, which actually passed the value back. This is the most obvious choice for basic types, but some have asked why arrays are always serialized across the network. The basic reasoning is, usually if the user is passing around an array, he must be planning to do some serious computation with it. The latency involved with reading even a few values across the network would quickly overtake the performance gained by not passing the array as a big block in the first place.

Objects are an exception to this rule of pass by copy. Objects may be passed either by

copy or by reference. In fact, pass by reference is the default behavior. In order for an object to be passed by copy, the argument must be tagged with the "copy" keyword in SIDL, and the object must extend sidl.rmi.Serializable. For example:

```
copy foo.Bar mthdCopy(in copy foo.Baz bz);
```

This SIDL declares a method that passes in a foo.Baz object by copy, and returns a foo.Bar object by copy If either foo.Bar or foo.Baz do not implement Serializable, Babel's SIDL parser will halt with an error.

Internally, pass-by-reference means passing the object URL as a string to the recipient; pass-by-copy means serializing all the data in the object as a string, and recreating it on the recipient system.

5. Client Side Deletion or Disconnect

All Babel objects are reference counted and when reference count goes to zero, the object's destructor is invoked. RMI objects are basically the same, but more complex. A Babel remote object has three distinct reference counts. It has a reference count on the stub, a reference count on the Instance Handle, and a remote reference count on the server (for more on reference counting, see page 13). When any of these reference counts reaches zero, the object at that point will be destroyed. The object is completely destroyed when the remote reference count reaches zero.

Babel neither requires nor enforces that the reference count be known exactly at any one location. For instance, the reference count for an object in the IOR could be 2, but one of those could be from Python where Python has 5 internal references to the extension module. In this case, only when all 5 internal references are collected in Python would the IOR's reference count be decremented by one, and only when the IOR's reference count actually reaches zero (presumably whoever has the other reference will deleteRef it) would the instance be destroyed.

Remote objects are similar to the Python example. There is both a remote reference count on the remote object, and a local reference count on the stub and the InstanceHandle. Only when the remote reference count reaches zero is the object actually destroyed. This remote reference count counts the number of InstanceHandles that point to that object.

So, in conclusion, The normal add and deleteRef functions will increment and decerement a reference count that only refers to THAT stub. When a stub is collected, it will deleteRef's the remote object. All of this is transparent to the user, who may freely reuse objects in the normal Babel manner. (Except that cast addRefs as of Babel 0.11.0)

6. Client Side Cast

With the addition of RMI, casting will now always result in incrementing the reference count, and will sometimes result in a new stub. In other words, after a cast, the user must deleteRef the pre-cast reference if they do not wish to continue using it. This is because, if a remote object is downcast, Babel may have to create a new stub to represent the new type. This means there will be two stubs pointing to the same remote object, so the user must explicitly collect both. For more on RMI casting, see page 14, for more on reference counting see page 13.

7. Nonblocking

There are two types of nonblocking RMI in Babel, nonblocking and oneway. Both are declared as attributes on the method in SIDL. The difference is that with oneway communication, the client does not expect any return values. A oneway method will not even return an exception, unless it occurs during communication with the server. On the other hand, a non-blocking call can have return arguments. The user will send a request, and get a \SIDL{sidl.rmi.Ticket}. Later, the user may use the \SIDL{Ticket} to receive the out arguments.

For nonblocking or oneway RMI to actually work, the protocol must support it. All protocols must at least emulate support, even if the emulation loses the benefits of real nonblocking calls. Currently there are no protocols that actually support non-blocking RMI. ``Simple Protocol'' simply emulates support. However, there are at least two protocols currently under development that do support it.

Here is an example of SIDL declerations of nonblocking code:

```
package foo version 0.2 {
  class Bar {
    nonblocking double runSimulation(in double x, inout y, out z);
    oneway void initSimulation(in string name, in int flags);
  }
}
```

Notice that the nonblocking call may take any arguments, but only in arguments are allowed for the oneway call. In fact, Babel's SIDL parser will halt with an error if a oneway call declares out arguments or an exception.

Calling the oneway method looks just like calling a normal Babel RMI blocking call, but nonblocking requires MPI-style send and receive calls. The _send returns a sidl.rmi.Ticket, which the user later calls receive on to get the return arguments.

```
foo_Bar b1 = foo_Bar__createRemote("simhandle://pc1:9999",&_ex);
sidl_rmi_Ticket t = NULL;
double x, y, z;
```

```
foo_Bar_initSimulation(b1, "Test Simulation 1", 0, &_ex);
t = foo_Bar_runSimulation_send(b1, x, y, &_ex);
/* ... Work ... */
foo_Bar_runSimulation_recv(b, t, &y, &z, &_ex);
sidl_rmi_Ticket_deleteRef(t , &_ex);
```

There is another interface, sidl.rmi.TicketBook, which is used to organize a set of nonblocking calls, so that the user can maximize their performance by making a set of calls and dealing with their return values as the come in, rather than in a specific hard coded order.

The protocol implementor should implement the sidl.rmi.Ticket and TicketBook interfaces for their protocol, even if nonblocking calls are not supported.

B. Server Side Interactions

The server side interactions are not as strongly scripted as the Client side, because they effect the user much less directly, and Babel RMI does not have any rules on wire communications. However, the protocol implementor must write a Babel Object Server (BOS) for their protocol, and there are a few interfaces they must implement and rules they must follow in order to correctly interface with the generated Babel RMI code. This section is a high level view of the BOS and Babel RMI interactions.

A Babel BOS will have 3 basic functions a client may call on it: creation of objects, allowing function calls on existing objects, and requesting an object be serialized. Babel already has a mechanism to create instances based on typeName via the sidl.Loader. Babel RMI adds the ability to both (1) access instances based on a string objectID and (2) to execute methods on instances by methodName.

1. Starting a BOS

Every user of BabelRMI must be able to access at least one BOS across the network. A BOS may be started as a standalone server by a small driver program, or it may be started in the background on a running client program for peer-to-peer communications. In order to make this easy for users, every BOS must implement two important interfaces, sidl.rmi.BaseServer, and sidl.rmi.ServerInfo. The sidl.rmi.BaseServer interface is for starting up the BOS, and the sidl.rmi.ServerInfo interface is for communicating with the Babel generated code. Babel can communicate with the BOS only if it's ServerInfo is registered with the sidl.rmi.ServerRegistry. (Note: Currently the ServerRegistry can only handle 1 BOS at a time.

```
interface BaseServer {
```

```
      int init(in string url, in int flags);
      long run();
    }

    interface ServerInfo {
      string getServerURL(in string objID);
      string isLocalObject(in string url);
      array<sidl.io.Serializable,1> getExceptions();
    }
```

The normal sequence of events looks like this example of running the Simple Protocol BOS:

```
sidlx_rmi_SimpleOrb echo=NULL;
char* url = "simhandle://localhost:9999";
int tid;
sidl_rmi_ServerInfo si = NULL;

echo = sidlx_rmi_SimpleOrb__create(&ex);SIDL_CHECK(ex);
sidlx_rmi_SimpleOrb_init( echo, url, 1, &ex);SIDL_CHECK(ex);
tid = sidlx_rmi_SimpleOrb_run( echo, &ex );SIDL_CHECK(ex);
si = sidl_rmi_ServerInfo__cast(echo,&ex);SIDL_CHECK(ex);
sidl_rmi_ServerRegistry_registerServer(si, &ex);SIDL_CHECK(ex);
sidl_rmi_ServerInfo_deleteRef(si,&ex);SIDL_CHECK(ex);

pthread_join(tid, NULL); //Optional PTHREAD join
```

The basic idea is, create the BOS, initialize and run it. Once it's running, cast it to a ServerInfo and register it with the ServerRegistry so that Babel can use it to export remote objects.

2. The InstanceRegistry

The measure of whether or not an object is available remotely is whether it is registered with the sidl.rmi.InstanceRegistry or not. The InstanceRegistry is a singleton class that maps objectIDs to local objects and vice-versa. Objects are added if they were created by a createRemote call from a remote machine, or when they are passed as arguments to a remote machine, or they may be added explicitly by the user. Objects may be registered in the InstanceRegistry under multiple objectIDs (aliases). They may be removed explicitly, or the Babel runtime system automatically removes them when the object reference count reaches zero and the object is destroyed. Note that there is no way for the Babel internals to differentiate between a remote and a local reference. so the object will persist in the registry until *all* references are deleted.

3. BOS Functionality

A BOS normally supports three basic functions: creation of new objects, serialization of an existing object to a new server, and executing a method on an existing object. How these are communicated to the BOS is completely up to the BOS implementor, but there are some nice interfaces to make the actual actions easy to implement. In general sidl.rmi.Call and sidl.rmi.Return should be used for the in arguments and out arguments respectively. Babel RMI requires these interfaces for passing in and out the arguments for method execution, but it is probably easiest to use them for creation and serialization as well.

1. Generic object creation usually consits of using the Babel's sidl_Loader and sidl_DLL classes to fload and create the object by classname. Then the new object must be added to the InstanceRegistry and the objectID returned to the user.
2. Object serialization occurs when a remote client needs a local copy of an object. They can therefore request the BOS send them a serialization of the object. The BOS implementor only needs to look up the object in the InstanceRegistry, cast it to sidl.io.Serializable, and call packSerializable on it to return the serialization to the remote client.
3. Method execution is similar. Babel provides the _exec builtin method on every object specifically for invoking methods by name. The BOS implementor gets the objectID and the method name. They then look up the object in the InstanceRegistry, and call _exec on the object passing in the method name and the Call and Return interfaces associated with the call. After executing the call, it returns the Return interface, which now holds all the out arguments.

4. Server Side Deletion & Disconnect

Objects are deleted and removed from the InstanceRegistry when their reference count goes to zero. This means that if a user wishes to continue to publish an object over RMI they should continue to hold a reference to an object as long as they wish to publish it.

Users also may explicitly remove an object from the InstanceRegistry by using the remove functions. This however, is highly discouraged. The user has no way of knowing what remote users may be accessing that object. Explicitly removing objects from the InstanceRegistry could have sever unintended consequences

The protocol implementor generally does not have to worry about removing things from the registry. The remote client will send deleteRefs, which can be called through the same _exec method that exists on every object. When the reference count reaches 0, the object will automatically be destroyed and removed from the registry.

5. Nonblocking

The BOS doesn't really need to know much about nonblocking. Nonblocking is really mostly a client side abstraction. If the protocol implementor decides to implement a nonblocking protocol, than the BOS will need some way to contact the client when the call is finished, but Babel RMI provides no interfaces or mechanisms for this, it's all up to the protocol implementor.

III. Implementation details

This section describes a handful of items covered in the last section in more detail. Most people may want to skip this section unless they have specific questions.

A. Client Side Interactions

1. Interface Connection

An interesting problem arises with connecting to remote interfaces. Connection to a remote object requires a local Babel RMI stub, which is basically a Babel IOR object with it's virtual function table filled in with RMI stub functions rather than local skel functions. The problem is that, by definition, Babel cannot instantiate interfaces, only classes. There is no place to even hold the InstanceHandle in an interface pointer.

The solution was to generate an anonymous class for each interface. The anonymous class only implements the interface it was generated for. On connection, this anonymous class in instantiated and cast up to the required interface. The anonymous class name is simply the interface name with a prepended underscore. Obviously, the user can only call _connect on this anonymous class, there is no way to _remote[Create] such a class.

2. Reference Counting

Babel RMI reference counting is even a little more complex than previously described. A Babel remote object has three distinct reference counts. It has a reference count on the stub, a reference count on the Instance Handle, and a remote reference count on the server (see Illustration1). When any of these reference counts reaches zero, the object at that point will be destroyed. The object is completely destroyed when the remote reference count reaches zero. The remote reference count really counts the number of InstanceHandles that point to that object.

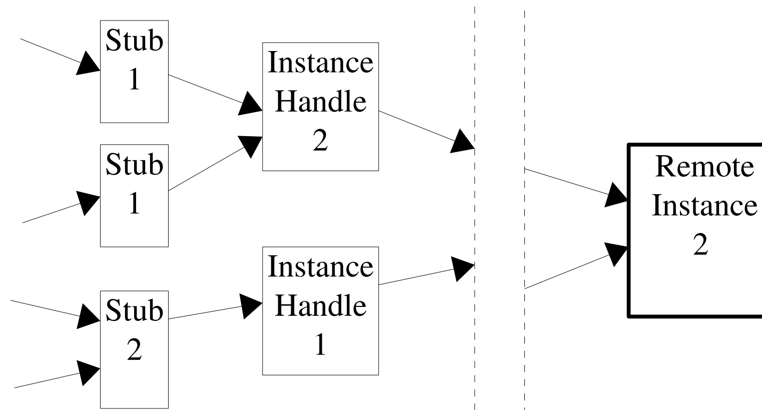

Illustration 1: This shows reference counts on a possible Babel RMI object.

It is possible for multiple InstanceHandles to a given remote object to exist on the same client. However, since creating and maintaining remote references is expensive, we try to reuse InstanceHandles as freely as possible.

So, in conclusion, The normal add and deleteRef functions will increment and decerement a reference count that only refers to THAT stub. When a stub is collected, it deleteRef's its InstanceHandle, and when an InstanceHandle is collected, it deleteRefs the remote instance. All of this is transparent to the user, who may freely reuse objects in the normal Babel manner. (Except that cast addRefs as of Babel 0.11.0)

3. RMI Cast

Traditional Babel casting is fairly simple because the IOR object is always really a complete representation of the most derived type. Casting merely consists of checking if the cast is legal, and, if the cast is to or from an interface, passing back a pointer to a different place in the IOR structure.

In RMI, if an object is passed as an argument, all we know about the object is the location of the object and the declared argument type. We do not know the actual type of the object. We therefore create an stub of the argument type, and connect it to the remote instance. If, as may often be the case, the argument type is not the most derived type of that object, we will have created a local stub that does not truly represent the “real” type of the remote instance. Therefore, a later RMI object downcast may need to create a new client stub that uses the same InstanceHandle. This is why we must addRef on cast, and why casting may result in a new stub. To understand this, consider the case where a stub, with one reference, is downcasted and creates a new stub. In order to maintain the old babel standard behavior, we would have to deleteRef the old stub, which would destroy it! The user would probably not expect that behavior.

Because the InstanceHandle only hold data about how to contact the remote object, not it's type or methods, we can reuse it. We reuse the InstanceHandle by calling a special internal Babel function defined in the stub called _IHConnect(). (InstanceHandleConnect) _IHConnect creates a stub, just like _connect, but instead of creating a new InstaceHandle though the ProtocolFactory like _connect does, it simply addRef's the existing InstanceHandle and reuses it.

There is one final odd issue with IHConnect. Since IHConnect is only called when an object is downcasted, there is no way to statically link in the required IHConnect function. Imagine downcasting sidl.BaseClass to a more derived type. There is no way to know statically all the derived types of sidl.BaseClass, there are simply too many. So instead, there is another singleton class called ConnectRegistry. The ConnectRegistry maps type names to IHConnect methods dynamically. Types are loaded into the registry when they are first seen in Babel, so they are accessible by the time users are able to cast to them.

B. Server Side Interactions

1. InstanceRegistry (again)

i) Adding Objects

There are three different ways an object may added to the InstanceRegistry, and therefore be published over RMI.

1. A remote machine may call create[Remote] on the currrent host. In this case the object is created and added to the InstanceRegistry by the BOS. This is a pure remote object, the current host's user code knows nothing of the object, and will only learn of it if the object is passed to the local host as a function argument. The object begins with a reference count of 1, which is held by the remote client. If that remote client calls deleteRef, the object is destroyed.
2. A local object may be implicitly added to the InstanceRegistry if it is passed as a function argument to a remote host. Thus the object becomes accessible through RMI, and the remote host gets a reference to the object. The object will remain accessible until its reference count goes to zero and the object is destroyed.
3. A local object may also be explicitly added to the InstanceRegistry by the user. This is often called "publishing an object." The easiest way would be to simply call getURL on the local object, which will automatically add the object to the InstanceRegistry if it is not already there and return the object's URL. However, the _getURL, method, along with every automatic method, gives the object an automatically generated objectID. The user can give an object a specific objectID by

adding it to the InstanceRegistry manually with the sidl.rmi.InstanceRegistry.registerInstance[ByString] method. Note that the InstanceRegistry allows aliasing. In other words, the user can add the same object under multiple names. However, two different objects cannot be entered under the same name. Therefore, when adding an object by name, there are four possible outcomes:

1. In the normal case, no other object has the same name, and the object is added to the InstanceRegistry under the user suggested name.
2. If the object already exits in the registry under a different name, the object is still added to the registry under the suggested name. The object will be in the registry twice, once under the old name, and once under the new suggested name.
3. If the object already exists in the registry under the suggested name, the call is idempotent, nothing happens.
4. However, if a different object already has the same name, the object will be added to the registry under a similar, but unique name. If the user wishes to remove the conflicting object, they may do that with removeInstance, and re-add the new object. However, the user should keep in mind that explicitly removing objects can be extremely dangerous, and discussed on Page 12.

ii) Reference Counting and the InstanceRegistry

Reference counting in the InstanceRegistry doesn't follow the normal Babel rules. When adding an object to the InstanceRegistry, that object is NOT addRef'd. This is so that the user will still have the only reference to the object. When the user wishes to destroy the object, they only need to deleteRef their handle to it, and the object will be removed from the InstanceRegistry. However, when an object is gotten from the InstanceRegistry, it is addRef'd, so the user of that object actually has a real handle to the object, and no one can destroy the object while they are using it. Finally, if removeInstance[ByString] is called, the object is not only removed from the InstanceRegistry, but a reference is passed back to the caller. This means that, like getInstance[ByString] the object is addRef'd when it is removed.

2. Passing Object References

i) Identifying Local Objects

It is quite possible that a client may pass an object to a server, the object is actually local to the server being called (or vice versa, if the client has a BOS). Imagine a case where a client has a remote object foo where the remote instance exists on server A. The client then calls a method on A, and passes foo by reference into that method. Babel can

automatically determine that the object is local to server A, and optimize out the network by using the local reference instead of making a remote reference.

This check is performed in the _connect() call, before the remote instance is created, Babel internals use the ServerRegistry singleton class to call the isLocalObject(url) method in the ServerInfo. This method is implemented by the protocol writer, and it should be able to use the information in the url to determine if the object is actually a local or a remote reference, and return true or false depending.

ii) Reference Counting

This section is devoted to detailing exactly how the Babel RMI generated code handles passing references to remote objects. This is actually fairly tricky, because a remote object has a locally instantiated stub and Instancehandle that we want to collect correctly as well as the remote object. In normal Babel there is an understood set of rules for how the user is expected to addRef and deleteRef Babel objects when passing them in and out of Babel function. The system described here is to allow the user to apply the same rules to passing object references through Babel RMI. This solution may not be an optimal solution, but it is simple and it works.

First, here is a review of what Babel expects the **user** to do:

- In arguments: The caller does not addRef before passing in an in argument. The callee does not addRef, **unless** the callee wishes to keep a reference to the object. The callee **must not** deleteRef an in argument, unless they had previously addRef'd it.
- Out/Return arguments: When an object is passed back as an out argument or a return value, the callee function is expected to pass back its own reference. In other words, if the callee wishes to save a reference to the object it is returning, it should addRef it. This allows the caller to deleteRef the object they receive at will.
- Inout arguments: Inout arguments are similar to out arguments, but in both directions. The caller passes in its reference to the object. The callee is therefore free to deleteRef the object and replace it with a new/different one. Again, the callee also passes its reference back to the caller, so it must addRef if it wishes to keep a copy of the object it is returning.

Now, here is what BabelRMI does to allow the user to follow exactly the same rules whether the object is remote or local:

i. In Object Arguments

When a remote reference is passed as an in argument through Babel, the reference count

on the remote server is automatically addRef'd on connect. This creates an RMI object with a stub reference count of one, and instance handle reference count of one, and an incremented remote reference count. When the callee function is exited, the in object reference is deleteRef'd. If the callee did not keep a reference, (ie the library code did not addRef the in object argument) the last deleteRef destroys the stub and instanceHandle, and decrements the remote reference count; thereby maintaining the same reference count in the remote object. If the callee does keep a reference, it will addRef and the local stub count will go up to 2. The deleteRef on return will simply decrement the stub count to 1, the connection is maintained, the remote count will retain it's incremented reference count.

ii. Out Object Arguments and Return Values

The idea of a remote callee returning its reference to the remote caller creates a unique problem for Babel. Theoretically, the object should not be deleteRef'd because the reference is passed back to the caller. However, if the object being passed back is remote, not deleteRefing will cause the stub and instanceHandle to be leaked! However, if we pass the reference back and deleteref, the object may be destroyed by the deleteRef before the caller has time to addRef the newly received object. Therefore, we need to "transfer the reference." This means that the server side Babel internals should "remote addRef" the object, pass the reference back, and then deleteRef the local copy of it. The caller should then connect without addRefing. (The addRef has already happened remotely on the callee.) (See illustration 2) For more information on _raddRef, see page 28, for more on connection without addRefing see page 27.

iii. InOut Object Arguments

Inout arguments are similar to out arguments, and so the solution is similar. Babel transfers the reference both ways. The caller backend code remote addrefs and local deleterefs when calling, and the callee does the same on the return.

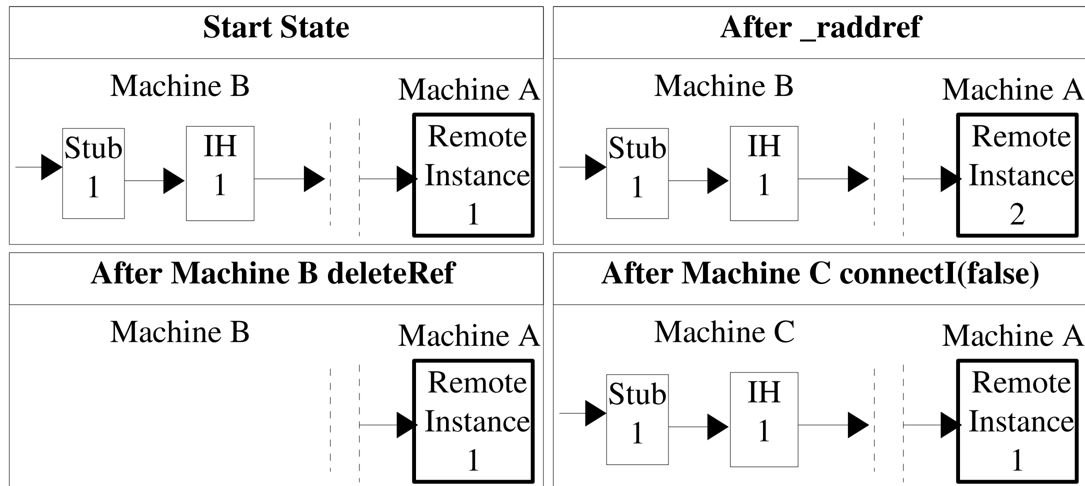

Illustration 2: Example of transfering a reference from Machine B to Machine C

3. Passing Objects By Copy/Serialization

When passing an object by copy through Babel RMI, it must be serialized. This means the object and all it's data must somehow be serialized into a byte-stream that can be reconstructed into an object on the receiving side. This is surprisingly straightforward.

The protocol implementor implements two methods declared by sidl.io.Serializer and sidl.io.Deserializer, packSerializable, and unpackSerializable. packSerializable starts up the serialization process, usually by serializing the most derived SIDL name of the class so that unpackSerializable can use the Babel runtime system to create the object by name. It then calls on the object implementor's code to serialize the object data.

As mentioned previously, any object that a user intends to be passable through Babel RMI by copy, must implement sidl.rmi.Serializable. sidl.rmi.Serializable declares 2 functions:

```
interface Serializable {
   void packObj( in Serializer ser );
   void unpackObj( in Deserializer des );
}
```

These, serialize and unserialize the data the object holds. packObj takes a serializer so the protocol implementor may stream data directly to the network if they so desire, and a gigantic buffer doesn't need to be passed around. The object implementor also needs to call super.packObj to unserialize the data of the superclasses. It doesn't really matter if super.packObj is called first, or if the local data is serialized first, as long as it's done the same way in unpackObj.

How to serialize the object data is completely up to the object implementor. Usually they will probably want to use the sidl.io.Serializer's basic data serialization methods, but in certain circumstances they may choose a difference approach, such as using the Java

serialize capability to serialize a Java data structure. The protocol implementor should not make any assumptions about how object data is serialized.

4. Exceptions

All exceptions are passed though Babel by copy. Therefore, sidl.BaseException extends sidl.io.Serializable. The Babel defined exceptions have packObj methods written for them, including sidl.SIDLException, a commonly used base class for exceptions. A user that creates a new exception, must make sure that any important data is correctly serialized.

Of course, if the user chooses to extend a Babel Runtime defined exception, the data defined in the Babel Runtime exceptions is already serialized. For sidl.SIDLException, this is a string message and a trace that are included with the exception. For sidl.rmi.NetworkException, this is both the data from sidl.SIDLException (serialized by a call to super.packObj) and a "network hop count" that is incremented each time the exception is serialized. This is useful for figuring out where exactly the exception came from.

5. Connecting and Casting from the IOR

The _exec method implementation is in the IOR so that the correct version can be called virtually on a sidl.BaseClass object. Unfortunatly, there is a cost to this ability. Any arguments that are passed into the method must be resolved inside _exec. Of course, these object arguments will rarely have any connection to the object the method is actually being called on.

If an object argument is passed by reference, somehow _connect() must be called on that reference to get an actual instantiation of the object. If the object argument is passed by copy, the protocol and Babel internals can take care of instantiating the object, but they can only return a Serializable, so the object must be cast to the type the method being called is actually expecting. Somehow we need a way of calling _connect() and _cast(), which are both declared and defined in the Stub, from the IOR of a totally different object. The IOR has no possible way of doing this directly because it does not include or link to any stub files. However, depending on the language binding, either the Skel file or the Impl file does. Therefore, Babel generates special call-though functions in the Skel and Impl files. They are called fcast and fconnect, and one is generated in each file for each object type that is passed as an argument. The user, and the protocol writer, can safely ignore these functions. For more on _fcast and _fconnect, see page 27.

III. Protocol Implementor's Guide

This section gives an overview of what the protocol implementor needs to write in order

to interact correctly with Babel RMI and the user. It also includes a few general hints on parts that might be confusing.

A. Overview

The protocol implementor must implement the following interfaces as part of the client side of their protocol:

```
sidl.rmi.InstanceHandle      The heart of the client
sidl.rmi.Invocation          Argument Serializer (in arguments)
sidl.rmi.Response            Argument Deserializer (out args)
sidl.rmi.Ticket              Nonblocking response support
sidl.rmi.TicketBook          A bunch of tickets
```

The protocol implementor must implement the following interfaces as part of the server side of their protocol:

```
sidl.rmi.Call                Argument Deserializer (in args)
sidl.rmi.Return              Argument Serializer   (out args)
sidl.rmi.BaseServer          Basic ORB user interface
sidl.rmi.ServerInfo          Basic ORB registry interface
```

The following interfaces are therefore indirectly implemented:

```
sidl.io.Serializer           A set of marshalling methods
sidl.io.Deserializer         A set of unmarshalling methods
```

1. Serializer

sidl.io.Serializer is an interface for serializing any arguments that may be passed through Babel. It is extended by both the sidl.rmi.Invocation (for clients) and the sidl.rmi.Return (for servers) interfaces. Serializer has methods for Serializing each Babel basic type, and arrays of those basic types, as well as Serializable objects, arrays of Serializable objects, and generic arrays. (Object references are passed as urls, so they use packString directly.) (See page 28 for the functions in Serializer)

Each argument is serialized with a string key as well as the data. This is for protocols that may reorder arguments in transport. Babel RMI always automatically inputs the keys for arguments, and serializes and deserializes in the same order, so the protocol writer may choose to either maintain names or ordering across the network, both do not need to be maintianed.

The protocol implementor is free to implement argument serialization however they wish.

This could mean serializing into a buffer, or streaming across the network. It could also mean serializing to a binary or a text based format. Babel RMI doesn't care.

1. Deserializer

sidl.io.Deserializer is the opposite of sidl.io.Serializer. It is an interface for deserializing any arguments that have been passed through Babel. It is extended by both the sidl.rmi.Resonse (for clients) and the sidl.rmi.Call (for servers) interfaces. Deserializer has methods for unpacking each Babel basic type, and arrays of those basic types, as well as Serializable objects, arrays of Serializable objects, and generic arrays. (Object references are passed as urls, the _exec call unpacks the url as a string and calls connect on it directly from the IOR.)(See page 29 for the functions in Deserializer)

Each argument was serialized with a string key as well as the data, and Babel RMI will deserialize them in the same order and with the same names as they were serialized with.

B. Client Side

1. InstanceHandle

sidl.rmi.InstanceHandle is the heart of any client side protocol implementation. An InstanceHandle generally represents one remote object, and is the main way of a client maintaining it's link to that object. However, multiple stubs may point to the same InstanceHandle. InstanceHandle provides all the basic methods that a user may need on an object, including: initializing a link to a remote object, getting an object's url, and starting an method invocation on an object. In fact, InstanceHandle is the sole source of the Invocation interface, which it returns from the createInvocation method. (To see all the methods in InstanceHandle, see page 31).

There is one odd method on the InstanceHandle. initUnserialize returns a Serializable. This method is used when server needs an object to be copied to it from another server. For example, a client has a foo object on Server B, and passes that foo object by copy to Server C. Rather than having to serialize the object through the client to C, we would prefer to have C be able to request the object from B, and have B serialize the object directly to C. This is what initUnserialize does. It requests a copy of an object by url.

2. Invocation

sidl.rmi.Invocation is how the client calls methods on remote objects. Invocation extends sidl.io.Serializer. It is returned by sidl.rmi.InstanceHandle.createInvocation. It declares three related methods:

```
Response invokeMethod();
Ticket invokeNonblocking();
```

```
void invokeOneWay();
```

After serializing the in arguments via the calls inherited from sidl.io.Serializer, BabelRMI will call one of the above functions, depending on whether the call is blocking, nonblocking, or oneway. The protocol writer must support all three calls, although strictly blocking protocols may simply emulate nonblocking and oneway as “Simple Protocol” does.

InvokeNonblocking is the only way to get sidl.rmi.Ticket object, but both invokeMethod and a Ticket object can return a sidl.rmi.Response.

3. Response

sidl.rmi.Response is simpler than sidl.rmi.Invocation. It's sole purpose is deserializing the out arguments from a method. It only declares one new function, sidl.BaseException getExceptionThrown(). This is a safe function for checking if an exception was thrown by the remote call. If this method returns NULL, it is safe to unpack the arguments from the Response.

4. Nonblocking Implementation

The implementation of nonblocking will vary between protocols, however, we generally expect that implementors will use something like a nonblocking write in combination with a select call to wait for the response. In this case, all calls would be dispatched with a nonblocking call, and their connections left open. select() would then be used to wait on all ports where requests have been sent, waiting for a response. The user can use the Ticket or TicketBook interfaces to find out when the response arrives, or block waiting for it. Blocking calls would be implemented through the same system, they would just always block waiting for the response immediately after the call is made, inside the Protocol.

In a strictly blocking protocol, there are two options for emulating nonblocking calls. “Simple Protocol” uses the simplest. It doesn't support anything like a nonblocking call. When invokeNonblocking is called, blocking call is made, and the call actually blocks until the method returns.

Another possible way would be to have the client have an open accept port, so when a nonblocking method finishes, the remote sever would open a new connection to the client to return the out arguments. This would allow real nonblocking calls an a strictly blocking protocol, but the system has serious disadvantages.

Both sidl.rmi.Ticket and sidl.rmi.TicketBook can be read about in the Babel User's guide.

B. Server Side

1. ServerInfo

sidl.rmi.ServerInfo is an interface that all BOSs must implement one way or another. It provides three important functions that are accessed through the sidl.rmi.ServerRegistry singleton class.

```
string getServerURL(in string objID);
string isLocalObject(in string url);
array<sidl.io.Serializable,1> getExceptions();
```

getServerURL returns the URL of the server this ServerInfo represents. If the objID argument is NULL, it only returns the server URL, otherwise, the objectID is incorperated in the URL.

IsLocalObject is uses the URL to determine if the object referenced is local to this BOS.

GetExceptions is a different sort of function. In most cases, BOS exceptions should be returned to the client. However, some of these exceptions, such as network failure, cannot be returned. Instead they are logged in the server. They may be retrieved from the BOS via the getExceptions call. This will hopefully be useful in troubleshooting.

2. Call

sidl.rmi.Call's sole purpose is deserializing the in arguments from a method call. It does not declare any new functions, it only extends sidl.io.Deserializer. It's only purpose is to logically compliment sidl.rmi.Return. It is one of the arguments to _exec.

3. Return

sidl.rmi.Return extends sidl.rmi.Serializer, it serializes the out arguments from a method call back to the client. It declares one new method: throwException, which takes a an exception to throw. Because there is no way to know when an exception could be thrown, throwException may have to erase any previously serialized out arguments, and even change the returning method data to alert the client that an exception was thrown. This, however, is all internal to the protocol. The client side Babel RMI will call sidl.rmi.Response.getExceptionThrown() first after receiving a response, so the client need someway to know immediately if there has been an exception. However, Babel RMI does not dictate how this must be done.

4. Basic BOS Calls

There are three basic calls that clients may make that any BOSs must support. They are:

object creation, object serialization, and method invocation (on an existing object.) These mirror the init functions on the InstanceHandle. The normal sequence of events for these calls:

- Object Creation:
  1. Get the class name from the call
  2. Use sidl.Loader and sidl.DLL to instantiate an object of that type
  3. Register the object with the InstanceRegistry
  4. Return the objectID to the client
- Object Serialization:
  1. Get the objectID from the call
  2. Get the object from the InstanceRegistry
  3. Call sidl.rmi.Return.packSerializable on the objection
  4. return to the client
- Method Execution
  1. Get the objectID and the method name from the Call
  2. Get the object from the InstanceRegistry
  3. Call _exec on the object with the method name, the sidl.rmi.Call, and the sidl.rmi.Return.
  4. Return to the client.

V. Babel RMI Reference Manual

This section is a reference manual for all of the interfaces and methods in Babel RMI, both builtin and SIDL declared. This section mostly duplicates what can be found in sidl.sidl, but hopefully more clearly.

A. RMI Builtin Methods

Builtin methods start with a leading underscore (which is not expressible in SIDL) and get generated by the backends for each language. This section is divvied up by whether the function is exposed to the user or not, and whether it is static or virtual.

I) User-Exposed Static Methods

1. _create[Remote](in string url)

Similar to _create(), this is behaves like a static final method. It differs from _create in that the url argument is used to determine on which server (BOS) this argument is to be created on. Babel RMI copies the first portion of the url, up to the first non-alphanumeric character to find the protocol in the protocol factory. The complete, unmodified url is then passed to the protocol itself for further parsing. The url might take the form:

protocol://server:port/

but the specifics of the format are up to the protocol developer. Babel is only concerned with the first portion (the “scheme”).

_createRemote() exists only for classes and, in the case of failure, returns NULL and throws an exception indicative of the failure.

2. _connect(in string url)

Instead of creating a new remote instance, this method creates a new connection to an existing remote object. The object's reference count will be incremented for the duration of the connection. Another important difference between _connect and _create is that interfaces can be connected to remote instances.

Like the url in _create[Remote](), Babel is not concerned with the url beyond the protocol section, but unlike _create[Remote](), the objectID of the remote instance must be communicated somehow. The url might take the form:

protocol://server:port/objectID

II) User-Exposed Virtual Methods

1. _getURL()

This function returns the URL of the object it is called on. This URL may be used for connection. This function is mostly useful for passing an object by reference, Babel uses it internally, but it also automatically adds objects to the InstanceRegistry if they are not already there. It may be useful in a variety of ways for the user.

2. _isRemote/_isLocal()

_isRemote is a builtin function that returns true if the object it is called on is implemented remotely, false otherwise. _isLocal is the logical opposite. This is used inside of Babel to regulate reference counting, it is exposed to the user for convenience and performance reasons.

_isLocal is not actually built into the epv, it is implemented in the stub as logical not _isRemote.

2. _exec(in string methodName, in sidl.rmi.Call inArgs, in sidl.rmi.Return outArgs)

This method provides a fundamentally new capability for Babel objects. The _exec function allows all the methods in an object to be invoked by name. The only stipulation being, method name must always be in the long form, since overloading is impossible to support. Essentially, for each method a function is defined that unpacks the in arguments from inArgs , executes the normal EPV entry, packs the out arguments into outArgs, and returns. The _exec function consists for a static table of function string name-function pointer pairs. When called, the _exec function looks up the correct function pointer by name in the table, and calls it.

Although static methods are not supported in RMI, in order to more fully support reflexivity, a static exec builtin function is planned for Babel, it will be called _sexec.

Furthermore, in order to support reflexivity in normal Babel, a pseudo protocol is planned that will serialize arguments locally, rather than over a network.

III) Internal Static Methods

1. _IHConnect(in InstanceHandle instance)

This connect is for internal Babel use only. This connect is used in remote object downcasts when the stub cannot be reused. The InstanceHandle can always be reused in these cases, however. For more information on InstanceHandle reuse, see page 14.

2. _connectI(in string url , in bool ar)

_connectI is the connect method used internally in Babel. It works the same as _connect as previously described, but also takes a boolean that determined if the remote instance will be addRef'd or not. Normally when a new stub and InstanceHandle are created on a remote instance, the remote instance is addRef'd as there is now a new reference to it. However, when transferring the reference for out and inout arguments, (as described on page 18) the remote instance has already been addRef'd in expectation of the creation of the new stub and InstanceHandle. In that case, the remote instance should not be addRef'd when it gets connected.

The user exposed _connect method always addRef's the remote instance.

3. _fconnect/_fcast

_fcast and _fconnect are special call through functions for performing _connect and _cast

on object arguments seen in the IOR function _exec. The IOR is no linked to any stubs, so in order to get access to _cast and _connect for object arguments, Babel calls though the skel files, and possibly the Impl files (depending on the language binding), which are linked to the object argument stubs. In order to avoid naming conflicts, _fcast and _fonnect have very long names:

```
skel_source_class_fconnect_target_class()
skel_source_class_fcast_target_class()
impl_source_class_fconnect_target_class()
impl_source_class_fcast_target_class()
```

Where source class is the class attempting to call _connect on the class named target class. The call is prepended with "skel" if the call defined in the skel, and "impl" if it is defined in the impl. The IOR always calls the skel version of the method, which, in turn, may either call the impl version or call the stub version directly, depending on the language binding.

II) Internal Virtual Methods

1. _raddRef()

This function is for internal Babel use only, it is not exposed to the user through the stub. This function increments the remote reference count on a remote object, without touching the reference count of the stub or the InstanceHandle. On a local object, it simply calls the normal addRef function. This is only used for "transferring the reference" as described the section 18.

B. Interfaces Added to sidl.sidl

1. sidl.io.Serializer interface

Can be implemented by RMI protocols, IO libraries, maybe even checkpoint/restart and process migration.

The normal pack functions are fairly straightforward, but the Array functions have a few interesting features. Normal Babel arrays can be reshaped into certain dimensions in passing, since Babel RMI if copying the arrays anyway, Babel RMI supports doing this during serializaion. It is possible to ensure an array is in a certain order by passing in ordering and dimension requirements. Ordering should represent a value in the sidl_array_ordering enumeration in sidlArray.h. If either argument is 0, it means there is no restriction on that aspect. The boolean reuse_array flag is set to true if the remote unserializer should try to reuse the array that is passed into it or not.

Also, the packGenericArray can pack arrays of any size, dimension, and type. It

automatically finds the metadata. However, this function cannot ensure an array is of a certain size or type. In the future we may remove all the array serialization methods except for packGenericArray, and factor in ensuring data and type into that method to simplify the interface.

```
interface Serializer {
  void packBool( in string key, in bool value );
  void packChar( in string key, in char value );
  void packInt( in string key, in int value );
  void packLong( in string key, in long value );
  void packFloat( in string key, in float value );
  void packDouble( in string key, in double value );
  void packFcomplex( in string key, in fcomplex value );
  void packDcomplex( in string key, in dcomplex value );
  void packString( in string key, in string value );
  void packSerializable( in string key, in Serializable value );
  void packBoolArray( in string key, in array<bool> value,
                      in int ordering, in int dimen,
                      in bool reuse_array );
  void packCharArray( in string key, in array<char> value,
                      in int ordering, in int dimen,
                      in bool reuse_array );
  void packIntArray( in string key, in array<int> value,
                     in int ordering, in int dimen,
                     in bool reuse_array );
  void packLongArray( in string key, in array<long> value,
                      in int ordering, in int dimen,
                      in bool reuse_array );
  void packOpaqueArray( in string key, in array<opaque> value,
                        in int ordering, in int dimen,
                        in bool reuse_array );
  void packFloatArray( in string key, in array<float> value,
                       in int ordering, in int dimen,
                       in bool reuse_array );
  void packDoubleArray( in string key, in array<double> value,
                        in int ordering, in int dimen,
                        in bool reuse_array );
  void packFcomplexArray( in string key, in array<fcomplex> value,
                          in int ordering, in int dimen,
                          in bool reuse_array );
  void packDcomplexArray( in string key, in array<dcomplex> value,
                          in int ordering, in int dimen,
                          in bool reuse_array );
  void packStringArray( in string key, in array<string> value,
                        in int ordering, in int dimen,
                        in bool reuse_array );
  void packGenericArray( in string key, in array<> value,
                         in bool reuse_array );
  void packSerializableArray( in string key,
                              in array<Serializable> value,
                              in int ordering, in int dimen,
                              in bool reuse_array );
}
```

2. sidl.io.Deserializer interface

The opposite of sidl.io.Serializer. We also expect this to be implemented by RMI

protocols, IO libraries, maybe even checkpoint/restart and process migration. The same suggestions apply to the Array functions as well.

```
interface Deserializer {
  void unpackBool( in string key, out bool value );
  void unpackChar( in string key, out char value );
  void unpackInt( in string key, out int value );
  void unpackLong( in string key, out long value );
  void unpackOpaque( in string key, out opaque value );
  void unpackFloat( in string key, out float value );
  void unpackDouble( in string key, out double value );
  void unpackFcomplex( in string key, out fcomplex value );
  void unpackDcomplex( in string key, out dcomplex value );
  void unpackString( in string key, out string value );
  void unpackSerializable( in string key, out Serializable value );
  void unpackBoolArray( in string key, out array<bool> value,
                        in int ordering, in int dimen,
                        in bool isRarray );
  void unpackCharArray( in string key, out array<char> value,
                        in int ordering, in int dimen,
                        in bool isRarray  );
  void unpackIntArray( in string key, out array<int> value,
                       in int ordering, in int dimen,
                       in bool isRarray  );
  void unpackLongArray( in string key, out array<long> value,
                        in int ordering, in int dimen,
                        in bool isRarray  );
  void unpackOpaqueArray( in string key, out array<opaque> value,
                          in int ordering, in int dimen,
                          in bool isRarray  );
  void unpackFloatArray( in string key, out array<float> value,
                         in int ordering, in int dimen,
                         in bool isRarray  );
  void unpackDoubleArray( in string key, out array<double> value,
                          in int ordering, in int dimen,
                          in bool isRarray  );
  void unpackFcomplexArray( in string key, out array<fcomplex> value,
                            in int ordering, in int dimen,
                            in bool isRarray  );
  void unpackDcomplexArray( in string key,
                            out array<dcomplex> value,
                            in int ordering, in int dimen,
                            in bool isRarray  );
  void unpackStringArray( in string key, out array<string> value,
                          in int ordering, in int dimen,
                          in bool isRarray  );
  void unpackGenericArray( in string key, out array<> value);
  void unpackSerializableArray( in string key,
                                out array<Serializable> value,
                                in int ordering, in int dimen,
                                in bool isRarray  );
}
```

3. sidl.io.Serializable interface

This would be implemented by classes that can pack and unpack themselves to a Serializer/Deserializer. All exceptions implement this interface, whether directly or

indirectly.

```
interface Serializeable {
  void pack( in Serializer ser );
  void unpack( in Deserializer des );
}
```

4. sidl.rmi.InstanceHandle interface

Implemented by the RMI library. (Was called "Connection" in earlier versions, but the name was changed in acknowledgment that this object may persist over multiple network connections. This interface is the heart of any RMI protocol implementation. It manages the connection to the remote object, or actively connects to the remote object on demand. This interface is not normally exposed to the user, Babel RMI passes user requrest on to this interface. This interface is held in the d_data pointer of a remote object, along with the stub reference count.

initCreate is called by _create[Remote](). InitCreate should actually handle the remote object creation request through communication with the remote server. It should return TRUE on success, FALSE and an exception on failure.

initConnect is called by _connect() and _connectI(). InitConnect should actually handle the remote object creation request through communication with the remote server. initConnect takes a boolean ar, which is the same boolean as in connectI(). If TRUE, the remote instance should be addRef'd, if FALSE, the remote instance should not be addRef'd. initConnect should return TRUE on success, FALSE and an exception on failure.

InitUnserialize is something of a special case, since the InstanceHandle it is called on is never really part of a remote object. It handles requesting the serialization of a remote object from a remote server to the local Babel process. initUnserialize is called through the ProtocolFactory singleton class.

```
interface InstanceHandle {
      bool initCreate( in string url, in string typeName );
      bool initConnect( in string url, in bool ar);
      sidl.io.Serializable initUnserialize( in string url);
      string getProtocol();
      string getObjectID();
      string getObjectURL();
      Invocation createInvocation( in string methodName );
      bool close();
}
```

5. sidl.rmi.Invocation interface

Implemented by the protocol. Mostly for serializing in arguments to a method on the client. When the serialization of in arguments is finished, one of the three methods below

is invoked, after which the Invocation is useless, and should be destroyed by the caller via deleteRef.

```
interface Invocation extends sidl.io.Serializer {
  Response invokeMethod();
  Ticket invokeNonblocking();
  void invokeOneWay();
}
```

6. sidl.rmi.Response interface

Implemented by the protocol. Encapsulates the response of a single method invocation. Mostly for deserializing out arguments on the client, but before beginning deserialization, getExceptionThrown() should always be called to check if an exception was thrown from the server. If the return value is NULL, then it is safe to deserialize the out arguments.

```
interface Response extends sidl.io.Deserializer {

  /** if returns null, then safe to unpack arguments */
  sidl.BaseException getExceptionThrown();
}
```

7. sidl.rmi.Call interface

Implemented by the protocol. Encapsulates the in arguments of a single method invocation on the serverside.. Mostly for deserializing in arguments on the server, in the _exec call. (It is one of the arguments to _exec.)

```
interface Call extends sidl.io.Deserializer { }
```

8. sidl.rmi.Response interface

Implemented by the protocol. Encapsulates the response of a single method invocation. Mostly for serializing out arguments on the server. It declares one special function for throwing exceptions from the server to the client. throwException() is expected to stopt the serialization of further arguments if it is called.

```
interface Return extends sidl.io.Serializer {
  void throwException(in sidl.BaseException ex_to_throw);
}
```

9. sidl.rmi.BaseServer interface

This interface must be implemented by all Babel Object Servers (BOSs). It is the user interface for the server, so the user will have a standard easy way to start up all servers. There may also be special server specific methods for defining extra options.

init takes a protocol specific url and a protocol specific set of flags. The url should communicate all the information needed about where the server should start up, and the flags should be used for controlling simple boolean options such as error output

verbosity.

```
interface BaseServer {
  int init(in string url, in int flags);
  long run();
}
```

10. sidl.rmi.ServerInfo interface

This interface must be implemented by the server side protocol, probably by the BOS itself. ServerInfo was mostly written to handle the problems that arise in exporting local objects as RMI objects. If a user wishes to have the ability to export local objects over RMI, they must run an BOS and resister that BOS's ServerInfo with the ServerRegistry. (Registration with the ServerRegistry could also be handled by the BOS during construction.)

The exception is the “getExceptions()” method, which returns an array of exceptions that were thrown from the BOS, but could not be returned to the client for some reason.

```
interface ServerInfo {
  string getServerURL(in string objID) throws NetworkException;
  string isLocal(in string url) throws NetworkException;
  array<sidl.io.Serializable,1> getExceptions();
}
```

C. Singleton Classes Added to the Babel Runtime System

1. sidl.rmi.InstanceRegistry (singleton)

Generates unique names for each registered instance, and serves up instances by name. It also reverse maps from object instance to registered name. Instances can be added under multiple names (aliasing) but multiple instances cannot be added under the same name.

InstanceRegistry does not addRef objects when they are stored, but does addRef objects when they are returned (gotten out of the registry). See page 16.

```
class InstanceRegistry {
  static string registerInstance( in sidl.BaseClass instance );
  static string registerInstance[ByString]( in sidl.BaseClass
             instance, in string instanceID);
  static sidl.BaseClass getInstance[ByString]( in string instanceID );
  static string getInstance[ByClass]( in sidl.BaseClass instance );
  static sidl.BaseClass removeInstance[ByString]( in string instanceID );
  static string removeInstance[ByClass]( in sidl.BaseClass instance );
}
```

2. sidl.rmi.ProtocolFactory class (singleton)

Serves up protocols (e.g. Proteus, “Simple Protocol”) using sidl.Loader based on a

protocol shortname (prefix) string that is found on the front of the URL and almost definitely not the same as the class name.

This singleton class is the starting point of creation and connection of objects because it needs to load up the protocol. It is also used when specifically requesting a copy of a remote object. (unserializeInstance)

```
class ProtocolFactory {
  static bool addProtocol( in string prefix, in string typeName );
  static string getProtocol( in string prefix );
  static bool deleteProtocol( in string prefix );
  static InstanceHandle createInstance( in string url,
                                        in string typeName );
  static InstanceHandle connectInstance( in string url, in bool ar);
  static sidl.io.Serializable unserializeInstance( in string url);
}
```

3. sidl.rmi.ConnectRegistry class (singleton)

This class is for Babel internal use only, it maps symbol names to IHConnect functions. This is part of the system used to downcast remote Babel RMI objects.

```
class ConnectRegistry {
  static void registerConnect( in string key, in opaque func);
  static opaque getConnect( in string key );
  static opaque removeConnect( in string key );
}
```

4. sidl.rmi.ServerRegistry class (singleton)

This singleton class is how Babel generally allows access to the current ServerInfo interface. It is recommended that only one BOS be run per Babel process, since the ServerRegistry does not offer support for having multiple BOSs.

The user will not have much use for this, except for the rare advanced user who needs getExceptions() for BOS debugging purposes.

```
class ServerRegistry {
  static void registerServer(in sidl.rmi.ServerInfo si);
  static string getServerURL(in string objID);
  static string isLocal(in string url);
  static array<sidl.io.Serializable,1> getExceptions();
}
```

D. Babel RMI Exception Hierarchy

With RMI comes all the blessings and curses of network communication. Among the curses of network communication is the fact that it is inherently unstable. The developer needs some mechanism to allow recovery in the case of network failure. The best way of doing this is to have any method that uses RMI throw a NetworkException.

(NetworkException extends RuntimeException, so any method may throw one.) However, not all network exceptions are equal, some may be recoverable in some cases, and in other cases fatal. Therefore Babel has a fairly extensive exception hierarchy defined in sidl.sidl in order to communicate specific types of network failure. Available Network Exceptions:

UnknownHostException - Thrown when Hostname lookup fails. (DNS failure)

BindException - Usually means that the requested port is in use

ConnectException - Connection Refused, Host Down, etc.

NoRouteToHostException - Router is probably down.

TimeOutException - Request timed out.

UnexpectedCloseException - Network dropped connection because of reset,

Software caused connection abort, Connection reset by peer, etc.

ObjectDoesNotExistException - No such object on the server

Protocol Exception - Non specific protocol error

MalformedURLException - The protocol cannot parse the URL

NoServerException – There is no local BOS registered.