UCLA
COMPUTATIONAL AND APPLIED MATHEMATICS

NetApp Wrappers : Higher level construction of distributed applications using the cam.netapp Package

Christopher R. Anderson

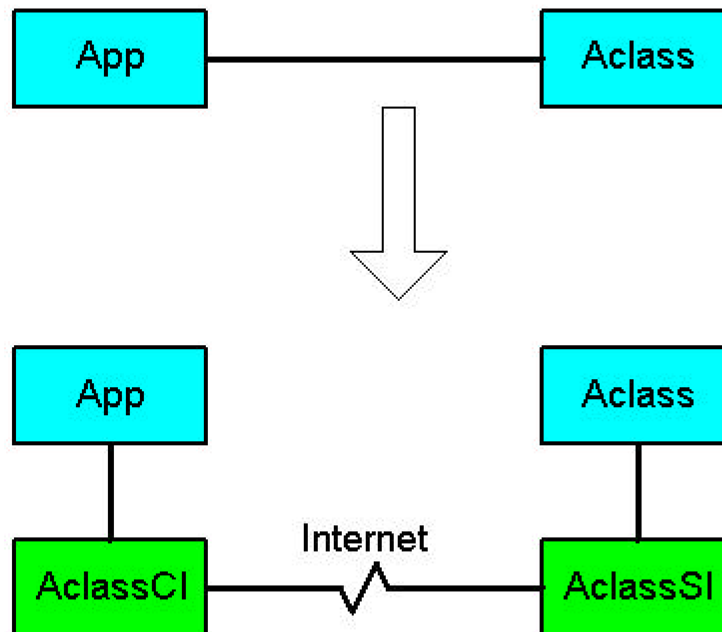
September 1998
CAM Report 98-40

Department of Mathematics
University of California, Los Angeles
Los Angeles, CA. 90095-1555

NetApp Wrappers :

Higher level construction of distributed applications using the cam.netapp package.

Abstract : This report describes a technique that enables one to create a distributed version of an application that is originally composed of classes interacting through standard method calls. The technique involves the construction and use of auxiliary or ``wrapper" classes. A utility program for automatically generating the wrapper classes is described. The cam.netapp package is used to provide the communications infrastructure. Extensions of the technique can be made to other communications infrastructures, e.g. that provided by the Java RMI package.



These software components were developed in conjunction with the research supported by Air Force Office of Scientific Research Grant F49620-96-I-0327 and National Science Foundation/ARPA Grant NSF-DMS-961584

Chris Anderson
Dept. of Mathematics
UCLA Los Angeles, CA 91555
anderson@math.ucla.edu
(C) UCLA 1998

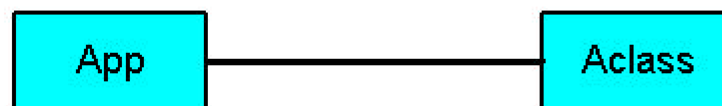
1. Introduction

Java contains many packages and constructs that facilitate the creation of "distributed" applications --- applications whose constituent classes are running on geographically separated machines. However, the process of learning about these constructs (or just remembering them) and creating the requisite code for a distributed application can be very time-consuming. What's needed is a software infrastructure that assists one in putting classes in a network accessible form and enables the creation of applications that utilize both ``local" and ``remote" class instances. Ideally the use of this infrastructure should be as simple as creating and ``publishing" Web pages.

In particular, the use of the infrastructure should not require knowledge of the networking/communications constructs that are are used to implement the infrastructure.

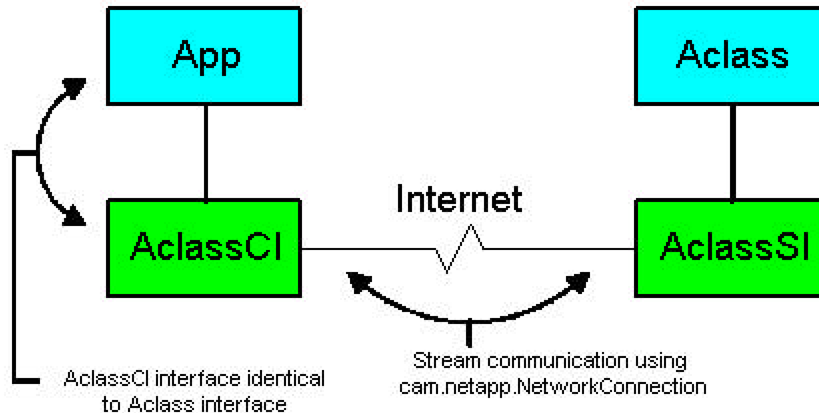
There are a variety of ways that this infrastructure can be created. For example, if the communication between components is constrained to be through stream connections, than one can create a modest infrastructure that possesses these features. The classes in the cam.netapp [1] [2] package provide such an infrastructure. While it is not difficult to create components that communicate via streams, it's a non-standard programming style. One therefore seeks an infrastructure that allows code components to communicate via standard method calls. One infrastructure that provides this is the Java RMI package . The general idea behind the RMI package is to provide an infrastructure that allows one to create client-server applications. The client classes execute on a local machine and communicate over the network with server classes executing on a remote machine. The client classes have the capability of invoking server class methods using standard method calls. The default use of this package requires one to explicitly create classes that implement either a ``client'' structure or a ``server'' structure. If one has an existing application then there can be a bit of programming to get the application into the appropriate client-server form. In an effort to minimize the amount of programming it takes to get components into a network accessible form, we describe an alternate infrastructure that supports code components that communicate via standard method calls.

The approach is based upon "wrapper" classes. The general idea for using wrapper classes is as follows: Suppose one has an application with the following structure :



In the application, we consider App the "front", or client, and Aclass, the "back" or server class. To keep things simple for now, we assume that App only accesses methods of Aclass. The challenge is to figure out how, with minimal programming, this application can be made to run in a distributed fashion e.g. App executing on one machine and Aclass executing on another.

One means of creating such a distributed application in which App and Aclass are separated is to have App communicate with another class possessing the same interface as Aclass, but whose functionality is provided by a remote instance of Aclass. This remote functionality is obtained through remote method requests sent via a stream connection (implemented using cam.netapp.NetworkConnection) to a class on a remote machine that invokes the appropriate methods of an Aclass instance and then returns the results. Thus a distributed version of the application has the form



Here AclassCI (which stands for Aclass (C)lient (I)mplementation) is the class whose interface is identical to Aclass and communicates with AclassSI (which stands for Aclass (S)erver (I)mplementation). AclassSI contains a local instance of Aclass to obtain its functionality. AclassCI and AclassSI are the wrapper classes utilized in the creation of this distributed implementation. Once the wrapper classes are created for Aclass, identical programming constructs can be use for either local or remote use of an Aclass instance.

The cost of creating a distributed application in this way is the cost of creating the wrapper classes. Fortunately this construction can be automated and the application cam.codegen.CreateCISI does this. Thus, the task of creating a distributed version of an application involves only a modest number of code changes (a constructor change, and a call specifying the address and port of the remote instance), as well as the invocation of the cam.codegen.CreateCISI application.

In the following section we present an extended an example where a distributed version of an application is constructed. In section 3, the nature of the Client Implementation and Server Implementation classes, and "how they work" is discussed. Callbacks from the server class to a client class are possible, and these are described in section 4. In section 5, we discuss how the use of class interfaces can minimize the amount of code that must change when one switches between the a local and a remote class instance. Lastly we discuss the use of "wrappers" with the Java RMI package.

For this work, our primary Java references were the books [3], [4] and [5].

2. An Example

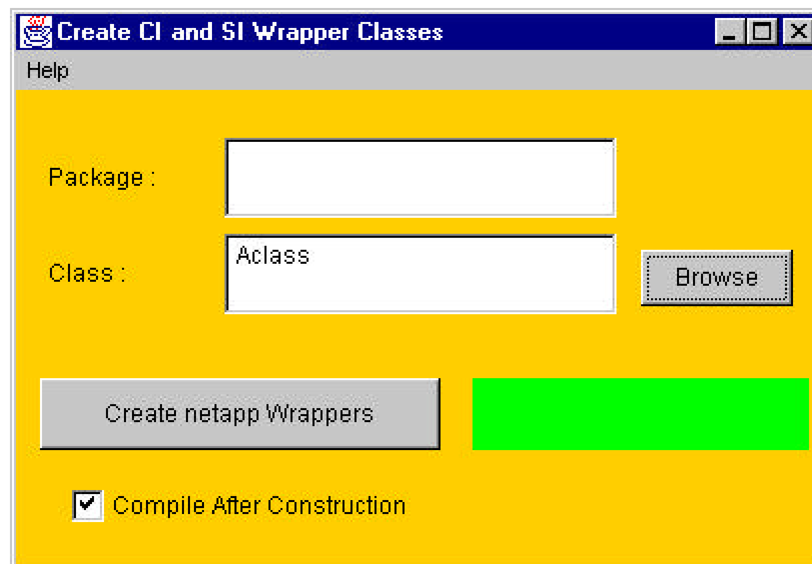
As an example, we consider the creation of an application with the two classes mentioned in the introduction, App and Aclass. The App class will have a method called `doCalculation(...)` that invokes an Aclass method `addTwo(...)`. Thus, the App class will be the "client" class and Aclass the "server" class.

The starting point is the construction of a purely local implementation. This implementation can be viewed in Sample1/App.java and Sample1/Aclass.java. The implemenation is as one would

expect; App contains the data member A -- an Aclass instance. The `doCalculation(...)` method merely invokes `A.addTwo(...)` to obtain the result. The App.main(...) routine tests the construction. The output from the program is

```
The result of the calculation = 7 (and should = 7)
```

The next step consists of creating the wrapper classes for Aclass. This is is accomplished by invoking the the cam.codegen.CreateCISI application. The interface to the program has the form



You must type in the name of the package that the class resides in (e.g. if it is in package cam.codegen, then one would type in `cam.codegen`). In this example, Aclass is in the default package, so one leaves the package field empty. In the Class field one must specify the name of the class. This program creates the wrapper classes, and, by default, internally invokes the Java compiler to compile them. If one doesn't want this program to compile the routines after construction, then clear the Compile After Construction checkbox.

Note that when running the CreateCISI application, the target class must reside in a directory that allows JAVA classes to be loaded; in particular, the concatenation of a directory listed in the CLASSPATH variable and the package name of the target class must yield a directory the target class resides in.

This result of using cam.codegen.CreateCISI is the creation of two classes AclassCI.java and AclassSI.java.

With the wrapper classes constructed and compiled, the next step is to modify the App class to utilize the client implementation class AclassCI. This version is contained in the file Sample2/App.java. The required changes consist of changing the declaration of the variable A from

```
   Aclass       A;
```

to

```
AclassCI     A;
```

and changing the constructor from

```
 A = new Aclass();
```

to

```
A = new AclassCI();
try
{
   A.setVerboseFlag(true);// remove to stop messages ..
   A.createServerInstance("127.0.0.1",6789);

}catch(Exception e)
{System.out.println(e.getMessage());}
```

This new code is compiled and then tested by invoking the cam.netapp.ServerManager, and running App.main(...). With the verbose mode on, the output from the program has the form:

```
NetworkConnection : Setting up Connection
NetworkConnection : Connecting to [127.0.0.1] Port : 6789
NetworkConnection : Connected  to [127.0.0.1] Start Up Port : 6815
NetworkConnection : Connected  to [127.0.0.1] Server   Port : 6816
NetworkConnection : Connected  to [127.0.0.1] Server   Port : 6817
NetworkConnection : Connected  to [127.0.0.1] Server   Port : 6818
NetworkConnection : Application AclassSI requested
NetworkConnection : Remote Class Found
NetworkConnection : Remote Class Instantiated
NetworkConnection : Streams Connected to Remote Class
NetworkConnection : Connection Complete
NetworkConnection : Starting Remote Application
AclassCI : Connection Complete
The result of the calculation = 7 (and should = 7)
```

At this point, one can test the program working between two machines. To accomplish this one needs to have App and AclassCI on the client machine and Aclass and AclassSI on the server machine. The server classes Aclass and AclassSI must reside in a directory where they can be loaded by the cam.netapp.ServerManager running on the server machine. Also, the address and port numbers utilized in `A.createServerInstance(...);`method should be set to the server machine and port where the cam.netapp.ServerManager is running..

Here's a summary of the steps associated with the construction of a distributed application

1. Develop the client and server class locally. Utilize standard JAVA programming tools and get the application running.

2. Invoke cam.codegen.CreateCISI and create the client (CI) and server (SI) impementation classes. Compile these classes (if one doesn't select the compile option of the cam.codegen.CreateCISI application).

3. Modify the client code to utilize the client (CI) implementation of the server class. Test the implementation locally (e.g. run a local cam.netapp.ServerManager and specify the loop back address (usually 127.0.0.1) as the machine location in the createRemoteInstance(...) method).

4. Export the server class and its associated server (SI) implementation class to a remote machine. Make sure the classes reside in a location that allows a cam.netapp.ServerManager instance to load. them. One needs to change the address and port number in the createRemoteInstance(...) method called by the client application.

To run the distributed application; start up the cam.netapp.ServerManager on the remote machine, and then start the client application.

3. How it's done

The code in the wrapper classes is just code one would construct ``by hand'' if one were directly using the cam.netapp classes. For example, if one considers the two classes AclassCI.java and AclassSI.java that are generated, one sees that the client implementation (CI) contains the public methods of Aclass. For each method in the CI implementation, a vector of parameters `methodData` is created and transmitted to the corresponding server (SI) class. The first element of `methodData` contains the index of the method, and the remaining elements of the vector contain the method parameters. All parameters are transformed to Object instances before transmission, e.g. an int parameter is transformed to an Integer object before being added to the vector of data.

The vector of data is transmitted over a connection established using a cam.netapp.NetworkConnection instance. There is one single transmission statement; `Os.writeObject(methodData)`. The `writeObject(...)` method automatically serializes all the elements of the data vector, and then writes the results to its associated stream. On the receiving end, the server (SI) class has a single read statement that listens for the vector of method data. Once this vector is received, the data is unpacked, and the appropriate method of a local instance of Aclass is called. Every routine in the server implementation creates a return vector that carries return status, return data, and any exceptions that may have been called. The client implementation (CI) reads the return data, un-packs it, and returns to the invoking program.

To construct the client (CI) and server (SI) wrapper classes the CreateCISI application uses a utility class CreateNetAppWrappers. This utility class first loads an instance of Aclass, then uses Java's class reflection methods (i.e. the methods of java.lang.Class) to determine the public methods, parameters, return types, etc. and then writes the wrapper class code using this information.

4. Implementing Callbacks

If an application has "callbacks", that is, the server class has the capability of invoking methods on the client, then the distributed implementation is a bit more complicated than when callbacks are not present. In particular, when there are callbacks, the client must act as a server as well, and

appropriate communication links must be established for this. In keeping with the philosophy that the migration from a local implementation to a distributed implementation should be as simple as possible, our goal was to have it so minimal programming changes are required when creating a distributed implementation of an application that possesses callbacks. Thus, the idea is that a programmer creates and debugs an application with callbacks and then creates a distributed implementation by making a small number of code modifications and by creating appropriate wrapper classes (classes that are automatically generated) .

As with the creation of distributed applications without callbacks, perhaps the best means of revealing the procedure and requisite programming constructs is through an example. So, consider a two component application of the form used previously :

However, in this version, Aclass contains a reference to App for callbacks, and one method of the App class (typically the constructor) calls an Aclass method to set the reference to App. To facilitate the conversion to a distributed implementation, we require Aclass set the reference to App by implementing the `cam.netapp.SetClient` interface.

In the purely local version of the application, the App class constructor has the form

```
public App()
{
   A = new Aclass();

   try{
   A.setClient((Object)this); // Using the method defined by
                              // cam.netapp.SetClient to set the
   }                          // callback
   catch(Exception e){};
}
```

while in the Aclass class, the method `setClient(...)` has the form

```
public void setClient(Object Ob)throws Exception
{
    appClient = (App)Ob;
}
```

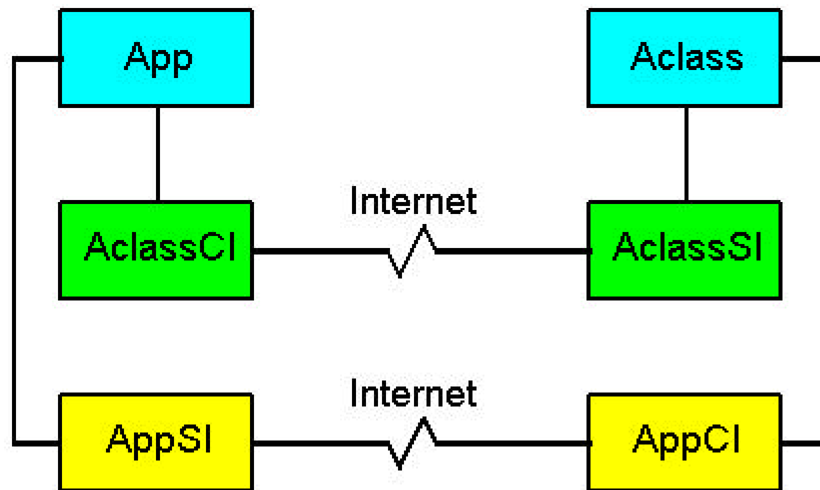
The complete codes are contained in the files Sample3/App.java and Sample4/Aclass.java.

Using these constructions gives the Aclass instance the capability of invoking App methods. For demonstration purposes App contains the methods `testCallBack(...)` and `printString(...)`. Aclass contains the method `useCallBack(...)`. The construction is tested by having `App.main(..)` create an App instance and invoke the `testCallBack(...)` method. This method immediately invokes the Aclass `useCallBack(...)` method which in turn invokes the client instance `printString(...)`method. The output from the `App.main(...)` is

```
The String "Callback Ok" Indicates Success
```

```
Callback OK
```

In a distributed implementation, the existence of a callback requires that the App class act as a server class. Thus, wrapper classes AppCI and AppSI are required to implement the connection between Aclass and App. Ultimately, the structure of the distributed application has the form



The additional wrapper classes consist of AppCI and AppSI, classes that enable the Aclass instance access (via AppCI methods) the methods of the App instance.

To create a distributed implementation from the local implementation consists of

1. Creating the classes AclassCI, AclassSI, AppCI, AppSI.

2. In App, changing the server type from Aclass to AclassCI.

3. In Aclass changing the callback reference from type App to AppCI. Note that the `setClient(...)`method requires modification as well.

4. Adding System.exit(0) to `App.main(...)`. If one doesn't do this, then the AppSI instance remains after App has exited.

The new App constructor has the form

```
public App()
{
   // Using local : uncomment the next block
   // #######################################

   // A = new Aclass();

   // #######################################
   //
   // Using remote : uncomment the next block
   // #######################################
```

```
   A = new AclassCI();
   try
   {
      A.setVerboseFlag(true);// remove to stop messages ..
      A.createServerInstance("127.0.0.1",6789);

   }catch(Exception e)
   {System.out.println(e.getMessage());}

   // ########################################
   //
   try{
   A.setClient((Object)this); // Using the method defined by
                              // cam.netapp.SetClient
   }
   catch(Exception e){System.out.println(e);};
}
```

while Aclass implementation of cam.netapp.setClient(...) now has the form

```
public void setClient(Object Ob) throws Exception
{
    appClient = (AppCI)Ob;
}
```

Sample4/App.java and Sample4/Aclass.java contain the source for the distributed version of the application.

With these changes, when the `setClient(...)` method of Aclass is invoked, then the callback structure described above is automatically instantiated. (This assumes the AppSI is located on the client machine and AppCI is located on the server machine).

The output from the `App.main(...)` with the verbose mode set has the form

```
NetworkConnection : Setting up Connection
NetworkConnection : Connecting to [127.0.0.1] Port : 6789
NetworkConnection : Connected  to [127.0.0.1] Start Up Port : 6791
NetworkConnection : Connected  to [127.0.0.1] Server   Port : 6792
NetworkConnection : Connected  to [127.0.0.1] Server   Port : 6793
NetworkConnection : Connected  to [127.0.0.1] Server   Port : 6794
NetworkConnection : Application AclassSI requested
NetworkConnection : Remote Class Found
NetworkConnection : Remote Class Instantiated
NetworkConnection : Streams Connected to Remote Class
NetworkConnection : Connection Complete
NetworkConnection : Starting Remote Application
AclassCI : Connection Complete
The String "Callback Ok" Indicates Success
Callback OK
```

So, the steps to the creation of a distributed version of a two component application with callbacks consists of

1. Creating and debugging a local version of the application, a version in which the `cam.netapp.SetClient` interface is used by the client class to set the callback reference.

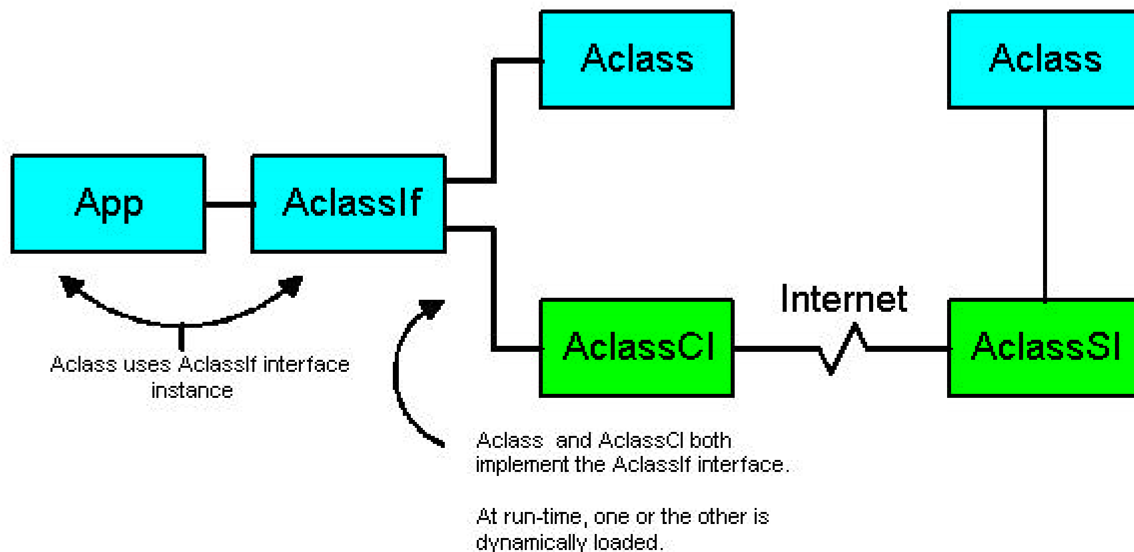
2. Generation of the client (CI) and server (SI) classes for both components.

3. Modification both components to use the client (CI) classes.

5. Using Interfaces to Minimize Code Modification

As described, the process of creating a distributed implementation from a local implementation requires code changes. While these are few, the fact is that one must change and recompile the code. This is a drawback, because it means that one has to worry about two versions of an application and not just one. By investing a little more time in the local implementation, one can make it so that the conversion to the distributed version requires no re-compilation; the use of a remote class instance can be enabled at run-time.

The idea is to define an interface class that contains the public methods (excluding constructors and static methods) for the server class. One then has the server class and its associated client implementation (CI) implement this interface. In the client class, one programs using an interface class reference, and then dynamically loads either the server class or its client interface depending upon if one wants a local or a distributed implementation. The structure for the sample two component application involving App and Aclass is the following



As an example, an interface describing the methods of Aclass used in the first example (Sample1/Aclass.java) has the following form

```
public interface AclassIf
{
    public int doCalculation(int D);
}
```

Now, by having Aclass implement this interface, and writing the code in App to only use this interface, one can create an App class that does not need to be recompiled when the distributed

implementation is used. (After creating Aclass so that it implements the AclassIf interface, then AclassCI and AclassSI are generated using cam.codegen.CreateCISI. The program cam.codegenCreateCISI will automatically have AclassCI implement the AclassIf interface if Aclass does.)

Sample5/App.java and Sample5/Aclass.java contain the implementations of App and Aclass that utilize this construction.
Of interest in these classes is the mechanism by which either the local or remote instance of Aclass is instantiated by the App class.
The App constructor has the following form, exhibiting how this is accomplished with dynamic loading;

```
public App(boolean localFlag)
{
    String serverClassName   = "Aclass";
    String serverClassNameCI = "AclassCI";
    String address = "127.0.0.1";
    int    port    = 6789;

    Class  theClass  = null;
    Object theObject = null;
    if(localFlag) // local implementation
    {
      try
      {
      theClass   = Class.forName(serverClassName);
      theObject  = theClass.newInstance();
      }
      catch(Exception ex)
      {System.out.println("Class Not Found : " + ex.getMessage());};
    }
    else // distributed implementation
    {
    try
    {
        theClass   = Class.forName(serverClassNameCI);
        theObject  = theClass.newInstance();
    }
    catch(Exception ex)
    {System.out.println("Class Not Found : " + ex.getMessage());};
    // request the remote instance
    try
    {
    ((cam.netapp.CIinterface)theObject).setVerboseFlag(true);
    ((cam.netapp.CIinterface)theObject).createServerInstance(address,port);
    }
    catch(Exception e){System.out.println(e);};
    }
//
//  Cast the object to type AclassIf
//
    A = (AclassIf)theObject;
}
```

So, depending on the flag passed into the App constructor, either a local or a distributed implementation is created. It is important to note that the method `createServerInstance(...)`

is accessed via the interface `cam.netapp.CIinterface`. The use of the cam.netapp.CIinterface interface ensures that the initialization code does not require the any explicit reference to the AclassCI class. (This is important because this ensures that the existence of AclassCI is not necessary for App to compile)

Here are the results obtained with `App.main(...)`

```
 Local Results
The result of the calculation = 7 (and should = 7)

 Distributed Results
The result of the calculation = 7 (and should = 7)
```

So, if one is using the NetApp wrapper constructs, in order to create applications in which the transition from a local to a distributed implementation requires minimal re-coding one needs to

1. Create both a server class **and** a server class interface that the server class implements.
2. Have all client applications utilize the server interface reference for all server method calls
3. Initialize the server instance by dynamically loading the server class or the server class client (CI) implementation by name.

The cost of carrying out this work is that of keeping the server class and the server class interface synchronized. This is a source of errors, but, fortunately, the compiler typically lets you know if they are not synchonized.

6. Using Wrappers and Java RMI

Our main emphaisis has been on the construction and use of wrapper classes that utilize the cam.netapp package to provide the underlying communication link, however, it is also possible to use the same techniques to create wrapper classes that utilize the Java RMI package as the communications link. In particular, the application cam.codegen.CreateRMI takes an existing class and creates the appropriate wrapper class and wrapper interface so that it is in the ``server'' form required by the Java RMI interface.

Consider again our first example, the application whose components are contatined in Sample1/App.java and Sample1/Aclass.java. The creation of a distrubuted version that uses the Java RMI package requires

- The creation of a server interface and server wrapper class for the Aclass component.
- Modification of the App client to use the Aclass server interface class.

The application cam.codegen.CreateRMI creates the required server classes. When applied to Aclass, the application creates the server wrapper class AclassRS.java ((R)MI (S)erver) and the server interface AclassRI.java ((R)MI (I)nterface). As with all RMI server classes, the JDK `rmic` utility program is applied to AclassRS to create the stub and skeleton classes AclassRS_Stub and

AclassRS_Skel.

The interface class, AclassRI.java, extends java.rmi.Remote and contains all the public methods of Aclass. The interface is not identical to that of Aclass, because each method throws java.rmi.RemoteExceptions (as required of any class that extends extends java.rmi.Remote). The server class AclassRS.java implements the AclassRI interface and obtains its functionality by invoking the appropriate methods of an Aclass instance that AclassRS contains. Also included in AclassRS is the requisite `main(...)` invocation that sets the SecurityManager and registers the class with the rmiregistry.

After this packaging of Aclass into a "server" form, the client appliction App must be modified to use a reference of type AclassRI. In addition to changing the data member declaration, the constructor requires modification and all AclassRI method calls must placed within a try-catch block The modified code is presented inSamples6/App.java.

Prepretory to running the distributed application requires appropriately placing class components on the local and remote machine. The location of the files is given in the following table

Local	Remote
App.class (uses AclassRI)	Aclass.class
AclassRI.class	AclassRS.class
AclassRS_Stub.class	AclassRS_Skel.class

Once the files are in place, one starts the `rmiregistry` on the remote machine, and then starts an instance of AclassRS (e.g. by running a statement of the form `java AclassRS`). The client application App is then invoked and it communicates AclassRS on the remote machine.

References

---

[1] Anderson, C.R., cam.netapp Package Documentation www.math.ucla.edu/~anderson/JAVAclass/CAMJava.html, 1997

[2] Anderson, C.R.,Creating Distributed Applications with the cam.netapp Package, UCLA Dept. of Mathematics CAM Report 98-39 1998.

[3] D.J. Berg and J.S. Fritzinger, *Advanced Techniques for Java Developers*, Wiley & Sons.,New York, 1997.

[4] M. Campione, K. Walrath, *The Java Tutorial Second Edition*, Addison-Wesley, Reading Massachusetts, 1998.

[5] D. Flanagan, *Java in a Nutshell*, O'Reilly, Sebastapol Ca., 1997.