UCLA
COMPUTATIONAL AND APPLIED MATHEMATICS

Creating Distributed Applications in Java with the cam.netapp Package

Christopher R. Anderson

September 1998
CAM Report 98-39

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

Creating Distributed Applications In Java Using cam.netapp Classes

Abstract： The purpose of this document is to describe the package cam.netapp, a collection of classes that provides a "minimal" software infrastructure for creating distributed applications. The process of creating a distributed application is discussed, and several examples are presented.

Contents

- Introduction
- NetworkConnection
- SetStreams
- ServerManager
- Operational diagram
- Data driven synchronization
- Communication between components
- Miscellaneous notes

Sample Programs

Front.java Back.java, Front2.java Back2.java, Front3.java Back3.java

Class Reference

cam.netapp package reference
cam.dataxchg package reference

Package Source

cam.netapp , cam.dataxchg


```
Chris Anderson
Dept. of Mathematics
UCLA Los Angeles, CA 91555
```



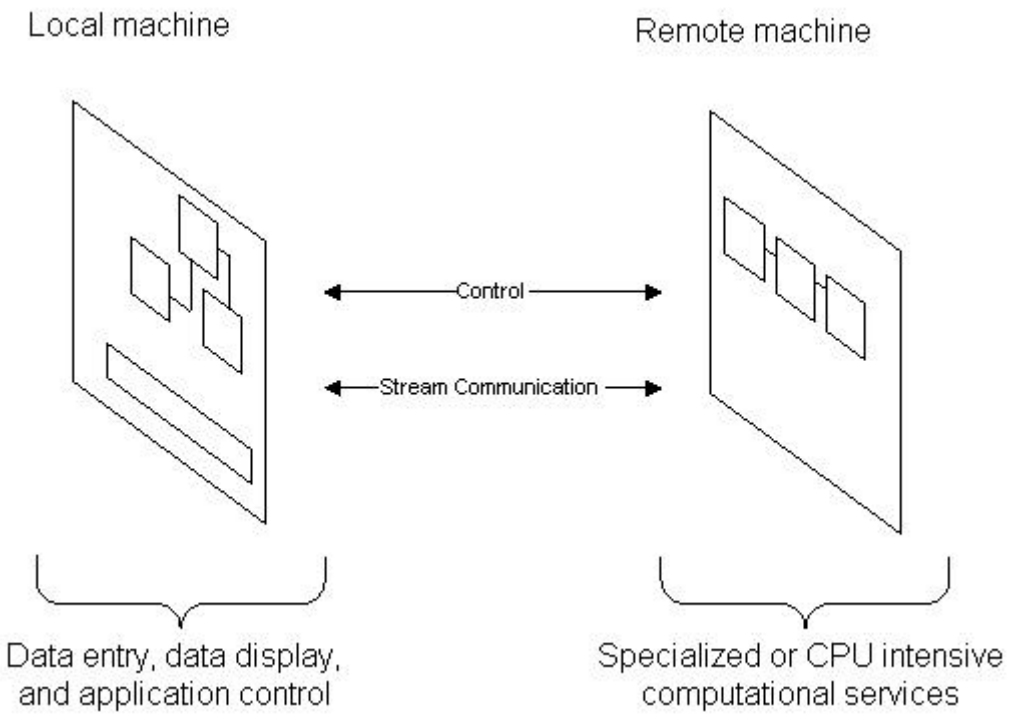
```
Dan Delsol implemented the original versions of the ServerManager and
NetworkConnection classes. His contributions are gratefully acknowledge
```

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

Updated 07/15/98 CRA

Introduction

The purpose of this document is to describe a "minimal" software infrastructure that facilitates the construction of applications/programs which have the form shown in the following figure



These applications consist of two components, a "front" component and a "back " component. Typically, the "front" component provides a user interface that allows data input, data display, and mechanisms for user control while the "back" component consists of a program providing a specialized or CPU intensive computational service. Because the two components of the application often (but not always) reside on different machines, we refer to such applications as "distributed" applications.

Within the realm of scientific/technical computing, distributed applications allow one to

- provide access to a particular piece of software or capability without having to port code to multiple machines.
- place CPU intensive components on specialized platforms and/or distribute computational tasks among multiple computational resources.

There are a number of software constructs that have been created to facilitate the development of distributed applications. (The Remote Method Invocation package contained within Java 1.1, ActiveX components, COM objects, CORBA objects, the

Message Passing Interface (MPI) library for parallel programs, etc.) However, the applications that motivated this work didn't require much of the functionality implemented by these constructs; and so we sought to create a "minimal" software infrastructure that would allow us to create distributed applications. The requirements of the software infrastructure we sought were

1. To coordinate/enable stream communication between "front" and "back" components.
2. The ability to start, stop, pause, and check the status of a "back" component.
3. The ability to request that a "back" component be loaded and connected (i.e. with respect to data streams and control) to the "front" component.

A combination of three Java classes in the cam.netapp package provide these capabilities --- the NetworkConnection class, the SetStreams class and the ServerManager class.

The NetworkConnection Class

Instances of the NetworkConnection class are included in the "front" component (one instance for each "back" component to which the "front" component is communicating). This class is responsible for

- Connecting to the "back" component (referred to as a "remote" application)
- Providing access to the streams connected to the "back" component.
- Providing control of the "back"component.

The manner in which these capabilities are provided is revealed by the following code fragments

```
   NetworkConnection netConnect = new NetworkConnection();

   try
   {netConnect.connectRemoteApplication("127.0.0.1",6789,"Back");}
   catch(Exception e){System.out.println(e);}
   }

   FrontInputStream  = netConnect.getInputStream();
   FrontOutputStream = netConnect.getOutputStream();

   netConnect.startRemoteApplication();
//
// Data communication between the "front" component and the "back"
// component occurs using FrontInputStream and FrontOutputStream ...
//
   netConnect.stopRemoteApplication();
```

In this instance, the "back" component is located on a machine with Internet address [127.0.0.1] (which is the loop-back address for my PC), is being serviced by a ServerManager listening on port 6789 (see below), and is a Java class named Back.class.

The "back" component does not require an instance of the NetworkConnection class --- it is only required to implement the Runnable interface and the SetStreams interface.

The SetStreams Class

The SetStreams class is an interface class; and the methods within it must be implemented by the "back" or "remote" component in order to communicate with a "front" component using a NetworkConnection instance. The "back" or "remote" components must also implement the Runnable interface.

There are only two methods in the SetStreams class; setInputStream(...) and setOutputStream(...). As the following code fragment from Back.java indicates, their implementation is rather simple

```
//
// SetStreams interface implementation
//
public boolean setInputStream(InputStream s)
{
    BackInputStream  = s;
    return true;
}
public boolean setOutputStream(OutputStream s)
{
    BackOutputStream  = s;
    return true;
}
```

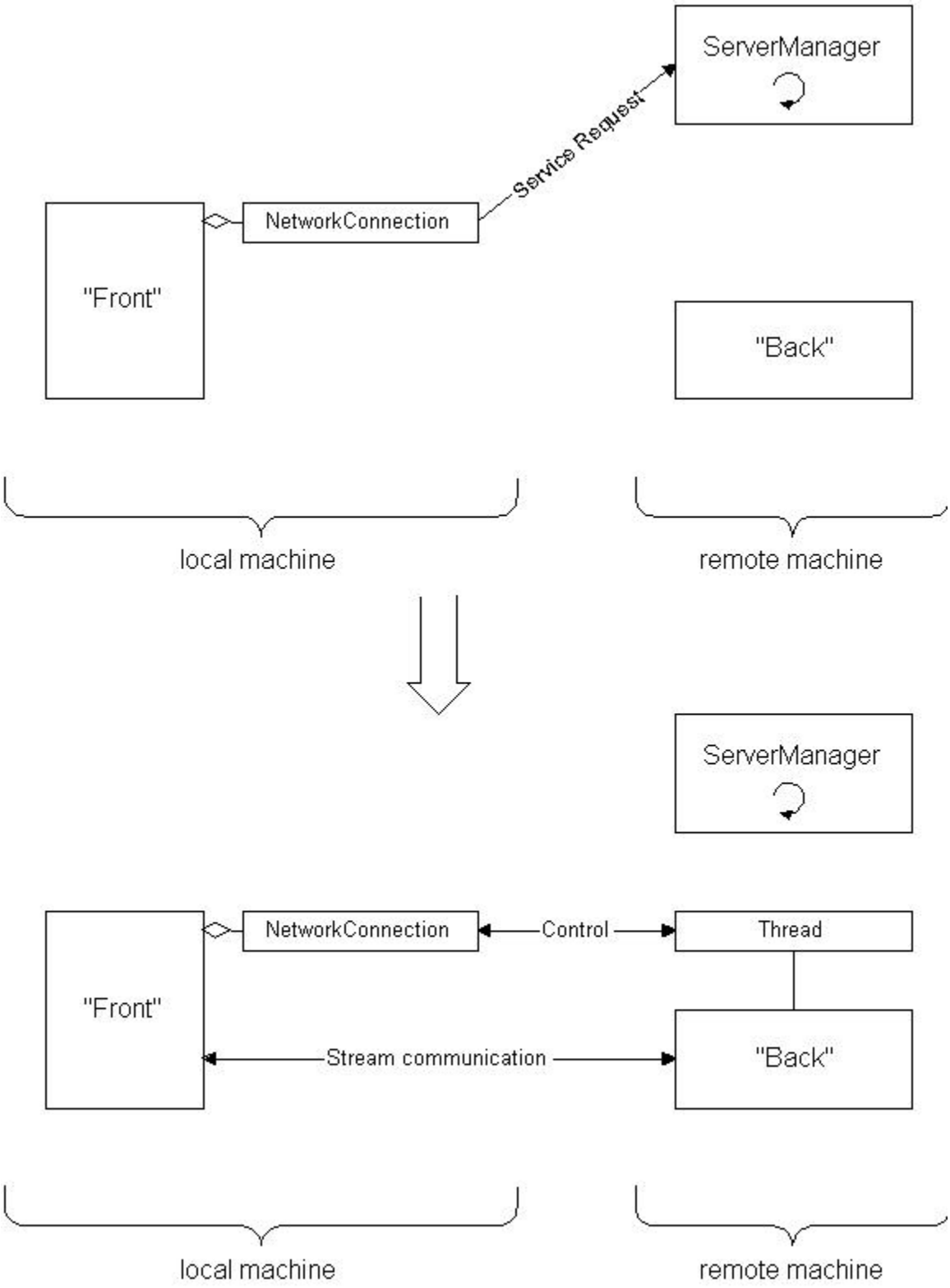
Here BackInputStream and BackOutputStream are class variables of type InputStream and OutputStream respectively.

The ServerManager Class

Instances of the ServerManager class are run on a remote machine to service requests for loading and running classes on the remote machine. The ServerManager class is typically invoked from the command line on a remote machine. An instance of the ServerManager class has a port associated with it; this is the port upon which the ServerManager is "listening" for connections. A NetworkConnection instance must use both the machine address and the port number to connect to a ServerManager. The default port number is 6789, alternate port numbers can be specified on the command line invocation of a ServerManager.

Operational Diagram

The following diagram illustrates how the NetworkConnection and ServerManager classes operate to create a functional distributed application.

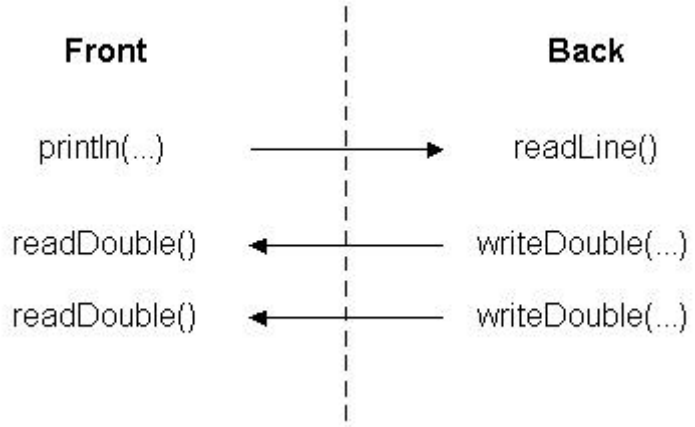

ServerManager
Service Request
NetworkConnection
"Front"
"Back"
local machine
remote machine
ServerManager
NetworkConnection
Control
Thread
"Front"
Stream communication
"Back"
local machine
remote machine

The top part of the diagram illustrates what occurs when the `connectRemoteApplication(...)` method of the NetworkConnection class is executed; a service request goes to an instance of the ServerManager running on a remote machine. This action initiates a sequence of events whose result is the loading of the requested class, the creation and connection of streams to that class (i.e. it invokes the SetStreams functions), and the creation of a thread in which the requested class will be run. Additionally, connections are made between the NetworkConnection instance and this latter thread, so that program control may be exercised over the requested class. The net result of these actions is the situation depicted in the bottom part of the diagram.

In order for this process to be carried out, it is necessary that the requested class on the remote machine be located within the classpath associated with the ServerManager instance. One way to ensure this is to place the remote classes in the same directory as the ServerManager. Alternately, one can append the appropriate classpath when creating an instance of the ServerManager from the command line.

Data Driven Synchronization

The synchronization of the execution of code in the "front" and "back" components is data driven. Specifically, I/O in Java is blocking --- if an attempt is made to read data from a stream and none is available, the thread associated with that statement will block until data is available. Therefore, after the streams between components have been connected, one coordinates the execution of code in the components by alternating reads and writes. For example; in the run() methods of Front.java and Back.java we have the sequence of reads and writes as depicted in following diagram.



For both components, when a read statement is executed in one component and no data is available, execution halts until the write statement is executed in the other component. Because this method of synchronization often leads to the "front" component being in a

blocked state, it is a wise idea to have the communication code and the user interface code run in separate threads. (So that when the communication code is blocked a user is still able to interact the interface.)

Communication Between Components

The sample programs give one an idea how simple data types (numbers, strings, etc.) can be exchanged between the front and back components. Its clear that if complex collections of data are exchanged the communication code quickly becomes unwieldy, cluttered, and difficult to debug. One can simplify the communication process and improve its reliability by using the idea of serialization. If you are using Java 1.0 you have to implement serialization yourself; if you are using Java 1.1 you can use the classes java.io.InputObjectStream and java.io.OutputObjectStream contained within the java.io package.

Serialization means that a class has methods capable of writing information to a stream in such a way that the information can later be read from a stream and that instance of the class reconstructed. An example of a class with this capability is the ParameterList class in the cam.dataxchg package. This class has routines

```
void output(OutputStream D)

void outputBinary(OutputStream D)

void input(InputStream D)

void inputBinary(InputStream D)
```

To communicate a ParameterList across a stream, one needs just use a matched pair of Output/Input or OutputBinary/InputBinary method invocations. The following code fragment from Front2.java illustrates this.

```
    netConnect.startRemoteApplication();
//
//  Exchange data with the remote class
//
//  create a ParameterList
//
    cam.dataxchg.ParameterList P = new cam.dataxchg.ParameterList("Test
    P.addDouble(" dt ",1.0);
    P.addInt(" N ", 100);
//
//  send
//
    P.output(FrontOutputStream);
//
//  receive (using the binary version as an example...)
//
    P.inputBinary(FrontInputStream);
//
```

```
//  output results to the screen
//
    P.output(System.out);

    netConnect.stopRemoteApplication();
```

In the run() method of Back2.java, corresponding input and output statements occur.

The ParameterList class was created before Java 1.1 was available, and so the methods Input, InputBinary, Output, OutputBinary were implemented by the programmer. With the java.io.ObjectInputStream and java.io.ObjectOutputStream classes in Java 1.1 one can utilize serialization procedures without having to create ones own serialization methods. Using these constructs the process of transmitting a ParameterList class instance across a stream becomes

```
    netConnect.startRemoteApplication();
//
//  Exchange data with the remote class
//
//  create a ParameterList
//
    cam.dataxchg.ParameterList P = new cam.dataxchg.ParameterList("Test
    P.addDouble(" dt ",1.0);
    P.addInt(" N ", 100);
//
//  send
//
    try
    {
    java.io.ObjectOutputStream Os = new java.io.ObjectOutputStream(Fron
    Os.writeObject(P);
    }
    catch(Exception e){System.out.println(e);};
//
//  receive
//
    try
    {
    java.io.ObjectInputStream Is = new java.io.ObjectInputStream(FrontI
    P = (cam.dataxchg.ParameterList)Is.readObject();
    }
    catch(Exception e){System.out.println(e);}

    P.output(System.out);

    netConnect.stopRemoteApplication();
```

These code fragements come from the Front3.java. (Back3.java has the corresponding input and output statements).

Miscellaneous Notes

No security mechanism is implemented in the ServerManager class.

Once a remote class is loaded by the ServerManager, the class remains loaded even after the execution of the remote class stops. Subsequent requests for the class do not cause a re-load to occur. This means that the ServerManager must be restarted if one wants the ServerManager to use a new version of a remote class.