KnittingFactory: An Infrastructure for Distributed Web Applications TR 1997–748

A. Baratloo, M. Karaul, H. Karl, and Z. M. Kedem
Department of Computer Science
Courant Institute of Mathematical Sciences
New York University

November 13, 1997

Abstract

While Java and applets have created a new perspective for Web applications, some problems are still unsolved. Among these are the question of how Java applets can find other members of the collaboration session, how to deal with the restrictions imposed by the Java security model, and how to overcome the inability of applets to communicate directly, even if they belong to the same distributed application. KnittingFactory addresses the problem of finding other members of a collaboration session by providing a distributed registry system where the search is performed within a Web browser without violating its security model; the problem of arbitrary placement of applications by providing the core functionality for downloading applets from an arbitrary node; and finally the problem of direct applet-applet communication by using the Java Remote Method Invocation mechanisms to give applets information on how their fellow applets can be reached. Two example applications validate this concept and demonstrate the ease of use of KnittingFactory.

Contents

1 Introduction

The advent of the World Wide Web along with the pervasive availability of cheap networking resources has permitted the idea of distributed parallel computation or collaborative work on a large scale. But while parallel computing on a local network of workstations is common practice (with systems like PVM [20] as examples), this is not the case for distributed computing over the World Wide Web. Similarly, collaborative work over the Internet is faced with a number of practical problems. Some of the obstacles common to both distributed computing and collaborative work are the heterogeneity of the participating systems, security concerns by users of remote and local computers, difficulties in administering a distributed application and finding partners, and often high communication delays. We will consider both parallel and collaborative application scenarios as “distributed applications”.

Some of these problems were successfully addressed by the Java language/system. Namely the security concerns where solved by the “applet”-concept (in combination with the notion of a “sandbox”) which allows the execution of foreign code without putting the system’s integrity at risk. Additionally, Java’s platform independence solves the problem of heterogeneity. Today’s performance problems of Java applets are expected to be solved by just-in-time compilation and other techniques.

This makes Java an attractive choice for the realization of distributed applications. An attractive way of designing such distributed applications is to use a *team* consisting of a stand-alone program running on one machine and a set of distributed applets.[1]. But Java imposes it’s very own restrictions on such teams. Typically, the location of a Java application working together with distributed applets is restricted to a machine also running an HTTP server. However, such a machine is not available to all users. Furthermore, the Java security model restricts the communication pattern between applets and application thereby limiting both communication flexibility and fault tolerance of such teams.

Implementing distributed teams on top of Java is thus faced with the following three obstacles:

- looking for and finding potential partners for a distributed team,
- having to run the stand-alone application of a distributed team on a machine also running an HTTP server,
- applets are not capable of direct communication.

The *KnittingFactory* project addresses these obstacles and provides well-integrated solutions that do not interfere with the Java security model and completely adhere to all standards.[2]

[1] In the Java terminology, such a stand-alone program is usually called an application (as opposed to an applet). We will use this term, too, when there is no danger of confusing it with the application as a whole, i.e., the team of stand-alone program and applets

[2] Experiments were conducted using Linux kernel version 2.0.29, Java Developers Kit version 1.1.3, and HotJava Browser Version 1.1. The novel features of Java 1.1 used in our implementation are not yet completely supported in any other off-the-shelf browser. Therefore, we could only experiment with our system while relying on HotJava and the JDK1.1.3’s appletviewer, and and we expect the system to carry over to other standard standard implementations of Java 1.1.

The rest of this paper is organized as follows. Section 2 describes the design of the KnittingFactory, Section 3 gives information about the implementation, Section 4 will show how to use the KnittingFactory with two simple examples, Section 5 gives an overview of related work and Section 6 discusses KnittingFactory in the context of some of these projects. Finally, the paper concludes with Section 7.

2 Design

KnittingFactory is concerned with providing functionality to facilitate the interaction and improve the efficiency of interacting distributed computations written in Java. There are several ways of designing such an application. To take advantage of the security features of Java, the distributed code should be implemented as an applet. Code executing on local machine can be implemented as a stand-alone Java application, which allows to use extended functionality compared to applets. Thus, a distributed application can be designed as a team consisting of a stand-alone Java application and a number of distributed applets. Depending on the semantics of the distributed application, the stand-alone program might be an "initiator," a "coordinator," or a "master," etc. depending on the context; the applets might be "workers," "servers," "user–agents," *etc.*—we will use the term *partner* to refer to them in a uniform way.

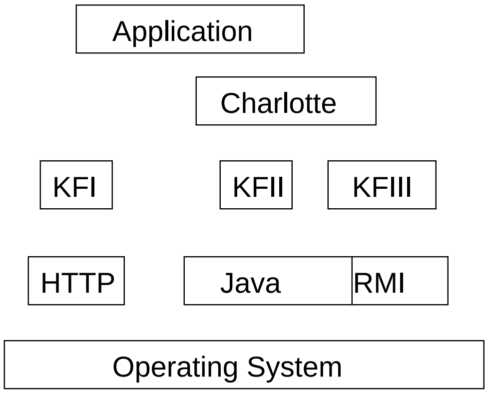

Figure 1: Overview of Knittingfactory design. KFI, KFII and KFIII refer to parts described in Subsection 2.1, 2.2 and 2.3, respectively.

Figure 1 gives an overview of a system using KnittingFactory. The boxes KFI, KFII and KFIII refer to parts of KnittingFactory desribed in Subsection 2.1, 2.2 and 2.3, respectively. Charlotte (cp. 6.1.2 and Section 6.1) is used as an example for a parallel programming environment using KnittingFactory, and an application sits on top of both KnittingFactory and Charlotte.

2.1 Registration and lookup

In a distributed computation organized, e.g., in a master-worker like fashion, a typical problem for a master is to find potential workers, and for workers (i.e., users willing to donate resources to a distributed computation) to find any masters. This might be of less concern to a collaborative application such as whiteboards, since one usually knows with whom one wants to cooperate. But there are other cases in which this problems occurs even for some collaborative-type applications such as games.

In either case, a distributed matching problem occurs between applications and potential partners. While there are many classical ways of doing this,[3] we want to make use of the existing Web-infrastructure as far as possible. In particular, no new agents for either providing or searching information should be necessary.

To this end, we use existing HTTP servers to act as a *registry* for partner requests and standard Web–browsers as search engines. A registry accepts requests for partners, stores these requests, and deletes them again upon request. Also, a registry knows about a list of other registries (which can be updated as well) which, in concert, form a directed graph of registries. A request for partners must contain the hostname and port number where the application can be contacted (in standard URL format) and can be enhanced by additional information such as Quality of Service parameters (e.g., only machines with at least a certain amount of memory are useful) or a descriptive comment. A registry is implemented using so-called cgi-bin scripts using standard HTTP server interfaces. Information is stored in a format that is directly accessible to standard browsers.

A user who wants to participate in a distributed computation needs only to know about one (or several) registries as starting points. The user can use a standard browser to search the requests stored in this registry. If there is no suitable request found on a given registry, the search proceeds in a breadth-first fashion along the graph of neighboring registries until a suitable request for partners is found or all registries[4] have been visited. Breadth-first search is used to find applications on "nearer" registries before applications registered with more remote registries—other search strategies are possible. As soon as a matching partner request is found, the browser jumps to the URL described by this request. The main point to notice here is that the search is executed completely within the user's browser which downloads the registry information from respective HTTP servers.

2.2 Arbitrary origin of applets

In a typical distributed team written in Java, applets are downloaded from an HTTP server and connect to an application to be able to communicate with other parts of the team. Since the Java security model usually employs the so–called "host-of-origin" policy, applets can only connect to the host from which they were downloaded. While other policies are conceivable, it is most likely that this will be the standard policy used by most Web browsers. This means that the stand-alone application of the team has to run on the same machine as the HTTP server that provided the applets to allow

[3] For example, by using a naming-service like system

[4] in this connected component

them to successfully connect to the application. This is often inconvenient at best and impossible at worst; a user who wants to initiate a distributed computation might not have access to an HTTP server, and such a machine will become a bottleneck if a number of distributed computations is run.

KnittingFactory addresses this problem by providing the essential functionality of an HTTP server for downloading applets from the application itself. A browser can connect to the application and retrieve all necessary HTML or class files directly from this process. Obviously, it is then possible for the applet to reconnect to this application to communicate with it. This allows the application to run on any host connected to the Web and not just the ones running HTTP servers.

Additionally, the application can register itself with one or more registries. A search as described in Subsection 2.1 (or a manual click on the corresponding link) will then result in a browser connecting to the application and downloading the applet. As an added convenience, KnittingFactory also provides the applet with the port number at which the application is accepting its connection requests.

2.3 Direct applet–applet communication

Due to the Java security policy of “host-of-origin”, it is not directly possible for different applets to establish direct communication with each other—even if they belong to the same team.[5] The main *raison d’être* for this security policy is to limit the amount of information exchanged between any two applets. But since we are considering applets belonging to the same distributed computation, which can exchange arbitrary information by means of relaying it through the application, there is no inherent reason in the security model why these two applets should not be allowed to communicate directly.

The advantages of such a direct communication are obvious: the application can easily become a bottleneck and it imposes a single point of failure for the applets (depending on the kind of application scenario, it might be perfectly acceptable if the application dies after the applets have been set up).

KnittingFactory uses the Remote Method Invocation of Java 1.1 to realize a direct applet–applet communication. To use RMI, a task needs a reference to a remote object to be able to invoke any methods remotely. Each applet creates such a remotely accessible object and passes its reference to the stand-alone application. In the simplest case, the application redistributes these references back to all other applets. Thus, all applets in one team learn about remote objects in all the other applets. This allows them to directly invoke methods at other applets, which implies that they can communicate directly.

In a more elaborate scenario, an applet might only want to know about a subset of applets. The set of applets is thus divided in different groups. These groups may overlap, since the remotely accessible objects are informed about other remote objects and not the applet itself—an applet only has to create one of these objects for each group in which it wants to participate.

[5]Unless both applets run on the machine from which they were downloaded.

3 Implementation

3.1 Registration and lookup

The implementation of the registration and lookup services conveniently divides into two parts: the registry and the lookup. The registry consists of four cgi-bin scripts written in Perl that update an HTML-page. This KnittingFactory main page contains links to other known registries and applications looking for partners as well as the lookup script written in Javascript. Additionally, a number of HTML-pages are provided to form a simple user interface to these cgi-bin scripts. This allows simple adding and deleting of registries and manual registration and deregistration of applications. The HTML-forms used for this communicate the following parameters to their corresponding cgi-bin scripts:

Adding a registry

newurl: The URL of the KnittingFactory main Web page of a new registry. This adds a link pointing from the registry where this form is submitted to the given URL.

password: to secure consistency of the registry — optional.

Script name: `register.cgi`

Removing a registry

remurl Remove the given URL from the list of registries on this registry.[6]

password: to secure consistency of the registry — optional.

Script name: `deregister.cgi`

Registering an application

host Hostname where the application is running.

kfport The port number where this application can be contacted by a Web browser to obtain an applet.

description An arbitrary description of this application (optional).

This can optionally be enhanced by Quality of Service parameters.

Script name: `add_app.cgi`

Deregistering an appplication

host Hostname where the application is running.

kfport The corresponding port name.

Script name `rem_app.cgi`

Figure 2: The main page of a KnittingFactory registry

Both registries and applications are stored as HTML-links to the corresponding sites. Figure 2 shows an example of such a page.

This main KnittingFactory page contains a Javascript program (`searcher.js`) that performs the actual search. Since this main page is manipulated only via cgi-bin scripts, the search script can rely on the syntax of this page. To store the state information necessary to perform a breadth-first search, the `tag`-element of the URL is used. This part of the URL is usually used to let a browser navigate to certain "anchors" in an HTML-document. Since this page does not contain any anchors, the Web browser just ignores the `tag`, but this information is still available to the Javascript program.

The tag is organized in the following way:

```
TAG ::= search+TODO-LIST+SEEN-LIST

TODO-LIST ::= ε |
              TODO-LIST-P

TOTO-LIST-P ::= URL |
                URL#TODO-LIST-P

SEEN-LIST ::= ε  |
              SEEN-LIST-P

SEEN-LIST-P ::= URL |
                URL#SEEN-LIST-P
```

where ϵ is the empty string and URL represents any well-formed http-address.

[6] In the current implementation, this is called `newurl`.

The searcher script first checks if the `tag` actually starts with the string `search`. If not, it stops immediately. If it does, it first examines the links on this page to check if any applications are registered with this registry. An application is distinguished from a registry by their relative position on the page. If some applications are found, one is randomly selected and the browser is directed to go this URL. Additionally, if any Quality of Service parameters are given, the searcher script will only consider applications where these requirements are met.[7] Other selection possibilities, e.g., based on a best fit or best price principle, can easily be integrated.

If no (matching) application is found on a given page, the searcher script will proceed with a breadth-first search. First, the URL of this page is appended to the `seen`-list of the `tag` string. Next, all registries contained in this page are appended to a `todo`-list. The first element of the `todo`-list is extracted and gives the address of the next page to visit. To this address, the `#search` tag, the new `todo`- and the new `seen`-list are appended to form the new URL and finally, the browser is instructed to go to this URL. Since this is, by construction, also the main KnittingFactory Web page of some other registry, it also contains the same searcher script which will proceed with the search using the additionally information obtained from the previous page and included in the tag. And since this page provides the searcher script code again, this code is free to inspect the contents of the newly downloaded page.[8] This process ends if either an application is located or no new registries can be found (which produces a suitable message to the user).

As an alternative to a pure breadth-first search that accepts the first request found, an exhaustive search with limited time or a QoS-based search are conceivable—here is a wide range of open possibilities.

This search is completely executed by the Web browser; no new processes need to be installed nor does the HTTP servers need to execute any search-related programs (they only provide Web pages). It is susceptible to error if any selected HTTP server happens to be unreachable—in this case the search aborts without result. This can be amended by randomizing the order in which registries are added to the `todo`-list and reexecuting the searcher script via a watchdog Javascript program which executes in another frame and periodically checks the progress of the actual search. The problem here is how the watchdog can distinguish between a search in progress, a successfully terminated search (an executing applet), an unsuccessfully terminated search (an attempt to contact an application that is no longer running or otherwise not responding) and a failed search (due to, e.g., an unreachable HTTP-Server).

A potential problem with this approach might be misuse of the lookup service. Since anybody should be able to register a request, more or less arbitrary Web-sites can be registered—even some completely unrelated to a distributed computation—with the only objective of attracting (unvoluntary) viewers. The cgi-bin scripts might be used to protect such misuse, e.g, based on a password scheme.

[7] Checking for QoS-parameters is currently not implemented.

[8] Which is not possible if Javascript code and Web page come from different machines — their contents is so-called “tainted”.

3.2 Arbitrary origin of applets

The goal of this part is to be able to run the stand-alone application on any machine—independent of whether an HTTP-Server is running on that host. The problem here is that an applet is only allowed to connect to the machine it is downloaded from. Therefore, to allow it to connect to the application's machine, it must be downloaded from this machine. This second part of KnittingFactory provides the necessary functionality to do so by implementing core HTTP-Server capabilities.

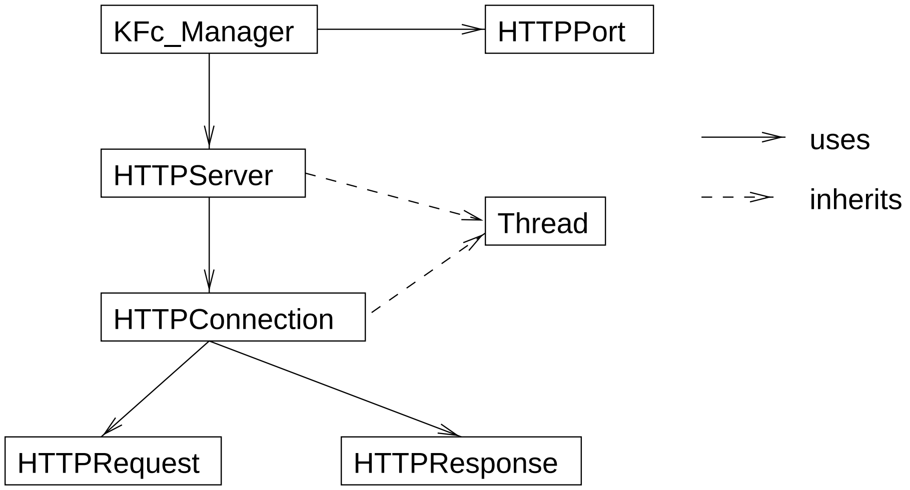

Figure 3: Class hierarchy for Subsection 3.2

The API is very simple: in the application, the programmer creates an object of class `KFc_Manager` (see Figure 3 for an overview of the class hierarchy). The constructor of this class takes two parameters: the name of the applet class that is to be given to any partner, and, optionally, the port number which the application is expecting partners to contact it. `KFc_Manager` itself creates an object of class `HTTPServer` (which extends `Thread`) that creates a server socket on which it expects a browser to connect to download the applet. Once a connection is made, `HTTPServer` creates a new `HTTPConnection` object to handle a request. This connection handler parses the request (using `HTTPRequest` as a helper class). Since we only want to serve one particular applet to the browser,[9] an HTML-page containing the applet name and optional parameters is generated and sent back to the browser (using an `HTTPRespone` object). To execute the applet, the browser will then connect again to the `HTTPServer` object and request various class files as needed. Again, the server creates an `HTTPConnection` object, the class name is extracted from the request, the class file is searched along the `CLASSPATH` and sent back to the browser. Thereby, an applet can be downloaded from the application's machine and executed without having to run an HTTP server.

[9]An extension to multiple, different applet classes is very simple.

Although any applet can be used this way, KnittingFactory provides the convenience class `KFc_Applet` which extends `Applet`. This class provides a method `getPort()` which returns the port number where the application expects the applet to contact it. The actual applet would thus be derived from `KFc_Applet` instead of `Applet`; it has to call the `init()` function of its parent class to allow correct initialization.

The integration with the registration service of KnittingFactory is achieved by two methods, `register()` and `deregister()`, of class `KFc_Manager`. They take one or more registry URLs as parameters and register/deregister the application with these registries.[10] The class `HTTPPort` is used to sent a `POST`-request to the registries, resulting in the execution of the appropriate cgi-bin scripts. `register()` automatically inserts the port number of this application's `HTTPServer` object in the registration message. Thereby, a browser which has found an application via the searcher script will contact the applicatioin at the correct port and can immediately download the applet without further manual intervention.

3.3 Direct applet-applet communication

The objective here is to allow two applets within one distributed team to communicate directly with each other, no matter on which machines they are executed. The Remote Method Invocation (RMI) interface of Java 1.1 allows to invoke methods at a remote object once a reference to this remote object is obtained. Usually, these references are obtained from the so-called RMI-Registry. Since we want to inform other applets of new applets joining the team, we need an active component and can not solely rely on the RMI-Registry.

This active component is the Manager-Registry. It is itself remotely accessible and thus has a remote interface `KF_Registry` extending the class `java.rmi.Remote` that describes the methods that can be invoked remotely and the actual implementation of this interface `KF_RegistryImpl`[11]—see Figure 4 for an overview of the class hierarchy described in this subsection.

A `KF_RegistryImpl` object

1. registers with the RMI-Registry under a given service name to be visible to other objects,
2. offers the method `register()` in its remote interface to allow other remote objects to register with it,
3. collects all references for the remote objects created at the various applets with this service name, and,
4. upon receiving a new registration, redistributes them to all other known objects.[12]

The core functionality for a KnittingFactory remotely accessible object—namely the ability to register at a KF_Registry and to gather references distributed by the

[10] `register()` also takes an optional description parameter.

[11] which extends `java.rmi.server.UnicastRemoteObject`

[12] In particular, an object will receive a reference of itself.

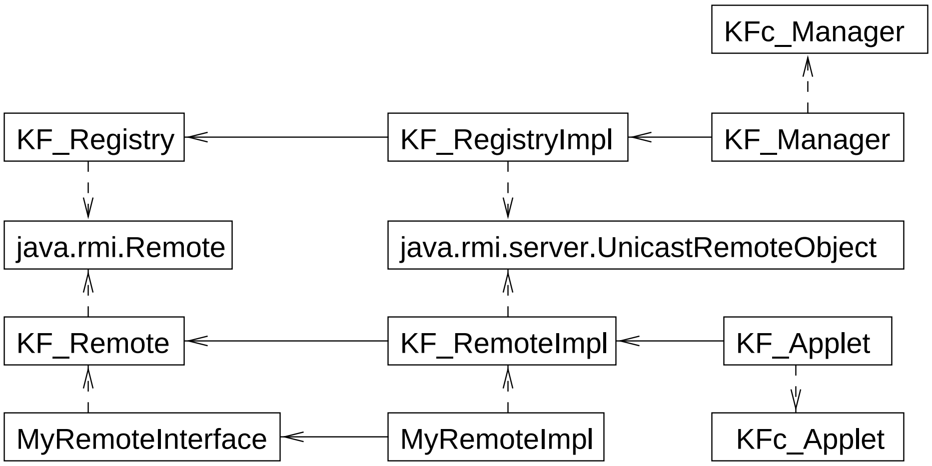

Figure 4: Class Hierarchy used for applet-applet communication (with MyRemoteInterface/MyRemoteImpl as example extensions of the KnittingFactory classes)

KF_Registry—is contained in the `KF_RemoteImpl` remote object implementing the interface `KF_Remote`. `KF_Remote` is the remote interface used by the KF_Manager and offers two methods: `addRemoteObject()` to inform an object of another remote object, and `setId()` to allow the KF_Manager to set unique ids for all remote objects. This interface is implemented by `KF_RemoteImpl`. In particular, this class also provides an `initialize()` method. Parameters to this method are the host where the RMI-registry is found and a service name identifying the KF_Registry on this host. In the `initialize()` method a `KF_RemoteImpl` object looks up the KF_Registry from the RMI-Registry on the given host[13] under the given service name. Once it has obtained a reference to the KF_Registry, it registers with it via the remote method `register()` passing a reference to itself to the KF_Registry. The registry will then send this reference via the `addRemoteObject()` to all known `KF_RemoteImpl` objects. The applet can obtain references to the other remote objects by calling `getWorker(id)` at a `KF_RemoteImpl` object.

`KF_Remote`/`KF_RemoteImpl` do not provide any methods to actually exchange information between distributed applets. To do so, interface and implementation must be extended to match the distributed computation's needs. For example, an implementation of the streams interface on top of the RMI-based applet-applet communication is conceivable (see subsections 3.3.2 for details). In Figure 4, `MyRemoteInterface` and `MyRemoteImpl` exemplify this.

To hide the necessity to explicitly create a `KF_RegistryImpl` object on the application side and the creation of and initialization calls for the `KF_RemoteImpl` objects on the applet side, we provide a `KF_Manager` class for the application and a `KF_applet` for the applet side which are built on top of the `KFc_Manager` and

[13] both application and RMI-Registry must run on the same host for the applet to be able to connect to the latter

KFc_Applet class, respectively. The stand-alone application only has to create a KF_Manager object, the constructor of which takes a service name parameter in addition to the class name and port described in subsection 3.2. This will not only create and initialize the KF_Registry, it will also provide the HTTPServer object with the service name so it can be passed along as a parameter to the applet once it is downloaded. KF_Applet will extract this parameter and setup static information in the KF_RemoteImpl class where both the RMI-registry and the KF_Registry can be found and under which service name the KF_Registry can be lookup up from the RMI registry. The actual applet can thus be derived from KF_applet, and, provided the KF_applet's init() function is called, any object of a class derived from KF_RemoteImpl can simply be created with new and will automatically register with the Manager-Registry and receive references for all other remote objects in this distributed computation.

3.3.1 Possible Enhancements

Applet-based updates: In the current implementation, the KF_Registry passes the reference of a new remote object to all previously known remote objects itself. Since this needs $O(n)$ remote method invocations per joining remote object, it does not scale well and raises the danger of the KF_Registry becoming a bottleneck.

A simple solution would be to let the joining object himself inform the other remote objects. This is easily possible and puts the burden on the newly joined object. But it does not take advantage of the possibility to distribute this information in parallel since other applets can pass this reference on, too.

Therefore, one solution might be to use the ids of the remote objects to form a binary tree and pass a new reference along this tree. This takes only $O(\log n)$ time as opposed to the $O(n)$ before. The disadvantage of this approach is that it is not fault-tolerant towards the failure of (non-leaf) remote objects. To amend this, other interconnection structures like a Beneŝ-network should be investigated; this ties in with work done on interconnection networks for parallel computer architectures.

Deregistering: While the current implementation does not provide explicit deregistration of remote objects, it is easy to do so as long as no KnittingFactory-related functionality is given to the remote objects like the applet-based update described above. This poses problems depending on the solution chosen for the interconnection network and needs further investigation.

Multiple service names: As discussed in Subsection 2.3 it might not be necessary or even desirable for all applets to know about each other. A distributed computation might divide the applets in logical groups that do interact directly within but only communicate with the application otherwise (an example for such an approach would be to consider these groups as replicated objects). One applet might be a member of one or several such groups.

A simple way to do this is to provide several service names and KF_Registries. One group is identified with one service name and administered by one KF_Registry object. The application creates such an object for every group it wants to use (possibly dynamically). An applet that wants to join a certain group creates a new remotely accessible object and passes the group/service name to the constructor of this object. This service name is used to look up and register with the corresponding KF_Registry. If an applet wants to join multiple groups, it creates one remotely accessible object for each group.

Providing callbacks in `KF_RegistryImpl`: The stand-alone application might want to be informed of applets joining or leaving the team even without having the applets to explicitly contact the application. An example for this might be the need to manipulate data structures containing the references to the remote objects—in a Chess program, all information about the remote objects might be deleted once two players have joined and connected to each other.

A callback mechanism is provided to allow a programmer to react flexibly to such events. Before and after a joining remote object is entered into `KF_RegistryImpl`'s data structures, the methods `pre_register()` and `post_register()` are called respectively. In the `KF_Registry` class, these are empty methods, but they can be redefined by extending `KF_Registry`, overwriting these methods, and reacting in a program-specific way to these events.

Doing without RMI-Registry: Currently, the official RMI-Registry is needed for bootstrapping purposes: a remote object needs the RMI-Registry to look up a reference of the corresponding KF_Registry. It might be advantageous to incorporate this functionality in the KnittingFactory itself.

3.3.2 Streams on top of RMI

In Java, the usual way of doing I/O of any kind is by using the I/O streams provided by the `java.io` package. To integrate the KnittingFactory applet-applet communication seamlessly into Java environments, this streams interface should be usable to communicate between applets, too.

An implementation of streams on top of KnittingFactory can be done by extending `KF_Remote` / `KF_RemoteImpl` to corresponding `KF_Stream` / `KF_StreamImpl` which maps a `write()` method call to a corresponding remote method invocation and putting the data in a remote buffer. The issue here is how to maintain this buffer efficiently while still guaranteeing correct semantics. Every `KF_StreamImpl` objects maintains a read buffer for all other objects a channel has been opened to. Bandwidth and latency of such an implementation (which reflects bandwidth and latency of the underlaying RMI implementation) should be investigated.

4 Examples

4.1 Passing a token

A simple example to demonstrate the ease–of–use of KnittingFactory is distributed token–passing: all the applets in the distributed computation pass a token from one to another, integrating new applets and continuing even after the application has died.

The source code for the stand–alone application, here called `Server`, is given in Figure 5. Notable is the creation of a `KF_Manager` object in line 10.

The token itself is represented by the color of a circle in the browser. So passing the token means resetting one's own color and setting the color of the logical neighbor. The remote interface this is shown in Figure 6.

Implementing this interface requires little work. `RingWorkerImpl` (Figure 7) does this in a straightforward way. Note that it also has a reference to the applet itself to be able to initiate a repaint once it has received the token.

Eventually, the core functionality of the applet itself is shown in Figure 8 (in particular, `try-catch` pairs are omitted—see Appendix A for a complete listing). This part has the longest code since it mostly implement the actual drawing of circles, holding the token for some time, etc. Note that the actual KnittingFactory related code is very small. The `Client` applet is derived from `KF_applet` and so has immediate access to most of the functionality. Note that in line 14 the `init()` function of `KF_applet` is called to extract the RMI–related parameter and prepare the `RingWorkerImpl` for the initialization. Thereby, in line 15 only a `worker = new RingWorkerImpl()` needs to be written - and worker refers to an object that knows about all other applet's remote objects and will be updated dynamically.

4.2 A shared whiteboard

Using KnittingFactory, a shared whiteboard is very simple to implement. Basically, all one has to do is write an applet implementing the actual drawing functionality and use KnittingFactory classes to make the drawing functions accessible to other applets so that drawing a figure can be broadcasted to all other applets. Figure 9 shows a screen shot of our shared whiteboard applet.

The complete source code for the whiteboard example can be found in Appendix B.

5 Related Work

The Web has the potential to be the infrastructure in integrating remote and heterogeneous computers into a (single) global computing resource. The Java programming language has the potential of removing many of the difficulties associated with untrusted and heterogeneous computing environments. And the growing number of Java capable browsers make them an ideal choice in seamlessly bringing distributed computing to every-day computer users. There is growing body of work on how to best

```java
package ring;

import knittingfactory.*;

public class Server
{
  public static void main (String av[])
  {
    KF_Manager s=null;
    try {
      s = new KF_Manager ("ring.Client.class", 0, "ring");
      // 1.) tell the browser to download "Class.class"
      // 2.) Server does not use any port
      // 3.) use "ring" as identifying service name
    }
    catch (java.rmi.RemoteException e){
      System.err.println ("couldn't create Manager: " +
                          e.getMessage());
      e.printStackTrace();
      return;
    }

    s.register ("RMI Test",
             "http://milan.milan.cs.nyu.edu/cgi-bin/");
    // register with the description "RMI Test" at
    // the registry running on milan.milan.cs.nyu.edu

    System.out.println ("Server started");
    try {
      Thread.sleep(30000);
    } catch (InterruptedException e) {
      ;
    }

    System.out.println ("Server unregistering...");
    s.deregister ("http://milan/cgi-bin/");
    // and deregister
  }
}
```

Figure 5: The server code for the token ring example:Server.java

```java
package ring;
import knittingfactory.*;
import java.awt.Color;

public interface RingWorker extends KF_Remote {
  public void setColor (Color color) throws java.rmi.RemoteException;
}
```

Figure 6: The remote interface: RingWorker.java

```java
package ring;
import knittingfactory.*;
import java.awt.Color;
import java.rmi.*;
import java.applet.*;

public class RingWorkerImpl extends KF_RemoteImpl implements RingWorker {
  private Client myApp;

  public RingWorkerImpl() throws RemoteException
  {
    myApp = null;
  }

  public RingWorkerImpl(Client c) throws RemoteException
  {
    myApp = c;
  }

  public void setColor(Color color) throws RemoteException
  {
    if (myApp != null) {
      myApp.changeColor(color);
      myApp.repaint();
    }
  }
}
```

Figure 7: The remote object implementation: RingWorkerImpl.java

```
import knittingfactory.*;
import java.applet.*;
import java.awt.*;
import java.rmi.*;
import java.net.InetAddress;

public class Client extends KF_Applet
    implements Runnable {
  final int width  = 300;
  final int height = 200;
  RingWorkerImpl worker;

  public void init() {
    super.init();
    worker = new RingWorkerImpl ();
    worker.setApplet (this);

    // ... code cut here...
  }
      // ... code cut here ...
  public void run() {
    while (true) {
      sleep (2000);

      if (worker.getColor() != Color.black) {
        int size   = worker.size();
        if (size > 1) {
          int myId   = worker.getId();
          int nextId = ((myId+1) % size);
          RingWorker nextWorker =
            (RingWorker) worker.getWorker(nextId);
          worker.setColor(Color.black);
          nextWorker.setColor(Color.red);
        }
      } else
        java.lang.Thread.yield();
    }
  }
}
```

Figure 8: The applet: Client.java (abbreviated version)

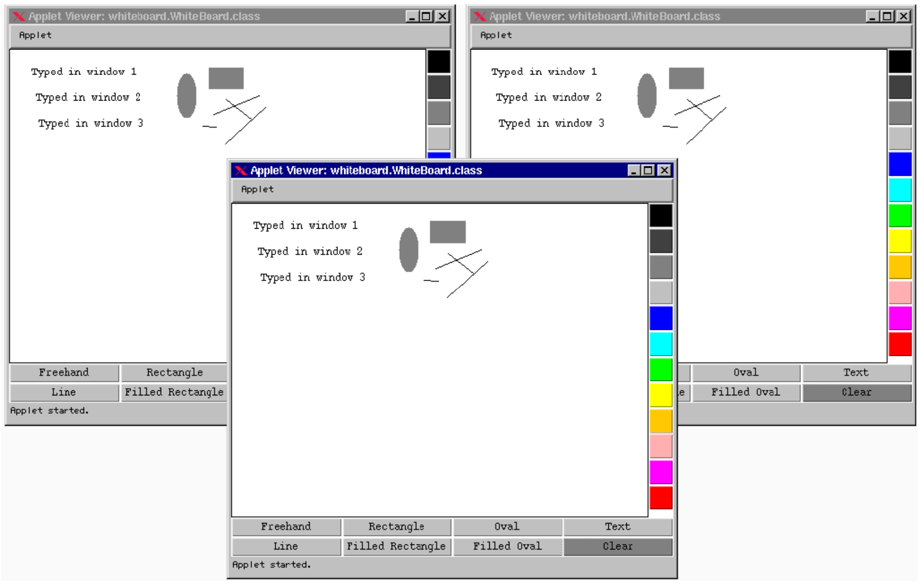

Figure 9: Screenshot of shared whiteboard showing three applets running on different machines

utilize the combination of these new technologies. This work can be loosely categorized based on their emphasis: those that focus on providing a parallel programming environment, or a metacomputer, on the Web; and those that focus on providing the infrastructure for collaborative applications on the Web.

5.1 Metacomputing on the Web

The projects that focus on a Java-based parallel computing include ATLAS [1], Charlotte [2], Java-MPI [21], JavaParty, [17], Javelin [6], JPVM [10], ParaWeb [5], and WebFlow [4]. Similarly, Web-enabled concurrent virtual machines have been proposed [12] as a High Performance Computing and Communications (HPCC) platform, as a basis for SPMD programming model [14]; and it has been proposed [11] to extend Java with global pointers and remote service request mechanisms from the Nexus communication library.

JPVM and Java-MPI provide message passing interfaces, based on the PVM and MPI programming models respectively, to applications. Java is utilized only for heterogeneity. Both systems lack the ability to download the application code from the network, and users are required to explicitly start daemon processes on each participating machine. These limit their applicability to local networks of workstations.

Another class of systems extend either the Java programming language, or its runtime environment (i.e., the Java Virtual Machine) to provide seamless distributed computations. Projects such as JavaParty, ParaWeb and the work proposed by Hummel *et al.*are such examples. JavaParty introduces the keyword *remote* to the programming language. Programs are then preprocessed by an extended Java compiler to standard Java with RMI calls. A runtime system that comprises of a single *manager* process and a *LocalJP* process on each site supports the execution of JavaParty programs. ParaWeb is an implementation of the Java Virtual Machine. It allows spawning threads on remote machines, provides message passing and shared memory semantics. Both systems suffer from a tight integration to the current Java implementation, and require a non-standard implementation. Hence, unlike KnittingFactory, they are not able to execute on remote autonomous machines, and specifically, within a users Web browser.

ATLAS is a Java based software that is based on Cilk's programming model and work-stealing distributed scheduling. Their hierarchy of manager applications provides a scalable method for idle worker processes to find, and hence, to steal threads form other busy workers. ATLAS relies on native methods and, in the current implementation, there is no support for shared variables among threads. The use of native methods prevents the applicability of ATLAS to the Web since portability, security, and the lack of the ability to download the application code becomes an issue.

Charlotte and Javelin are two 100% Pure Java software systems that were specifically designed for parallel programming on the Web with simplicity in mind. Their design and how they can benefit by using the infrastructure provided by KnittingFactory will be discussed in Section 6.

5.2 Collaborative applications on the Web

The second category of projects focus providing a software infrastructure for collaboration applications. Examples of such applications are distributed white boards, calendars, and editors where multiple users collaborate towards one goal. Software systems that support collaborative applications include the E programming language [9], Caltech Infosphere project [7], HORB [18], iBus [16], Jada/PageSpace [8], Java Collaborator Toolset [15], Java Remote Method Invocation (RMI) [19], JavaSpaces [13], and TANGO [3].

The E programming language and HORB extends Java in several ways. The E language adds message passing through channels and richer security through cryptographic and authentication. HORB is a Object Request Broker (ORB) for Java and adds a well understood programming model, RPC, and persistent objects. Both HORB and E are tightly integrated with a given Java implementation.

Java Remote Method Invocation (RMI) as a standard part of the Java 1.1 specification provides RPC mechanisms to Java. KnittingFactory builds on the client/server RPC model to provide seamless integration of applets with RMI. Jada and JavaSpaces integrate the concepts of Linda with Java. Where as Jada uses a Linda-like tuple model, JavaSpaces utilizes object hierarchies for tuple matching. PageSpace is a collaboration software package that is built on top of Jada. Because of their communication patterns, these systems can function as applications and not as applets, hence, limit their applicability to local networks of workstations.

Java Collaborator Toolset and TANGO provide the infrastructure for collaborative Java applications. They are of special interest to us since they both utilize Web browsers as users front-end, and hence, both have to work within the Java applet security model. Their design and how they can benefit by using the infrastructure provided by KnittingFactory will be discussed in Section 6.

6 KnittingFactory in Context

This section discusses how some related projects could benefit from using KnittingFactory. As in Section 5, we will treat projects for parallel programming and collaborative work separately.

6.1 Metacomputing on the Web

As examples of the research efforts aiming to provide a metacomputer on the Web, we focus on the Javelin and Charlotte projects.

6.1.1 Javelin

Javelin is a project at University of California, Santa Barbara, supporting the development and execution of parallel applications on the Web. Javelin is 100% Pure Java, and hence, it utilizes Java's security model and its ability to download the application code over a network.

There are three main components in a Javelin computation: a *broker*, a *client*, and *hosts*. A broker is a system-wide Java application that functions as a repository for the Java applet programs, and matches the client tasks with hosts. A client submits a task to the broker through a Web browsers by generating an HTTP POST-request, and periodically polls the broker for the results. Hosts are Web browsers (generally running on idle machines) that repeatedly contact the broker for tasks to perform, download the applet code, execute it to completion, and return the results to the broker. The communication between any two applets is routed through the broker as stated in [6] (Section 2.4)

> In general, messages between applet must be routed through the broker, because an applet cannot open a network connection to any site other then the one from which it was loaded.

On top of this framework, Javelin provides language support for synchronization, message passing, and a tuple space model of communication.

While the Javelin is a great step towards parallel programming on the Web, it is hindered by Java applet security model. For example, since hosts require a direct connection to the broker, an HTTP server (and possibly and FTP server) must be running on the same machine as the broker. Scalability might also become an issue since every message of every parallel application is routed through the broker.

While the functionality of Javelin transcends KnittingFactory's goals, KnittingFactory can provide a flexible infrastructure. First, KnittingFactory can provide the mechanism for client and host applets to search the Web and find each other directly. Second, the embedded light-weight Java class server of KnittingFactory can possibly be used to offload some of the responsibilities of the system-wide broker. And finally, the applet-to-applet communication mechanism provided by KnittingFactory can remove the message routing by the broker for higher scalability.

6.1.2 Charlotte

Charlotte is a research project at New York University providing a metacomputing environment on the Web. Similar to Javelin, Charlotte is written in Java and does not rely on any native code, and thus, takes advantage of Java's security model and its ability to download the application code over a network.

Charlotte is based on Calypso's programming model to provide parallel routines and shared memory abstraction on distributed machines. It relies on two integrated techniques, *eager scheduling* and *two-phase idempotent execution strategy* for load balancing and fault masking. A Charlotte program is a Java program with embedded parallel steps to perform the computationally intensive parts of the program. A parallel step is composed of one or more *routines*. A routine is analogous to a standard Java thread, except for its ability to execute remotely. A distributed shared name-space provides shared variables among routines.

There are two main components in a Charlotte program: a *master*, and one or more *workers*. The master is a Java application that creates an entry in a Web page at invocation. A worker is a Web browser generally running on an idle machine. By clicking on an entry, a worker downloads and executes the applet code. A direct socket connection provides the manager-worker communication.

As with the Javelin project, Charlotte is also hindered by the Java applet security model. For example, the manager is required to run on a machine running an HTTP server, otherwise worker applets could not establish a direct communication channel to the manager. In addition, Charlotte does not address the question of how a Web browser, i.e., the worker, finds work. The current implementation requires manual input of the URL location. KnittingFactory was designed to address both of these questions. As a proof of concept, we have ported Charlotte to use KnittingFactory as the underlying infrastructure. It required less than 12 lines of code.

6.2 Collaborative applications on the Web

As examples for projects geared towards collaborative work, we focus on TANGO [3] and the Java Collaborator Toolset [15].

6.2.1 TANGO

TANGO is a system built to support web-based collaboration applications. Main goals of TANGO are system flexibility and the capability of integrating existing applications. TANGO provides functionalities such as session management, user authentication, event logging, and communication between collaborating applications. While this certainly transcends KnittingFactory's goals, a closer look at TANGO's system architecture shows how it can benefit from KnittingFactory's capabilities.

The main elements of TANGO's architecture are a *local daemon* which is mainly responsible for communication between different TANGO components, and a *central server*. This server provides event logging, maintenance of system state, and relays messages between the local daemons. Since the local daemon is implemented as a plug-in for Netscape, it is free to communicate with other daemons, but it is no longer trusted code and requires local installation.

On the other hand, [3] states explicitly (in Section 2.5) that

> Using pure Java model we would have to create a client-server model in which all communication and data distribution are located in one host. This idea had to be rejected because of its inflexibility.

Thus, the local daemon is merely an attempt to circumvent the restrictions imposed upon the communication patterns by the Java security model. KnittingFactory presents a method for reaching the same goal—namely direct and arbitrary communication between remote applets–without relying on native-code plug-ins or having to relay messages through a central server. As opposed to the local daemon, KnittingFactory is fully implemented in Java, it does not face portability problems that native-code methods naturally have to cope with, and it is fully trusted code, respecting the Java security model, and does not require installation on the user's side.

6.2.2 Java Collaborator Toolset

The Java Collaborator Toolset is a platform supporting the design and implementation of collaborative Java applications. It is realized as a replacement for the standard Java Abstract Window Toolkit (awt), *collawt*. The main functionality of these collawt classes is that every event (e.g., a mouse click) is send to a central *event distributor* process that redistributes these events to all other applets. Since the event distributor is used to forward events of multiple applications, events carry (among other information) an application id so that they are only forwarded to applets belonging to the same distributed application.

This centralized event distributor shows the typical dangers: it represents a single point of failure (for multiple, independent applications on top of it) and has a severe chance of becoming a bottleneck. Additionally, this event distributor must run on the HTTP server host that provides the applet code to allow the applets to open a connection to it.

Using KnittingFactory, this design can be improved upon in an obvious fashion. Replacing both the event distributor and the HTTP server, a KnittingFactory process serves the applet code to a browsers and sets up the direct applet-applet communication. Thereby, applets can send their events directly to other interested applets improving scalability. It also improves fault tolerance since, unlike the event distributor, the KnittingFactory process is only needed during set-up time.

One advantage of a centralized distributor is the implicit synchronization it provides. But providing (virtual) synchrony on top of the KnittingFactory broadcast capabilities is merely a matter of implementing appropriate protocols, e.g., the token-based Totem protocol.

Note that using KnittingFactory would only require changes in the implementation of collawt; the user interface remains unchanged.

7 Conclusions

KnittingFactory provides solutions for three problems with which collaborative and parallel Internet applications are faced.

The registry service solves the problem of how collaborative applets find each other. This is achieved by executing breadth-first search in the Web browser of a prospective partner, and adheres to the browsers security model. By embedding the core functionalities of an HTTP server, namely serving Java applet code, applications can be started on any machine connected to the network, applets can be downloaded directly from the application process, and can therefore establish direct connections. Thus, the initiator of a collaborative application does not need to have access to a machine running an HTTP server. KnittingFactory also allows larger flexibility in designing a distributed application by providing means for direct applet-applet communication. Depending on the actual application, this can result in improved fault tolerance, scalability and efficiency. KnittingFactory completely adheres to all relevant standards and security models.

The validity and ease of use of KnittingFactory is demonstrated with two example applications: token passing and a shared whiteboard. In both applications we demonstrate the efficiency of our embedded applet server and applet-to-applet communication without an intermediary. The techniques validated through the KnittingFactory are applicable not only to collaborative systems, but to parallel programming systems such as Charlotte to overcome many of the difficulties imposed by the Java security model. It is important to repeat that KnittingFactory completely adheres to the Java security model.

8 Acknowledgements

This research was sponsored by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-96-1-0320; by the National Science Foundation under grant number CCR-94-11590; by the Intel Corporation; and by Microsoft. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

References

[1] J. E. Baldeschwieler, R. D. Blumofe, and E. A. Brewer. ATLAS: An Infrastructure for Global Computing. In *Proc. of the 7th ACM SIGOPS European Workshop: Systems support for Worldwide Applications*, Connemara, Ireland, September 1996.

[2] A. Baratloo, M. Karaul, Z. Kedem, and P. Wyckoff. Charlotte: Metacomputing on the Web. In *Proc. of the 9th International Conference on Parallel and Distributed Computing Systems*, Dijon, France, September 1996.

[3] L. Beca, G. Cheng, G. C. Fox, T. Jurga, K. Olszewski, M. Podgorny, P. Sokolwski, and K. Walczak. Web Technologies for Collaborative Visualization and Simulation. Technical Report SCCS-786, Northeast Parallel Architectures Center, January 1997.

[4] D. Bhatia, V. Camuseva, M. Camuseva, G. Fox, W. Furmanski, and G. Premchandran. Web-Flow – a Visual Programming Paradigm for Web/Java Based on Coarse Grain Distributed Computing. *Concurrency: Practice and Experience*, March 1997.

[5] T. Brecht, H. Sandhu, M. Shan, and J. Talbot. ParaWeb: Towards World-Wide Supercomputing. In *7th ACM SIGOPS European Workshop*, pages 181–188, Connemara, Ireland, September 1996.

[6] P. Cappello, B. Christiansen, M. F. Ionescu, M. O. Neary, K. E. Schauser, and D. Wu. Javelin: Internet-Based Parallel Computing Using Java. In *ACM Workshop on Java for Science and Engineering Computation*, 1997. http://www.cs.ucsb.edu/~schauser/papers/.

[7] K. M. Chandy, A. Rifkin, P. A. G. Sivilotti, J. Mandelson, M. Richardson, W. Tanaka, and L. Weisman. A World-Wide Distributed System Using Java and the Internet. In *IEEE International Symposium on High Performance Distributed Computing*, Syracuse, NY, August 1996. http://www. infospheres.caltech.edu/papers/index.html.

[8] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Redesigning the Web: From Passive Pages to Coordinated Agents in PageSpaces. In *Third International Symposium on Autonomous Decentralized Systems*, Berlin, Germany, April 1997. http://www.fokus.gmd.de/ws/isads97/.

[9] Electric Communities. The E Programming Language. http://www.communities/e/epl.html.

[10] A. Ferrari. JPVM – The Java Parallel Virtual Machine. http://www.cs.virginia.edu/~ajf2j/jpvm.html.

[11] I. Foster and S. Tuecke. Enabling Technologies for Web-Based Ubiquitous Supercomputing. In *Proc. 5th IEEE Symp. on High Performance Distributed Computing*, pages 112–110, 1996.

[12] G. Fox and W. Furmanski. Towards Web/Java Based High Perfomance Distributed Computing – an Evolving Virtual Machine. In *Proc. of the Symposium on High Performance Distributed Computing*, 1996.

[13] R. Guth. Sun's JavaSpaces is Foundation for Future Distributed Systems. IDG News Services, http://www.javaworld.com/javaworld/jw-09-1997/jw-09-idgns.javaspaces.html, September 1997.

[14] S. Hummel, T. Ngo, and H. Srinivasan. SPMD Programming in Java. *Concurrency: Practice and Experience*, March 1997.

[15] B. Kvande. The Java Collaborator Toolset, a Collaborator Platform for the Java(tm) Environment. Master's thesis, Department of Computer Science, Old Dominion University, August 1996.

[16] S. Maffeis. iBus—The Java Intranet Software Bus. http://www.olsen.ch/export/proj/ibus/.

[17] M. Philippsen and M. Zenger. JavaParty—Transparent Remote Objects in Java. In *ACM 1997 PPoPP Workshop on Java for Science and Engineering Computation*, Las Vegas, NV, June 1997.

[18] Hirano Satoshi. Preliminary Performance Evalutaion of a Distributed Java: HORB. In *France-Japan Workshop on Object-Based Parallel and Distributed Computation*, Toulouse, France, 1997. http://ring.etl.go.jp/openlab/horb/.

[19] Sun microsystems, Mountain View, CA. *RMI—Remote Method Invocation (JDK 1.1.3)*.

[20] V. S. Sunderam, A. Geist, J. Dongarra, and R. Mancheck. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing*, 20:531–546, April 1994.

[21] S. Taylor. Prototype Java-MPI Package. http://cisr.anu.edu.au/~sam/java/java_mpi_prototype.html.

A The Client code for Token passing

This appendix shows the complete code for the Client–Applet from Section 4.

```
package ring;
import knittingfactory.*;
import java.applet.*;
import java.awt.*;
import java.rmi.*;
import java.net.InetAddress;


public class Client extends KF_Applet implements Runnable {
  final int       width   = 300;
  final int       height  = 200;
  Color           myColor;
  RingWorkerImpl worker;


  public void init() {
    super.init();
    try {
      worker = new RingWorkerImpl (this);
    } catch (RemoteException e) {
      e.printStackTrace();
    }
    System.out.println ("created RingworkerImpl");
    if (worker == null)
      System.out.println ("but it is null!!!");

    if (worker.size() ==1)
      changeColor(Color.red);
    else
      changeColor(Color.black);

    Thread artist = new Thread(this);
    artist.start();
  }


  public void paint(Graphics g) {
    int size = worker.size();

    int maxx = width / 2;
    int maxy = height / 2;

    double del   = 2*java.lang.Math.PI / size;
    double angle = 0;
    for (int i=0; i<size; i++) {
      int x = (int)(maxx * java.lang.Math.cos(angle));
      int y = (int)(maxy * java.lang.Math.sin(angle));
      String str;
      if (i == worker.getId()) {
        str = "me";
        g.setColor(getColor());
      } else {
        str = "worker:"+i;
```

```
        g.setColor(Color.black);
      }
      g.drawString(str, maxx+x, maxy+y);
      g.drawOval(maxx+x, maxy+y, 20, 20);
      angle+=del;
    }
  }

  synchronized public void changeColor(Color c) {
    myColor = c;
  }

  synchronized public Color getColor() {
    return myColor;
  }

  static void sleep(int i) {
    try {java.lang.Thread.sleep(i);} catch (Exception e) {}
  }

  public void run() {
    while (true) {
      repaint();
      java.lang.Thread.yield();
      sleep (3000);

      if (getColor() != Color.black) {
        int size    = worker.size();
        if (size > 1) {
          int myId    = worker.getId();
          int nextId = ((myId+1) % size);
          RingWorker nextWorker = (RingWorker) worker.getRemoteObject(nextId);
          try {
            // System.out.println ("throwing from " + myId + " to " + nextId);
            nextWorker.setColor(Color.red);
            changeColor(Color.black);
          } catch (RemoteException e) {
            e.printStackTrace();
          }
        }
      } else {
        java.lang.Thread.yield();
      }
    }
  }
}
```

B The Whiteboard code

B.1 Server.java

```java
package whiteboard;
import knittingfactory.*;


public class Server {
  public static void main (String av[]) {
    KF_Manager s=null;
    try{
      s = new KF_Manager ("whiteboard.WhiteBoard.class", 0, "whiteboard");
      // 1.) tell the browser to download "whiteboard.WhiteBoard.class"
      // 2.) Server does not use any port
      // 3.) use "whiteboard" as identifying service name
    } catch (java.rmi.RemoteException e) {
      System.err.println ("Couldn't create Manager: " + e.getMessage());
      e.printStackTrace();
      return;
    }

    s.setAppletDimensions(500, 400);

    System.out.println ("Server started");
    // let's forget about the Part1 - registering/deregistering for the moment:

    /*
    s.register ("RMI Test", "http://milan.milan.cs.nyu.edu/cgi-bin/");
    //s.register ("RMI Test", "http://localhost/cgi-bin/");
    // register with the description "RMI Test" at
    // the registry running on milan.milan.cs.nyu.edu

    System.out.println ("Server started");
    try { Thread.sleep(30000); }
    catch (InterruptedException e) { }

    System.out.println ("Server going down");
    s.deregister ("http://milan/cgi-bin/");
    //s.deregister ("http://localhost/cgi-bin/");
    // and deregister

    */
  }
}
```

B.2 RemoteBoard.java

```java
package whiteboard;
import knittingfactory.*;
import java.awt.Color;


public interface RemoteBoard extends KF_Remote {
```

```
  public void drawLine(Color color, int startx, int starty, int endx, int endy) throws java.rmi.RemoteException;
  public void drawRect(Color color, int x, int y, int width, int height) throws java.rmi.RemoteException;
  public void fillRect(Color color, int x, int y, int width, int height) throws java.rmi.RemoteException;
  public void drawOval(Color color, int x, int y, int width, int height) throws java.rmi.RemoteException;
  public void fillOval(Color color, int x, int y, int width, int height) throws java.rmi.RemoteException;
  public void showKey(Color color, int x, int y, char ch) throws java.rmi.RemoteException;
  public void clearBoard() throws java.rmi.RemoteException;
}
```

B.3 RemoteBoardImpl.java

```
package whiteboard;
import knittingfactory.*;
import java.applet.Applet;
import java.awt.*;


public class RemoteBoardImpl extends KF_RemoteImpl implements RemoteBoard {
  WhiteBoard myApplet;

  public RemoteBoardImpl(WhiteBoard applet) throws java.rmi.RemoteException {
    myApplet = applet;
  }


  public void drawLine(Color color, int startx, int starty, int endx, int endy) throws java.rmi.RemoteException {
    myApplet.drawLine(color, startx, starty, endx, endy, false, true);
  }


  public void drawRect(Color color, int x, int y, int width, int height) throws java.rmi.RemoteException {
    myApplet.drawRect(color, x, y, width, height, false, true);
  }


  public void fillRect(Color color, int x, int y, int width, int height) throws java.rmi.RemoteException {
    myApplet.fillRect(color, x, y, width, height, false, true);
  }


  public void drawOval(Color color, int x, int y, int width, int height) throws java.rmi.RemoteException {
    myApplet.drawOval(color, x, y, width, height, false, true);
  }


  public void fillOval(Color color, int x, int y, int width, int height) throws java.rmi.RemoteException {
    myApplet.fillOval(color, x, y, width, height, false, true);
  }


  public void showKey(Color color, int x, int y, char ch) throws java.rmi.RemoteException {
    myApplet.showKey(color, x, y, ch, false);
  }


  public void clearBoard() throws java.rmi.RemoteException {
```

```
    myApplet.clearBoard(false);
  }
}
```

B.4 WhiteBoard.java

```
package whiteboard;
import knittingfactory.*;
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;


// added a boolean parameter remotely_invoced to the draw... routines
// this *should* fix the problem of wrong colors when drawing from a
// remote whiteboard                    - hk

public class WhiteBoard extends KF_Applet
implements Runnable, MouseListener, MouseMotionListener {
  private final int FREEHAND = 0;          // freehand drawing mode
  private final int LINE     = 1;          // line drawing mode
  private final int RECT     = 2;          // rectangle drawing mode
  private final int FILLRECT = 3;          // fill rectangle drawing mode
  private final int OVAL     = 4;          // oval drawing mode
  private final int FILLOVAL = 5;          // fill oval drawing mode
  private final int TEXT     = 6;          // text mode

  private Thread animator  = null;         // does the actual drawing
  private Image  offScreen = null;         // for doublebuffering

  private Color  XorAlternateColor = new Color(255,150,255);
                              // light pink

  // various buttons
  private Button blackButton, blueButton, cyanButton, darkGrayButton,
                 grayButton, greenButton, lightGrayButton, magentaButton,
                 orangeButton, pinkButton, redButton, yellowButton;
  private Button freehandButton, lineButton, rectButton, fillRectButton,
                 ovalButton, fillOvalButton, textButton, clearButton;

  private Panel  modePanel;                // panel of various drawing modes
  private int    currMode;                 // currently active mode
  private Panel  colorPanel;               // panel of various foreground colors
  private Color  currColor;                // current foreground color
  private Choice fontChoice;               // choice of available fonts

  private boolean mouseDown;               // true if mouse button is pressed
                              // mouseDown also serves as indicator wether or not to
                              // use XOR-drawing mode or standard mode
                              //                   - hk

  private int    downX, downY;             // coords when mouse button was pressed
  private int    currX, currY;             // current mouse coords if pressed
  private int    lastX, lastY;             // mouse coords at last update() call
  private int    imageWidth, imageHeight;  // size of drawing area
```

```
private int       offX, offY;              // coord offsets for drawing area

private RemoteBoardImpl remoteBoard;



// ————————————————————————————
// Standard applet methods in order of invocation
// ————————————————————————————

public void init() {
  //System.err.println ("WhiteBoard: before calling super.init()");
  super.init();
  //System.err.println ("WhiteBoard: after calling super.init()");
  setBackground(Color.white);

  // create buttons
  blackButton = new Button("  ");
  blackButton.setBackground(Color.black);
  blueButton = new Button("  ");
  blueButton.setBackground(Color.blue);
  cyanButton = new Button("  ");
  cyanButton.setBackground(Color.cyan);
  darkGrayButton = new Button();
  darkGrayButton.setBackground(Color.darkGray);
  grayButton = new Button();
  grayButton.setBackground(Color.gray);
  greenButton = new Button();
  greenButton.setBackground(Color.green);
  lightGrayButton = new Button();
  lightGrayButton.setBackground(Color.lightGray);
  magentaButton = new Button();
  magentaButton.setBackground(Color.magenta);
  orangeButton = new Button();
  orangeButton.setBackground(Color.orange);
  pinkButton = new Button();
  pinkButton.setBackground(Color.pink);
  redButton = new Button();
  redButton.setBackground(Color.red);
  yellowButton = new Button();
  yellowButton.setBackground(Color.yellow);

  freehandButton = new Button("Freehand");
  freehandButton.setBackground(Color.lightGray);
  lineButton = new Button("Line");
  lineButton.setBackground(Color.lightGray);
  rectButton = new Button("Rectangle");
  rectButton.setBackground(Color.lightGray);
  fillRectButton = new Button("Filled Rectangle");
  fillRectButton.setBackground(Color.lightGray);
  ovalButton = new Button("Oval");
  ovalButton.setBackground(Color.lightGray);
  fillOvalButton = new Button("Filled Oval");
  fillOvalButton.setBackground(Color.lightGray);
  textButton = new Button("Text");
  textButton.setBackground(Color.lightGray);
  clearButton = new Button("Clear");
```

```
clearButton.setBackground(Color.gray);

// created choice for fonts
fontChoice = new Choice();
String[ ] fontList = Toolkit.getDefaultToolkit().getFontList();
for (int i = 0; i < fontList.length; i++)
  fontChoice.addItem(fontList[i]);
fontChoice.select("Courier");
setFont(Font.decode("Courier"));

// create panels
colorPanel = new Panel();
colorPanel.setLayout(new GridLayout(12,1));
//colorPanel.add(new Label("Color"));
colorPanel.add(blackButton);
colorPanel.add(darkGrayButton);
colorPanel.add(grayButton);
colorPanel.add(lightGrayButton);
colorPanel.add(blueButton);
colorPanel.add(cyanButton);
colorPanel.add(greenButton);
colorPanel.add(yellowButton);
colorPanel.add(orangeButton);
colorPanel.add(pinkButton);
colorPanel.add(magentaButton);
colorPanel.add(redButton);

modePanel = new Panel();
modePanel.setLayout(new GridLayout(2,4));
//modePanel.add(new Label("Mode"));
modePanel.add(freehandButton);
modePanel.add(rectButton);
modePanel.add(ovalButton);
modePanel.add(textButton);
modePanel.add(lineButton);
modePanel.add(fillRectButton);
modePanel.add(fillOvalButton);
modePanel.add(clearButton);
//modePanel.add(fontChoice);

setLayout(new BorderLayout());
add("East", colorPanel);
add("South", modePanel);

// choose defaults
currMode = FREEHAND;
currColor = Color.black;

offX = colorPanel.minimumSize().width + 1;
offY = modePanel.minimumSize().height + 1;

// initialize event handling
this.addMouseListener(this);
this.addMouseMotionListener(this);

// initialize remote board
```

```java
    try { remoteBoard = new RemoteBoardImpl(this); }
    catch (java.rmi.RemoteException e) { e.printStackTrace(); }
  }

  public void start() {
    animator = new Thread(this);
    animator.start();
  }

  public void run() {
    // initialize drawing buffers and context
    imageWidth  = size().width - offX - 1;
    imageHeight = size().height - offY - 1;
    offScreen   = this.createImage(imageWidth, imageHeight);

    System.err.println ("Hello from WhiteBoard");

    // draw outline of drawing area
    Graphics g = offScreen.getGraphics();
    g.setColor(Color.black);
    g.drawRect(0, 0, imageWidth + 1, imageHeight + 1);

    g = this.getGraphics();
    while (true) {
      Thread.currentThread().yield();

      g.drawImage(offScreen, 1, 1, this);
      repaint();

      try { Thread.sleep(100); }
      catch (InterruptedException e) { }
    }
  }

  public void update(Graphics g) {
    getGraphics().setColor(Color.black);
    getGraphics().drawRect(0, 0, imageWidth + 1, imageHeight + 1);
  }


  public String getAppletInfo() {
    return "Mehmet's little KnittingFactory demo: \n--- A simple shared whiteboard. ---";
  }

  public void stop() {
    if (animator != null)
      animator.stop();
  }


  // ------------------------------------
```

```
// Event handling routines
// ------------------------------------------

public boolean action(Event event, Object arg) {
  // System.out.println("got event: " + event.target + ":" + arg);
  if (event.target == blackButton)
    currColor = Color.black;
  else if (event.target == blueButton)
    currColor = Color.blue;
  else if (event.target == cyanButton)
    currColor = Color.cyan;
  else if (event.target == darkGrayButton)
    currColor = Color.darkGray;
  else if (event.target == grayButton)
    currColor = Color.gray;
  else if (event.target == greenButton)
    currColor = Color.green;
  else if (event.target == lightGrayButton)
    currColor = Color.lightGray;
  else if (event.target == magentaButton)
    currColor = Color.magenta;
  else if (event.target == orangeButton)
    currColor = Color.orange;
  else if (event.target == pinkButton)
    currColor = Color.pink;
  else if (event.target == redButton)
    currColor = Color.red;
  else if (event.target == yellowButton)
    currColor = Color.yellow;

  else if (event.target == freehandButton)
    currMode = FREEHAND;
  else if (event.target == lineButton)
    currMode = LINE;
  else if (event.target == rectButton)
    currMode = RECT;
  else if (event.target == fillRectButton)
    currMode = FILLRECT;
  else if (event.target == ovalButton)
    currMode = OVAL;
  else if (event.target == fillOvalButton)
    currMode = FILLOVAL;
  else if (event.target == textButton)
    currMode = TEXT;
  else if (event.target == fontChoice) {
    fontChoice.select((String)arg);
    setFont(Font.decode((String)arg));
  }
  else if (event.target == clearButton)
    clearBoard(true);
  else
    return super.action(event, arg);
  return true;
}


public boolean keyDown(Event event, int key) {
```

```
    char ch = (char)event.key;
    if ((currMode == TEXT) && (event.id == Event.KEY_PRESS)) {
      switch(ch) {
      case '\t':
        currX += 15;
        return true;
      case '\n':
        currX  = 7;
        currY += 15;
        return true;
      default:
        showKey(currColor, currX, currY, ch, true);
        repaint();
        currX += 7;
        return true;
      }
    }
    return false;
  }


  public void mousePressed(MouseEvent e) {
    lastX = downX = currX = e.getX();
    lastY = downY = currY = e.getY();
    mouseDown = true;
  }


  public void mouseDragged(MouseEvent e) {
    currX = e.getX();
    currY = e.getY();

    // draw over the old figure and draw the new one
    if (mouseDown){

      //System.out.println  ("mouseDragged: lx=" + lastX + " ly="+lastY
      //                         +" cx=" + currX + "cy=" + currY );

      switch(currMode) {
      case FREEHAND:
        drawLine(currColor, lastX, lastY, currX, currY, true, false);
        break;
      case LINE:
        drawLine(currColor, downX, downY, lastX, lastY, false, false);
        drawLine(currColor, downX, downY, currX, currY, false, false);
        break;
      case RECT:
      case FILLRECT:
        drawRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
                 Math.abs(downX - lastX), Math.abs(downY - lastY), false, false);
        drawRect(currColor, Math.min(downX, currX), Math.min(downY, currY),
                 Math.abs(downX - currX), Math.abs(downY - currY), false, false);
        break;
      case OVAL:
      case FILLOVAL:
        drawRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
                 Math.abs(downX - lastX), Math.abs(downY - lastY), false, false);
```

```
        drawOval(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
                Math.abs(downX - lastX), Math.abs(downY - lastY), false, false);
        drawRect(currColor, Math.min(downX, currX), Math.min(downY, currY),
                Math.abs(downX - currX), Math.abs(downY - currY), false, false);
        drawOval(currColor, Math.min(downX, currX), Math.min(downY, currY),
                Math.abs(downX - currX), Math.abs(downY - currY), false, false);
        break;
      }
    }

    lastX = currX;
    lastY = currY;

  }


  public void mouseReleased(MouseEvent e) {
    switch(currMode) {
    case LINE:
      drawLine(currColor, downX, downY, lastX, lastY, false, false);
      mouseDown = false;
      drawLine(currColor, downX, downY, currX, currY, true, false);
      break;
    case RECT:
      drawRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
              Math.abs(downX - lastX), Math.abs(downY - lastY), false, false);
      mouseDown = false;
      drawRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
              Math.abs(downX - lastX), Math.abs(downY - lastY), true, false);
      break;
    case FILLRECT:
      drawRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
              Math.abs(downX - lastX), Math.abs(downY - lastY), false, false);
      mouseDown = false;
      fillRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
              Math.abs(downX - lastX), Math.abs(downY - lastY), true, false);
      break;
    case OVAL:
      drawRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
              Math.abs(downX - lastX), Math.abs(downY - lastY), false, false);
      drawOval(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
              Math.abs(downX - lastX), Math.abs(downY - lastY), false, false);
      mouseDown = false;
      drawOval(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
              Math.abs(downX - lastX), Math.abs(downY - lastY), true, false);
      break;
    case FILLOVAL:
      drawRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
              Math.abs(downX - lastX), Math.abs(downY - lastY), false, false);
      drawOval(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
              Math.abs(downX - lastX), Math.abs(downY - lastY), false, false);
      mouseDown = false;
      fillOval(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
              Math.abs(downX - lastX), Math.abs(downY - lastY), true, false);
      break;
    }
  }
```

```
// Unused methods from MouseListener interface
public void mouseClicked(MouseEvent e) { }
public void mouseEntered(MouseEvent e) { }
public void mouseExited(MouseEvent e) {
  if (mouseDown){
    // remove anything painted:
    switch(currMode) {
    case LINE:
      drawLine(currColor, downX, downY, lastX, lastY, false, false);
      mouseDown = false;
      break;
  case RECT:
    drawRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
             Math.abs(downX − lastX), Math.abs(downY − lastY), false, false);
    mouseDown = false;
    break;
  case FILLRECT:
    drawRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
             Math.abs(downX − lastX), Math.abs(downY − lastY), false, false);
    mouseDown = false;
    break;
  case OVAL:
    drawRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
             Math.abs(downX − lastX), Math.abs(downY − lastY), false, false);
    drawOval(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
             Math.abs(downX − lastX), Math.abs(downY − lastY), false, false);
    mouseDown = false;
    break;
  case FILLOVAL:
    drawRect(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
             Math.abs(downX − lastX), Math.abs(downY − lastY), false, false);
    drawOval(currColor, Math.min(downX, lastX), Math.min(downY, lastY),
             Math.abs(downX − lastX), Math.abs(downY − lastY), false, false);
    mouseDown = false;
    break;

    }
  }

}

// Unused methods from MouseMotionListener interface
public void mouseMoved(MouseEvent e) { }



// ——————————————————————
// Drawing and broadcasting routines
// ——————————————————————

public void drawLine(Color color, int startX, int startY, int endX, int endY,
                     boolean bcast, boolean remotely_invoced) {
  Graphics g = offScreen.getGraphics();
  if (mouseDown && (currMode != FREEHAND) && (!remotely_invoced))
    // all the remoyely invoced stuff is drawn with setPaintMode -> otherwise
```

```
    //      wrong colors
    // but FREEHAND is a special case: always drawn with setPaintMode
    g.setXORMode (XorAlternateColor);
  else
    g.setPaintMode();
  drawLine(g, color, startX, startY, endX, endY);

  if (bcast) {
    int size = remoteBoard.size();
    int me    = remoteBoard.getId();
    for (int i=0; i<size; i++)
      if (i != me) {
        RemoteBoard other = (RemoteBoard) remoteBoard.getRemoteObject(i);
        try { other.drawLine(color, startX, startY, endX, endY); }
        catch (java.rmi.RemoteException e) { }
      }
  }
}


public void drawRect(Color color, int x, int y, int width, int height,
                     boolean bcast, boolean remotely_invoced) {
  Graphics g = offScreen.getGraphics();
  if (mouseDown && (!remotely_invoced))
    g.setXORMode (XorAlternateColor);
  else
    g.setPaintMode();
  drawRect(g, color, x, y, width, height);

  if (bcast) {
    int size = remoteBoard.size();
    int me    = remoteBoard.getId();
    for (int i=0; i<size; i++)
      if (i != me) {
        RemoteBoard other = (RemoteBoard) remoteBoard.getRemoteObject(i);
        try { other.drawRect(color, x, y, width, height); }
        catch (java.rmi.RemoteException e) { }
      }
  }
}


public void fillRect(Color color, int x, int y, int width, int height,
                     boolean bcast, boolean remotely_invoced) {
  Graphics g = offScreen.getGraphics();
  if (mouseDown && (!remotely_invoced))
    g.setXORMode (XorAlternateColor);
  else
    g.setPaintMode();
  fillRect(g, color, x, y, width, height);

  if (bcast) {
    int size = remoteBoard.size();
    int me    = remoteBoard.getId();
    for (int i=0; i<size; i++)
      if (i != me) {
        RemoteBoard other = (RemoteBoard) remoteBoard.getRemoteObject(i);
```

```
        try { other.fillRect(color, x, y, width, height); }
        catch (java.rmi.RemoteException e) { }
      }
  }
}

public void drawOval(Color color, int x, int y, int width, int height,
                     boolean bcast, boolean remotely_invoced) {
  Graphics g = offScreen.getGraphics();
  if (mouseDown && (!remotely_invoced))
    g.setXORMode (XorAlternateColor);
  else
    g.setPaintMode();
  drawOval(g, color, x, y, width, height);

  if (bcast) {
    int size = remoteBoard.size();
    int me   = remoteBoard.getId();
    for (int i=0; i<size; i++)
      if (i != me) {
        RemoteBoard other = (RemoteBoard) remoteBoard.getRemoteObject(i);
        try { other.drawOval(color, x, y, width, height); }
        catch (java.rmi.RemoteException e) { }
      }
  }
}

public void fillOval(Color color, int x, int y, int width, int height,
                     boolean bcast, boolean remotely_invoced) {
  Graphics g = offScreen.getGraphics();
  if (mouseDown && (!remotely_invoced))
    g.setXORMode (XorAlternateColor);
  else
    g.setPaintMode();
  fillOval(g, color, x, y, width, height);

  if (bcast) {
    int size = remoteBoard.size();
    int me   = remoteBoard.getId();
    for (int i=0; i<size; i++)
      if (i != me) {
        RemoteBoard other = (RemoteBoard) remoteBoard.getRemoteObject(i);
        try { other.fillOval(color, x, y, width, height); }
        catch (java.rmi.RemoteException e) { }
      }
  }
}

public void showKey(Color color, int x, int y, char ch,
                    boolean bcast) {
  Graphics g = offScreen.getGraphics();
  showKey(g, color, x, y, ch);

  if (bcast) {
```

```
    int size = remoteBoard.size();
    int me   = remoteBoard.getId();
    for (int i=0; i<size; i++)
      if (i != me) {
        RemoteBoard other = (RemoteBoard) remoteBoard.getRemoteObject(i);
        try { other.showKey(color, x, y, ch); }
        catch (java.rmi.RemoteException e) { }
      }
  }
}


public void clearBoard(boolean bcast) {
  Graphics g = offScreen.getGraphics();
  clearBoard(g);

  if (bcast) {
    int size = remoteBoard.size();
    int me   = remoteBoard.getId();
    for (int i=0; i<size; i++)
      if (i != me) {
        RemoteBoard other = (RemoteBoard) remoteBoard.getRemoteObject(i);
        try { other.clearBoard(); }
        catch (java.rmi.RemoteException e) { }
      }
  }
}


// ------------------------------------
// Local drawing routines
// ------------------------------------

void drawLine(Graphics g, Color color, int startX, int startY, int endX, int endY) {
  g.setColor(color);
  g.drawLine(startX, startY, endX, endY);
}


void drawRect(Graphics g, Color color, int x, int y, int width, int height) {
  g.setColor(color);
  g.drawRect(x, y, width, height);
}


void fillRect(Graphics g, Color color, int x, int y, int width, int height) {
  g.setColor(color);
  g.fillRect(x, y, width, height);
}


void drawOval(Graphics g, Color color, int x, int y, int width, int height) {
  g.setColor(color);
  g.drawOval(x, y, width, height);
}
```

```
  void fillOval(Graphics g, Color color, int x, int y, int width, int height) {
    g.setColor(color);
    g.fillOval(x, y, width, height);
  }

  void showKey(Graphics g, Color color, int x, int y, char ch) {
    g.setColor(color);
    g.drawString(String.valueOf(ch), x, y);
  }

  void clearBoard(Graphics g) {
    g.setColor(this.getBackground());
    g.fillRect(0, 0, imageWidth, imageHeight);
    g.setColor(currColor);
  }
}
```