A JAVA TOOL FOR COLLABORATIVE EDITING OVER THE INTERNET

Rhonda Chambers
Dean Crockett
Greg Griffing
Jehan-François Pâris

Department of Computer Science
University of Houston
Houston, TX 77204-3475

ABSTRACT

We present a distributed Java application that allows several people, possibly at different locations, to edit the same file at the same time while exchanging comments on the changes they are making to the file. While still being a prototype, our application provides a vivid demonstration of the benefits of collaborative editing.

1. INTRODUCTION

Computing networks have changed the way many of us work because they allow us to access remote data all over the world and provide a fast convenient way to communicate with people living and working far away from us. It is not unusual today to find people located in different states or different countries working together on the same report. Whoever has been involved in such an endeavor has noticed the lack of good software tools for collaborative editing. As a result, too many co-authors settle for a routine where they take turns editing their joint document while exchanging comments through telephone or electronic mail. What is lost in the process is the synergy occurring when the co-authors are in the same room, see the same version of the document and edit it together.

A good tool for collaborative editing should allow this synergy to occur. Hence it should allow several people, possibly at different locations, to edit the same file concurrently and exchange comments on the changes they are bringing to the file. To be truly effective, the tool should also be easy to port to a new computing environment and provide an easy to use graphical interface. Up to very recently, these two last requirements were contradictory because applications with graphical interfaces were much more machine dependent than their text-based counterparts. (Even though X Windows has become the de facto standard for UNIX platforms, they are not popular within the personal computer community.) This situation has dramatically changed with the recent introduction of the Java programming language. Java was specifically designed to be portable over a wide range of platforms and operating systems. It includes an extensive library of methods that allow the rapid design of graphical user interfaces.

There was another reason for selecting Java: the language contains less widely known features for interprocess synchronization, among which the first widely available implementation of the monitor concept proposed by Brinch Hansen (1973) and Hoare (1974) more than twenty years ago.

The remainder of our paper is organized as follows. Section 2 discusses some of the relevant features of the Java programming language. Section 3 presents our user interface. Sections 4 and 5 respectively discuss the client and server parts of our applications. Finally, Section 6 has our conclusions.

2. JAVA

As we expect most readers to be already familiar with the most salient features of Java, we will only mention them here before discussing in more detail some of the interprocess synchronization features of the language (Flannagan, 1996, Walnum, 1996). It suffices thus to say that Java is an interpreted object-oriented language designed to run on many different platforms and allowing end-users to eliminate most, if not all, of the security risks they might incur while running on their own machine Java applets downloaded from the Internet.

Since the early days of operating systems development, systems designers have found it difficult to synchronize concurrent accesses to a shared resource, let it be a shared variable, a shared memory segment or a shared file. While many solutions to the problem have been proposed, among which events, semaphores, conditional critical regions path expressions and many other constructs (Silberschatz and Gavin, 1994), the general consensus is that Brinch Hansen and Hoare's monitors provide the best solution to the problem.

Monitors control access to shared variables by requiring all access to any shared variable to be performed through one of the *procedures* of the monitor attached to that particular instance of the shared variable. Since these procedures can only be executed one at a time, mutual exclusion is automatically achieved. Monitors also provide a *wait* primitive allowing one monitor procedure to wait until a given condition is satisfied and a *signal* primitive allowing another procedure to signal to the first procedure that the condition on which it was waiting is now satisfied.

A major advantage of this approach is the fact that the processes accessing the shared variable do not need to do anything special to synchronize their access to the shared variable because everything has been taken care of within the monitor procedures. Moreover, unsynchronized accesses to a shared variable are guaranteed not to happen under any circumstances.

Central to Java is the concept of *class*. Each Java class describes a data object that is a set of *methods* that can be used to create new instances of the data object and manipulate them. The Java equivalent of a monitor is a Java class whose access methods have been declared **`synchronized.`** The first thread invoking a **`synchronized`** method on an instance of a data object will lock that object. Any other thread invoking a **`synchronized`** method on the same instance of the object will block until the lock is removed. Java also provides **`wait()`**and **`notify()`** primitives but without *named conditions*: whenever a **`synchronized`** access method does a **`wait()`**, it cannot specify the condition it wants to wait on. This would be a serious limitation if Java did not provide a **`notifyAll()`** primitive waking up all waiting threads in a class instance. The effect of a pair of **`wait(Condition_1)`**being waken up by a **`notify(Condition_1)`** can thus be achieved by replacing the **`notify(Condition_1)`** by a **`notifyAll()`**and the. **`wait(Condition_1)`** by a:

```
while(Boolean_Condition_1)wait()
```

where **`Boolean_Condition_1`** is a Boolean expression that is true if and only if the condition is satisfied. The **`notifyAll()`** will wake up *all* waiting threads and the **`while()`**before the **`wait()`** allows the waiting threads to decide themselves which one should be allowed to continue while all others should return to their waiting state.

3. THE USER INTERFACE

The user interface was designed with one major objective in mind, namely the ease of use in editing and communicating with a user's collaborators. To this end, we broke its functionality into four primary areas:

1) the user access module,
2) the non-editable text display module,
3) the text editing window module, and
4) the chat facility.

3.1 User Access

Even though this program is primarily a proof-of-concept application, some sort of entry-point is still necessary. In a full-fledged application of this type, a user would be expected to choose from among several existing collaborative efforts, provide a password as proof of eligibility to join the topic, be able to upload and download files on which to work, and start new topics as necessary. For this effort, however, all that was necessary was that a user provide a user alias upon entry to the only topic available. The system takes that user alias and associates it with a unique color (at the moment hard-coded into the program instead of chosen by the user) and starts an editing session for that user. When a new user joins a session, all other users are notified and the new user's alias appears in the user alias display box in the color associated with that user's activities.

3.2 The Text Display Window

The main feature of this application is the text display window. The document upon which the users are collaborating is displayed in a non-editable, scrollable window that takes up most of the display area. All users see the same text, but have independent control over which area of the text is displayed at any given time. Every user has several options available to them regarding the text displayed in this window. A user can mark, or highlight, a section of text, and that text will change to match that user's color on *all* users' screens. In this way users can indicate exactly which text they wish to point out for discussion. In addition to marking a section of text in this manner, the user can lock that block of text so the other users cannot access it, and engage an editing window in which to make changes to this text. When one user is editing a section of text, that text changes to the color of the user doing the editing in all the users' windows, and also becomes bold so that all other users can tell the text is being edited as opposed to merely being marked. When the user who is editing the text indicates he is finished by pressing a "done" button, the text that was highlighted on every users' screen is replaced by the new text in real-time so that all users can see any changes made as soon as the work is done.

3.3 The Text Editing Window

The text editing window is an editable window that holds only that text being edited, and is only displayed when a block of text is actively being edited. When each editing session is done, the window is hidden until it is needed again.

3.4 The Chat Facility

Since one of the primary purposes of this project was to create a real-time document collaboration tool, some sort of communication facility was required. We considered relying exclusively on Netscape's Internet Phone technology available for Netscape 3.0, but we decided that this limited the audience for this application, so we built in a standard chat facility with some additional functionality customized to our application. The uniqueness of this chat function is that it is tied into the color-coding of each user on the system, and displays all chat messages from them in the appropriate color, as well as providing their user alias. When a user wishes to send a chat message to the other users, he enters the text in a chat entry field. When the enter key is pressed, the text is sent to all the other users and appears in

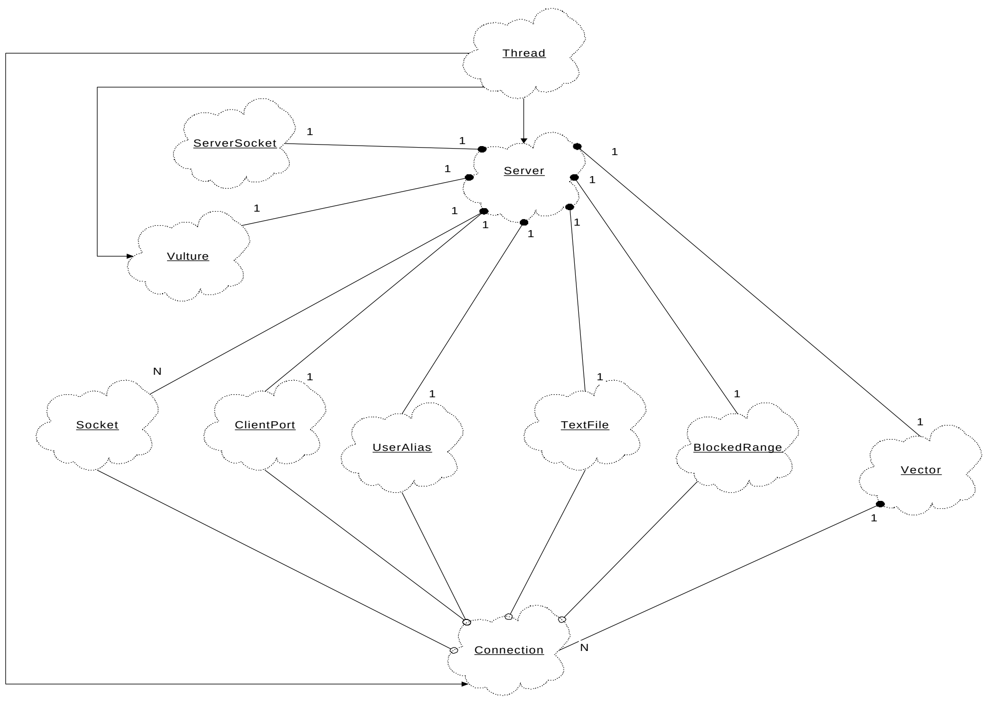


Figure 5.1: Server Architecture

alias and in the user's color. This color matches that to which the text is changed when that user marks or edits a section of the text.

4. THE CLIENT

The client is actually a combination of the GUI and a messaging facility which handles the dialog of the collaboration server.

4.1 The GUI Event Mechanism

The GUI uses an event mechanism to handle mouse clicks, key-strokes, etc. This event mechanism is not synchronized with the message stream from the server which poses a problem. The client uses a thread to asychronously intercept server messages and inserts an equivalent message in the GUI's event queue. This permits server messages to interleave with conventional GUI events in a non-intrusive way. Originally we tried directly manipulating the GUI components via GUI method calls in response to server messages. This strategy caused the GUI to become unstable since it might be processing an event while simultaneously being modified by the server messaging code. After careful study of the GUI event mechanism we discovered a means of introducing application specific events to the GUI's event queue.

4.2 The DataObject Class

The client process uses an instance of the DataObject class to parse server generated messages. As the server also uses this same class we ensure an agreement between the two processes regarding message format and content. The Client class exports a set of methods which the GUI classes used to converse with the server. This approach centralizes all server communication within the Client class.

5. THE SERVER

The server is responsible for handling all communication with a client. The following subsections describe how the server handles this communication. Topics to be discussed include the following:

1) the server architecture,
2) the management of client connections,
3) shared data and client communication, and
4) synchronization issues.

5.1 Server Architecture

The server architecture is illustrated using the Booch methodology for object-oriented design and analysis (Booch 1994). Figure 5.1 contains the class diagram depicting the relationships and inheritance between the classes used to implement the server. For an explanation of the Booch notation refer to the Appendix.

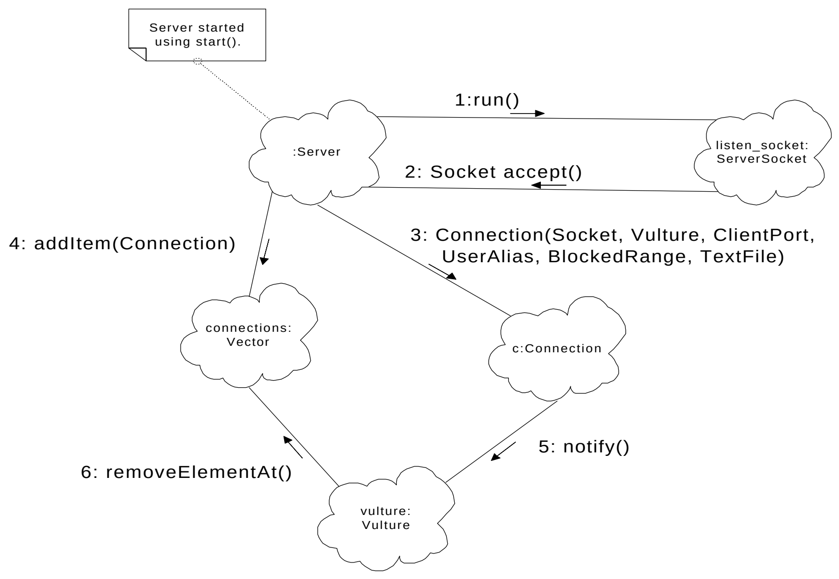

Server started using start().
1:run()
:Server
listen_socket: ServerSocket
2: Socket accept()
4: addItem(Connection)
3: Connection(Socket, Vulture, ClientPort, UserAlias, BlockedRange, TextFile)
connections: Vector
c:Connection
5: notify()
6: removeElementAt()
vulture: Vulture

Figure 5.2: Object Diagram of a Client Connection

5.2 Managing Client Connections

The server must have the ability to handle more than a single client. The server provides this ability using a Java class called the *ServerSocket*. The ServerSocket only listens on a specified port for clients to connect. When a client connects, the ServerSocket creates a new thread with its own socket for that client to communicate through. All subsequent interactions with that client are handled by the server thread created for that client. The object diagram in Fig. 5.2 depicts the role of the ServerSocket class and the addition/removal of clients.

The numbers preceding the method names indicate the sequence of events that repeatedly occur within the server. Each client that connects to the server is stored in a vector called connections. A vector is a built in Java class that implements an array which grows in size as necessary. Figure 5.3 shows a diagrammatic view of the vector of connections. Each connection object stored in the vector has a separate thread of execution. When the client disconnects it is removed from the vector of connections using the Vulture class. This is depicted in steps 5 and 6 above.

1	**2**	**. . . . .**	***n***
Connection 1 separate thread of execution	Connection 2 separate thread of execution	. . . . .	Connection *n* separate thread of execution

Figure 5.3: Vector of connections

Shared Data and Client Communication. Shared data and client communication is achieved using the classes in the class diagrams represented in Fig. 5.4.

Socket. Since Java was designed from the ground up with distributed Internet applications in mind, it has several classes that make communication through sockets a very simple process. The *Socket* class is a built in Java class that implements a socket for interprocess communication. The class uses two other Java classes that allow reading from the socket and writing to the socket as if one were reading and writing to a file. These classes are *DataInputStream* and *PrintStream*.

ClientPort. The *ClientPort* class is a class used to write information from a client to all other clients involved in collaborating on the shared document. It is created to take advantage of the event driven Java socket class. Whenever a client joins the document sharing session, the client's PrintStream (i.e.-the location where it receives data from the server) is stored in a vector that is part of ClientPort. Any time a message from one client needs to be broadcast to all other clients, ClientPort cycles through it's vector of *PrintStreams* writing out the message. The advantage to this approach is that it eliminates the need for the server to store and manage messages between clients.

TextFile. The *TextFile* class is used to encapsulate all operations performed on the shared document. These operations include opening the file and preparing it for editing, updating it's contents and saving it to disk. The file is opened and loaded into memory as soon as the server is started. All updates to the shared document occur at the request of a client and the file is saved to disk when all clients exit.

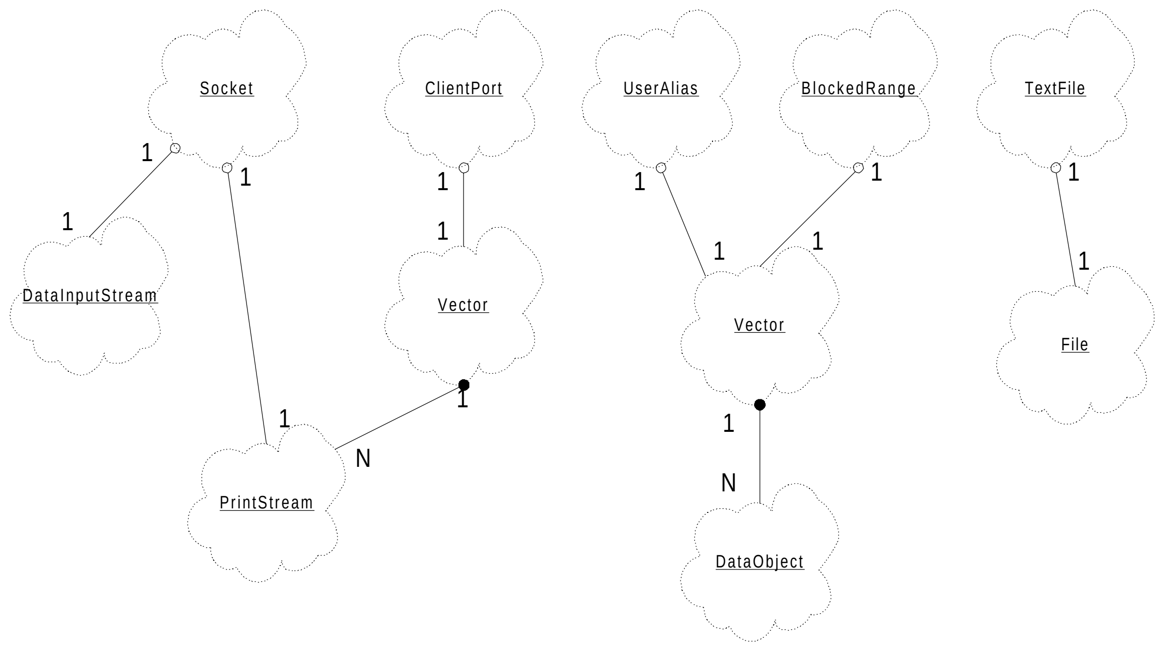


Figure 5.4: Shared Data Class Diagram

BlockedRange. The *BlockedRange* class works hand in hand with the TextFile class in managing the shared document. The BlockedRange class uses a vector to store each client's begin and end location of the range of text they are editing. Each time an update takes place, the range blocked by the client requesting the update is retrieved and used in updating the file. An important aspect to consider during the update process is that when a client requested update takes place, the other client's blocked range values are no longer valid. To handle this aspect, the TextFile class updates each of the remaining clients blocked range.

UserAlias. The *UserAlias* stores the user name of each client that joins the document sharing session. This information is important so that all clients know who is participating in the session.

Shared Data Script. Figure 5.5 displays a script diagram showing the interactions of the shared data and client communication classes. It demonstrates the passing of messages between the classes in relative order of when they occur.

5.3 Synchronization

The main issue faced in implementing the server was synchronizing access to the shared data. As depicted earlier, each client communication (a Connection object) has its own separate thread of execution. This implies that a connection object or many of the connection objects may want access to the shared data (i.e.- request a file update or request a range block, and so forth) at the same time. Synchronized methods are used to ensure that only one connection is changing the shared data at any given time. Java provides this ability through the synchronized keyword. When that keyword is placed before a class method, it indicates to Java that the method modifies the internal state of the class. Before Java runs the method, it obtains a lock on the class. This ensures that no other threads are modifying the class simultaneously.

6. CONCLUSIONS

We have presented a distributed Java application that allows several people, possibly at different locations, to edit the same file at the same time while exchanging comments on the changes they are bringing to the file. Even though it is still a prototype, it provides a vivid demonstration of the benefits of collaborative editing.

We would also like to mention that this was the first Java application for the authors. Despite a few initial misgivings about the lack of some customary programming constructs such as C pointers, we found Java ideally suited for the rapid development of portable distributed applications.

REFERENCES

Booch, G., 1994, *Object-Oriented Analysis and Design*, Second edition, Addison-Wesley,1994.

Brinch Hansen, P., 1973, *Operating Systems Principles*, Prentice-Hall.

Flannagan, D., 1996, *Java in a Nutshell*, First edition, O'Reilly and Associates.

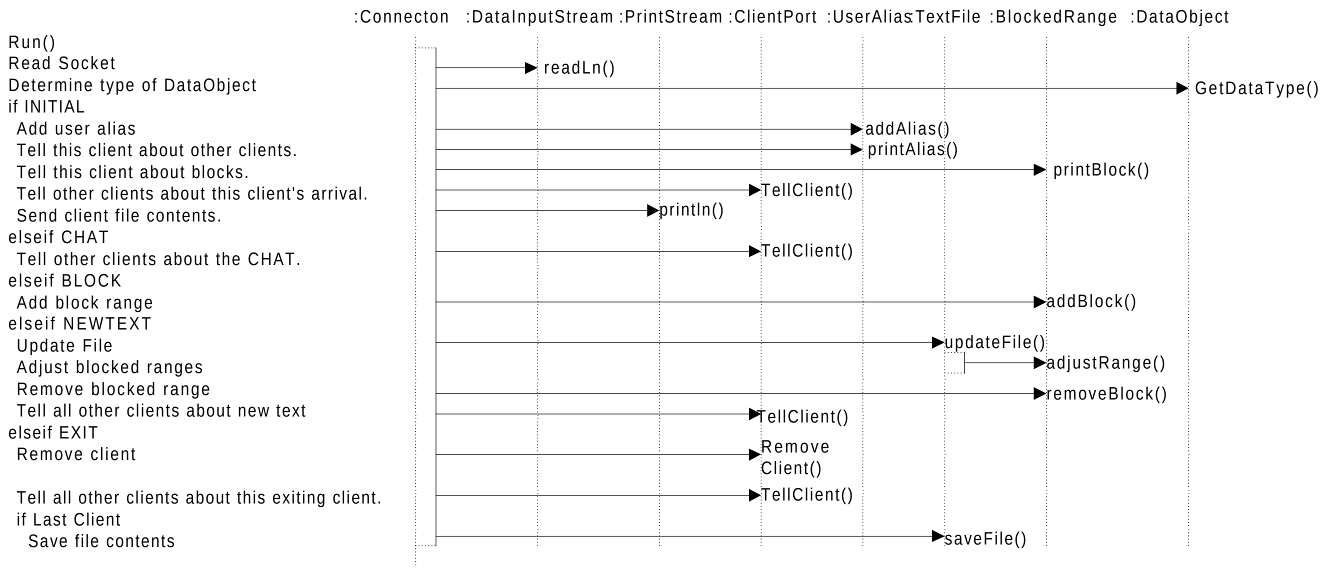


Figure 5.5: Interaction Diagram of Shared Data and Client Communication Classes

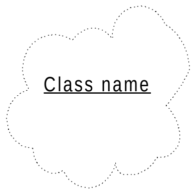
Hoare, C. A. R., 1974, "Monitors: An Operating Systems Structuring Concept," *Communications of the ACM*, Vol. 17, No. 10, pp. 549-557. Erratum in *Communications of the ACM*, Vol. 18, No. 2, pp. 95.

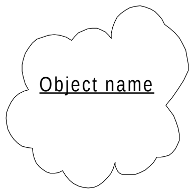
Silberschatz. A. and Gavin, P., 1994, *Operating Systems Concepts*, 4th Edition**,** Addison-Wesley.

Walnum, C., 1996, *JAVA by Example*, Que Corporation.

APPENDIX:
THE BOOCH NOTATION

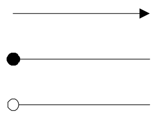
Class Icons





A	object name only
:C	object class only
A:C	object name and class

Class relationships



Notes