A Simple Mechanism for Sharing Variables
Across the Internet

by

Jake Harris

Submitted to the Department of Electrical Engineering and
Computer Science
in partial fulfillment of the requirements for the degrees of
Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1998

© Massachusetts Institute of Technology 1998. All rights reserved.
A Simple Mechanism for Sharing Variables Across the Internet

by

Jake Harris

Submitted to the Department of Electrical Engineering and Computer Science on May 8, 1998, in partial fulfillment of the requirements for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Computer Science and Engineering

Abstract

This thesis describes a lightweight yet powerful approach for writing distributed applications using shared variables. Our system (called SHAREHOLDER) is inspired by the flexible and intuitive model of information access in the World Wide Web. The distributed applications targeted by our approach all share a weak consistency model and loose transaction semantics, similar to a user’s model of accessing email, bulletin boards, chat rooms, etc. on the Internet. The SHAREHOLDER infrastructure has several advantages. Its highly object-oriented view of shared variables simplifies their initialization and configuration. A shared variable’s distribution mechanism is specified through a separate configuration object, and the programmer does not need to implement extra code to support the sharing mechanism. Configuration objects can be initialized dynamically, thus providing tremendous flexibility in dynamic control of distribution at runtime. Finally, the programmer can treat shared variables and local variables interchangeably, thus simplifying conversion of a serial application into a distributed application.

Thesis Supervisor: Vivek Sarkar
Title: Visiting Associate Professor
Acknowledgments

This thesis would not be possible without the help of many along the way. Many thanks to Professor Sarkar for suggesting the original idea and offering guidance and wisdom as it developed into something more substantial. To my parents for providing both emotional and sometimes financial support to keep the project on track. To my sister for being cool just the way she is. To all my other friends who probably have had their share of hearing me gripe about working on this project. To Tosci’s and Newbury Comics for the coffee and music that have kept me awake at my computer. And of course to MIT for providing the academic environment that makes all of this possible.
Contents

1 Introduction 8

2 The SHAREHOLDER Programming model 11
   2.1 Shared Variables ........................................ 11
   2.2 Configurations ............................................ 13
   2.3 The Proxy .................................................. 13

3 Examples 16
   3.1 Extracting information from a home page .................. 16
   3.2 A Simple Chat system ....................................... 20
   3.3 A Voting System ............................................ 21

4 Details of the SHAREHOLDER System 26
   4.1 The Application ............................................ 26
      4.1.1 The Reload Operation ................................ 27
      4.1.2 The Update Operation ................................ 28
      4.1.3 Blocking and Nonblocking Invocations ............... 28
      4.1.4 The Event Framework ................................ 29
   4.2 Configuration Objects ..................................... 30
   4.3 The Proxy .................................................. 32
      4.3.1 Communication Among Proxies ......................... 32
      4.3.2 The Proxy Implementation of SharedVar Methods ....... 34

5 Extensions 38
5.1 Changes to the Application ........................................... 38
5.2 Changes to the Configuration ......................................... 39
5.3 Changes to the Proxy .................................................. 41

6 Related Work ................................................................. 44

7 Conclusions ................................................................. 50
List of Figures

2-1 Different possible server connections for a proxy ............... 13
3-1 The StockPrinter example program ......................... 16
3-2 Pseudocode for StockQuote class ......................... 17
3-3 Pseudocode for StockQueryThread and StockPrinter classes .... 18
3-4 The SimpleMessage chat program ......................... 19
3-5 Pseudocode for SharedMessage class ................... 19
3-6 Pseudocode for SimpleMessage and MessageServer classes .... 21
3-7 The VoteClient application ........................ 22
3-8 Pseudocode for VoteRecord class ......................... 23
3-9 Pseudocode for VoteTally class ......................... 23
3-10 Pseudocode for VoteServer and VoteClient application ....... 24
4-1 An example of the update mechanism ......................... 27
4-2 The SharedVar Event Notification Framework ................. 30
4-3 The Proxy Communication Layers .......................... 35
4-4 The Proxy’s Reload Mechanism ............................ 36
4-5 The Proxy’s Update Mechanism ............................ 36
6-1 JavaSpaces Pseudocode for the Simple Chat Program ......... 46
6-2 JavaSpaces Pseudocode for the Voting Program ............... 47
6-3 JavaSpaces Pseudocode for the Voting Program, part 2 ........ 48
List of Tables

4.1 Communication mechanisms currently supported ............... 31
4.2 Potential future extensions to the communications mechanisms ... 31
4.3 Proxy messages received by the server. ......................... 33
4.4 Proxy messages received by the client program. ............. 33
Chapter 1

Introduction

The last few years have seen an explosive increase in the growth of the Internet and the number of networked computers worldwide. Despite the increasing connectivity of computers, distributed applications are still somewhat difficult for the typical programmer to create. This is largely because most distributed applications communicate through the most basic approach of message passing. Such message passing systems are efficient, but their lack of abstraction makes it difficult to write programs and to reuse code across multiple distributed applications. Two notable approaches that have more abstraction than message passing are remote method invocation (RMI) [13] and distributed shared memory (DSM) [10, 5, 2]. Unfortunately, the extra level of abstraction in these approaches is usually accompanied by extra runtime overhead and additional API’s (application programming interfaces) that impose an extra level of programming complexity on the programmer.

JavaSpaces [12] is a recent approach that builds upon past work on RMI and on Linda-style [4] concurrent systems. JavaSpaces allows the programmer to create and use multiple “virtual spaces”, each of which can be used as a persistent shared networked store by a heterogeneous collection of distributed applications. The heterogeneity is one of the key differences between JavaSpaces and DSM models; sharing in a DSM only occurs among multiple threads/processes that are homogeneous i.e., that belong to the same application. Other important characteristics of JavaSpaces include its strong typing, its support for template matching/lookup in a space,
its transaction mechanism that can support multi-space updates using a two-phase commit model. Though JavaSpaces can simplify the problem of writing transaction-based distributed applications, this simplification comes at the cost of a reasonably heavyweight mechanism. In addition, management of multiple spaces imposes an extra level of complexity in writing distributed applications. A comparison between our approach and JavaSpaces can be found later in chapter 6.

In this thesis, we address the problem of making the mechanism for using shared variables as lightweight and flexible as possible. The inspiration for our research is the World Wide Web, in which large volumes of information are shared very effectively on a daily basis with a weak consistency model and a loose transaction semantics that are understood intuitively rather than formally. For example, a set of web pages may be updated asynchronously on multiple servers and reloaded by multiple clients, without any requirement that the reload operations obey a strong consistency model such as sequential consistency [9]. Instead, the clients work with a very weak consistency model in which a reload operation is only guaranteed to return a "more recent" value than (or the same value as) a previous reload of the same web page. Similarly, multiple email messages sent from the same source may reach their destinations out of order; in fact, it is possible for an email message to fail and not reach its destination at all. Web users communicate daily via email, bulletin boards, chat rooms, etc. without depending on a strong transaction semantics or on a strong notion of atomic update of multiple resources. Instead, web users work with a very loose transaction semantics based on notifications of failed operations and/or acknowledgments of successful operations. The high reliability of email transmissions has made this a practical model for day-to-day collaboration.

Now, imagine support for a similar level of information access and sharing among computer programs! Our goal is to enable an application to harvest information from and interact with a variety of remote sources as simply and efficiently as possible. The programmer should not have to rewrite his code significantly to handle sharing and remote access, or hardwire the mechanics of sharing into his application code.

Our solution is to create a middleware infrastructure for shared variables on
the Internet that is extremely lightweight. This middleware infrastructure is called SHAREHOLDER, and has been implemented as a small library (approximately 3,000 lines of code) written in Java [8]. The SHAREHOLDER infrastructure is highly object oriented; the user creates a shared variable by declaring it to be of a type (class) that derives from a basic SharedVar type definition declared in the SHAREHOLDER library. All operations for sharing are inherited from this basic class and can even be overridden by the programmer for customized behavior. In addition, SHAREHOLDER builds on the reflection and object serialization mechanisms of Java in several places, and avoids burdening the programmer with additional interfaces to support reflection and serialization in the middleware.

The key advantages of the SHAREHOLDER infrastructure are as follows. It allows the programmer to use shared variables and local variables interchangeably in code, thus simplifying conversion of a serial application into a distributed application. The distribution information for a shared variable is stored in a configuration object for the variable. Configuration objects can be initialized at run-time thus allowing tremendous flexibility in determining the sharing of an application’s variables at runtime. For example, it is possible to run the same program with many different configurations. Configuration objects also promote modularity by decoupling distribution logic and application logic in writing a program. SHAREHOLDER configurations are versatile, and can allow shared variables to read data not only from specialized server programs but also from information sources as simple as remote home pages. Finally, the system allows communication through shared variables to be built upon a variety of communication protocols (e.g., sockets, datagrams, http, email).

The rest of the thesis is organized as follows. Chapter 2 outlines the SHAREHOLDER programming model from the user’s point of view. As illustration, chapter 3 discusses three example distributed applications that have been written using the SHAREHOLDER system. Chapter 4 contains details of the current design and implementation of the SHAREHOLDER, while chapter 5 outlines some extensions which could be made to the SHAREHOLDER design in the future. Chapter 6 discusses related work, and chapter 7 contains our conclusions.
Chapter 2

The ShareHolder Programming model

The following sections outline the SHAREHOLDER programming model from the user’s point of view. Section 2.1 describes how shared variables can be created. Section 2.2 describes how configurations can be associated with shared variables. Section 2.3 summarizes the user’s view of the proxy used in the SHAREHOLDER infrastructure.

2.1 Shared Variables

Shared variables are the basic building blocks of any distributed application in the SHAREHOLDER infrastructure. The user creates a shared variable by declaring it to be of a type (class) that derives from the SharedVar type declared in the SHAREHOLDER library. The primary benefit of this approach is that type checking and inheritance properties (constructors, etc.) can be used to ensure that shared variables are properly initialized and only shared variables are shared. In addition, class inheritance avoids the need for adding new keywords/modifiers to the base programming language, and hence avoids the need for modifying existing compilers.

Each shared variable inherits several useful methods from the SharedVar class. The reload() method can be called by the program to reload the contents of the shared variable from its home location. As discussed in section 2.2, this home location
can refer to a variety of possible sources, depending on the configuration. For example, it may be a location in local memory, or on a remote web page, or on a remote server that acts as a repository for shared variables. The update() method is called to write the new local value of the shared variable to the home location and subsequently to any other applications that are sharing the variable (we refer to these applications as subscribers of the shared variable). These updates can occur asynchronously in general; a programmer should be aware that the value of a shared variable can change at any time after an update() method is performed by any subscriber. Hence, shared variables must always be declared with the volatile modifier in Java, so as to ensure that they are not incorrectly optimized by a Java compiler [8]. The update() method is also declared as synchronized to ensure that the update value does not get partially corrupted by some other local thread in the application while the update() method is running.

The reload() and update() methods can be called in either blocking or non-blocking invocations. Using a blocking invocation allows the programmer to halt execution in a thread until the latest value has been reloaded or an update has been received and acknowledged by the server. Loss of connectivity may mean that calls to these methods may never return. As an alternative, the programmer can make a non-blocking invocation which waits for at most a limited period of time. Such non-blocking invocations will never hang indefinitely, but they do not guarantee that the operation has completed successfully when execution resumes.

With these two basic operations (reload and update), the programmer has all he needs to read and write to a shared variable. As a convenience, SHAREHOLDER also provides an event-based mechanism for reporting when changes to a shared variable occur. Similar to the event handling mechanism in Java JDK 1.1, this framework allows any type to register as a listener to shared variable events (described by the SharedVarEvent class). When such an event is thrown (e.g., after a shared variable's local value is modified by a local reload operation or a remote update operation), a corresponding helper method in the listener is called. This mechanism allows applications to be informed of changes to variables without having to continuously
Figure 2-1: Different possible server connections for a proxy

reload variables to see if they have changed. This is especially useful for applications that display shared variables in graphical interfaces that only need to be updated when a change occurs.

2.2 Configurations

Configurations are the means by which a programmer specifies the distribution of shared variables in a program. Unlike shared variable declarations, a configuration is associated with an individual object rather than with an entire class. Multiple objects may have the same configuration. Each shared variable object stores its configuration in an ObjectConfig field after initialization. This configuration can be viewed by other classes, but variables are not allowed to modify their configuration after initialization.

2.3 The Proxy

A configuration contains all the necessary information about how a variable is shared. This information describes the home location for a shared variable. This home location is the single central location of a shared variable (e.g., whether it is a web page, an entry stored at a remote server, or strictly local). Local instances of the variable in applications can be thought of as locally cached versions of the home
location. In the case of an object whose internal fields are loaded from remote web pages, the configuration includes the URLs of all the remote pages. In the instance where an object is stored on a remote server, the configuration consists of a special URL with a "path name" to identify the variable on the remote server. A degenerate configuration for an object can also state that the variable is "unshared", in which case only a local copy is maintained of the variable.

Currently, this approach requires the programmer to create the configuration for an object and then pass it as a parameter to the constructor for the class. However, a future goal of the system allows variables to retrieve their associated configuration from an external configuration file at runtime. The use of an external configuration file would decouple the sharing behavior at runtime from the program text, allowing the same application to be run with different networked behaviors merely by passing it a different configuration file at startup.

The Proxy is the subsystem that handles all of the details of sharing automatically for the programmer. The proxy is executed as a separate thread from the application and server nodes. It contains library routines that implement the reload() and update() method calls invoked on shared variables. The proxy also contains code for communicating with other proxies and for maintaining a central repository server for shared variables. The use of a proxy enables a shared variable to be initialized from a variety of sources, as illustrated in figure 2-1. In general, we see that a client application's proxy can harvest information from such read only sources as remote web pages and push media, as well as have full interaction with another proxy acting as a server.

Since the proxy runs in a separate thread, a Proxy object only needs to be created once at the startup of the application. The application should then pass a reference to this proxy to all of its shared variables' constructors. This is necessary because several of the inherited methods from SharedVar use the proxy to function. In turn, the helper methods of the proxy use a shared object's configuration to work correctly. The proxy thread runs until the main program thread terminates. In the case where we want to implement servers which use the SHAREHOLDER system, we want to keep
the server application from terminating. This is efficiently accomplished through joining on the Proxy’s server thread\(^1\). To enable such joins on the Proxy, the programmer is provided with a `joinServer()` method.

\(^1\)In the context of threads in Java, joining means to suspend execution of your thread until the thread you’re joining on has terminated. There are also some other versions of this invocation which allow the programmer to specify a maximum time period to wait for the other thread to terminate, allowing the execution of the calling thread to continue whichever comes first.
Chapter 3

Examples

Perhaps the best introduction to the simplicity and flexibility of this framework is through examples. Sections 3.1, 3.2, and 3.3 discuss three simple example distributed applications that have been written using the SHAREHOLDER system.

3.1 Extracting information from a home page

The first example application shows the simplest usage of the SHAREHOLDER framework. This StockPrinter application is a program that periodically reloads stock quote information from remote web pages, and updates a display as shown in figure 3-1. The shared variable used in this application is quotes, an array of StockQuote objects. As shown in figure 3-2, the StockQuote class extends the

Figure 3-1: The StockPrinter example program
import SharedVar.*;

public class StockQuote extends SharedVar {
    private String name; private float quote;
    public StockQuote( ObjectConfig objConf, Proxy proxy, String name) {
        super(objConf, proxy);
        this.name = name; this.quote = -1;
    }
    public String getName() { return name; }
    public float getQuote() { return quote; }
}

Figure 3-2: Pseudocode for StockQuote class

SharedVar class (all program text specific to SHAREHOLDER is shown in italics in figure 3-2 and in all later figures that contain code fragments).

Figure 3-3 contains the definitions of the StockQueryThread and StockPrinter classes, which make up the rest of the application. The StockQueryThread periodically reloads a list of stocks (represented by an array of StockQuote objects), and then displays the quote values. The only SHAREHOLDER primitive used in this class are the periodic reload() it makes. Most of the code which uses shared variables would be found in auxiliary classes like this which use shared variables rather than the class definitions for the variables themselves.

The StockPrinter class initializes and executes the program. The creation of a proxy and configuration objects in this class illustrates such objects are created at startup and provided to the constructors for the shared variables. The configuration for each element of StockQuotes identifies the home page from which the information is to be retrieved. For simplicity, this mechanism currently looks for special HTML tags in the homepage that look something like <vartag name=AAPL value=26.3>. This is similar to the convention used by the Java applet param tag. The configuration for a shared variable which looks for this vartag specifies this target appending an identifier (e.g., AAPL) to the end of the URL for the homepage. The proxy then uses this locator to search for a vartag with a matching name to get the value for the field. This mechanism for extracting information is simple and effective. Furthermore, such tags are not displayed by browsers since they are nonstandard HTML; this allows designers to embed information for use by shared applications without having to
public class StockQueryThread extends Thread {
    private volatile StockQuote quotes[];
    private int sleepLength;
    private StockDisplay display;

    public StockQueryThread(StockQuote[] quotes, int sleepLength) {
        this.quotes = quotes;
        this.sleepLength = sleepLength;
        // display creation code...
    }

    public void run() {
        while (true) {
            StringBuffer quoteText = new StringBuffer();
            for (int i=0; i < quotes.length; i++) {
                quotes[i].reload();
                quoteText.append(quotes[i].getName()+": " + quotes[i].getQuote()+"\n";
            }
            display.setText(quoteText.toString());
            sleep(sleepLength);
        }
    }
}

class StockPrinter {
    public static void main(String[] argv) {
        volatile StockQuote quotes[] = new StockQuote[argv.length];
        Proxy proxy = new Proxy();
        for (int i=0; i < quotes.length; i++) {
            Config nameConfig = new Config("local", Config.READ);
            Config valueConfig =
                new Config("http://www.mit.edu/~harrisj/stock-test.html"+argv[i], Config.READ);
            ObjectConfig objConf = new ObjectConfig();
            objConf.setFieldConfig("name", nameConfig);
            objConf.setFieldConfig("quote", valueConfig);
            quotes[i] = new StockQuote(objConf, proxy, argv[i]);
        }
        StockQueryThread sqt = new StockQueryThread(quotes, 6000);
        sqt.start();
    }
}

Figure 3-3: Pseudocode for StockQueryThread and StockPrinter classes
public class SharedMessage extends SharedVar implements SharedVarListener {
    private String message;
    transient private ChatFrame cf;

    public SharedMessage(ObjectConfig config, Proxy proxy) {
        super(config, proxy);
        message = null;
    }

    public void postMessage(String message) {
        this.message = message;
        update();
        System.out.println("Message posted");
    }

    public String getMessage() { return this.message; }
    public void reloadMessage() { reload(); }

    public void sharedVarUpdated(SharedVarEvent evt) {
        if ((evt.getSource() == this) && (cf != null))
            cf.append(message);
    }
}

Figure 3-5: Pseudocode for SharedMessage class

modify the look of a page. It should also be possible to add other ways for extracting
text from homepages, but this basic implementation should be sufficient for many uses.

Figure 3-4: The SimpleMessage chat program
3.2 A Simple Chat system

The *SimpleMessage* program is a simple application for allowing multiple users to chat over the Internet. The chat design allows multiple clients to join in or leave a chat hosted by a central server at any time, logging the chat as shown in figure 3-4. Like an electronic bulletin board, this chat program does not guarantee that messages are recorded by clients in the chronological order they were received at the server or even that a given client eventually sees all of the messages sent by other clients in a chat.

The *SharedMessage* class shown in figure 3-5 is the type for the central shared variable used in the communication. It contains a String message, and a reference to a graphical display (which is declared as *transient* [8] to indicate that it should not be shared over the network). As in the previous example, the `reload()` method is used to refresh the value of a shared variable. In addition, the call to `update()` posts messages to the server and shares it with other clients. Note that this class implements the *SharedVarListener* interface *i.e.*, it uses the *SharedVarEvent* mechanism to detect when the value of the shared variable has changed.

Figure 3-6 contains the pseudocode for the *SimpleMessage* and *MessageServer* classes. The *MessageServer* acts a central repository for the *SharedMessage* variable. This means that it is responsible for updating a locally stored copy of the variable and communicating this change to all other participating clients. However, as can be seen in figure 3-6, this functionality is completely handled within the *Proxy* abstraction. The client program merely has to create a *Proxy* server thread and then join on it.

The *SimpleMessage* class is the program’s top level. This class creates the proxy and sets the configuration for the shared variable. Unlike the previous example, the configuration here is set for the entire object (not individual fields of the object), and the protocol used for communication is different from http (e.g., sockets, datagrams, or even email may be used as a mechanisms by the *Proxy* for this application).
public class SimpleMessage {
    public static void main(String args[]) {
        volatile SharedMessage sm;

        // code for reading in source and ident from the command line...
        Proxy proxy = new Proxy();
        Config conf = new Config(source, Config.READ—Config.WRITE);
        ObjectConfig objConf = new ObjectConfig(conf);

        sm = new SharedMessage( objConf, proxy );
        // creation of the ChatFrame display...
    }
}

public class MessageServer {
    public static void main(String args[]) {
        Proxy proxy = new Proxy();
        System.out.println("Ready for connections!");
        proxy.joinServer();
    }
}

Figure 3-6: Pseudocode for SimpleMessage and MessageServer classes

3.3 A Voting System

This Voting application is another example of how the SHAREHOLDER model can be used to create complex networked programs without much effort. This simple voting system allows multiple networked clients to select votes from a list of choices provided by the server. Votes are not anonymous; instead, each client is provided with tallies of the votes and a list of the clients that voted for each choice. These tallies are updated and propagated by the server whenever a vote is cast by any client. Each client is allowed only one vote, but they can change their vote during the process. A sample display is shown in figure 3-7. Furthermore, client applications are able to enter and exit at any point during the voting process and should still be able to cast votes and view results. One interesting difference from the previous example is that this application uses two different shared variable types (VoteRecord or VoteTally) — one for each direction of communication between the client and server.

The VoteRecord class (shown in figure 3-8) contains the basic information associated with a vote (i.e., the voter’s name and choice). A shared variable of this type is used by the client application to send a vote to the server by calling the variable’s
The VoteClient application uses the `update()` method.

The VoteTally class serves as a repository for the voting results and statistics. As a shared variable, it is used for communicating voting results from the server to clients. This tally is sent to all subscribing clients whenever the server receives a new vote. Though this might appear to be excessive communication, this data structure is not too large to transmit and allows any client to join in the middle of the voting process.

The VoteServer is a simple top level program that reads in a list of choices from a file and starts a server thread that listens for and handles votes from clients. Again, the Proxy keeps tracks of the subscribers and broadcasts updates to the VoteTally object. All the server does is to detect when new votes are received and update its tally. The server uses the same SharedVarEvent mechanism as previous examples to listen for updates to its VoteRecord variable. It then uses this vote information to update its tally and thus propagate it outwards to the clients.

The final component of the system (the VoteClient class) is a graphical client application that displays voting information and allows the user to cast or change his vote. This application monitors changes to its shared VoteTally object with the event listening mechanism; when changes are detected, it updates its display accordingly. The client sends votes to the server through the update mechanism in the VoteRecords.makeVote() method. As an effect of the update mechanism, all
import SharedVar.*;

public synchronized class VoteRecord extends SharedVar {
    String voterName; int voteChoice;

    public VoteRecord(ObjectConfig objConf, Proxy proxy) {
        super(objConf, proxy);
        this.voterName = null; this.voteChoice = -1;
    }

    public void makeVote(String voterName, int voteChoice) {
        this.voterName = voterName; this.voteChoice = voteChoice;
        this.update();
    }

    public String getName() { return voterName; }
    public int getChoice() { return voteChoice; }
}

Figure 3-8: Pseudocode for VoteRecord class

public class VoteTally extends SharedVar {
    // internal tables for storing choices and votes...

    public VoteTally(ObjectConfig config, Proxy proxy) {
        super(config, proxy);
        // remaining initialization code...
    }

    public VoteTally(ObjectConfig objConf, Proxy proxy, String[] choices) {
        // similar to previous constructor...
    }

    public String getChoice(int index) { returns the choice with the given index }
    public String[] choices() { // returns the list of choices }
    public int getVotes(int index) { // returns number of votes for a given choice. }

    public void addVote(String name, int vote) {
        // stores a vote that has been cast.
    }

    public String voteCountsToString() { // returns tallies for choices }
    public String[] voteCounts() { // returns tallies for choices. }
}

Figure 3-9: Pseudocode for VoteTally class
public class VoteServerThread extends Thread implements SharedVarListener {
    volatile VoteTally tally;
    volatile VoteRecord record;
    Proxy proxy;

    public VoteServerThread(String[] choices) {
        proxy = new Proxy();
        Config tallyConfig = new Config("local:tally", Config.READ—Config.WRITE);
        ObjectConfig objConf = new ObjectConfig(tallyConfig);
        tally = new VoteTally(objConf, proxy, choices);
        Config recordConfig = new Config("local:vote", Config.READ);
        objConf = new ObjectConfig(recordConfig);
        record = new VoteRecord(objConf, proxy);
        record.addSharedVarListener(this);
    }

    public void run() {
        proxy.joinServer();
    }

    public void sharedVarUpdated(SharedVarEvent e) {
        if (e.getSource() == record) {
            tally.addVote(record.getName(), record.getChoice());
            tally.update();
        }
    }
}

public class VoteClient extends Frame implements ActionListener, SharedVarListener {
    private volatile VoteTally tally;
    private volatile VoteRecord record;
    private String name;

    public VoteClient(String name) {
        this.name = name;
        Proxy proxy = new Proxy();
        Config tallyConfig = new Config("socket://localhost/tally", Config.READ);
        ObjectConfig objConf = new ObjectConfig(tallyConfig);
        tally = new VoteTally(objConf, proxy);
        tally.addSharedVarListener(this);
        tally.reload();
        Config recordConfig = new Config("socket://localhost/vote", Config.WRITE);
        objConf = new ObjectConfig(recordConfig);
        record = new VoteRecord(objConf, proxy);
    }

    public void sharedVarUpdated(SharedVarEvent evt) {
        // update the display to reflect
        // the results in the VoteTally....
    }
}

Figure 3-10: Pseudocode for VoteServer and VoteClient application
of the other clients involved in the voting process will received updated values for their VoteRecord objects when this client casts its vote. The class is synchronized to make any operations on the local VoteRecord atomic in order to avoid potential multithreading problems this may cause. Future versions of SHAREHOLDER may allow for a non-propagating update operation to reduce communication by avoiding sending updates from the server to other clients.

This class also provides an example of how a configuration’s privileges can enforce the access of an object. In this example, the configurations ensure that the applications follow the communications model of the program – where the VoteRecord is written only by the client, and the VoteTally is written by the server. Any attempt by the application to illegally read or write (through a call to reload() or update()) will generate a exception.
Chapter 4

Details of the ShareHolder System

A complete application in the SHAREHOLDER system consists of three modular components — the application code, configuration objects, and the proxy. The application code is a networked application containing shared variables written by the programmer. Configuration objects represent configuration information associated at runtime with the shared variables in the application code. These objects may be written by the programmer or by an application user; the decoupling between application code and configuration objects makes it practical for these two components to be implemented by different persons. Finally, the proxy is a component provided by the SHAREHOLDER library for handling all the communication details of sharing variables.

4.1 The Application

The main goal of the SHAREHOLDER system is to allow programmers to easily adapt non-networked code to use shared variables with networked models of information access. Thus, the real challenge has been to create a system that is flexible enough to use for a variety of applications, but is also simple enough to require minimal changes to standalone programming styles. We feel that the programming interface described in chapter 2 achieves this goal within in a simple and extensible object oriented framework. This chapter does not repeat the basic descriptions of how the
programmer uses the SHAREHOLDER framework that have been given alread; instead it explains the system design decisions and implementaion details.

### 4.1.1 The Reload Operation

Each shared variable has two associated methods for reading and writing its contents — `reload()` and `update()`. Furthermore, in the case of reloads, there are two basic models (called *piecemeal* and *wholesale*) of how `reload` can be used to load the value at the variable’s home location into the local copy. The piecemeal model is for shared objects whose component fields are loaded individually from various sources. A good example of this is the StockQuote in the first example program (section 3.1). Because it is used to load components from read only sources (web pages, etc.), this piecemeal mechanism is currently only allowed for reloading shared fields that are read but never updated by the application over the network. The wholesale model is for loading entire objects from remote servers. This is made possible through communication between proxies, and can be seen in the other two examples (sections 3.2 and 3.3). In both examples, a proxy acts as a server repository for shared variables used by clients. Regardless of the reload model, it is always true that the reload returns the value of the object at the server when the reload request was received.

---

Figure 4-1: An example of the update mechanism.
4.1.2 The Update Operation

The `update()` method sends the contents of a shared variable to the server proxy that is the home location for the shared variable. Upon receiving the update, this server then propagates the new value outward to other clients. Figure 4-1 is a graphical depiction of this process. At the first step, a client which is a subscriber for the shared variable posts an update to the variable’s source. This update is then propagated by the server proxy out to the other subscribers in step 2. When a new client B posts an update in step 3, it is both added to the list of subscribers for the variable and the new value is propagated out to the other subscribers in step 4. When a client proxy receives an updated value for a shared variable from the server, it asynchronously updates the values of its local variable. Such updates to local variables can happen at any time during execution; correct programs using shared variables should take into account that their values may change at any time. Therefore, shared variable declarations are required to be marked `volatile` to indicate that the variable’s value may change suddenly and should not be optimized by the compiler. As mentioned in the Introduction, multiple updates — whether from different applications or the same client program — are not guaranteed to be received in any particular chronological order (unless blocking invocations of `update()` are used). Furthermore, because of changing network conditions, there is also no upper bound on the time when an update is received at the server or subsequently by any subscriber for the shared variable; indeed, there is no guarantee that it is even received at all. Also, while it is possible for clients to know if their variable update has reached the server or an error has occurred in transmission, there is no way for a client to know if another client has received any particular update.

4.1.3 Blocking and Nonblocking Invocations

Since reloads and updates can take potentially a long time to complete, both operations are implemented to run in separate threads. In a nonblocking invocation, the programmer is allowed to specify a maximum time limit to block on waiting for
the command to complete successfully; if that limit is exceeded, the caller continues execution even though the reload or update operation may not have completed. A blocking invocation forces the caller to wait indefinitely for a reload/update to complete. Although this could potentially result in applications hanging due to networking problems, this may be useful in certain cases. For instance, blocking on a reload acts as a guarantee that further computation continues only with the most recent value from the server. Likewise, blocking on an update can guarantee that further code is executed only after the update has successfully been transmitted to the server. Using such blocking invocations allows programmers to support loose memory coherency within their applications. Regardless of whether an update is blocking or non-blocking, any modification to the local value of a shared variable is performed within a synchronized method (in Java) in the local proxy. This ensures that updates are atomic with respect to local threads.

4.1.4 The Event Framework

In addition to the reload and update methods, the SHAREHOLDER system also supports an event notification framework for reporting when shared variables have changed (presented in figure 4-2). Any object which wishes to act as a listener must implement the SharedVarListener interface. This interface specifies several defined listener methods. SharedVarListener objects can then register themselves as listeners for events in any shared variable. When an event occurs, the appropriate event handling method inside the listener is called. Examples of such event handling were seen in the Chat and Vote example programs in sections 3.2 and 3.3. Currently there is only one event type, VAR_UPDATED, which is triggered when the internal value of a shared variable is changed (whether as a result of a reload or update); however, this framework can easily be extended for other event types in the future.
public class SharedVar {
    public void addSharedVarListener(SharedVarListener listener);
    public void removeSharedVarListener(SharedVarListener listener);
    void notifyListeners(SharedVarEvent evt);
}

public class SharedVarEvent {
    SharedVarEvent(SharedVar source, int eventType);
    public SharedVar getSource();
    public int getType();
}

public interface SharedVarListener {
    public void sharedVarUpdated(SharedVarEvent evt);
}

Figure 4-2: The SharedVar Event Notification Framework

4.2 Configuration Objects

The configuration objects are the second aspect of a distributed application in the SHAREHOLDER system. As described previously, the configuration information associated with a SharedVar object specifies the sharing behavior of the variable at runtime. Corresponding to the two models of reloading object values outlined in section 4.1.1, a shared variable can either have one global configuration or individual configurations for each of its fields (with the global configuration acting as a default for unspecified fields). The first case — called wholesale reloads — is for situations where we wish to share the entire variable, and thus want to specify a single configuration for the entire object. In the case where we reload the fields of a shared object individually, we would want the piecemeal configuration described by the latter case.

The information required for a configuration object is rather simple. Each configuration is made up of two parts — the source used for retrieving the object’s information and the privileges the object is granted with that information source. The source is a pseudoURL\(^1\) which declares the source to use for both reloads and updates. This source contains the protocol and machine to use for operations. Some examples

\(^1\)We call it a pseudoURL, because the source can specify strings that are not valid URLs but follow the same stylistic convention. Furthermore, like real URLs, the pseudoURL string specifies both a mechanism (e.g., http, ftp, email) and a location (e.g., www.mit.edu/people/harrisj/) for retrieving information.
Table 4.1: Communication mechanisms currently supported

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>http</td>
<td>retrieve a read-only individual field value from a special tag located in the source document.</td>
</tr>
<tr>
<td>socket</td>
<td>communicate with a remote proxy server using sockets</td>
</tr>
<tr>
<td>local</td>
<td>indicates that the variable is either strictly local or possibly shared but residing on the local machine.</td>
</tr>
<tr>
<td>email</td>
<td>communicate with a remote proxy server via email (e.g., using SMTP and conventional unix mail handling only for now)</td>
</tr>
</tbody>
</table>

Table 4.2: Potential future extensions to the communications mechanisms

of current protocols and potential future extensions are provided in tables 4.1 and 4.2. The privileges in a configuration specify what operations are allowed for the object; currently, this just specifies whether a SharedVar can be reloaded or updated, but this could be extended in the future.

Two examples should give a rough idea of how typical configurations are specified. As an example of piecemeal configurations, the quote field within a StockQuote variable might be given a configuration that resembled a conventional URL:

source: http://quoteserver.quote.com/cgi/get-quote?AAPL\verb+\+AAPL
privs: read only

In this case, the configuration tells the proxy to search the page specified by the URL for a VARTAG with the name AAPL (this mechanism is more thoroughly described in section 3.1). The information from this tag is then extracted and used to update the contents of the field. As another example, the VoteClient application (section 3.3) contained a VoteTally object which was shared by the clients and a server. The configuration for this object had the basic form

client source: socket://proxy/identifier privs: read
server source: local:identifier privs: read and write

In this case, proxy is the hostname of the server, and identifier is an identical string in both configurations used to reference the object in the server's directory. Also, note
how the configurations are being used to enforce a model where the server writes the object, but the client can only read it. More concrete examples of these configuration objects can be found in the example code.

4.3 The Proxy

Last but not least, there is the Proxy. The SHAREHOLDER system requires that a single proxy thread must exist on each node/system that a distributed application runs on. The proxy is a library component which hides all of the details of network programming and shared variables behind an abstraction layer. Components of the systems contained within the proxy include the core operations for all of the SharedVar methods as well as structures for communication between proxies.

4.3.1 Communication Among Proxies

For a more flexible and scalable design, the proxy uses an abstract communication layer which can be implemented on top of any particular protocol, resulting in a two-tier communication framework as shown in figure 4-3. On the top level, communication between proxies can be viewed as an exchange of ProxyMessage objects. These message objects contain basic information used for any communication between proxies (i.e., the source and destination proxies, a message type, shared variable identifier, and optional shared variable value). At this level, all inter-proxy communication is one-way and connectionless; that is, ProxyMessages sent are not guaranteed to be received at their destination in any particular order, or even at all. This allows the proxy to thus use connectionless and/or one-way protocols (as well as more reliable connection-based ones) for its underlying message transport. Tables 4.3 and 4.4 present examples of messages which can be received by server and client proxies.

The lower tier of the proxy’s communications uses various threads for transmitting and receiving messages in the supported protocols (e.g., sockets, datagrams, email,
<table>
<thead>
<tr>
<th>Type</th>
<th>Components</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscribe</td>
<td>name</td>
<td>adds the source proxy as a subscriber for the named shared variable</td>
</tr>
<tr>
<td>Unsubscribe</td>
<td>name</td>
<td>removes the source proxy from the list of subscribers for a the named variable</td>
</tr>
<tr>
<td>Create ID</td>
<td>name, value</td>
<td>If there is no entry associated with the name in the server’s directory, create an entry with the given name and value. Otherwise, if the variable associated with the name is the same type, add the source proxy as a listener. The behavior for when the types differ is currently undefined.</td>
</tr>
<tr>
<td>Remove ID</td>
<td>name</td>
<td>Remove the entry associated with the name from the directory. The server is currently not obliged to notify any subscribers of such an action.</td>
</tr>
<tr>
<td>Reload</td>
<td>name</td>
<td>Request the latest value of the named variable from the proxy. If the requesting source is not a subscriber for the named variable, it is added to the list. If the named entry does not exist in the directory, the behavior is currently undefined.</td>
</tr>
<tr>
<td>Update</td>
<td>name, value</td>
<td>Indicates an update for the named variable. If there is an entry associated with the name with the same type, the server updates the value of the variable, adds the message source as a subscriber, and notifies all other subscribers of the new value with the update_value messages. If there is no entry associated with the name in the server, creates a new entry and adds the source as a subscriber. If the named entry has a different type, the behavior is currently undefined.</td>
</tr>
</tbody>
</table>

Table 4.3: Proxy messages received by the server.

<table>
<thead>
<tr>
<th>Type</th>
<th>Components</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reload Value</td>
<td>name, value</td>
<td>The reply to a reload request. Contains the name used to request the object and the latest value of the variable.</td>
</tr>
<tr>
<td>Update Value</td>
<td>name, value</td>
<td>Indicates that a new value for the named entry has been received at the message source. Triggers an automatic update in the value of the local SharedVar associated with the name. If there is a type mismatch between the received value and the local version, the behavior is currently undefined.</td>
</tr>
</tbody>
</table>

Table 4.4: Proxy messages received by the client program.
etc.). At startup, the Proxy creates a ProxyServerThread which listens for incoming messages. These messages are then translated into higher-level ProxyMessage objects which can then be handled by more general methods inside of the proxy. Similarly, outgoing messages begin as ProxyMessage objects and are translated appropriately before being sent out via a particular communication mechanism. The SHAREHOLDER system uses object serialization\(^2\) for sending shared variables contained in ProxyMessage objects. As a result, any fields that should not be shared should be declared transient. The final part of the lower-level communication system contains the VarDirectory structure which contains pointers to all SharedVars hosted at the server proxy as well as their subscribers. This information is used by the server to propagate client updates outwards to other subscribers. Clients also use this structure as a pointer to local cached copies of shared variables, but such variables in their directories have no subscribers. Subscriptions in the directory have no preset expiration times, but are instead verified with every update; when a client reloads or updates a variable it is added as a subscriber if necessary. Furthermore, whenever the proxy encounters an error while sending a message to a client, the client is removed from the list of subscribers. In this way, the list of subscribers stored at the server roughly reflects the actual shared usage of the variable.

4.3.2 The Proxy Implementation of SharedVar Methods

The life cycle of a typical shared variable demonstrates how these messages are used by the proxy. When a variable whose configuration specifies that it is hosted on a remote proxy is initialized in a client program, the client proxy creates an entry in its own local directory — containing a pointer to the local variable and the variable's identifier — before sending a SUBSCRIBE or CREATE_ID message to the remote server. These two message types serve largely similar purposes of notifying the server that the client wishes to be added as a subscriber for the shared variable, but they differ in how they act when there is no variable associated with the requested identifier at

\(^2\)Using the Java Serialization API in JDK 1.1.
the server (see figure 4.3 for more details). When that server receives this message, it adds the client as a subscriber and creates an entry for the variable associated with the identifier if necessary. Similarly, when a variable is no longer used and is finalized by the garbage collector, the client proxy notifies the server that it is no longer a subscriber by sending a UNSUBSCRIBE or REMOVE_ID message as appropriate to the remote proxy which notifies the server that it can remove the client as a subscriber. The REMOVE_ID message is meant to provide future support for remote clients using a server to remove objects no longer being shared by any clients, but its usage is currently undefined.

When the client application requests a reload of a shared variable, the client proxy sends a RELOAD request for the variable to the remote server. This invocation may then block (issues pertinent to such blocking were discussed in section 4.1.3) until a reply is received from the server through a RELOAD_VALUE message. This blocking for the reload is accomplished through joining on a ReloadWaitThread which waits for the reply to the reload from the server. This received value is then used to update the local copy of the shared variable which was to be reloaded. This entire mechanism is presented graphically in figure 4-4. Similarly, an application call to update() triggers
Figure 4-4: The Proxy's Reload Mechanism

Figure 4-5: The Proxy's Update Mechanism
the client proxy to send an UPDATE message to the server proxy. The server updates its directory variable with the contents of the message and propagates this new value out to all of the other subscribers within UPDATE.VALUE messages. These clients then update their local copies of the shared variable in turn. The process for this is illustrated in figure 4-5.

When new values for variables are received, the shared variable’s contents are updated asynchronously. This seemingly simple goal presents some interesting problems in an object-oriented system. We might like to directly replace the contents of the shared variable’s address in local memory with the newer value; however, this is impossible in modern object oriented languages which allow classes to contain protected fields. Instead, it is necessary for shared variables to implement a method called updateObjectValue which recursively copies the fields from the new value over. This function basically exists as a “back door” into the object’s internals for the Proxy’s update mechanism. This method can be inherited from the SharedVar class, but it works only in cases where all internal fields are public. Otherwise, it must be inserted into shared variable class definitions manually by the programmer. Fortunately, there is a generic version of updateObjectValue() which uses the Java Reflection API (part of Java JDK 1.1) to determine all the internal fields of a variable. This function can easily be added by programmers or could be automatically inserted into classes by a customized compiler. As a silver lining, this approach does provide the means for a sophisticated programmer to customize the way his shared variables are updated.
Chapter 5

Extensions

There are many ways in which the current design and implementation of the SHAREHOLDER system can be extended and improved in the future. This chapter briefly outlines a few of the most pertinent extensions that could be made.

5.1 Changes to the Application

There are several important changes that could be made to the programming interface. For starters, there is currently extra code that has to be inserted by programmers into the most basic programs, as can be seen in the examples. This extra code includes the creation of the Proxy and ObjectConfig objects which have to be created explicitly at the top level and then passed to the constructors for any SharedVar. Furthermore, any shared variable with protected internal fields must contain a generalized updateObjectValue method (not included in the examples for brevity) in order for object updates to work. These code insertions diminish the simplicity of our system, and it should be possible in future versions to further reduce the amount of code that has to be added by the programmer. It remains to be seen whether this can be accomplished through a better system design or a clever compiler that inserts such code automatically.

We can also envision extensions to the programming semantics of reload and update. In addition to the blocking and nonblocking invocations, it may be a good
idea to extend the reload and update commands with extra arguments to specify other useful parameters. For instance, an extra parameter could specify whether or not updates should be propagated to other clients when received at the server (the unnecessary propagation of the VoteRecord class to other client in the Voting application of section 3.3 is a good example of this option’s usefulness). Other parameters may be used to set a time limit and/or data transfer rate limits which must be met for message transmission, or else the operation fails\(^1\). Other possible extensions to the semantics include support for security, locking and transactions. Even in the absence of any locking mechanisms, the system could also be extended to support an atomic “fetch-and-add” operator for shared variables. It may even be possible to establish some sort of temporal ordering, where clients would be able to detect out-of-order messages from the server and reject dated updates with a more complicated protocol (reminiscent of TCP/IP). Such changes should be possible but would require a careful analysis to ensure that the system will work even under bad network conditions.

5.2 Changes to the Configuration

There are several possible changes to the ways in which users specify and use configuration objects that could be considered in the future. The biggest change is to implement the functionality to allow shared variables to load their configurations from a file (or some other data source) at runtime. This would still require the ObjectConfig parameter to be passed to a shared variable’s constructor, but the contents of the configuration could be loaded externally. Using configuration files decouples the runtime configuration from the program code, allowing the same program to be run with multiple configurations without recompilation. It also makes it easier for the programmer to write networked code without having to worry about every detail of its usage. In the current system without this functionality, the separation of compilation and runtime behavior from can be achieved through modular programming.

\(^1\)This interface for setting limits on transfers was suggested by [3].
Configurations could be loaded from the file through requesting them by name. If there is no configuration associated with the requested name in the file, a default configuration specifying that the object is local is returned. This method for retrieving configurations from a file is similar to that used for resources in the X Windowing System or applet parameters. This scheme requires the programmer to also publish a list of names used for retrieving configurations with his code, but it would make up for the inconvenience with the extra flexibility that it provides.

In addition to modifying the loading mechanism, it might also be good to extend actual the configuration information associated with objects to support more complex forms of communication and access. For instance, in the case of piecemeal reloads from a web server, we might want to be able to specify backup servers to query in case an application’s connection to the main server is lost. Other possible extensions might be to allow users to specify certificates or other security mechanisms to use with secure forms of access, as well as other types of access permissions or even time and rate limits for communication. Some of these have been mentioned as extensions to the application programming interface, but there are some good reasons why they could be specified in the configuration.

The current implementation requires the program user to select a protocol to use for sharing variables with server proxies. It should also be possible for the user to let the proxy “automagically” select an appropriate protocol to use for communication. The selection criteria could include such factors as the size of the object, some information about the state of the networks, and requested rates for speed and efficiency.

There are also some extensions we can make to the ways an application can load configurations from files. For instance, it should be possible to load files from over the network. Furthermore, we could extend the basic model of loading configuration files, perhaps creating a more flexible cascading configuration model. This framework — similar to the Cascading Style Sheets [14] proposed by the Web Consortium — allows for configuration files to be arranged in an inheritance heirarchy. For instance, it should be possible for companies to define global configuration files, which can be
inherited and potentially overridden by department-wide configurations, by individual
users, and so on.

5.3 Changes to the Proxy

The Proxy could also be extended in a variety of ways. For starters, support for the
future communications mechanisms described in table 4.2 of the System Overview
could be added with only a few simple changes to the existing code. With some appro-
priate additions to the configurations, it should be possible for the proxy to use other
high-level systems as protocols for sharing, essentially making the SHAREHOLDER
system a simpler front-end for other systems. For instance, it should be possible to
use JavaSpaces, RMI, and possibly even CORBA OMG as back-end protocols if so
desired. This would require only some potential additions to the configurations and
some additions to the proxy to interface with such systems.

The VarDirectory used by the proxy for tracking subscribers ought to have a more
precise naming scheme for keys e.g., a concatenation of an identifier and the fully
resolved name of the object’s class — which would resemble identifier$classname.
Using this indexing scheme would eliminate the undefined cases described in figure 4.3,
where a client attempts to reload or update a shared variable identifier of a different
type than is associated with the identifier at the server. Such naming conventions
could be added automatically by the client proxy and would serve to avoid namespace
collisions by objects of different types.

Security and authentication could also be added to the proxy. The use of authen-
tication would allow the designer to control which clients are able to use the server
and what permissions they are granted. These permissions could cover such actions
as creating, removing, reloading, or updating objects in the server directory. Such
authentication and permissions could be implemented with certificates and access
control in a way similar to that used by existing secure web services. It should also
be possible to add encryption for protecting sensitive data that might be intercepted
in transit from the client to the server or vice versa. Such mechanisms need not be
required for all applications, but should be requested in specific configurations.

For better performance on a machine running multiple applications, the proxy’s functionality could be split into two components. Instead of interfacing with a full-scale proxy, each SHAREHOLDER application would contain a thin stub proxy. Each stub would communicate with a single central proxy acting as a daemon on the machine. All communication (i.e., updates and reloads) from an application to a remote server proxy can then be relayed by the stub to the central proxy which would then communicate with the remote proxy. Replies are received at the central proxy which then could pass it to the stub. This approach adds an extra communications layer but requires less system resources for multiple applications.

Inter-proxy communication can be improved in other ways. For starters, it should be possible to devise more efficient variable encoding schemes for different communication protocols. For instance, while straight serialization may be sufficient for sockets, a different encoding might be better for datagrams. Of course, any encoding in our current approach sends the entire object; this can be inefficient for reporting small changes to large shared variables with many fields. It might be possible to instead use a delta encoding which only specifies changes instead of entire structures. Another more change to the communication would be to add a tag representing the time the message was sent for outgoing messages from the server to the client. This would allow client proxies to detect and reject out of order messages from the server. More complex time ordering would be difficult or even impossible, but even this simple ordering would be useful to protect applications from being thrown off by receiving messages from the server out of order.2

One other major change to the communication mechanism is useful for the situation where a shared variable may have many internal fields, but the state of these are dependent on only a few internal variables. For these cases we can define a transformation function which takes a few arguments and uses them to define values

---

2 As an example, we can imagine a server could send out two different UPDATE_VALUE messages to a client in the course of a few minutes with the second accidentally arriving first. The client would then be updated with a stale copy of the server’s variable when the first message from the server arrives.
for all of the internal fields. The idea is similar to that of a constructor in modern object oriented language. Under this approach, programmers would be able to specify a transformation method as a method in a shared variable (which would probably take in an array of parameters) that is to be used by the proxy to update the internal state of the object. Proxies would share variable values merely by sending a list of arguments for the transformation function, which would be called by the local proxy to update the local copy of the object. The default case inherited from SharedVar would have the same functionality as the current updateObjectValue() method. However, this interface would allow programmers with an easy mechanism for overriding the transformation method and thus customizing how their shared variables’ values are updated.
Chapter 6

Related Work

As was mentioned in the Introduction (chapter 1), message passing is the most basic approach for writing distributed applications. However, the lack of abstraction associated with message passing makes it difficult to write simple programs and to reuse code across multiple distributed applications. Distributed Shared Memory (DSM) systems pioneered the metaphor of shared memory for homogeneous distributed applications. Through scalable topologies and efficient implementations, some notable recent implementations (Munin [5] and TreadMarks [2]) have even approached the speed of raw message-pasing systems for certain applications [6].

Java RMI [13] is another recent system for simplifying the writing of distributed applications. An object oriented version of Remote Procedure Calls (RPC), RMI allows the programmer to call certain methods of objects over the network. This basically allows the programmer to think of network applications as procedure calls; the underlying system translates this into the network communication. Despite this abstraction, the RMI programmer still has to do a fair amount of low-level programming when writing a distributed application, such as implementing stubs and redefining objects appropriately to use RMI. A handful of other collaborative systems have been built with Java, but these either provide modified low-level mechanisms [1, uses modified AWT events] or specialized high level components [7, provides client and server tools for “hablets” written in Java].

JavaSpaces [12], a system built on top of RMI, is probably the closest system
to SHAREHOLDER in design and motivation. Both systems create an abstraction of a virtual networked store (similar to the shared memory of DSM) which multiple applications can use; the programmer is able to write networked code in terms of reads and writes on this shared space. Unlike conventional DSM implementations, both systems also are designed for the sharing of object oriented structures and provide a notification mechanism for reporting object changes using events [11].

Despite these similarities, there are some important differences between JavaSpaces and the SHAREHOLDER system, largely stemming from a difference in metaphor. JavaSpaces is built upon a metaphor of a "space" as a non-indexed persistent networked store that acts as the intermediary in the exchange of objects between applications. Under this approach, objects are retrieved from the space using templates of specified values and/or wildcard values for an object's fields. It is possible for several objects to match a read() with a given template, and a write() to the space only adds a new object to the space; it does not ever overwrite an object. In order to remove an object, an application can use the take() method to read and remove objects from the space. In order for takes and writes to be atomic, JavaSpaces also provides a transaction mechanism that can support multi-space updates using a two-phase commit model.

In contrast, the SHAREHOLDER system is based loosely on the metaphor of a shared memory. Under this system, there is only a single "space" which is a table of shared variables indexed by unique string identifiers. There can only be one variable associated with a given identifier in the space. A read or write operation only accesses the value of a single variable in this space. Furthermore, under this metaphor, this table entry in the space is the location for the shared variable. Ultimately, SHAREHOLDER is a more lightweight and flexible system than JavaSpaces.

Two examples should illustrate the differences of the two approaches. The first example (presented in figure 6-1) is a translation to JavaSpaces of the Simple Chat program from section 3.2. In this code, there is no MessageServer class, since the clients just use the JavaSpaces server. In order to emulate the behavior of update(), a client must first take the SharedMessage object before writing to the space. This
import java.rmi.*;
import net.jini.event.*;
import net.jini.transaction.*;

public class SharedMessage extends BasicEntry implements RemoteEventListener {
    transient private JavaSpace space;
    private String channel;
    private String message;
    transient private ChatFrame cf;
    private transient EventRegistration eventReg;

    public SharedMessage(JavaSpace space, String channel) {
        message = null; this.channel = channel;
        this.space = space;
        this.eventReg = space.notify(this, null, this, Lease.FOREVER, null);
    }

    public void postMessage(String message) {
        this.message = message;
        SharedMessage template = new SharedMessage();
        space.takeIfExists(template, null, 0);
        space.write(this, null, 0);
    }

    public String getMessage() { return this.message; }

    public void reloadMessage() {
        SharedMessage template = new SharedMessage(this.space, this.channel);
        SharedMessage reloadVal = (SharedMessage) space.read(template, null, timeout);
        this.message = reloadVal.message;
    }

    public void notify(RemoteEvent evt) throws UnknownEventException, RemoteException {
        if (evt.getlDO == this.eventReg.getlDO) {
            reloadMessage();
            if (cf != null)
                cf.append(this.message);
        }
    }
}

public class SimpleMessage {
    public static void main(String args[]) {
        SharedMessage sm;
        JavaSpace space = getSpace();
        sm = new SharedMessage(space, args[0]);
        ChatFrame cf = new ChatFrame(sm, ident);
        sm.setChatFrame(cf);
    }
}

Figure 6-1: JavaSpaces Pseudocode for the Simple Chat Program
import java.rmi.*;
import net.jini.event.*;

public class VoteRecord extends BasicEntry {
    String voterName;
    Integer voteChoice;
    transient JavaSpace space;

    public VoteRecord(JavaSpace space) {
        this.voterName = null; this.voteChoice = new Integer(-1);
        this.space = space;
    }

    private static VoteRecord template() { return new VoteRecord(); }

    public void makeVote(String voterName, int voteChoice) {
        this.voterName = voterName; this.voteChoice = new Integer(voteChoice);
        VoteRecord template = this.template();
        space.takeIfExists(template, null, 0);
        space.write(this, null, Lease.FOREVER);
    }

    public String getName() { return voterName; }
    public int getChoice() { return voteChoice; }
}

public class VoteTally extends BasicEntry {
    private VoteTally() { // code removed for brevity }
    public VoteTally(String[] choices) { // removed for brevity }
    public static VoteTally template() { return new VoteTally(); }
    public void addVote(String name, int vote) { //removed for brevity }
}

Figure 6-2: JavaSpaces Pseudocode for the Voting Program

would be encapsulated in a transaction to make it atomic, but that has not been included for brevity. Also, note the usage of an extra element called channel in the SharedMessage class. This is similar in function to the identifier used in the SharedVar configuration, allowing multiple channels of communication to exist on the same server.

The second example shows a translation of the Voting application to JavaSpaces (in figures 6-2 and 6-3. Unlike the previous example, there is a server which uses the Space. In this system, the clients cast votes by placing VoteRecords into the JavaSpace. The Space then notifies the server which removes the votes and replaces its tally object with a combined take and write. The server retrieves votes by
public class VoteServerThread extends Thread implements RemoteEventListener {
    VoteTally tally;
    JavaSpace space;
    EventRegistration eventReg;

    public VoteServerThread(String[] choices) {
        if (choices == null) tally = new VoteTally();
        else tally = new VoteTally(choices);
        this.space = getSpace();
    }

    public void run() {
        space.write(tally);
        eventReg = space.notify(VoteRecord.template(), null, this, Lease.FOREVER, null);
        // code for joining goes here.
    }

    public void notify(RemoteEvent evt) {
        if (evt.getID() == eventReg.getID()) {
            VoteRecord template = VoteRecord.template();
            while (true) {
                VoteRecord record = (VoteRecord) space.take(template, null, 0);
                if (record != null)
                    tally.addVote(record.getName(), record.getChoice());
                else
                    break;
            }
            space.take(VoteTally.template(), null, 0);
            space.write(tally, null, 0);
        }
    }
}

public class VoteClient extends Frame implements ActionListener, RemoteEventListener {
    private VoteRecord record;
    private String name;
    private JavaSpace space;
    private EventRegistration eventReg;

    public VoteClient(String name, int port) {
        this.name = name;
        this.space = getSpace();
        record = new VoteRecord(space);
        eventReg = space.notify(VoteTally.template(), null, this, Lease.FOREVER, null);
        // ALL THE GUI CREATION CODE FOLLOWS
    }

    public void notify(RemoteEvent evt) {
        if (evt.getID() != eventReg.getID()) { return; }
        VoteTally tally = (VoteTally) space.read(VoteTally.template(), null, 0);
        if (tally == null) { return; }
        // update display with tally
    }
}

Figure 6-3: JavaSpaces Pseudocode for the Voting Program, part 2
querying the space for matches for each choice, creating some extra overhead. The clients in turn wait for notification of updated tallies being placed into the Space and uses the new value to update their displays.
Chapter 7

Conclusions

In this thesis, we have described the SHAREHOLDER system which is an extremely lightweight infrastructure for supporting shared variables on the Internet. This system was designed to provide programmers with a powerful but simple framework for developing networked applications. Our motivating inspiration was the “less is more” basic model of information access common to the World Wide Web. By not inherently requiring any consistency or coherency for shared memory access, we are able to support more flexible methods of information sharing than other distributed programming systems. This in turn results in less overhead in the programming interface, allowing any programmer to quickly adapt existing standalone code to use this loose model of networked sharing. Furthermore, more complex protocols can be supported by the system library later without requiring any real changes to the application code.

This simplicity of design has also made the SHAREHOLDER system relatively simple to implement. I wrote a basic implementation of the system in 3,000 lines over the course of 3 months. This example implementation already supports most of the functionality described in the system overview (chapter 4) and is flexible enough to be easily extended with the proposed modifications described in chapter 5. A final implementation would probably contain less than 10,000 lines of code with multiple interfaces for plugging in future extensions (e.g., support for different protocols, modules for security, libraries for locking and transactions, etc.) This may not seem
like much in itself, but it is interesting to compare to more large-scale and general sharing systems such as CORBA or even JavaSpaces. After it has been cleaned up somewhat, the SHAREHOLDER library written in Java will be made available for public use in the near future.
Bibliography


