Load Plateau: A Toolkit for Building Scalable Collaborative Applications

by

Rodrigo Leroux

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1999
June 1999

Author .....
Department of Electrical Engineering and Computer Science
May 21, 1999

Certified by .....
Judith Donath
Associate Professor of Media Arts and Sciences
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Load Plateau: A Toolkit for Building Scalable Collaborative Applications

by

Rodrigo Leroux

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 1999, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

Load Plateau is a tool kit that will aide software developers in the creation of scalable, real-time distributed collaborative applications (DCAs). DCAs are multi-user applications that allow people to work cooperatively to achieve some common objective. Cooperative work involves the sharing of a common state among the different hosts running the application.

To support efficient state sharing in a real-time DCA, Load Plateau implements a distributed shared memory with a weak consistency model based on an atomic broadcast protocol. State sharing improves the speed at which clients can access the shared state. However, the broadcasting protocol to maintain state consistency introduces scalability problems in applications that need to support a large number of users.

In a real-time DCA, the number of broadcast updates that each client has to process grows faster than the number of users in an application, potentially overwhelming clients that can't keep up with the fast update rate. Load Plateau addresses scalability issues using update prioritization, a strategy that exploits weak consistency constraints in real-time DCAs. Instead of processing updates in the order in which they are recieved, clients can process updates in order of importance, improving application responsiveness and distributing the load placed on each client over time. Since the priority of different updates is application specific, Load Plateau delegates the task of defining rules that prioritize updates to developers, giving them a reliable and flexible infrastructure to do it.

Thesis Supervisor: Judith Donath
Title: Associate Professor of Media Arts and Sciences

Acknowledgments

Many people helped throughout the development of this project. I would like to thank them all, especially my advisor, Judith Donath, for challenging my ideas and pushing me to give the best I could. This document is orders of magnitude better thanks to her assistance. Also, I'd like to express my gratitude to Fernanda Viegas and all the people in the Sociable Media Group for their valuable feedback and contributions in the makings of Load Plateau.

To my brother Xavier and my sister Ximena, thank you for always being exemplar models in my life. I love you very much. To Steven and Alex, I just want to remind you that Losers Inc. will rock the world. I sincerely hope we keep sharing our dreams and accomplishments, at least until each one of us gets married. To my friends Ben, Lydia, Oliver and Phil, I would like to express my immense admiration and prompt you guys not to forget about me after you become powerful and famous. To Luis Eduardo, Ceci, Carmen, Ivan, Memo, Carlos, Dani, Aletia and Amalia, now that most of us are on the other side of the bridge, I want to thank you all for sharing the best moments with me and helping me through the hardest ones. To Dina, thank you so much for being such a wonderful companion. Your inconditional love, support, and understanding have made this, my last year at MIT, an exceptionally happy one.

Finally, I would like to dedicate this thesis to my parents. There's so many things I would like to thank you for, that I'd rather mention only one, and then show my appreciation for the others by always striving to make you feel proud of me. Thank you for loving each other so much.

Contents

List of Figures

Chapter 1

Introduction

Network environments like the World Wide Web have allowed people to share information easily with others regardless of their physical location. However, as of today, web users still want and expect more from this now ubiquitous medium of communication. Beyond the ability to share information, people also want to share workspaces so that they can interact with others in a way that resembles more closely how people interact with others in the real world. This need has motivated a great research interest in building collaboration tools [3, 7, 14, 18], figure 1-1. As network technologies evolve and the infrastructure to support them gets faster and better, the widespread deployment of distributed collaborative applications (DCAs) will become feasible, making them increasingly important as the software tools people use to work, communicate, and play.

This thesis describes the design and implementation of Load Plateau, a Java-based toolkit that aims at improving support for building real-time DCAs. Plateau's objective is to simplify the creation of performance-driven and scalable real-time DCAs through the use of distributed shared memory (DSM)[1] with a weak consistency model. Weak consistency allows improvement in application scalability while still guaranteeing that client states will converge, even when state updates may be applied out of order.

Load Plateau features two main components:

1. A simple framework based on a client/server model using DSM. The framework isolates developers from the details of implementing a DSM application, allowing them to concentrate on the application specific logic of a real-time DCA. The goal of using DSM is to increase the speed of shared state access by allowing clients to keep local copies, thus avoiding delays incurred from queries to other hosts. Different protocols to maintain state consistency exist [6] that provide different consistency guarantees. Plateau uses an atomic broadcast protocol [2, 5] to ensure that update broadcasts are received in the same order by all clients. It does not guarantee that at a specific moment, all state copies will be consistent. Instead, it guarantees that state copies will converge. Section 1.2.2 explains why this weak consistency model is appropriate for real-time DCAs.

2. An update prioritization utility that addresses scalability issues encountered when clients in a DCA using DSM can't cope with the amount of updates received. Update Prioritization is a strategy that exploits weak consistency to improve performance when a large number of clients are connected by controlling the order in which client updates get processed by their peers. The utility lets developers create application specific prioritizers that determine the behavior of incoming and outgoing message queues. It also allows developers to guard messages so that they can be reprioritized once they have been enqueued.

1.1 Overview

This thesis is organized as follows:

- The rest of this chapter presents background in the field of DCAs. It starts by describing the differences between asynchronous and real-time DCAs, and then proceeds to explain the challenges involved in building real time DCAs and the common design strategies used. It then explains the motivation behind creating Load Plateau, and describes the major architectural decisions made,

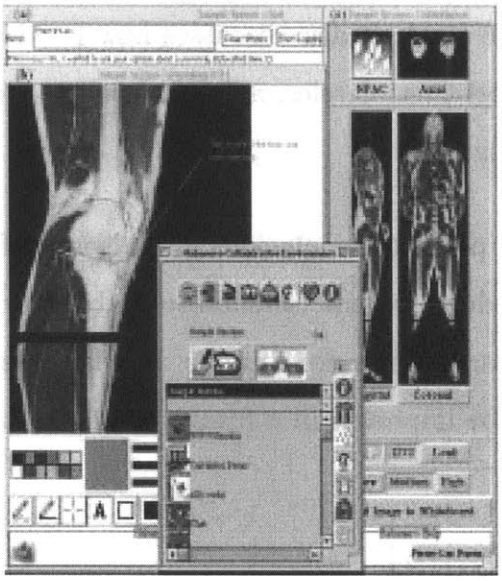

Figure 1-1: The HDF Viewer application allows medical interns to share their knowledge and diagnosis concerning some of their patients' cases. HDF Viewer is based on the Habanero Framework [4]

briefly justifying them. It finally describes a financial DCA as an example of applications whose implementation would be simplified using Load Plateau.

- Chapter two describes relevant work in the field of collaborative software, focusing on DCAs meant to work in real-time environments.

- Chapter three describes Update Prioritization in detail, why it can increase scalability and improve performance, and application characteristics to look for when evaluating whether update prioritization is a viable solution.

- Chapter four describes in detail and justifies the design and implementation of the Plateau framework that facilitates the creation of DCAs using distributed state sharing.

- Chapter five presents the Plateau Prioritization Utility, explaining its design and implementation.

- Chapter six explains the implementation of Plateau Scribble, a proof-of-concept applet that shows how Load Plateau can cut down on the development effort to

build a scalable DCA. It also discusses how Load Plateau was used to improve the scalability of Chat Circles, a DCA developed at the Sociable Media Group in the Media Lab.

- Chapter seven presents conclusions and ideas for future work. It evaluates the complexity of development, scalability and performance of applications using Load Plateau and draws conclusions on what design decisions where correct, and what can be improved in a future version of Load Plateau.

1.2 Background

Described in a broad context, a distributed collaborative application (DCA) allows people that are physically separated to join efforts interactively with the purpose of achieving some common goal. DCAs are similar to other types of distributed applications because computation happens in several processors that are connected in a computer network. Therefore, the design of a DCA involves (but is not limited to) a subset of the following issues: Coordinating activities that happen in different hosts, synchronizing copies of data that might exist in order to keep them consistent, and dealing with performance and capacity issues related to the underlying transport mechanisms. DCAs are different from other types of distributed software because their main objective is to allow people to interact with each other through a computer interface, relaxing the requirement of physical proximity necessary for some types of collaboration in the real world. Email, instant messaging, and competitive games like Quake are good examples of DCAs.

DCAs are becoming increasingly widespread as network technologies and infrastructure become better and faster. A clear example of how new technologies have enabled DCA is the World Wide Web. While initially, the WWW was just viewed as a user-friendly way to provide information over the Internet, eventually researchers realized that web technologies could be used in conjunction with web browsers to design, create, distribute and execute collaborative applications [11]. Currently, sev-

eral popular websites offer DCAs such as chat and game environments, collaborative virtual environments (CVE), and business simulations (websims) among others. Still, better underlying technologies can be built that will enable powerful DCAs such as reliable audio and video-conferencing environments and software to effectively manage collaborative work processes and documents. An example of a DCA that could be viable given the right technologies is described in 1.4. Load Plateau aims at improving support for building performance-driven real-time DCAs like the one described in that section.

1.2.1 Real-Time vs. Asynchronous DCAs

DCAs can be classified in two groups, based on the frequency of interaction between their users. Synchronous, or real-time, DCAs allow users to interact with each other during a period of time where all are active, and this continuous activity needs to be communicated to the whole group. Prompt broadcasting of activity is essential for the correct operation of these application since users need to be aware of activity at the moment when it happens. This type of DCAs requires that all clients are continuously connected so that activity can be broadcasted at any time with a bounded delay. Some collaborative and competitive simulations and games, instant messaging, and chat software are examples of real-time DCAs.

Asynchronous DCAs involve collaboration over a larger period of time. Activity is less frequent and correctness in the transmission of activity is more important than promptness. Email, network file systems, project management software, and multi-user development environments are examples of asynchronous DCAs.

This thesis is relevant to the development of real-time DCAs. It focuses on systems where clients are continuously connected and where performance is directly connected to how effectively are activity updates processed.

1.2.2 Design Strategies in Real-Time DCAs

DCAs enable collaboration by allowing users to work interactively viewing and modifying a set of common software objects that I will label the common state. The users work on client hosts that communicate user actions to its peers. When the actions modify the common state, all clients must be aware of the modification so that subsequent updates do not create inconsistencies in the clients' view of the common state. The minimum common state in most DCAs is the set of users in the application. When users leave, or new users join, the common state is changed to reflect this. There are two main architectural models that can be used to implement an application with the previous requirements:

Client/Server model In a client/server model, All clients connect directly to a central server. Communication between any two clients, a sender and a receiver, involves a message from the sender to the server, followed by a message from the server to the receiver. Application logic and data can be split between client and server in different ways, and clients usually run the same code. In a DCA using a client/server model, broadcasting is achieved by having the server broadcast a message received from a sender. Multicast protocols can cut down on the traffic generated when broadcast is heavily used. The common state can live as a unique copy in the server, as a shared copy in each client, or as a combination of both.

Peer-to-Peer model In this model, communication between any two clients can be achieved using a direct connection or using any number of clients as intermediaries. The common state lives as a shared copy in every client. Peer-to-peer models are efficient when broadcast messages are not heavily used but maintaining consistent copies of the common state is a challenging task.

Load Plateau uses a client/server architecture for the sake of simplicity and to allow clients to have access to an up-to-date version of the common state promptly

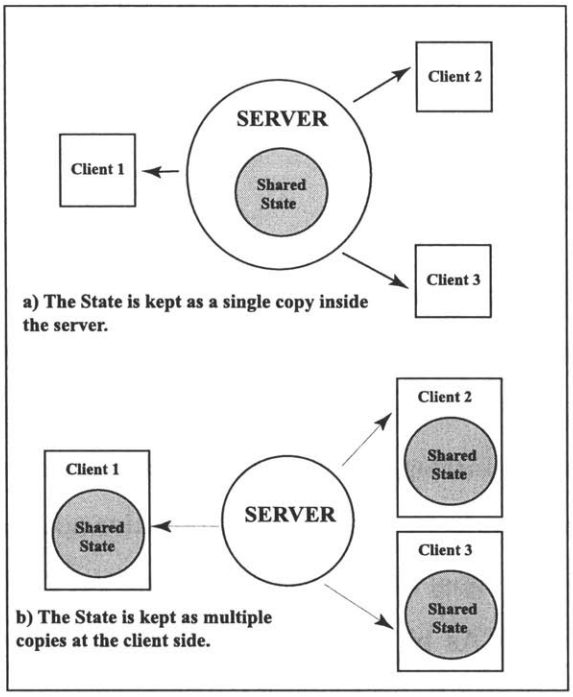

Figure 1-2: Two different strategies to keep the shared state in a Client/Server model

using distributed state sharing. In a peer-to-peer architecture, it would be very complex to implement a distributed shared state and it is not clear that broadcasting of updates would be more efficient. It would still be an interesting research topic to explore possible implementations of Load Plateau using a Peer-to-Peer communication model. Chapter ?? under the future work section explores this possibility further.

The most commonly used strategy to share state in a client/server application is to make the server the bookkeeper and filter of client activity updates, allowing clients to poll the server for information about the current state. This approach has the advantage of trivially preserving consistency of the shared state by keeping only one copy of it. However, the approach hinders performance by forcing clients to contact the server every time they need to access or modify parts of the common state. In applications where the client needs prompt notification about any change to the shared state, keeping state on the server hinders performance greatly. Instead, these types of applications rely on distributed state sharing, allowing all clients to keep a copy of the common state and broadcasting activity updates when they happen.

Load Plateau uses this second approach as real-time DCAs require prompt notification about common state updates.

1.3 Motivation

DCAs share sufficient characteristics and requirements to make exploring a general framework for building them an idea worthy of pursuit. Other research groups have explored the idea, and have came up with interesting frameworks that are sufficiently generic to serve as a starting point in the implementation of a particular DCA. 2 describes other frameworks for building collaborative applications and compares them to Load Plateau. The goal of the Plateau Framework will be to simplify the task of building scalable DCAs by taking care of the details of implementing DSM and allowing for a transparent integration of the update prioritization utility, addressing scalability problems that result from using a DSM approach.

The update prioritization utility will address scalability problems observed in DCAs relying on DSM. In these applications, clients rely heavily on the broadcast of activity updates among themselves, usually mediated by the server to keep a consistent state. However, the number of updates each client needs to process as the number of hosts running the same application increases is prohibitive. Fast clients might not have a problem with the number of updates that need to be processed, but slow clients will be overwhelmed forcing the DCA to perform at suboptimal levels. This problem is especially critical in applications built on slow runtime environments like Java or other interpreted languages.

The idea behind using update prioritization is that not all activity updates need to be processed as soon as they are received by a client for the application to operate correctly. As long as those that need to be processed immediately get priority over others that might be able to wait in a client-side queue for some time, DCA clients will cope much better with the update traffic as the number of users increases. Plateau's weakly ordered eventual consistency model is simple, scalable, and adequate for real-

time DCAs in which performance is more important than strict consistency.

1.4 A Motivating Example

Javaworld (http://www.javaworld.com) featured an article in its March, 1999 edition about Lotus and IBM working on a series of Java-based applications dubbed Lotus components for the Internet. The objective of these components it to let users access the Internet either from their SmartSuite applications or from a Web browser. The article describes an example of a brokerage business that lets investors access its Web site to analyze stock portfolios. With the portfolio information presented in a Lotus spreadsheet component, investors can do "what if" analyses, view the results in a Lotus chart component, and then send any changes back to their brokers.

Based on this idea, we could imagine a simulation implemented as a real-time DCA in which several investors trade stocks based on information displayed in a spreadsheet. The information would be influenced both by external factors and by the operations of other simulation users. If the spreadsheet is very large (thousands of rows and columns) and if the dependencies between cells in the spreadsheet are defined by complex and time-consuming operations, then reflecting updates received from operations happening in other clients would be a very time-consuming task. In a simulation where a large amount of investors are participating, prioritizing the activity updates so that cells that the investor is viewing and those they depend on are refreshed first, the number of users that the DCA can handle would certainly increase.

Implemented with Plateau, the simulation could allow hundreds of investors to participate at the same time while still allowing every client to hold data for the whole spreadsheet. Using prioritization, updates that are relevant to the activities that the investor is doing would take precedence over other updates, cutting down on the number of updates that need to get processed before the investor can get the results for a particular analysis or for trading stock. For example, if the investor is analyzing

a real-time chart about the value of a particular stock, updates that influence the value of that stock would always be processed first. When the investor moves on to analyze some other stock, then the updates relevant to the new stock would get high priority among all updates that need to be processed.

Chapter 2

Related Work

While other frameworks support the development of DCAs using distributed shared memory, none of them address scalability issues related to the use of DSM. Instead, most focus on underlying mechanisms used to give strong guarantees about consistency. In the first section, this chapter describes four such frameworks: Bayou, Habanero, DistView, and Orca, comparing their DSM approach to Load Plateau's. Table 2.1 summarizes the main characteristics of these frameworks.

We believe that there is a class of real-time DCAs that have weak consistency constraints and strong performance and scalability requirements. Load Plateau is unique since its main focus is on addressing these needs by using update prioritization, a technique that exploits weak consistency constraints. Prioritization, as a means for building scalable distributed systems, has also been studied, although most work targets lower layers in the network model. The second section of this chapter describes quality of service and priority dropping, two prioritization schemes used to improve performance in network routers and video multicasting, respectively. Plateau's update prioritization is based on knowledge that developers have about application level characteristics, targeting a higher layer than quality of service or priority dropping.

Table 2.1: DCA Framework Characteristics

Framework	DCA oriented	DSM	Consistency	Architecture	Scalable
Bayou	yes	yes	weak	peer-to-peer	no
Habanero	yes	yes	weak	client/server	no
DistView	yes	yes	strong	client/server	no
Orca	no	yes	strong	both	no
Load Plateau	yes	yes	weak	client/server	yes

2.1 DCA Frameworks

2.1.1 Bayou

The Bayou system [19] offers one solution for supporting the needs of DCAs. The goal of Bayou is to support distributed shared data applications with client-side updates in an environment with disconnected operation. Updates to shared state are propagated to clients using a variant of Golding's time-stamped anti-entropy (TSAE) protocol [12] for ordered group communication. This protocol enforces the policy of eventual consistency: in the absence of new updates, all caches converge to the same consistent state.

Plateau has similar goals to Bayou. Both provide eventual consistency of state for distributed collaborative applications. However, Bayou's implementation of eventual consistency uses merge and rollback procedures to automatically handle conflicting updates to shared data. Plateau has no notion of a conflict, and therefore has no need for merge or rollback procedures.

Another difference between Bayou and Plateau is in architecture. Bayou uses a peer-to-peer structure and TSAE to propagate updates. Peer-to-peer communication can improve scalability, but it is difficult to maintain when the set of peers is likely to change. In a peer-to-peer system, clients pair up to exchange updates. If the set of clients changes, a stranded client must be aware of other peers to continue with the TSAE protocol. DCAs with client/server communication can avoid this overhead

and state. Plateau addresses scalability issues in a client/server DCA where clients are performance bottlenecks by using update prioritization 3. Server side scalability issues can be addressed using the brute force approach: getting a faster, more capable server.

2.1.2 Habanero

The Habanero framework [4] is designed to give developers the tools they need to create collaborative Java applications. Habanero provides the necessary methods that make it possible to create or transition existing applications and applets into collaborative applications.

Habanero works by replicating applications across clients and then sharing all state changes in those clients. When a new client joins a session, it is sent information about which applications are running in that session. Each application is then sent enough information to completely replicate the important state being shared by the existing copies of that application. Habanero also ensures that all clients see the same state changing events in the same order, which results in applications appearing the same to all clients. Habanero allows programmers a great deal of flexibility in determining what exactly a state-changing event is. Habanero also provides a general floor control object called an arbitrator, which allows finer control over what users can take what actions at a given time.

Habanero can be seen as a mature implementation of the Plateau framework. Like Plateau, it uses a client/server architecture supported by Java's object serialization capabilities to do object replication. Like Plateau, its java-based nature allows deployment over a wide variety of operating systems and hardware platforms. Habanero is more mature because in addition, the environment includes a server that hosts several sessions and a client that interacts with sessions using a variety of applications called Hablets. Sessions can be recorded, are persistent, can be access restricted and can even be anonymous. The Habanero client provides the interface to define, list,

create, join and interact with a session. The client provides session information, user identification, a notification mechanism, record and replay capabilities, and security. However Habanero lacks the update prioritization utility that allows Plateau applications to deal with scalability problems when the number updates caused by a large number of users overwhelms slow clients.

2.1.3 DistView

DistView [16] suggests a simple synchronous collaboration paradigm in which the sharing of the views of user/application interactions occurs at the window level within a multi-user, multi-window application. It supports building of collaborative applications in which users can share some of their application windows with other users while still keeping other application windows private.

Like Plateau, DistView is targeted at synchronous collaboration over wide-area networks and uses an object-level replication scheme, in which the application and interface objects that need to be shared are replicated.

Unlike Plateau, it focuses on window-level sharing and makes use of locking techniques to guarantee replica consistency. In DistView like in Orca, clients must lock a shared object before modifying it. Using locks has the advantage that, once the locks are acquired, operations on objects can be done locally first and then broadcast, giving good interactive response times for object modification. However, locks slow down browsing operations since they can proceed only on unlocked objects. In Plateau, locking is avoided by first broadcasting a common state update and then relying on a local update handler to execute the update.

2.1.4 Orca

The Orca distributed shared object system and language [1] focuses on optimizations for parallel programming. It uses a custom compiler to determine read/write semantics of operations, and makes object access and location decisions based on those

semantics. For instance, read-only methods can run simultaneously on multiple machines, but methods that perform writes require locking the object. Objects can be replicated, moved, or accessed remotely. Replica consistency is enforced by a totally ordered group communication [13]: all clients see updates in the same order. Updates are automatically propagated to clients, and updates to objects are sent as method signatures with parameters.

Orca has a number of elements that are useful for DCAs including object replication, an update-based consistency protocol, and global ordering for updates. However, Orca allows unsynchronized access only for read-only methods, but locks objects to execute methods that modify the object. These locks prevent other clients from simultaneously reading the object. Plateau's goal is to experiment with a weaker consistency model that will allow faster, unsynchronized access. Plateau avoids locks altogether for three reasons: (1) to eliminate the server state necessary to keep track of clients holding locks, (2) to improve state access speed, and (3) to prevent denial of service because a client couldn't acquire a lock, or because a remote client couldn't release a lock.

2.2 Prioritization Schemes

2.2.1 Quality of Service

The current Internet delivers one type of service, best-effort, to all traffic. In this model the network allocates bandwidth among all of the instantaneous users as best as it can and attempts to serve all of them without making any explicit commitment as to rate or any other service quality. A number of proposals have been made concerning the addition of enhanced services to the Internet. The rationale behind these proposals is that providing different guarantees improves performance and scalability, usually measured as user satisfaction. In differentiated service architectures, the resource being requested and allocated is bandwidth, and packet information tells routers what quality of service to apply to them.

Update Prioritization at the client side can be categorized as a similar approach, where the resources being allocated are the client's processor cycles. While quality of service architectures are applicable to much wider scale systems, eg. the Internet, it is interesting to note how the same approach, differentiating load based on some metric is used to improve performance and scalability. While one of the hardest problem in quality of service architectures is to provide a service that makes no assumptions about the type of traffic using it, in update prioritization as applied in Load Plateau, the application developer has the ability to specify application-specific rules that will be used to do message differentiation.

2.2.2 Priority Dropping for Layered Video

Prioritization techniques are used in Layered Video [17] to allow the network to selectively drop low priority packets when congestion is detected. Packets belonging to the base layer of hierarchically encoded video stream are marked as high priority while packets belonging to each successive enhancement layer are marked as successively lower priority. During times of congestion, the network preferentially drops the low priority packets, protecting the base layer from significant loss. Shifting loss away from more important and towards less important packets improves picture quality.

Fortunately, in Load Plateau, clients don't necessarily need to drop messages, as queues only need to hold messages for the number of clients that are currently connected in the DCA. Still, like priority dropping in layered video, Plateau clients rely on message priorities to service those that directly influence performance first. In an extreme case, update prioritization can also specify that certain messages need to get dropped, if, like layered video, some of the message data is only necessary to provide a better quality for the same specification.

Chapter 3

Update Prioritization

Update Prioritization is a strategy in which a message consumer queues update messages and then processes them according to their priorities. Priorities are usually assigned to the messages when they are inserted into the queue, and can change while the message is waiting to be dispatched inside the queue. The purpose of using prioritization in a DCA is to allow messages that are in a client's critical path with respect to performance to take precedence over messages whose delay will not degrade the performance of a client. Prioritization allows clients to effectively cope with the large number of activity updates caused by an increasing number of users in a DCA.

This chapter will describe how can DCAs take advantage of update prioritization to address scalability issues that result from relying on DSM to keep consistent states. It identifies the particular characteristics that a DCA should exhibit for update prioritization to be a viable solution, and points out potential pitfalls of using it.

3.1 First in, First out (FIFO) vs. Out-of-Order Processing

As described in section 1.2.2, DCAs using distributed state sharing rely on activity updates to keep a consistent common state. The conventional way in which activity

messages are processed in a DCA is using a first in, first out (fifo) approach. The advantage of using this approach is that updates are applied in the same order to the common state of all clients. This allows state consistency to be preserved in a straightforward way.

Our premise is that for certain DCAs, transient inconsistencies caused by out-of-order processing of updates are acceptable. As long as we can prove that at any particular point in time, updates can be applied so that clients converge to the same state, update prioritization will not compromise state consistency. Allowing out-of-order update processing by using prioritization, however, could improve performance by allowing messages that are in the client's critical path to be processed first. The definition of what is the client's critical path depends greatly on the nature of an application. An example will help illustrate its meaning.

In a video conferencing application, the interface could allow a user to display a subset of the people who are participating in the conference. The client will be receiving messages from all other clients, allowing it to render the image of the participants' face. If only a subset of the participants are being shown at one time in one client, performance would be increased if the client processes only the messages with information about the users that are being displayed. In this example, messages in the application's critical path are those with video information for the users that are currently being displayed. Other messages could be safely queued, or even thrown away, in the case of information like real-time video.

3.2 Potential Pitfalls of Out-of-Order Processing

There are a few risks that a developer must incur into when using update prioritization. First, proving that a particular communication protocol is correct will be much harder, as the order in which messages are sent is not necessarily the order in which they are received.

Prioritization also increases the complexity of a DCA, potentially making it harder to maintain and to extend. If a designer has a poor understanding of the consistency constraints of a particular application, using update prioritization will most likely compromise it. Shared objects might need to be locked in certain DCAs to make a series of operations look as an atomic one. For example, in a collaborative document editor, a person might be required to lock part of the document before editing it, so that other users can't concurrently change it. While locking is not supported in the main Plateau infrastructure, an extended message protocol can implement them. Unfortunately, implementing is not a trivial task, and should probably be handled by the platform instead of handed off as an additional taks that a developer needs to worry about.

The costs of using update prioritization are design tradeoffs that result from focusing on a strategy that will considerably improve scalability. Careful considerations need to be made to evaluate the relative importance of simplicity and consistency vs. performance and scalability. In general, prioritization is safe when objects are shared but still only one agent has the permissions to modify it. When objects are shared by several agents with permissions to modify them, it is harder to specify rules that will correctly preserve consistency, especially when concurrent object updates can occur. While locking is a useful strategy to preserve consistency when more than one agent can modify an object concurrently, Load Plateau does not provide the primitives to lock parts of the shared state. Locking is discussed further in the Future Work section (7.1).

3.3 When to use Update Prioritization

The following characteristics in an application make update prioritization a good candidate for increasing scalability.

1. The average number of update messages increases asymptotically faster than the number of users in the application. This growth breaks a DCA using distributed

state sharing since clients will get overwhelmed by the amount of updates that need to be processed.

2. Update processing accounts for a large part of the client's processing time, making performance directly dependent on it. If a client can't process messages at a faster rate than the rate at which messages are arriving, then a queue of messages will form. This is usually a good indicator that message processing is in the client's critical path.

3. Throughout the life of a client, it is necessary only to have a consistent subset of the common state. Real-time DCAs are usually heavily based on delivering views of the state to its users. In most cases, these views are incomplete representations of a common state. Therefore, for a particular view, it is only necessary for the data that produces that view to be consistent.

4. Out-of-order message processing is possible. Given the nature of DCAs, this is usually true and is also a consequence of the previous item. Of course, there could be specific ordering constraints between a subset of the messages. However, as long as there is some flexibility in the final ordering of all messages, performance improvements could be achieved. According to local considerations, each client could choose a particular processing order that fits it best. In some types of applications, however, processing updates in the same order throughout all clients could be a requirement for correct operation. This is usually the case for asynchronous DCAs, like process management applications or network file systems.

5. The rate at which clients receive activity updates varies over time. Given that most updates are triggered by human actions that need to be communicated to all participants, a variable update rate is generally the case in real-time DCAs. One of the effects that update prioritization aims at achieving is to distribute the load that a client bears over time. In this way, slow clients will not be overwhelmed when spikes of activity generate an unmanageable number

of activity updates.

3.4 Update Prioritization in Load Plateau

The Plateau utility relies on a message wrapper used to communicate activity updates between clients. It provides an implementation of a priority queue specialized to handle these messages, and abstract classes that allow developers who want to write their own implementation of a priority queue within the context of Plateau to do it.

Parallel to quality of service architectures in network design, when doing update prioritization, no generic prioritization rules will be applicable to all types of real-time DCAs. Therefore, in Load Plateau, the responsibility of specifying rules is delegated to the developer. Only the designer of the collaborative application has that knowledge. Factors like what the interface shows about the shared state and what each client needs to work properly are essential in choosing those updates that should be processed first. This approach follows the end-to-end argument in systems design [8]. Instead of trying to predict what updates are important at a lower layer in the design of the application, Plateau will allow developers who have much better insight about the nature of a specific application to define the rules by extending the Prioritizer interface.

The Prioritization utility was designed to be used as an incoming filter of update messages. However, it can also be used to dispatch outgoing messages at a rate that can be adjusted depending on the nature of the application. If more messages than the ones that can be sent given the outgoing dispatch rate need to go out, the queue will grow and priority rules specified by the application designer will be applied to determine the order in which outgoing messages leave.

Chapter 4

The Plateau Framework

The Plateau framework provides a series of base components for the creation of Java-based, real-time DCAs using a client/server architecture and distributed shared memory (DSM). It isolates developers from the details of networking and from implementing DSM, allowing them to concentrate on the client logic and user interface of the application. The framework also allows for a transparent integration of Plateau's update prioritization utility. The Java packages described in table 4.1 comprise the Load Plateau toolkit.

Table 4.1: Plateau Packages

plateau.server	Includes all the components in the client/server framework.
plateau.queue	Includes the update prioritization utility classes.
plateau.test	Contains a set of classes for testing the framework and the prioritization utility, and sample code, including the source code for Plateau Scribble.
plateau.util	Contains miscellaneous classes used to implement Plateau.

4.1 Main Features

4.1.1 Connection oriented Client/Server model

In a DCA implemented with Plateau, each user works at a client that connects to the Plateau server. The server's three main tasks are (1) Managing the initialization of new clients and finalization of leaving clients, (2) mediating message communication between clients, and (3) Logging status information. Section 4.2.3 describes the server design and functionality. Since a real-time DCA requires prompt notification of activity to all users, the framework is based on clients that establish a continuous connection to a server. In a connectionless model, the time it would require to set up a connection when activity is detected is prohibitive. Connections are socket-based and require that the Plaeau server is already running and listening for incoming connections. Disconnected activity is not supported in this first version, and the framework does not address recovery from a client crash or support for unreliable connections.

4.1.2 Communication based on Message Broadcasting

Distributed applications rely on a communication protocol between hosts to operate correctly. Developers creating a DCA with Plateau will need to extend the Message class to create an application specific communication protocol. In Plateau, low-level host communication is based on Java sockets writing and reading serialized Messages. While developers don't need to deal with the details of these operations, they need to be aware that all objects contained inside a Message subclass should be serializable.

4.1.3 State Sharing using Distributed Shared Memory

Clients share state by keeping a local instance of the CommonState object. Developers need to extend the CommonState class to represent for a particular application the state that will be shared by all clients. Upon connection, new clients receive an up-to-date instance of this object. To preserve CommonState consistency, clients follow

an atomic broadcast protocol based on State Messages, a particular kind of Message that signals all clients to modify the state. The protocol guarantees that update operations can be applied in the same order to all instances of the CommonState, allowing state copies to converge. It does not guarantee, however, that at a specific moment, all copies will be consistent.

4.1.4 Flow Control of Incoming and Outgoing Messages

In addition to the broadcast protocol, weak consistency constraints observed in most real-time DCAs can be exploited using update prioritization to address the scalability issues of DSM, as described in chapter 3. Transparent integration of the update prioritizer utility is achieved by managing outgoing and incoming messages in a client sender and receiver. The client sender receives messages from the local application and sends them to the server. It can hold components from the prioritization utility to control the order and rate at which messages get sent. The client receiver receives messages from the server and dispatches them to a local Update Handler. It can also hold prioritization components to control the order in which messages are dispatched.

4.1.5 Traffic Optimization

The server optimizes message traffic by queueing client messages and broadcasting them at a rate specified by the developer. Periodic broadcasting increases the server's performance by cutting down the number of write operations to a socket, especially when a large number of clients are sending messages at the same time.

4.2 Plateau Framework Design

This section describes the design of the plateau framework. The first section describes in detail the atomic broadcast protocol used to allow clients to keep consistent copies of a shared state. The next two sections describe the client and the server components in detail. All components are part of the java.server package. The javadoc for publicly

accessible classes can be found in Appendix A.

4.2.1 Data Consistency Protocol

Plateau relies on a distributed shared memory to give clients quick access to a common state, an important requirement for real-time DCAs. When a client connects to the server, it receives an up-to-date copy of a CommonState object that represents the data that all clients are sharing. To keep all the CommonState objects consistent, the following atomic broadcast protocol is used:

Instead of directly modifying the CommonState, clients first send a State Message to the server. The Message is received by the server and broadcasted to all the clients, including the original sender. Even when messages might be generated concurrently by clients, the server will broadcast them in a serial way, establishing an absolute order which will be the order in which each client will receive all messages. Each client owns an Update Handler that processes incoming Messages. When a client receives a State Message, the Update Handler modifies the CommonState according to the contents of the Message.

This protocol does not guarantee that at a specific moment, all clients will have exactly the same CommonState. The latency of each client connection will affect the time in which State Messages are received by each client. However, the protocol guarantees eventual consistency since the order in which the server broadcasts Messages will determine the exact order in which every client receives them. Figure 4-1 illustrates this broadcast protocol.

Clients must not modify the CommonState directly. While the CommonState has a defined interface with mutator methods, the only object that can call these methods is the UpdateHandler. The UpdateHandler owns the CommonState and modifies it inside its handleUpdate method. The method receives a State Message object as an argument, and modifies the CommonState according to the type of Message and the data it contains. Clients indirectly modify the CommonState by broadcasting

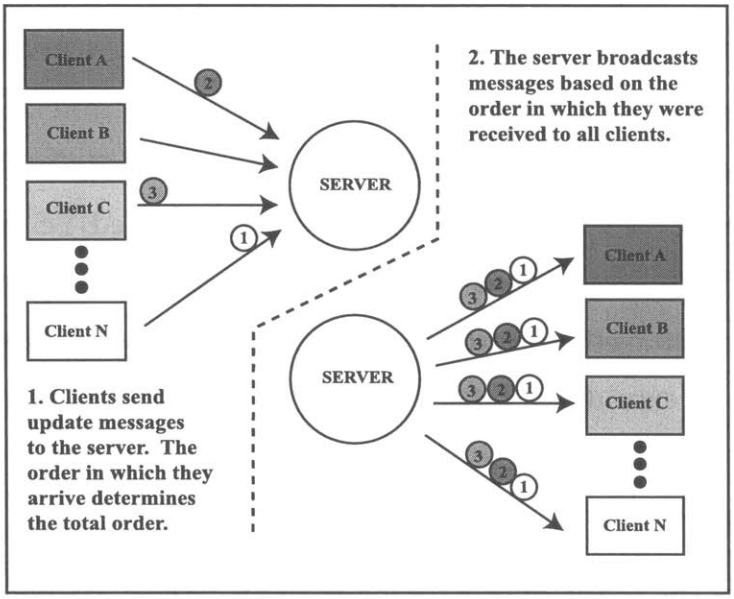

Figure 4-1: Broadcast protocol used in Plateau

State Message objects which get received by a ClientReceiver and then passed to the UpdateHandler's handleUpdate method.

To adapt this protocol to the needs of a particular application, a developer must take the following steps:

1. Define an application specific protocol based on Message objects. The developer can define different types by extending the Message into different subclasses or by extending the Message into one subclass that defines different Message ids. Data is passed inside an Object that gets passed to the Message in its constructor. The data can be accessed calling its getContents method. 4.2.1 describes the Message class in detail.

2. Extend the User class to represent an application specific user. The basic User only holds a unique id that gets assigned by the server.

3. Extend the CommonState class to represent an application specific shared state. The generic CommonState only holds the current users connected at the moment.

4. Extend the UpdateHandler to specify what operations should be done upon receipt of Normal Messages, and what CommonState modifications should be made upon receipt of State Messages. Typically, a different operation is done for each different type of State Message. The generic UpdateHandler adds and removes users as a consequence of the system Message.BYE and Message.NEW types. The developer must make sure to call the base UpdateHandler's super handleUpdate method so that this behavior does not get overwritten. It can, however, be extended.

Update Handler

Each client and the server owns an instance of an UpdateHandler. This class owns the CommonState, and defines how each Message will modify it in its handleUpdate method. At the client side, the Client Receiver (4.2.2) receives messages from the server and dispatches them to the UpdateHandler so that they get processed. At the server, the Sender (4.2.3) enqueues Messages into a StateKeeper after broadcasting them to all clients. The purpose of having an update handler at the server side is to keep a consistent copy of the CommonState that will be used to initialize new clients.

The basic UpdateHandler handles the Message.NEW type by adding a new User to the CommonState, and the Message.BYE type by removing a User. Developers should extend the UpdateHandler so that it also handles application specific Message types.

CommonState

The CommonState represents the state that is shared by all clients. It is owned and modified by an UpdateHandler. The basic class holds the Users that are currently connected in a hashtable indexed by a unique User id. The CommonState supports methods for adding, accessing and removing users. Developers should extend it to incorporate all other application specific data that needs to be shared in a DCA.

User

Users represent a person working at a client. They are used as the basic components of a CommonState and also identify the sender of a Message. The first thing a client does afer connecting to the server is to send a User object to the server so that it gets a unique id assigned. The server assigns a unique id and then broadcasts a Message.NEW message to all clients to let them know that a new user has connected. The User object is then added to the CommonState and sent back to the new client inside an UpdateHandler. Methods in the basic CommonState class allow developers to tell the local user apart from remote users.

Messages

A Message is used as the basic unit of communication between hosts in a DCA. As such, it is a serializable object that can be written and read from an ObjectStream using Java's object serialization. Its constructor takes an Object which will contain the actual application specific data used in the communication protocol.

Message properties are shown in table 4.2. Refer to the javadoc in appendix B for the specification of the methods used to access these properties. These properties allow the UpdateHandler to decide what actions to take and how to modify the CommonState. They are also used by a Prioritizer in the Plateau Prioritization Utility to assign message priorities (See section 5.1.2).

Typically, a communication protocol is based on different types of messages that get passed between hosts. The Message object has an id that can be set and read in order to differentiate messages types in the protcol. Developers can either use this property or extend the Message into different subclasses, one for each different message tpye used. The generic message defines two types which get handled by the generic UpdateHandler:

Message.NEW This message tells a client that a new user has joined. The contents of the Message is a User object. The UpdateHandler add the new user to the

Table 4.2: Message Properties

sender id	This property allows the original sender of Message to be identified. The sender id is the unique id of the User that is local to the client that broadcasted the message.
label	This is an auxiliary property that developers can use if they want a readable description or representation of the Message.
timestamp	This property is useful in applications where timing is important. If clients' clocks are synchronized, the timestamp can be used to define timing constraints and prioritization rules based on the time when a Message was created.
contents	Contains application specific data that needs to be sent with a Message. The Message constructor takes an Object argument which is set to be the contents of the Message.
id	Identifies the Message type. Message defines the Message.NEW and Message.BYE types, representing a new user and a user that leaves the application respectively. Developers should extend Message to define an application specific communication protocol based on Message type.
state	For the purpose of maintaining consistency, there are two classes of Messages: State Messages and Normal Messages. State Messages have the state flag set. An update handler is free to take any action as a result of getting a normal message, as long as the action does not modify the CommonState. Messages that trigger actions that will modify the CommonState should have the state flag set.
priority	Used in the Prioritization Utility. Represents the importance of a message. A low number represents a high priority. High priority messages will be dispatched to an Update Handler before lower priority ones 5.

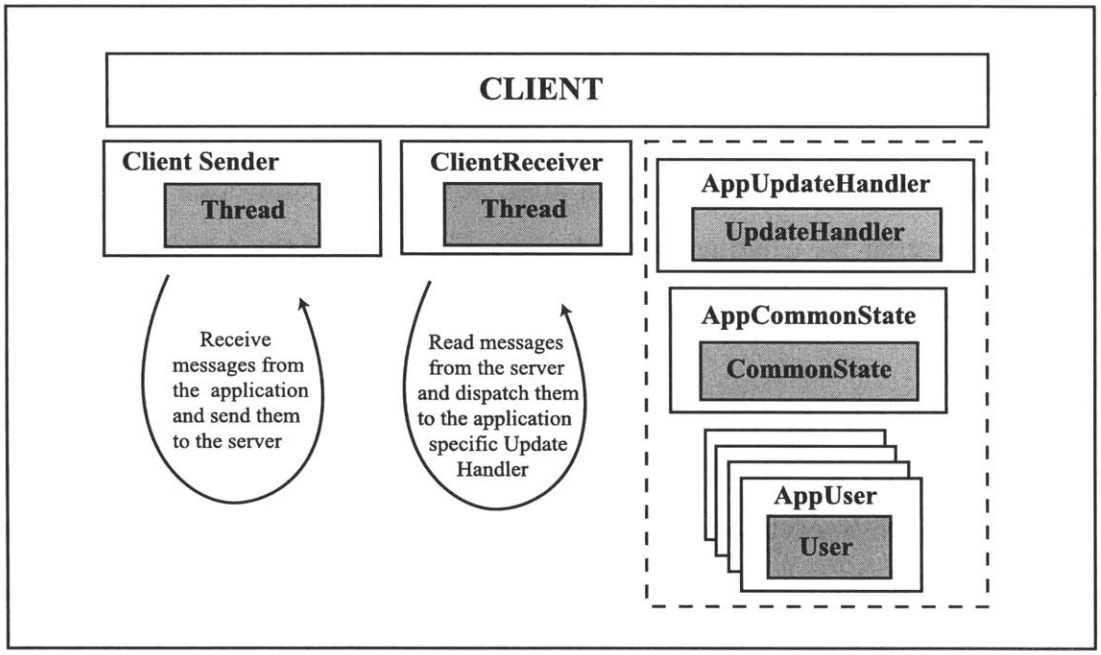

Figure 4-2: Client Architecture

CommonState.

Message.BYE This message tells a client that a user has left. The UpdateHandler uses the message's sender id property to remove the leaving user from the CommonState.

4.2.2 Client Components

The client components isolate the developer from the details of client-side networking. Still, most of them are base classes that a developer should extend to give them application-specific functionality. The client side components also offer the possibility of transparently integrating Plateau's prioritization utility.

Client

The Client class takes care of opening a connection with the Plateau server, obtaining an up-to-date CommonState, and managing incoming and outgoing messages. It

relies on the functionality of ClientSender and ClientReceiver to allow the local user to collaborate with users working at remote clients.

When a Client object is instantiated, it starts a connection with the Plateau server. In order for existing clients to be notified about the new client, the Client's start method must be called.

The client's broadcast method allows Messages to be sent to all the clients in the DCA, including the local client. The flow of outgoing messages can be controlled using the ClientSender class. Internally, a thread listens for incoming messages and places them in the ClientReceiver. The ClientReceiver can immediately dispatch the Messages to the client's UpdateHandler, or can rely on a MessageQueue to prioritize the Messages before they get consumed by the UpdateHandler.

ClientSender

The ClientSender is responsible for sending broadcast Messages to the server. Developers use this class to control the flow of outgoing Messages. Upon construction, a ClientSender can optionally take a MessageQueue argument. If no MessageQueue is specified, then the ClientSender will simply broadcast messages in the order in which it receives them. If a MessageQueue is specified, Messages will firt be queued, prioritized, and sent according to their priorities. Refer to section 5.1 for a description of MessageQueue and how the Plateau Prioritization Utility works.

Messages are passed to the ClientSender by calling its enqueue method. Enqueue can take a Message or a queue of Messages as an argument. If Prioritizaton is used to control the flow of outgoing messages, messages will be sent asynchronously: one thread will be placing messages inside the sender, while a different thread will extract and send them to the server.

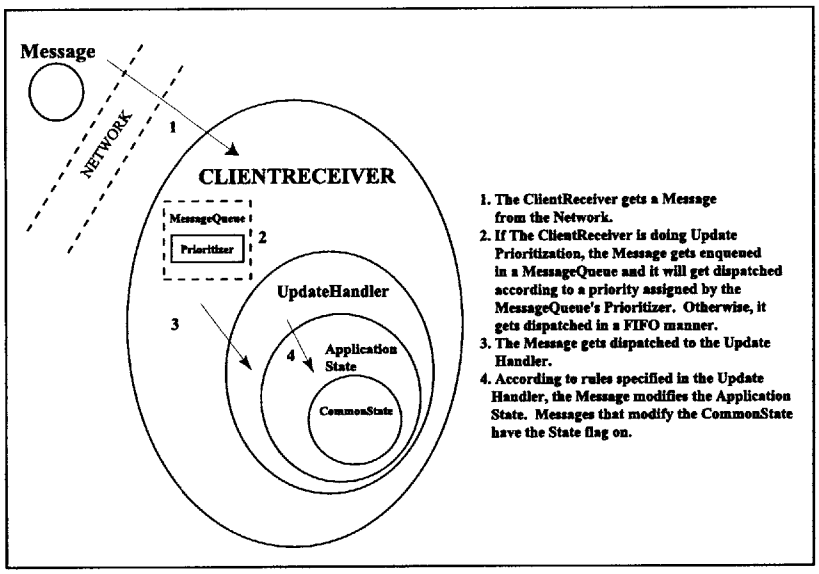

Figure 4-3: Message Flow in the Client

ClientReceiver

When the Client's thread receives a Message from the server, it will immediately place the newly received Message in a ClientReceiver. The ClientReceiver's task is to dispatch the incoming Messages to the client's Update Handler. Just like the ClientSender, the ClientReceiver can take a MessageQueue in is constructor to control the order in which incoming Messages get dispatched to the Update Handler. If a MessageQueue is not specified, Messages will be dispatched in the order in which the client receives them.

4.2.3 Server Components

The server components comprise a fully functional module that requires no additional extension to implement a full-fledged collaborative application. Parameters can be set to control the number of simulataneous connections the server will allow, logging behavior, and the rate at which messages should be broadcasted to clients. While the generic server offers no administrative interface, a few handles to internal components are offered with the purpose of allowing a developer to custom-craft an administrative

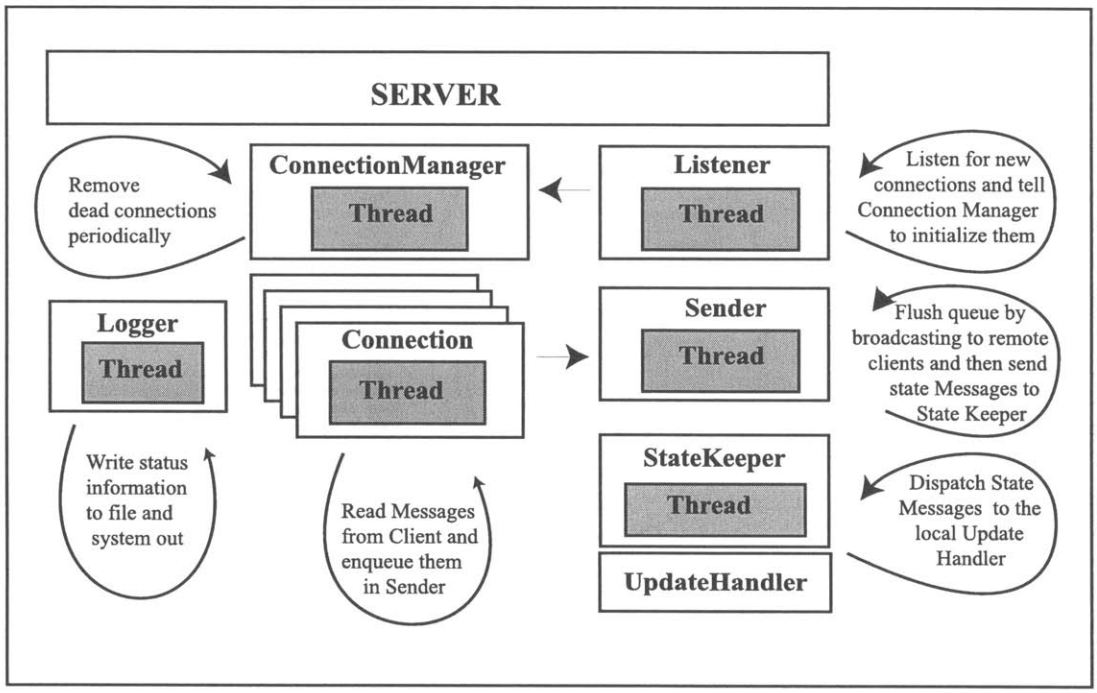

Figure 4-4: Server Architecture

interface.

We will describe the design using a top-down approach.

Server

The framework's main server class is plateau.server.Server. It is launched by invoking any Java interpreter from a command line. Like any other java application, the command line arguments are passed as String parameters to the main method. Server supports the following command-line arguments:

```
java plateau.server.Server [-sysout] [<port>]
```

- -sysout If sysout is included, all the status information that gets logged will also be written to the default system out stream.

- port specifies the port number where the server will listen for incoming connections. If no port argument is specified, the server will listen in its default port

which is 7878.

The server owns three components: A Logger, a Listener, and a ConnectionManager. If the server is launched from the command prompt, its startServer method gets called, and the three the three components get initialized and started. Server can also be part of the private data of an application specific server class. In that case, the constructor takes the same two arguments that are specified in the command prompt. However, the server's startServer method must be called after the server gets instantiated. Logger takes care of writting status information to a specified outputstream, which is generally a log file. Listener is a thread that waits for incoming connections, and then passes the newly created connection to the ConnectionManager. ConnectionManager receives incoming connections and creates a Connection object that will manage all communication between the new client and the server. Connections are socket-based and read and write operations are based on Java's object serialization. As described in 4.2.1, the basic unit of communication is the Message class.

Logger and Listener

Logger Logger is a threaded class that gets created by passing it a PrintWriter where all status information will be written and a flag that specifies whether the information will also be written to system out. The default PrintWriter is a file named after the date at which the Server gets launced. However, this default behavior can be changed by getting a handle to the Logger and setting a different PrintWriter. The System Out stream that Logger uses can also be changed at will. Status information is written by calling the Logger's println method.

By default, Plateau logs key events including when the server is launched and when connections are created and closed. Developers can extend this functionality by obtaining the Logger from the Server class and calling its println method as desired.

Logger also features support for periodic flushing. By default, PrintWriters au-

tomatically flush data when it is written to them. This behavior can be changed in the constuctor of a PrintWriter. If data is not automatically flushed, the Logger can be used as a thread that periodically flushes the PrintWriter. Like any other thread class in Java, it can be started, stopped, suspended and resumed. Its setPeriod method allows developers to specify the period at which the PrintWriter will be flushed. Flushing data periodically can increase the server's performance by reducing the number of costly flush operations executed.

Listener Listener allows the Plateau server to support multiple connections without blocking. To do this, it just sits inside a Server instance and waits for connection requests from remote clients. When a request is received, the resulting socket is passed to the connection manager's addConnection method so that an actual connection is created. Developers can't get a handle to the Listener since no additional functionality should be extended within this class. Listener owns a handle to the Logger instance to log the times when remote clients connect.

ConnectionManager

The main objective of the ConnectionManager, as its name suggests, is to manage all the concurrent connections to the server. When the server's listener receives a connection request, it passes the newly created socket to the ConnectionManager's addConnection method, which will create a new Connection and store it. At periodic times, the ConnectionManager will iterate through all the existing connections to make sure they are still up, and will remove the ones that are not.

The ConnectionManager also owns a Sender object which enqueues Messages received by all connections and periodically broadcasts them to all clients.

Finally, The ConnectionManager offers a series of accessor methods that are very useful for a developer who wants to create an administrative user interface for the server.

Connection The connection constructor takes care of the most critical server operation, which is to initialize a new client in a way such that its copy of the CommonState is synchronized will all the other copies. This task is difficult because while initialization happens, other clients could be broadcasting state messages. At some point, the server must stop updating its local copy of the CommonState to send it to the new client. All state messages that are recieved after this operation must be queued until the whole initialization process finishes, so that when they are broadcasted, the new client also receives them. 4.3.1 describes the implementation of this crucial operation.

Each Connection is a different thread that listens for incoming messages. Once a connection is created, the thread is started, and every time that a message is received, it is placed inside the Sender's message queue. Enqueueing a message is an extremely fast operation, so the connection thread will only be blocked for a fraction of a second before it can receive the next message. This technique can drastically improve the server's performance when clients broadcast messages at a very fast rate.

Sender and State Keeper The Sender object is a thread that queues Messages received by all Connection instances. All connections concurrently place Messages in the Sender, and by queueing them up inside, a total Message order will be established. The order in which Messages are queued in the server will be the order in which Clients receive them. Periodically, the sender will flush its queue and broadcast it to all connected clients. Periodic broadcast is done in order to minimize the number of socket writes, an operation that is computationally expensive.

After broadcasting messages, the sender iterates over them and places state messages inside the State Keeper. The State Keeper is an instance of the ClientReceiver class which lives locally in the server. Exactly like in the client side, the StateKeeper will dispatch messages to an UpdateHandler that will process them and modify the CommonState depending on the Message type.

4.3 Implementation Details

4.3.1 Client Initialization

Perhaps, the most critical operation throughout the life of a real-time DCA implemented with Load Plateau is client initialization. While existing clients might be continuously sending state messages that need to be broadcasted and reflected in the server's local Common State, the server must send a copy of the Common State to the new client, and make sure that all state messages are either processed by the StateKeeper before the copy is sent, or broadcasted to the new client after initialization has completed.

To guarantee that all messages are either (1) processed by the server's StateKeeper or (2) sent to the new client after initialization, the following initialization protocol is used.

1. A client requests a connection to the server.

2. The server's listener gets this requests and forwards it to the Connection Manager. If the maximum number of connections has been reached, the request is declined and the client is informed. Otherwise, it sends an "Initialize" message if this is the first client to connect, or a "Success" message if other clients are already connected. At this point, a connection is instantiated. The connection thread takes care of the rest of the client initialization, allowing the server's Listener to continue waiting for new connections.

3. The client sends a serialized User. When the server receives it, it assigns a unique id to the User object, and sends it back to the new client.

4. If the client received an "Initialize" message, it sends a copy of the application specific UpdateHandler. This UpdateHandler holds the Common State, so the server now has a copy of the state, and the class that holds the rules for updating it. At this point, the client has been successfully initialized, and can start

broadcasting messages. The server will assign the new UpdateHandler to its State Keeper. The server will time-out and the new connection will be disposed of if the client delays its sending of the Update Handler.

5. If the client received a "Success" message, it waits for the server to send an up-to-date copy of the CommonState.

6. The server locks and pauses the sender thread, inserts a message of type Message.NEW into the sender, flushes all messages in the sender's queue, and finally unlocks the server without resuming the thread.

7. Since the server paused the sender thread before flushing messages, each connected client will receive messages up to the one that announces the new client. Any message placed in the sender after it is unlocked, will wait in queue until the server resumes the sender thread.

8. The server waits for all flushed messages to be processed by the local Update Handler, and then sends a copy of the common state to the new client. At this point, all clients, including the new one have processed all Messages up to the one announcing the new user. The new client has successfully initialized, and can start sending messages to the server.

9. After the server sends the up-to-date common state to the new client, it starts the new connection thread, adds it to the ConnectionManager and resumes the sender thread. Messages received after the new user was announced are now broadcasted to all clients, including the newly initialized one.

Chapter 5

The Plateau Prioritization Utility

Load Plateau provides a Prioritization Utility in the plateau.queue Java package. The utility integrates transparently with the framework for building DCAs, and allows clients to control the order in which outgoing messages get sent and incoming messages get processed by the Update Handler. The package can also be used as a stand-alone utility for implementing update prioritization in applications that are not based on the Plateau framework, as long as the application uses Message objects (??) as the basic unit of communication between clients.

Developers using the prioritization utility only need to implement a Prioritizer, the class that holds the rules for assigning priorities to Messages. The utility implements the details of storing the messages and dispatching them to a consumer in the correct order (in the case of Plateau, the message consumer is an Update Handler). This is done via a MessageQueue, a priority queue that orders and dispatches messages according to their priority. Plateau also provides a mechanism for changing the priority of messages after they have been enqueued in a MessageQueue but before they have been dispatched to a consumer.

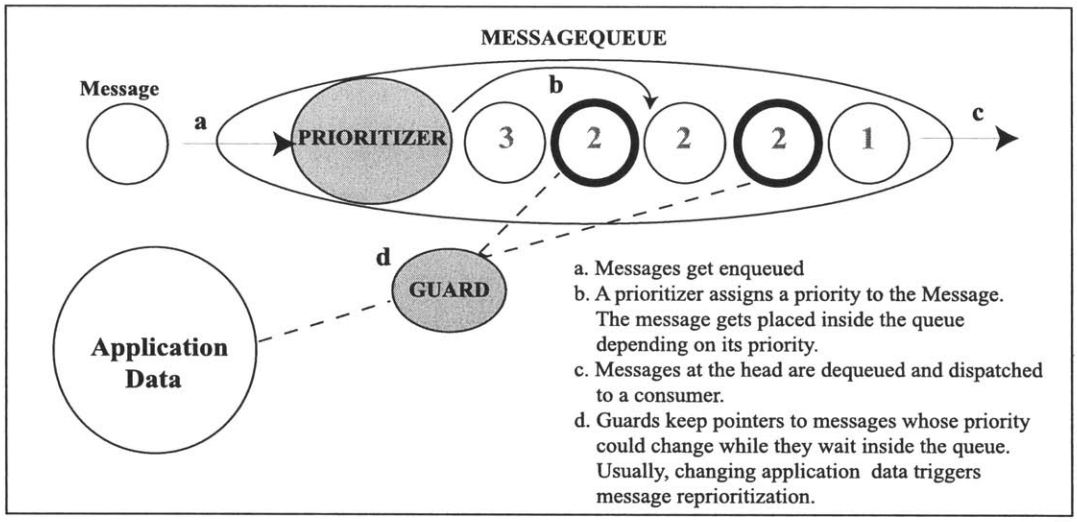

Figure 5-1: Message Flow in the Prioritization Utility

5.1 Design

The plateau.queue package is based on four components: Messages, Prioritizers, MessageQueue and Guards. Messages are the basic unit of communication between clients, and are also used in the plateau framework to keep a consistent common state between all the clients of a DCA. Prioritizers assign a priority to each Message. The MessageQueue is the object that holds and orders messages according to their priority, assigned by a Prioritizer when the message gets inserted into the MessageQueue. Guards are objects that tell a MessageQueue to reorder certain messages when an event makes their priority change.

The queue package also includes an implementation of MessageQueue named IndexedQueue 5.2 which uses queue indexing in order to provide fast relocation of Messages if their priority should change while they are inside the queue. 5.1.4 explains reprioritization in detail. [1] The Javadoc for the top-level classes of the prioritization utility can be found in Appendix B.

[1]For a description of implementation strategies used to build fast priority queues see [9]

5.1.1 Messages

A Message is used as the basic unit of communication between hosts in a DCA. As such, it is a serializable object that can be written and read from an ObjectStream using Java's object serialization. Its constructor takes an Object which will contain the actual application specific data used in the communication protocol.

For the purpose of the prioritization utility a Message can be treated as a generic object with application specific data that will allow a prioritizer to assign a priority to it. Refer to ?? for a full description of the Message class.

5.1.2 Prioritizers

Prioritizer is the interface that developers must implement to specify rules for prioritizing messages when they get enqueued into a MessageQueue. It defines the following methods:

- int getPriority(Message m): Returns the priority of a Message m. A developer defines the rules used to prioritize messages implementing this method.
- void setParent(MessageQueue m): When a MessageQueue gets created, it sets itself as the parent of a Prioritizer. This restricts a Prioritizer so that it can be used at most by one MessageQueue. The reason behind this decision is that in some occasions, developers will want to assign priorities based on the state of the queue where a Message is being placed. If two queues are using the same Prioritizer, a complex approach would be needed to let the prioritizer know what queue Messages correspond to.
- MessageQueue getParent(): Returns the MessageQueue object that is using the Prioritizer.

MessageQueues are created by passing them an instance of a Prioritizer. When a MessageQueue operation needs to figure out the priority of a Message, it calls the getPriority method of Prioritizer. Usually, information within the message will let

the Prioritizer decide what priority to assign. However, a Prioritizer can also point to application data that influences how messages get prioritized.

5.1.3 Message Queues

Load Plateau relies on a priority queue to hold update messages sent by other clients. A normal queue, just like in the real world, is a structure that holds a set of elements, and that uses a FIFO (first-in firt-out) policy to insert and remove elements from it. The queue has a head and a tail. When an element is inserted, it takes its place at the tail of the queue. The element removed is always the one at the head of the queue. Priority Queues are queues where elements get ordered depending on their priority. When the elements gets enqueued, they take a position in the line of elements depending on their priority. Elements still get removed from the head of the queue.

Prioritization is achieved by placing a Message inside a priority queue after it gets read from the network. Messages will then be processed by extracting messages from this queue instead of processing them directly after they are received. Message priorities are assigned by a Prioritizer (5.1.2) when they are enqueued into a Message Queue.

Plateau defines the MessageQueue abstract class. This class defines the methods that a MessageQueue must implement to become a holder of prioritized messages in Plateau. Subclasses of MessageQueue must be created passing a Prioritizer which will assign priorities to Messages that are enqueued.

The key methods that MessageQueues must implement are described next. For a complete specification of the MessageQueue abstract class refer to Appendix B.

- void enQueue(Message m): Will take the message, pass it to the prioritizer to get a priority, and then place it inside the queue depending on the priority.
- Message deQueue(): Will extract and return the Message placed at the head of the MessageQueue.

- void renqueueAll(): Will reprioritize all messages.

5.1.4 Guards

Once a message has been enqueued, it is possible that its priority could change before it gets dispatched by the MessageQueue. Plateau allows developers to assign a Guard to a Message. By doing so, the message's priority can be updated by calling the guard's update method. The update method will make the queue restructure itself to reflect the new position of the Message given its new priority.

Guarding is achieved by making queue Observable objects. The Observable and Observer classes are defined in the java.util package. The guard's update method will call the MessageQueue's udpate message, which should then restructure itself in an implementation specific way. IndexedQueue uses Message indexing to implement a fast restructuring operation 5.2.

Guards can be assigned to a specific message, to a set of messages, or to the whole queue.

5.2 Implementing Fast Priority Queues

The Prioritization utility includes an implementation of a MessageQueue called IndexedQueue. IndexedQueue offers fast restructuring operations so that the order in which messages are kept is quickly updated after a Guard's update method has been called.

IndexedQueue relies on a specialized subclass of Message named IndexedMessage. IndexedMessages get invalidated when their priority changes, and a copy of them gets enqueued so that they take the correct place in a queue. When an IndexedMessage gets dequeued, IndexedQueue makes sure that it has not been invalidated. If it has, then it disposes of the message and calls itself again until a valid message is found.

IndexedQueue is a prioritized queue that holds Messages with a fixed number of

integer priorities. The number of priorities is specified at construction time, and it will not change throughout the life of the object.

Internally, an IndexedQueue is an array of standard queues. A queue's index represents the priority of Messages stored in it. For example, the queue in index 4 of the array only holds Messages of priority 4. When an IndexedMessage gets enqueued, it gets placed in the standard queue with an index that corresponds to its priority. To dequeue a Message, the IndexedQueue dequeues a Message from the lowest index queue that is not empty.

When an IndexedMessage priority changes, the message is marked as invalid. A copy of it is made and enqueued again. This means that two or more copies of the same Message might be stored in the IndexedQueue at the same time. However, only one copy will be marked as valid. All invalid copies that are dequeued will be disposed of, and only valid copies will be exposed to a Message consumer.

5.3 Visualizing Priority Queues

The prioritization utility includes the GraphicQueue class, which allows MessageQueues to be displayed graphically on the screen. Refer to Appendix B for the full specification of GraphicQueue.

GraphicQueue will be very useful when developers need to debug prioritization schemes, as they will be able to visualize the message flow inside a priority queue. The only requirement necessary to visualize Messages is that they implement the Colorable class, which defines the getColor method. GraphicQueue relies on getColor to know what color to use when displaying messages in a Panel.

5.4 Summary

The four components of the java.queue package allow great flexibility in terms of how Messages get prioritized, and how the Queues get updated whenever outside

conditions make priorities change. Depending on the type of application, a developer might be interested in a MessageQueue that can update the priorities of all the messages in a queue in a very efficient way, or that only enqueues and dequeues messages fast. If the implementations provided by Load Plateau do not satisfy a developer's needs, then it will be possible for him to write his own implementation to fulfill his own requirements.

Chapter 6

Validation

This chapter describes how Load Plateau was used to implement two real-time DCAs: Plateau Scribble and Chat Circles. Plateau Scribble is a shared whiteboard where several users can collaboratively draw pictures while chatting with each other through a text-based interface. It is a simple, proof-of-concept application that illustrates the main ideas behind building a collaborative application using the toolkit. Chat Circles is an abstract graphical environment for synchronous conversation. Scalability and performance problems in the initial prototype of this research project motivated the initial development of Load Plateau. By solving those problem in a second Chat Circles implementation, Load Plateau's approach is validated as a viable one.

6.1 Plateau Scribble

Plateau Scribble is a shared whiteboard that allows users to draw pictures in it and communicate with each other using a text-based chat interface. The whiteboard is a large panel, and only part of it is visible to each user at once. Users can click and drag the mouse to scribble on the board, and can scroll a panel to see different parts of the board. Scrolling will make different users see different parts of the whiteboard at any given time. Appendix C contains Plateau Scribble's complete source code.

Since Plateau Scribble is built using the Plateau Framework, clients rely on Scrib-

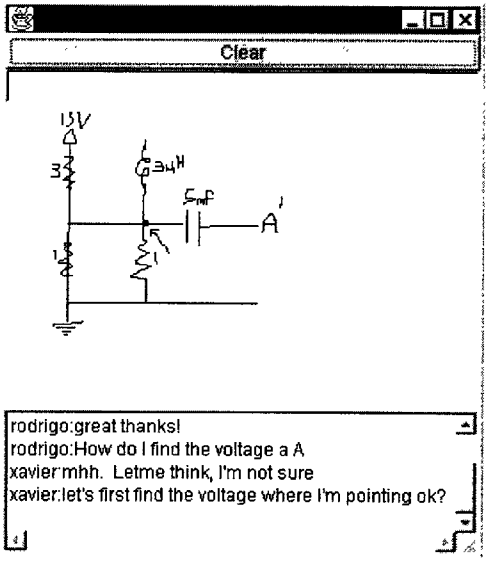

Figure 6-1: Plateau Scribble includes a shared whiteboard and a text-based chat interface.

bleMessage and ScribbleUser in order to communicate with each other. ScribbleMessage extends the basic Message class, and defines the following types:

ScribbleMessage.DRAG Contains a Line object, which represents a line drawn in the panel. The Line data structure contains a starting Point and ending Point. When a user drags the mouse inside the whiteboard, Drag messages are broadcasted so that all clients receive the scribbles drawn. Drag is a State Message, and it will therefore modify Plateau Scribble's common state.

ScribbleMessage.CLEAR There is no specific data in the clear message. It just signals that a user wants to clear the whiteboard. Clear is a state message and will modify the common state.

ScribbleMessage.SAYS Contains a String that represents a comment by a user. Says is used to implement the text-based chat interface. It is a normal message, so it will not modify the common state.

ScribbleUser extends the basic User class by including a name for each user.

6.1.1 Plateau Scribble's CommonState and UpdateHandler

Since all users need to share the whiteboard, the CommonState class is extended into ScribbleBoard. In addition to holding a table of current users, ScribbleBoard holds a DrawCanvas. DrawCanvas is a data structure that extends the java.awt.Canvas class. It holds a vector of Line objects that get painted inside the Canvas when DrawCanvas is painted. The two main mutator methods that ScribbleBoard offers are:

void add(Line line) Adds a line to the DrawCanvas so that it gets painted when the DrawCanvas is displayed. If the DrawCanvas is currently being displayed, the line gets painted immediately.

void clear() Removes all lines from the DrawCanvas, clearing it if it is currently being displayed.

At this point, the design strategy should be obvious: Whenever a user drags the mouse, DRAG messages will be broadcasted to all clients. When a client receives a DRAG message, its Update Handler will extract the line object and add it to the ScribbleBoard so that it gets displayed inside the Canvas. When a user clicks on the clear button, the CLEAR message will be broadcasted, prompting each client's UpdateHandler to clear the ScribbleBoard. When a user types a message in the textbox and then presses enter, a SAYS message will be broadcasted. The UpdateHandler will look at the message's sender id to figure out who sent the message, will query the ScribbleBoard to find the name that corresponds to that id and will display the message in the chat area, specifying who is the author.

6.1.2 The Scribble Class

Scribble is the top level client class. Its constructor takes the host name and port that it should use to connect, the name of the user, and a boolean flag that specifies whether the class is being created by an Applet or is being launched as a stand-alone application. Upon initialization, it instantiates the common state (ScribbleBoard), a prioritizer and message queue that will be used to perform update prioritization 6.1.3,

and a ScribbleUpdateHandler. With those, it creates a ScribbleUser, a ClientSender, and ClientReceiver, and finally initializes and starts the Client class.

At that point, Scribble will be connected to the server, and any messages broadcasted by remote clients will be received by the ClientReceiver and passed to the ScribbleUpdateHandler. The Scribble class contains the logic that will broadcast different ScribbleMessages according to the actions of the user.

6.1.3 Update Prioritization in Plateau Scribble

Imagine that the shared whiteboard is used by dozens of people at the same time. Even when a user does not see but a small part of the entire board, ClientReceiver will be receiving drag messages that update all parts of the board. If the number of users is very large, drag messages will begin queueing up at the ClientReceiver, and the delay between a user dragging the mouse and the board showing the new scribbles will be too long.

Obviously, not all users will be watching the same part of the board at the same time. Instead, users might start clustering in different parts of the board, working on different activities. In that case, to improve the performance of Plateau Scribble, we can use the update prioritization utility to give precedence to the drag messages for the part of a client's board that is visible at the time. If the user scrolls to some other part, then those messages that update the new visible part will get high priority.

Plateau Scribble performs update prioritization on messages in the way described in the previous paragraph. It implements a ViewPrioritizer that compares the coordinates of the line contained in a drag message with the coordinates of the area that is visible at the time. If any part of the line intersects the visible area, the message will get a priority of one. If the line does not intersect the visible area, the message will get a priority of two. Any other message gets a priority of one. The ClientReceiver holds an IndexedQueue, which at the same time holds the ViewPrioritizer. Whenever a Message is received, it is queued inside the IndexedQueue so that high priority mes-

sages are dispatched to the UpdateHandler first. In the case of PlateauScribble, all messages have the same priority but drag messages that belong to an area that is not visible at the time. This strategy will drastically reduce the delay of drag messages for a visible part of the board when a large number of users are connected at the same time.

It might seem that guarding messages with priority two is a good idea. In that case, when a user scrolls to a different part of the board, the guard's update method would be called and all priority two messages would get reprioritized. Those messages that become visible after the user has scrolled would get priority one. In the case of Plateau Scribble, the computation it takes to reprioritize messages is not justified by the improvement in performance that might result from guarding messages. Only those messages that are within the queue at the time when a scroll occurs will experience a slight delay. All subsequent messages will get correctly prioritized.

6.2 Chat Circles

Chat Circles is graphical interface for synchronous conversation. Color and form are used to convey social presence and activity, and proximity-based filtering is used to intuitively break large groups into conversational clusters. The system also features an integrated history interface, which visualizes archival chat logs. [1]. After a prototype implementation of Chat Circles was developed in the summer and fall of 98, it remained a challenge to design a robust and scalable system that incorporated the ideas put forth by the Chat Circles vision.

While the prototype's interface is adequate, the amount of information each client receives and needs to process limits the number of users that can be using Chat Circles without experiencing prohibitive delays in response time. There are two main reasons why the prototype implementation of Chat Circles does not scale well in terms of the number of users it can support.

[1]For a complete description of Chat Circles refer to [20]

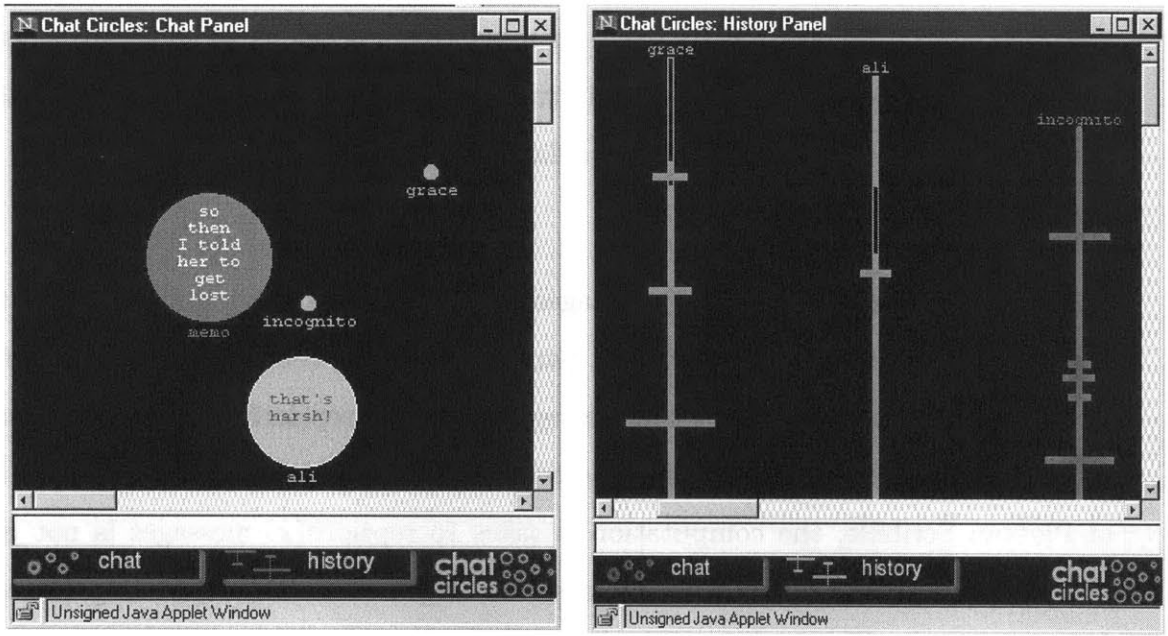

Figure 6-2: Chat Circles features a chat interface and a history interface for visualizing conversations.

First, chat programs are an example of collaborative application where the amount of per-user activity grows as the number of users increases. Since the interface requirements of Chat Circles require every client to receive all information about each user's activity, the amount of activity updates each client needs to process grows faster than linearly with respect to the number of users. Second, Chat Circles is written in Java, a programming language that still suffers from a slow runtime environment. Since the application is mostly a graphical environment, processing activity updates involves extensive graphic manipulation from each client, especially when the number of users is large. As described in section 3.3, these two characteristics hurt scalability considerably, calling for a technique like update prioritization to solve the problem.

Using Plateau to build a Chat Circles system that does not incur into performance penalties due to the requirements of the interface was a great way of testing the validity of the Plateau approach.

6.2.1 The Chat Circles Prototype

The initial implementation of Chat Circles experienced the following problems:

- After more than a dozen users logged into the system, it became slow and unresponsive, making it virtually unusable.

- Client initialization was not properly implemented. If the client crashed during initialization, or if too much activity was experienced at the moment, the server would hang. This symptoms indicated a potential flaw in the concurrency control of the server, and lack of failure tolerance, as the server should never hang if a client fails at the middle of initialization.

- At several points in the message flow, race conditions among threads trying to use the same resources made the server perform at suboptimal conditions.

- Beyond performance and scalability problems, the text-based communication protocol made it very difficult to maintain and extend the application's functionality.

6.2.2 The Beta Version of Chat Circles

The first step in the process of making a more scalable version of Chat Circles was to redesign most of the client in order to separate interface functionality from network functionality. Once that was done, the server framework of Load Plateau was used to build the server and the network layers of the client side. This improved the overall performance greatly, making use of the message traffic optimizations of Load Plateau, and fixing the concurrency control problems that would make the server hang. Finally, update prioritization was used to solve the scalability problem. While there are still performance issues related to costly graphical operations, the new version can support an increasing amount of users without crashing and without overflowing client queues with update messages.

Communication Protocol

Instead of using a text-based communication protocol, the protocol was redefined to make use of Plateau's absolute broadcast protocol by extending the basic client side components of Load Plateau. In the beta version of Chat Circles, the following client side components are used:

CCMessage extends Message, defining two message types: CCMessage.MOVE and CCMessage.SAYS.

CCUser extends User, adding properties like color, position and name.

CCCommonState extends CommonState. The shared state of plateau only includes the basic vector of users that are currently connected. Since each user knows its own position and color, the whole Chat Circles world can be displayed to any client.

CCUpdateHandler extends UpdateHandler. It defines exactly what modifications to the commonstate should be made as a result of receiving MOVE or SAYS messages.

The new design allows for great flexibility and extensibility. Several research ideas that were hard to incorporate in the prototype version will be easily introduced into the new version of Chat Circles.

Update Prioritization

Chat Circles, like Plateau Scribble, only shows part of the common state to each user at a time. As a consequence, not all state updates will be immediately apparent to a user. Chat Circles maintains an up-to-date interface by refreshing the graphics every 100 ms. The following steps are required to do this:

1. A timer triggers the refresh process every 100 ms.

2. The program iterates through each user, painting it in an off-screen graphics buffer. The user object has information about the position, color, size and contents of its graphic representation.

3. The graphic buffer gets painted into the visible panel.

We realized that there was great room for improvement based on the refresh process. First, only those user objects that are currently visible should be painted in the off-screen buffer. Second, only the visible part of the off-screen buffer should be painted in the panel. Third, if only the visible part of the whole Chat world is being refreshed, we should prioritize both MOVE and SAYS messages and give precedence to updates on parts of the common state that are currently visible to the local user.

To implement these changes we introduced two buckets in the client side code. At every moment, the main bucket holds users that are currently visible while the secondary bucket holds the rest. Buckets get updated when a user scrolls to other parts of the Chat world. Finally, the refresh process was modified so that only users in the main bucket would be considered.

In addition, we created a CCPrioritizer class that queries the buckets when broadcast messages are received. All messages are assigned a priority of 1 except for:

- MOVE messages from users that are in the secondary bucket get a priority of 2.

- SAYS messages from users that are in the secondary bucket are preprocessed so that the comment timer starts ticking. After that, they also get a priority of 2.

We expect the changes to make a large impact in the number of concurrent users that can be logged at the same time. While the initial prototype slowed down considerably after ten users where connected, we are aiming to support fifty or more users in the beta version. The improvement in the server efficiency and communication protocol already make the overall operation much smoother.

In the future, we want to experiment with three priority levels: One for users within the local user's hearing range, one for visible users, and the last one for the rest. Beyond improving the overall responsiveness of the system, prioritizing messages from users within the local user's hearing range will improve the response time for messages in each particular conversation. The challenging part of implementing this idea will be to have fast operations that update the contents of each user bucket as different users move around and scroll about the chat world. If these operations are computationally expensive, they will cancel out the gain in performance obtained from using three priority levels.

Chapter 7

Conclusion

As seen in Chat Circles, the prioritization utility helps developers exploit their unique knowledge about the dynamics of a real-time DCA. Usually, developers will not have enough a priori knowledge that will help them devise the best set of rules to do prioritization. Allowing them to first prototype the application using the Plateau framework, and then to introduce the prioritization rules based on performance and message flow analysis minimizes the possibility of introducing problems due to out-of-order message processing. The graphic queue visualizer is a very useful resource to understand and debug real-time DCAs after prioritization has been introduced.

In addition, using the Load Plateau framework forces developers to think clearly about the communication protocol that will allow clients to share information while keeping it consistent. This contributes to the creation of DCAs with clean designs, and allows for easy maintenance and extensibility of the application's functionality. Load Plateau also helps developers cut down on development time by providing a series of base components that implement generic DCA behavior and a server framework that takes care of efficiently broadcasting messages to all clients.

While update prioritization is a powerful strategy when performance and scalability are important, developers should carefully analyze the consistency constraints of the shared state they wish to implement, and weigh the relative importance of performance and scalability versus simplicity and consistency. Update Prioritization

does create more complex applications to debug, as it makes an application less deterministic by dynamically changing the order in which messages get processed. Also, in particular cases, the timing dependencies between messages will force them to be processed in strict FIFO order. In those cases, update prioritization will compromise consistency, and possibly prevent applications from operating correctly.

7.1 Future Work

Most of Load Plateau's shortcomings are a result of its lack of maturity as an established framework with a rich set of features. The following is a list of missing features that are worth exploring further:

Disconnected Activity Constant connections are required for operation. While most real-time applications require constant connections due to the bounded network delay they expect, it is sometimes desirable to allow a client who has disconnected to connect again with the possibility of synchronizing with the rest of the clients. We could accommodate disconnected operation through the use of queuing strategies: Updates could be placed in a queue when there is no connection. When a connection is made, the client and server would exchange the latest pending updates.

Unreliable Connections Support for unreliable connections would allow clients that disconnect for a short amount of time to recover all relevant information in order to synchronize again with the rest of the clients. This feature would increase the robustness of DCAs implemented with Load Plateau, as not all users have the luxury of getting high reliability connections from their Internet Service Providers. Unreliable connections can persist if the server buffers the client unique id for a short amount of time after it has disconnected. When the client connects again, the server would try to match the initialization data with its buffer of recently disconnected clients. A few undesirable consequences from having the server cache data for recently disconnected clients need to be

taken care of. For example, a malicious client could place a denial of service attack by intermittently connecting and immediately disconnecting, overflowing the server's memory buffer without necessarily reaching the maximum allowed number of connections enforced.

Crash recovery Load Plateau does not deal with crash recovery and therefore, shared state is not recoverable if a client crashes in the middle of operation. Crash recovery is harder to implement than support for unreliable connections, as the whole state needs to be recreated after the client goes down. It is arguable whether crash recovery in a real-time DCA is necessary or not, as in most cases, there will be no difference in service quality if a client starts over again or recovers from a crash.

Session-scoped Activity In the current version of Load Plateau, a server can only support one session at a time, which means that only one application specific CommonState is being kept by the server at any point in time. The session is considered created when the first client connects. The session dies after the last client disconnects. After the last client disconnects, the server cleans up its state, and requests a new one only after the next remote client connects. We could take advantage of Load Plateau's architecture to let the server support different applications, each within its own session at the same time. To do that, the server could provide services at different ports, each supported by its own Listener, Connection Manager, Sender, and Logger.

Automatic Server BootStrap A Plateau server needs to be up and running before any client attempts to connect. Plateau could allow DCAs to start even if no server is running by using Java's RMI to instantiate a server if a client requests its service. This feature is particularly useful when server hosts run several services and need to allocate resources only to the currently active services. Also, it relieves administrators from the task of making sure that the service is up at all times.

State Locking Plateau avoids locking techniques as its premise is to exploit weak consistency to improve performance and scalability. However, as discussed in section 3.2, locking is essential in DCAs where a series of operations by a client should look as one atomic operation to the rest in order to preserve state consistency. While the developer can incorporate locking into the application specific communication protocol, Plateau could take part of the load off the developer's shoulders by providing generic locking mechanisms that developers can extend in their applications if they deem locking as an essential requirement.

Priority Queues The Prioritization Utility includes one implementation of the MessageQueue class: IndexedQueue. IndexedQueue focuses on providing fast reprioritization mechanisms, and is targeted at applications where Guards will be heavily used. The trade off from using IndexedQueue is that the number of priorities used is fixed, and needs to be specified when IndexedQueue is instantiated. Some DCAs might need more flexibility in the number of priorities available, and in those cases, a developer needs to work on its own implementation of a MessageQueue. Plateau could provide more MessageQueue implementations, each with different dynamic characteristics. This would give developers more flexibility to devise prioritization strategies targeted to the particular needs of the DCA they are working on.

7.2 Summary

This thesis has explored the design of a toolkit that supports the creation of real-time DCAs with emphasis on improving performance and scalability. While other tools exist to support collaboration software, most work focuses on the sharing of state and preservation of state consistency. Load Plateau is unique in the field, as it introduces the idea of update prioritization as a strategy that exploits weak consistency constraints on a shared state in order to address performance and scalability problems.

Appendix A

plateau.server

Package Contents *Page*

Classes

A.1 Classes

A.1.1 *Class* User

User represents a generic user in a collaborative application. Users are kept and referenced by a unique userid in the CommonState that all clients keep. Extend User to make it specific to a particular collaborative application.

Declaration

public class User
extends java.lang.Object
implements java.io.Serializable

Constructors

- *User*
  `public User( )`

- Description
  - Constructor

Methods

- *setUserID*
  `protected void setUserID( long id )`
  - Description
    - Used by the plateau server to assign a unique id to a user
- *getUserID*
  `public long getUserID( )`
  - Description
    - Returns the unique id assigned to the user.

A.1.2 *Class* ClientReceiver

ClientReceiver manages Messages received from the network, and dispatches them to an application's UpdateHandler either in a FIFO way, or based on message priorities if a MessageQueue is being used to perform update prioritization.

Declaration

public final class ClientReceiver
extends java.lang.Thread

Constructors

- *ClientReceiver*
  `public ClientReceiver( plateau.server.UpdateHandler uh,`
  `plateau.queue.MessageQueue Mqueue )`
  - Description
    - Creates a new ClientReceiver. uh is the UpdateHandler used by the receiver. If Mqueue, is null, messages are passed in the order in which they are received to the UpdateHandler. Specifying and Mqueue, allows the developer to use a priority queue to control the order in which received messages get processed.
  - Parameters
    - uh - the UpdateHandler instance that will process Messages.

        * `Mqueue` - A prioritized Message Queue that dispatches messages according to their priority.

- *ClientReceiver*
  `public ClientReceiver( plateau.server.UpdateHandler uh )`
  - Description
    * Constructs a ClientReceiver that does no message prioritization.
  - Parameters
    * uh - the UpdateHandler instance that will process Messages.

- *ClientReceiver*
  `public ClientReceiver( )`
  - Description
    * Constructs a new ClientReceiver with the generic UpdateHandler and no message prioritization.

Methods

- *createMessageInstance*
  `public Message createMessageInstance( )`
  - Description
    * Usually, a MessageQueue will require a particular subclass of Message. Since plateau generates a few system messages, this messages need to be of the subtype required by the MessageQueue. this method allows plateau to generate these messages. When no MessageQueue is being used, createMessageInstance returns a generic message.

- *setUpdateHandler*
  `protected void setUpdateHandler( plateau.server.UpdateHandler uh )`
  - Description
    * Changes the UpdateHandler used to process messages. This method is used on the server side to specify the UpdateHandler that a particular application wants to use.
  - Parameters
    * uh - the new update Handler that will process received messages.

- *isPrioritized*
  `public boolean isPrioritized( )`
  - Description

    * Returns true if the Client Receiver is using a MessageQueue to prioritize messages.

- *getMessageQueue*
  `public MessageQueue getMessageQueue( )`
  - Description
    * Returns the MessageQueue used if the receiver is in prioritized mode (isPrioritized returns true). Returns null otherwise.
- *setPrioritized*
  `protected void setPrioritized( boolean prio )`
  - Description
    * if prio is false, no MessageQueue is used for sending messages to the update handler. If prio is true AND the updatehandler has the isPrioritized flag set to true, the receiver prioritizes messages before dispatching them to the update handler.
- *getUpdateHandler*
  `public UpdateHandler getUpdateHandler( )`
  - Description
    * Returns the UpdateHandler used by the Client Receiver to process messages.
- *getCommonState*
  `public CommonState getCommonState( )`
  - Description
    * Returns the CommonState instance kept by the Update handler
- *enqueue*
  `protected void enqueue( plateau.queue.Message m )`
  - Description
    * Called internally by the client when a message is received
- *flush*
  `protected void flush( )`
  - Description
    * Forces all messages to be processed by the UpdateHandler and returns only after the messages have been processed.
- *run*
  `public void run( )`
  - Description
    * The thread keeps dispatching received messages to the UpdateHandler. If a MessageQueue is being used (isPrioritized returns true), messages are dispatched according to their priority.

A.1.3 *Class* Logger

Logger writes textual data to a PrintWriter. At creation time, you can specify whether the textual data will be written to System.out too, or not.

Declaration

public final class Logger **extends** java.lang.Object **implements** plateau.util.Timed

Constructors

- *Logger*
  `public Logger( java.io.PrintWriter log, boolean systemout )`
  - Description
    - Creates a new Logger.
  - Parameters
    - `log` - The PrintWriter object where data is written to.
    - `systemout` - if true, the data is also displayed in system out.

Methods

- *setPeriod*
  `public void setPeriod( long millis )`
  - Description
    - If the PrintWriter does not do autoflush every time data is written to it, setPeriod specifies the time Logger waits before flushing the PrintWriter.
  - Parameters
    - `millis` - Time in milliseconds
- *suspend*
  `public void suspend( )`
  - Description
    - Suspends the thread that flushes the PrintWriter periodically.
- *resume*
  `public void resume( )`
  - Description

        * Resumes the thread that flushes the PrintWriter periodically.

- *setLog*
  `public void setLog( java.io.PrintWriter l )`
  - Description
    * Changes the PrintWriter used to write data to.
- *setOut*
  `public void setOut( java.io.PrintStream output )`
  - Description
    * Sets the destination where system out messages are being written. The default destination is System.out If out is null, an error is sent to standard error and the destination is not changed.
- *println*
  `public synchronized void println( java.lang.String text )`
  - Description
    * Logs status information
  - Parameters
    * `text` - the status information to log
- *fire*
  `public synchronized void fire( )`
  - Description
    * Called by a Timer periodically to flush the PrintWriter.

A.1.4 *Class* Server

A Server keeps information about clients that are connected and broadcasts messages sent by clients. It listens at a specific port for new connections, and synchronizes new clients so that they get a consistent copy of the CommonState kept by each client.

Declaration

public class Server **extends** java.lang.Thread

Constructors

- *Server*
  `public Server( int port )`
  - Description

    * Creates a server object with displaying status messages in system out.
  - Parameters
    * `port` - the port number at which the server will listen

- *Server*
  `public Server( boolean sysout )`
  - Description
    * Creates a server that will listen at the default port.
  - Parameters
    * `sysout` - displays log messages in system out if sysout is true.

- *Server*
  `public Server( )`
  - Description
    * Create a server listening at the default port and displaying status messages in system out

- *Server*
  `public Server( int portnumber, boolean sysout )`
  - Description
    * Creates a new server listening at port portnumber. If sysout is true, additional information about the server's status is written to system out
  - Parameters
    * `portnumber` - the port number at which the server listens.
    * `sysout` - If true, additional status information is logged.

Methods

- *getUpdateHandler*
  `public UpdateHandler getUpdateHandler( )`
  - Description
    * Returns the UpdateHandler used by the State Keeper. The update handler is set by the first client that connects after the server has been running with no connections established. After the server is first launched and before the first client connects, the server keeps a Default update handler (that is not specific to any collaborative application)

- *setMaxConnections*
  `public void setMaxConnections( int max )`

- Description
  - Sets the allowed number of simultaneous connections.
- Parameters
  - `max` - the maximum number of connections allowed.

- *getMaxConnections*
  `public int getMaxConnections( )`
  - Description
    - Returns the maximum number of connections that the server allows.

- *startServer*
  `public void startServer( )`
  - Description
    - Starts the listener and connection Manager so that new clients can connect.

- *getConnectionManager*
  `public ConnectionManager getConnectionManager( )`
  - Description
    - Returns the connection Manager. The connection Manager holds connections. Connection objects represent clients connected to the server.

- *getLogger*
  `public Logger getLogger( )`
  - Description
    - Returns the object in charge of logging status information to an output stream (by default a file and system.out)

- *setFlushPeriod*
  `public void setFlushPeriod( long time )`
  - Description
    - Sets the period at which messages get broadcasted to clients. The time unit is milliseconds. The default period is 100 milliseconds.
  - Parameters
    - `time` - the period in milliseconds that the server waits between broadcasts.

- *main*
  `public static void main( java.lang.String [] args )`
  - Description
    - Called if server is invoked from a command prompt. A server is instantiated and started.

A.1.5 *Class* Client

A Client class is created to initiate a connection with the Plateau server. After it is constructed, calling its start method will create a connection with the server and will retrieve the commonstate that is shared by all current users.

Declaration

public class Client **extends** java.lang.Thread

Fields

- public String ERROR
  - –

Constructors

- *Client*
  `public Client( java.lang.String hostname, int port )`
  - Description
    - Connects to a Plateau server at the specified host and port number.
  - Parameters
    - `host` - the name of the host where the server is running.

Methods

- *start*
  `public void start( plateau.server.User usr,`
  `plateau.server.ClientReceiver rec )`
  - Description
    - Initializes the client by retrieving the up-to-date common state and starting the thread that receives update messages.
  - Parameters
    - `usr` - The local user which is joining the application.
    - `rec` - The client receiver used by a client to process update messages.

- *start*
  `public void start( plateau.server.User usr, plateau.server.ClientSender sender, plateau.server.ClientReceiver rec )`
  - Description
    - Initializes the client by retrieving the up-to-date common state and starting the thread that receives update messages.
  - Parameters
    - `usr` - The local user which is joining the application.
    - `sender` - A sender object used to control the flow of outgoing messages.
    - `rec` - The client receiver used by a client to process update messages.
- *getUpdateHandler*
  `public UpdateHandler getUpdateHandler( )`
  - Description
    - Returns the update handler used by the client to process update messages.
- *getCommonState*
  `public CommonState getCommonState( )`
  - Description
    - Returns the up-to-date common state. CommonState is used to create an UpdateHandler, which at the same time is used to create a ClientReceiver. After the start method is called, the initial commonstate needs to be assigned to the object returned by this method in order to synchronize it with remote clients.
- *getUsers*
  `public Hashtable getUsers( )`
  - Description
    - Returns the table of users held by the CommonState.
- *getLocalUser*
  `public User getLocalUser( )`
  - Description
    - Returns the User instance that is local in that Client object. The client must be connected or a null object will be returned.
- *isLocalUser*
  `public boolean isLocalUser( plateau.server.User usr )`
  - Description
    - Returns true if the user passed as an argument it the local one.

- *isLocalUser*
  `public boolean isLocalUser( long userid )`
  - Description
    * Returns true if the userid passed as an argument is the user id of the local user.
- *isConnected*
  `public boolean isConnected( )`
  - Description
    * Returns true if a successful connection has been established with a plateau server.
- *disconnect*
  `public void disconnect( )`
  - Description
    * Closes the connection with the plateau server. This will generate a message broadcasted to all current users so that they remove this local user from their common state.
- *broadcast*
  `public void broadcast( plateau.queue.Message m )`
  - Description
    * Sends a message that is broadcasted and received by all clients connected to the server. The message sender will also receive this message.
- *setLogging*
  `public void setLogging( boolean lg )`
  - Description
    * If lg is true, status messages are written to a file. No logging can happen if client is used in the context of an applet because there is no file i/o allowed.
- *run*
  `public void run( )`
  - Description
    * This thread receives messages and places them in the ClientReceiver.

A.1.6 *Class* ConnectionManager

The Connection manager is a class that runs inside plateau.Server and, as it name indicates it, manages the connections established between remote clients and the server. Whenever the Listener accepts a connection from a remote client, it calls the Connection

Manager's addConnection method, so that an actual Connection instance is created. The connection manager makes sure that the maximum number of connections has not been reached before creating a new Connection. It keeps all existing connection instances in a vector. Whenever a connection is closed, the connection manager gets notified so that the connection is removed from the vector.

Declaration

public class ConnectionManager
extends java.lang.Thread

Constructors

- *ConnectionManager*
  `public ConnectionManager( java.lang.ThreadGroup` **group**`, int` **maxConnections**`, plateau.server.Server` **ws** `)`
  - Description
    - Creates a connection manager. Since it is a thread, it is associated to the thread group where all server related threads are running.
  - Parameters
    - `group` - The thread group in which all server related threads are associated.
    - `maxConnection` - the number of connections that the server will allow.
    - `ws` - The server object that owns this connection manager.

Methods

- *addConnection*
  `protected synchronized void addConnection( java.net.Socket` **client** `)`
  - Description
    - Takes the socket created when a listener socket accepts a connection. If the creation is successful (the connection limit has not been reached) Starts the connection (Connection is also a thread) and adds it to the vector of existing connections. If the number of connections has exceeded the limit, notifies the client, closes the connection and logs the error.
- *getUpdateHandler*
  `public UpdateHandler getUpdateHandler( )`

  - Description
    * Returns the update Handler kept by the local Client Receiver

- *endConnection*
  `public synchronized void endConnection( )`
  - Description
    * Called by a connection when it closes. The thread is awaken, and closed connections are removed from the connection vector.

- *setMaxConnections*
  `public synchronized void setMaxConnections( int max )`
  - Description
    * Changes the number of allowed connections

- *getSender*
  `public Sender getSender( )`
  - Description
    * Returns the sender instance used by the server to broadcast messages to all clients.

- *printConnections*
  `public synchronized String printConnections( )`
  - Description
    * Returns an array of strings. Each string represents a connection

- *removeConnectionAt*
  `protected synchronized void removeConnectionAt( int i )`
  - Description
    * Removes the connection located in index i in the vector, and notifies all other clients by sending a Message.BYE to each.
  - Parameters
    * `i` - the index of the connection to remove.

- *removeConnection*
  `protected synchronized void removeConnection(`
  `plateau.server.Connection c )`
  - Description
    * Removes the connection c if it is in the vector, and notifies all other clients by sending a Message.BYE to each.
  - Parameters
    * `c` - the connection to be removed.

- *run*
  `public void run( )`
  - Description
    * Sleeps until it is awaken by endConnection, then iterates through connection vector, removing those connections that are closed.

A.1.7 *Class* ClientSender

ClientSender is responsible for sending application messages to the server so that they get broadcasted to all clients. It also allows a client application to control the flow of outgoing messages by using a MessageQueue that will prioritize the order in which messages get sent.

Declaration

public class ClientSender **extends** java.lang.Thread

Fields

- public static final boolean DEBUG
  - –

Constructors

- *ClientSender*
  `public ClientSender( )`
  - Description
    - Creates a sender that sends messages as soon as it receives them from the the application.
- *ClientSender*
  `public ClientSender( plateau.queue.MessageQueue queue )`
  - Description
    - Creates a sender with a MessageQueue that will prioritize outgoing messages, unless queue is set to null.

Methods

- *setOutputStream*
  `protected void setOutputStream( java.io.ObjectOutputStream out )`
  - Description
    - Sets the outputstream used to broadcast messages. Usually it is a stream extracted from an initialized socket.

- *enqueue*
  `protected void enqueue( plateau.queue.Message msg )`
  - Description
    * Inserts an outgoing message to the MessageQueue. If no outgoing prioritization is being done, immediately sends the message to the server.

- *enqueue*
  `protected void enqueue( plateau.queue.Queue queue )`
  - Description
    * Enqueues a whole queue of messages. The queue is appended to a Message queue if it exists. Otherwise each message is sent.

- *run*
  `public void run( )`
  - Description
    * When prioritization is being done, this thread extracts messages from the MessageQueue and sends them to the server. The thread does not run when outgoing prioritization is not done, since messages get immediately sent when they are enqueued into the sender.

A.1.8 *Class* CommonState

CommonState represents the state that all clients will share. Usually CommonState will be extended by a particular application to hold all data that needs to be shared. All components of CommonState need to be Serializable because the CommonState is sent through the network to initialize new clients.

The generic CommonState holds the collection of Users that are connected and gets updated by the generic UpdateHandler when new users connect or users leave.

The generic UpdateHandler needs to be extended to handle a particular subclass of CommonState. the UpdateHandler's handleUpdate method will then modify the CommonState depending on the Message received.

Declaration

public class CommonState **extends** java.lang.Object **implements** java.io.Serializable

Constructors

- *CommonState*
  `public CommonState( )`
  - Description
    * Constructs a generic commonstate with an empty table of users.

Methods

- *setLocal*
  `protected void setLocal( plateau.server.User local )`
  - Description
    * Used internally by plateau to tag the user that is local in a client host.
  - Parameters
    * `local` - the user that will be set to be local at the client host where this instance of common state lives.
- *getLocal*
  `public User getLocal( )`
  - Description
    * Returns the user that is considered local by the commonstate
- *isLocal*
  `public boolean isLocal( plateau.server.User usr )`
  - Description
    * Returns true if usr is the user that is considered local by the commonstate
  - Parameters
    * `usr` - The user object being queried for locality.
- *isLocal*
  `public boolean isLocal( long userid )`
  - Description
    * Returns true if userid is the id of the user that is considered local by the commonstate
  - Parameters
    * `userid` - the user id of the user being queried for locality.
- *addUser*
  `public void addUser( plateau.server.User usr )`

- Description
  - Adds a new user to the CommonState's table. The user must have an assigned unique id so that it can later be retrieved from the table. addUser is called by the generic UpdateHandler when new clients connect.
- Parameters
  - `usr` - the User that be inserted into the CommonState's user table.

- *removeUser*
  `public User removeUser( long userid )`
  - Description
    - Removes the user with id userid from the user table and returns it. If no user with such usrid exists, return a null object. removeUser is called by the generic UpdateHandler when clients disconnect.
  - Parameters
    - `userid` - the unique usrid of the user to be removed.

- *getUser*
  `public User getUser( long userid )`
  - Description
    - Returns the user with unique id userid if that user is stored in the CommonState's table. Otherwise, returns a null object.
  - Parameters
    - `userid` - the unique user id of the User to be obtained.

- *getUsers*
  `public Hashtable getUsers( )`
  - Description
    - Returns the current table of users.

A.1.9 *Class* Sender

Sender is a thread that keeps a queue of Messages that need to be broadcasted to all clients. Instead of iterating through each message , sender periodically broadcasts the current queue of Messages to each connection object stored in the Connection Manager.

Declaration

public class Sender
extends java.lang.Thread

Fields

- public static final long DEFAULT_PERIOD
  - The default period between flush operations is 1 second

Constructors

- *Sender*
  `public Sender( plateau.server.ConnectionManager cm, plateau.server.Logger logger )`
  - Description
    * The constructor receives a handle to the Connection Manager to get access to the Vector of connections, and a handle to the Logger so that it can log activity that is going on.

      After broadcasting a queue, sender iterates through every message and enqueues it into a ClientReceiver if the message has the state flag on. This allows the server to keep a local CommonState that is consistent with the one kept by all clients, so that when a new Client connects, it gets a consistent copy.
  - Parameters
    * `cm` - The connection manager that holds connections
    * `logger` - the object in charge of logging activity

Methods

- *setFlushPeriod*
  `public void setFlushPeriod( long period )`
  - Description
    * Sets the period at which the sender flushes the message queue
  - Parameters
    * `period` - milliseconds in between flush operations.
- *enqueue*
  `public void enqueue( plateau.queue.Message msg )`
  - Description
    * adds a new Message to the outgoing queue
  - Parameters

* `msg` - The message to be added to the queue

- *enqueue*
  `public void enqueue( plateau.queue.Queue queue )`
  - Description
    * appends a queue of Messages into the outgoing queue.
  - Parameters
    * queue - a queue that holds message objects

- *getKeeper*
  `public ClientReceiver getKeeper( )`
  - Description
    * Returns the State Keeper, and instance of Client Receiver that is keeping the server's CommonState consistent. The server keeps a common state in order to initialize new clients.

- *flush*
  `public void flush( )`
  - Description
    * Broadcasts and processes all messages that are waiting in the queue.

- *run*
  `public void run( )`
  - Description
    * The thread sleeps and is waken up periodically by a NotifyTimer so that sender processes the current queue (if it is not empty)

A.1.10 *Class* UpdateHandler

UpdateHandler lives inside a ClientReceiver. It's handleUpdate(Message m) method gets called to process every message that is received. This class needs to be extended, and its handleUpdate method overwritten to handle application specific messages. Handling of messages usually involves updating the CommonState held by the UpdateHandler. If, a message updates the common state, it must have its isState flag set to true so that the server also updates its local copy of the CommonState.

Declaration

public class UpdateHandler **extends** java.lang.Object **implements** java.io.Serializable

Constructors

- *UpdateHandler*
  `public UpdateHandler( plateau.server.CommonState cmmn )`
  - Description
    - Creates a basic Update Handler.
  - Parameters
    - `cmmn` - the CommonState maintained by the Update Handler.
- *UpdateHandler*
  `public UpdateHandler( )`
  - Description
    - Create a new UpdateHandler with a generic CommonState.

Methods

- *setPrioritized*
  `protected void setPrioritized( plateau.queue.MessageQueue queue )`
  - Description
    - Used internally by ClientReceiver instances to allow the server to use a MessageQueue and process Messages in the same order they are processed on the client side.
  - Parameters
    - `queue` - The priority queue used to dispatch messages to the handler.
- *getMessageQueue*
  `public MessageQueue getMessageQueue( )`
  - Description
    - Returns the message queue used by the client receiver that holds this updatehandler. IF no message queue is used, or if no receiver holds this update handler, returns null.
- *isPrioritized*
  `public boolean isPrioritized( )`
  - Description
    - Returns true if the client receiver that holds the update handler prioritizes messages using a MessageQueue.
- *getCommonState*
  `public CommonState getCommonState( )`

  - Description
    * Returns the CommonState held by the handler.

- *handleUpdate*
  `public void handleUpdate( plateau.queue.Message message )`
  - Description
    * This method is called by the client receiver to process each message that a client receives. It must be overwritten to handle application specific messages. However, before doing application specific handling, the subclass' handleUpdate method must call super.handleUpdate to perform the system's update handling.

Appendix B

plateau.queue

Package Contents *Page*

B.1 Interfaces

B.1.1 *Interface* Prioritizer

This is the interface that developers must implement to specify rules for prioritizing messages when they get enqueued into a MessageQueue. MessageQueues are created by passing them an instance of a Prioritizer, and subsequent queue and enqueueAll operations make calls the getPriority method of Prioritizer to assign priorities to messages. Usually, information within the message will let the Prioritizer decide what priority to assign. However, The Prioritizer class can also have pointers to application data that influences how messages get prioritized.

Declaration

public abstract interface Prioritizer **implements** java.io.Serializable

Methods

- *getPriority*
  `public int getPriority( plateau.queue.Message msg )`
  - Description
    * The rules that assing message priorities should be specified in this method.
- *getParent*
  `public MessageQueue getParent( )`
  - Description
    * An instance of a Prioritizer should only be assigned to one MessageQueue. This method returns the MessageQueue that is using this prioritizer.
- *setParent*
  `public void setParent( plateau.queue.MessageQueue mq )`
  - Description
    * Should set a private property that holds the MessageQueue that uses this prioritizer and returns is when the getParent method is called.

B.1.2 *Interface* Guarded

Subclasses of Message that want to use Guard objects to be reprioritized when the Guard update method is called (if the guard has been assigned to the message) must implement this interface. IndexedMessage implements this interface.

Declaration

public abstract interface Guarded **implements** java.io.Serializable

Methods

- *isGuarded*
  `public boolean isGuarded( )`

- Description
  - Returns true if the message is guarded by a Guard

- *guardBy*
  `public void guardBy( plateau.queue.Guard g )`
  - Description
    - Assigns a guard to the message so that it get reprioritized when the the Guard's update method gets called.

- *getGuard*
  `public Guard getGuard( )`
  - Description
    - Returns the guard object that guards this message if the message is guarded. Returns null otherwise.

- *removeGuard*
  `public void removeGuard( )`
  - Description
    - If the message is guarded, removes the guard so that no reprioritization happens when the guard's update method is called.

B.2 Classes

B.2.1 *Class* Message

Message is the wrapper class that is used when messages are to be queued in a MessageQueue before being delivered to the agent that will process them. The actual message is passed as an Object in the constructor of the class. When a message is enqueued in a MessageQueue, it is assigned a priority by the queue that can be queried calling getPriority. Plateau uses the Message as the basic unit of communication among clients in a collaborative application, so Message implements Serializable so that it can be handled by different agents in a network.

Typically, a Message is extended to be used with a particular implementation of a MessageQueue. For example, IndexedMessage, a subclass of Message is used in conjunction with an IndexedQueue.

Declaration

public class Message
extends java.lang.Object
implements java.io.Serializable

Fields

- public static int NOT_QUEUED
  - –
- public static final int BYE
  - –
- public static final int NEW
  - –
- public static final int ERROR
  - –
- public static final int INIT
  - –
- public static final int GENERIC
  - –

Constructors

- *Message*
  `public Message( java.lang.Object cont, int id, boolean state )`
  - Description
    - The message constructor.
  - Parameters
    - `cont` - The actual content of a message.
    - `id` - The message id
- *Message*
  `public Message( java.lang.Object cont )`
  - Description
    - Creates a generic normal (non-state) message
  - Parameters
    - `cont` - The contents of the message.
- *Message*
  `public Message( java.lang.Object cont, boolean state )`
  - Description

        * Creates a generic message.
    - Parameters
        * `cont` - The contents of the message.
        * `state` - if true, creates a state message. Otherwise creates a normal message.

- *Message*
  `public Message( java.lang.Object cont, int type )`
    - Description
        * Creates a normal (non-state) message.
    - Parameters
        * `cont` - The contents of the message.
        * `type` - The message type

- *Message*
  `public Message( )`
    - Description
        * Creates a generic, normal (non-state) message with null contents.

Methods

- *setTimeStamp*
  `public void setTimeStamp( long t )`
    - Description
        * Sets the message's timestamp.
    - Parameters
        * `t` - the new timestamp (usually current time in milliseconds)
    - See Also
        * `method`

- *getTimeStamp*
  `public long getTimeStamp( )`
    - Description
        * Returns the messages timestamp. If no timestamp has been assigned, returns -1

- *setSenderID*
  `public void setSenderID( long sid )`
    - Description
        * Sets an id identifying the user who created this message.
    - See Also

        * `plateau.server.User` (in A.1.1, page 73)

- *getSenderID*
  `public long getSenderID( )`
  - Description
    * Returns the id of the User who created this message
  - See Also
    * `plateau.server.User` (in A.1.1, page 73)

- *isState*
  `public boolean isState( )`
  - Description
    * Returns true if this is an instance of a state message.

- *setState*
  `public void setState( boolean st )`
  - Description
    * If st is true, identifies this instance as a state message. Otherwise, identifies thie instance as a normal message.

- *setLabel*
  `public void setLabel( java.lang.String label )`
  - Description
    * Sets a textual label for the message.
  - Parameters
    * `label` - the textual label

- *getLabel*
  `public String getLabel( )`
  - Description
    * Returns the label of the message. If no label has been set, returns an empty string.

- *setID*
  `public void setID( int id )`
  - Description
    * Sets the message type.

- *getID*
  `public int getID( )`
  - Description
    * Returns the message type. Message whose type has not been set are of type Message.GENERIC

- *getContents*
  `public Object getContents( )`
  - Description
    * Returns the contents of a Message. Different agents can agree on what type of object this will be, or can use reflection to find out the class.
- *setContents*
  `public void setContents( java.lang.Object cont )`
  - Description
    * Sets the contents of the message.
- *getParent*
  `public MessageQueue getParent( )`
  - Description
    * If a Message is queued, returns the queue where it lives. Returns null if the message has not been queued.
- *isQueued*
  `public boolean isQueued( )`
  - Description
    * returns true if the message is queued. False if it isn't.
- *setQueued*
  `protected void setQueued( boolean q )`
  - Description
    * Specifies whether the message is queued or not. Only used by classes that extend Message to work with a particular subclass of MessageQueue
- *clone*
  `public Object clone( )`
  - Description
    * Retuns a copy of the message. All properties are cloned except for the contents. ie. The result is a new Message pointing to the same contents.
- *setParent*
  `protected void setParent( plateau.queue.MessageQueue parent )`
  - Description
    * Used by classes that extend Message to work with a particular implementation of MessageQueue
  - Parameters
    * `parent` - The MessageQueue where the Message has been enqueued

- *setPriority*
  `protected void setPriority( int prio )`
  - Description
    - Used by classes that extend Message to work with a particular implementation of MessageQueue
  - Parameters
    - `the` - priority of the Message.
- *getPriority*
  `public int getPriority( )`
  - Description
    - Returns the priority of a Message that is queued. If the message is not queued, it returns Message.NOT_QUEUED
- *toString*
  `public String toString( )`
  - Description
    - Returns the textual representation of the contents of the Message.

B.2.2 *Class* ColorableMessage

ColorableMessage can be enqueued in a GraphicQueue and holds necessary information to be represented graphically.

Declaration

public class ColorableMessage
extends plateau.queue.Message

Constructors

- *ColorableMessage*
  `public ColorableMessage( java.lang.Object cont, int id, boolean state )`
  - Description
    - The message constructor.
  - Parameters
    - `cont` - The content of a message.
    - `id` - The message type
    - `state` - if true, identifies a message that modify a common state when handled by an update handler.

Methods

- *setColor*
  `public void setColor( java.awt.Color c )`
  - Description
    - Sets the color of the message.
  - Parameters
    - `color` - the color used to paint the message.
- *getColor*
  `public Color getColor( )`
  - Description
    - Returns the color used to paint the message.
- *isMarked*
  `public boolean isMarked( )`
  - Description
    - Returns true if the message should be highlighted in its graphical representation
- *setMarked*
  `public void setMarked( boolean m )`
  - Description
    - Marks a message so that it gets highlighted in its graphical representation.

Methods inherited from class `plateau.queue.Message`

(in B.2.1, page 97)

- *setTimeStamp*
  `public void setTimeStamp( long t )`
  - Description
    - Sets the message's timestamp.
  - Parameters
    - `t` - the new timestamp (usually current time in milliseconds)
  - See Also
    - `method`
- *getTimeStamp*
  `public long getTimeStamp( )`
  - Description

        * Returns the messages timestamp. If no timestamp has been assigned, returns -1

- *setSenderID*
  `public void setSenderID( long sid )`
  - Description
    * Sets an id identifying the user who created this message.
  - See Also
    * `plateau.server.User` (in A.1.1, page 73)
- *getSenderID*
  `public long getSenderID( )`
  - Description
    * Returns the id of the User who created this message
  - See Also
    * `plateau.server.User` (in A.1.1, page 73)
- *isState*
  `public boolean isState( )`
  - Description
    * Returns true if this is an instance of a state message.
- *setState*
  `public void setState( boolean st )`
  - Description
    * If st is true, identifies this instance as a state message. Otherwise, identifies thie instance as a normal message.
- *setLabel*
  `public void setLabel( java.lang.String label )`
  - Description
    * Sets a textual label for the message.
  - Parameters
    * `label` - the textual label
- *getLabel*
  `public String getLabel( )`
  - Description
    * Returns the label of the message. If no label has been set, returns an empty string.
- *setID*
  `public void setID( int id )`
  - Description
    * Sets the message type.

- *getID*
  `public int getID( )`
  - Description
    * Returns the message type. Message whose type has not been set are of type Message.GENERIC
- *getContents*
  `public Object getContents( )`
  - Description
    * Returns the contents of a Message. Different agents can agree on what type of object this will be, or can use reflection to find out the class.
- *setContents*
  `public void setContents( java.lang.Object cont )`
  - Description
    * Sets the contents of the message.
- *getParent*
  `public MessageQueue getParent( )`
  - Description
    * If a Message is queued, returns the queue where it lives. Returns null if the message has not been queued.
- *isQueued*
  `public boolean isQueued( )`
  - Description
    * returns true if the message is queued. False if it isn't.
- *setQueued*
  `protected void setQueued( boolean q )`
  - Description
    * Specifies whether the message is queued or not. Only used by classes that extend Message to work with a particular subclass of MessageQueue
- *clone*
  `public Object clone( )`
  - Description
    * Retuns a copy of the message. All properties are cloned except for the contents. ie. The result is a new Message pointing to the same contents.
- *setParent*
  `protected void setParent( plateau.queue.MessageQueue parent )`
  - Description
    * Used by classes that extend Message to work with a particular implementation of MessageQueue
  - Parameters

    * `parent` - The MessageQueue where the Message has been enqueued

- *setPriority*
  `protected void setPriority( int prio )`
  - Description
    * Used by classes that extend Message to work with a particular implementation of MessageQueue
  - Parameters
    * `the` - priority of the Message.

- *getPriority*
  `public int getPriority( )`
  - Description
    * Returns the priority of a Message that is queued. If the message is not queued, it returns Message.NOT_QUEUED

- *toString*
  `public String toString( )`
  - Description
    * Returns the textual representation of the contents of the Message.

B.2.3 *Class* IndexedMessage

IndexedMessage is a subclass of Message that works together with an IndexedQueue. Since IndexedMessage implements the Guarded interface, they can be assigned a Guard by calling guardBy. By assigning a guard to the message, a developer can call the guard's update method so that the prioritizer of a MessageQueue reevaluates the message's priority, placing the message in the right position. This is useful because the priority of a Message could change once it has already been queued.

Declaration

public class IndexedMessage
extends plateau.queue.ColorableMessage
implements Guarded, java.util.Observer

Constructors

- *IndexedMessage*
  `public IndexedMessage( java.lang.Object cont, int type, boolean state )`
  - Description
    * The IndexedMessage constructor.

- Parameters
  - `cont` - The actual contents of a message.

- *IndexedMessage*
  `public IndexedMessage( java.lang.Object cont, int type )`
  - Description
    - Constructor. Creates a normal (non-state) new message with contents cont, of type type.
  - Parameters
    - `cont` - the contents of the message
    - `type` - the type of the message

- *IndexedMessage*
  `public IndexedMessage( java.lang.Object cont, boolean state )`
  - Description
    - Creates a generic state message with contents cont.

- *IndexedMessage*
  `public IndexedMessage( java.lang.Object cont )`
  - Description
    - Creates a generic normal (non-state) message with contents cont.

- *IndexedMessage*
  `public IndexedMessage( )`
  - Description
    - Creates a generic normal (non-state) message with null contents.

Methods

- *setMarked*
  `public void setMarked( boolean m )`
  - Description
    - if m is true, marks the message so that it gets highlighted in its graphical representation.

- *isMarked*
  `public boolean isMarked( )`
  - Description
    - Returns true if the message is mark.

- *getColor*
  `public Color getColor( )`

- Description
  - Returns the color of IndexedMessage.

- *setColor*
  `public void setColor( java.awt.Color c )`
  - Description
    - Sets the color of the IndexedMessage.

- *clone*
  `public Object clone( )`
  - Description
    - Retuns a copy of the IndexedMessage. All properties are cloned except for the contents. ie. The result is a new IndexedMessage pointing to the same contents. If the original message was guarded, the copy won't be guarded.

- *setParent*
  `protected void setParent( plateau.queue.IndexedQueue mq, int prio )`
  - Description
    - Used by IndexedMessageQueues.
  - Parameters
    - `mq` - the IndexedQueue object that will contain the IndexedMessage
    - `prio` - The priority that the IndexedQueue assigns to this Message

- *update*
  `public synchronized void update( java.util.Observable obv, java.lang.Object arg )`
  - Description
    - Used by a Guard object to reevaluate the priority of the IndexedMessage in a queue. Do not call this method. Update repositions the IndexedMessage in a queue by invalidating the original, and queueing a copy of it so that it gets reevaluated.

- *guardBy*
  `public synchronized void guardBy( plateau.queue.Guard guard )`
  - Description
    - Assigns a guard to the message so that every time that the update method is called in that guard, the message gets reevaluated in the queue. A message must be queued for a guard to be assigned to it. Nothing will happen if a guard is assigned to an unqueued message.
  - Parameters
    - `guard` - The Guard object that will update the message's priority

- *removeGuard*
  `public void removeGuard( )`
  - Description
    * If the message is Guarded, deletes the guard so that no reevaluations happen when the guard's update method is called.
- *isGuarded*
  `public boolean isGuarded( )`
  - Description
    * Returns true if a message has been assigned a guard. Returns false otherwise.
- *getGuard*
  `public Guard getGuard( )`
  - Description
    * If a guard is assigned to the message, returns the Guard object. Otherwise returns null.
- *isValid*
  `protected boolean isValid( )`
  - Description
    * This implementations of MessageQueue invalidates Messages that get reevaluated, and enqueues a copy of them. isValid returns false when a Message has been invalidated.
- *toString*
  `public String toString( )`
  - Description
    * Returns a textual representation of the message.

Methods inherited from class `plateau.queue.ColorableMessage`

(in B.2.2, page 102)

- *setColor*
  `public void setColor( java.awt.Color c )`
  - Description
    * Sets the color of the message.
  - Parameters
    * `color` - the color used to paint the message.
- *getColor*
  `public Color getColor( )`
  - Description

    * Returns the color used to paint the message.

- *isMarked*
  `public boolean isMarked( )`
  - Description
    * Returns true if the message should be highlighted in its graphical representation
- *setMarked*
  `public void setMarked( boolean m )`
  - Description
    * Marks a message so that it gets highlighted in its graphical representation.

Methods inherited from class `plateau.queue.Message`

(in B.2.1, page 97)

- *setTimeStamp*
  `public void setTimeStamp( long t )`
  - Description
    * Sets the message's timestamp.
  - Parameters
    * `t` - the new timestamp (usually current time in milliseconds)
  - See Also
    * `method`
- *getTimeStamp*
  `public long getTimeStamp( )`
  - Description
    * Returns the messages timestamp. If no timestamp has been assigned, returns -1
- *setSenderID*
  `public void setSenderID( long sid )`
  - Description
    * Sets an id identifying the user who created this message.
  - See Also
    * `plateau.server.User` (in A.1.1, page 73)
- *getSenderID*
  `public long getSenderID( )`
  - Description
    * Returns the id of the User who created this message
  - See Also
    * `plateau.server.User` (in A.1.1, page 73)

- *isState*
  `public boolean isState( )`
  - Description
    * Returns true if this is an instance of a state message.
- *setState*
  `public void setState( boolean st )`
  - Description
    * If st is true, identifies this instance as a state message. Otherwise, identifies thie instance as a normal message.
- *setLabel*
  `public void setLabel( java.lang.String label )`
  - Description
    * Sets a textual label for the message.
  - Parameters
    * `label` - the textual label
- *getLabel*
  `public String getLabel( )`
  - Description
    * Returns the label of the message. If no label has been set, returns an empty string.
- *setID*
  `public void setID( int id )`
  - Description
    * Sets the message type.
- *getID*
  `public int getID( )`
  - Description
    * Returns the message type. Message whose type has not been set are of type Message.GENERIC
- *getContents*
  `public Object getContents( )`
  - Description
    * Returns the contents of a Message. Different agents can agree on what type of object this will be, or can use reflection to find out the class.
- *setContents*
  `public void setContents( java.lang.Object cont )`
  - Description
    * Sets the contents of the message.

- *getParent*
  `public MessageQueue getParent( )`
  - Description
    - If a Message is queued, returns the queue where it lives. Returns null if the message has not been queued.
- *isQueued*
  `public boolean isQueued( )`
  - Description
    - returns true if the message is queued. False if it isn't.
- *setQueued*
  `protected void setQueued( boolean q )`
  - Description
    - Specifies whether the message is queued or not. Only used by classes that extend Message to work with a particular subclass of MessageQueue
- *clone*
  `public Object clone( )`
  - Description
    - Retuns a copy of the message. All properties are cloned except for the contents. ie. The result is a new Message pointing to the same contents.
- *setParent*
  `protected void setParent( plateau.queue.MessageQueue parent )`
  - Description
    - Used by classes that extend Message to work with a particular implementation of MessageQueue
  - Parameters
    - `parent` - The MessageQueue where the Message has been enqueued
- *setPriority*
  `protected void setPriority( int prio )`
  - Description
    - Used by classes that extend Message to work with a particular implementation of MessageQueue
  - Parameters
    - `the` - priority of the Message.
- *getPriority*
  `public int getPriority( )`
  - Description
    - Returns the priority of a Message that is queued. If the message is not queued, it returns Message.NOT_QUEUED
- *toString*
  `public String toString( )`
  - Description
    - Returns the textual representation of the contents of the Message.

B.2.4 *Class* IndexedColor

IndexedColor is the contents of a default IndexMessage.

Declaration

public class IndexedColor
extends java.lang.Object
implements java.io.Serializable

Fields

- public static int DEFAULT_INDEX

  –

- public static Color DEFAULT_COLOR

  –

Constructors

- *IndexedColor*
  `public IndexedColor( )`
- *IndexedColor*
  `public IndexedColor( java.awt.Color color, int index )`

Methods

- *setMarked*
  `public void setMarked( boolean m )`
- *isMarked*
  `public boolean isMarked( )`
- *getColor*
  `public Color getColor( )`
- *setColor*
  `public void setColor( java.awt.Color c )`
- *getIndex*
  `public int getIndex( )`
- *toString*
  `public String toString( )`

B.2.5 *Class* MessageQueue

MessageQueue is an abstract class that defines the method that a Priority Queue in the Plateau environment should implement. Plateau provides an implementation of a MessageQueue in IndexedQueue. IndexedQueue optimizes reprioritization operations in order to efficiently make use of the guarding capabilities of Plateau. If developers want to work on a different implementation that focuses on other optimizations, they can subclass MessageQueue.

Declaration

public abstract class MessageQueue
extends java.lang.Object
implements java.io.Serializable

Constructors

- *MessageQueue*
  `public MessageQueue( plateau.queue.Prioritizer prio )`
  - Description
    - Creates a new message queue.
  - Parameters
    - `prio` - the prioritizer that will assign message priorities.

Methods

- *getPrioritizer*
  `public Prioritizer getPrioritizer( )`
  - Description
    - Returns the prioritizer used by the message queue.
- *setPrioritizer*
  `public synchronized void setPrioritizer( plateau.queue.Prioritizer prio )`
  - Description
    - Sets a new prioritizer
- *plugToGraphic*
  `protected void plugToGraphic( plateau.queue.GraphicQueue gqueue )`
  - Description

        * Called by a GraphicQueue if this message qeueu is used to create it.

- *createMessageInstance*
  `public abstract Message createMessageInstance( )`
  - Description
    * Will return the subclass of Message that an implementation of MessageQueue needs to operate correctly.

- *getElements*
  `public abstract Enumeration getElements( )`
  - Description
    * Returns an enumeration of the messages queued

- *renqueueAll*
  `public abstract void renqueueAll( )`
  - Description
    * Reprioritizes all enqueued messages.

- *enqueue*
  `public abstract int enqueue( plateau.queue.Message msg )`
  - Description
    * Enqueues a message

- *dequeue*
  `public abstract Message dequeue( )`
  - Description
    * Returns and extracts the message at the head of the queue

- *isEmpty*
  `public abstract boolean isEmpty( )`
  - Description
    * Returns trute if the queue is empty

- *renqueue*
  `protected abstract void renqueue( plateau.queue.Message msg )`
  - Description
    * Reprioritizes a message

- *contains*
  `public abstract boolean contains( plateau.queue.Message msg )`
  - Description
    * Returns true if the queue contains msg

B.2.6 *Class* IndexedQueue

This implementation of MessageQueue optimizes message reprioritization by keeping a fixed number of message priorities. Message priorities in this queue must be positive integers.

Declaration

public class IndexedQueue
extends plateau.queue.MessageQueue
implements java.io.Serializable

Fields

- public static int DEFAULT_PRIORITY
  - –
- public static int DEFAULT_SIZE
  - –
- public static IndexedQueue.DefaultPrioritizer DEFAULT_PRIORITIZER
  - The default prioritizer assigns a priority of 1 to all enqueued messages.

Constructors

- *IndexedQueue*
  `public IndexedQueue( plateau.queue.Prioritizer prio )`
  - Description
    - Creates a new message queue with the default number of allowed priorities.
- *IndexedQueue*
  `public IndexedQueue( plateau.queue.Prioritizer prio, int bCount )`
  - Description
    - Creates a new message queue that will hold messages with priorities from 0 to bCount.
  - Parameters
    - `prio` - the prioritizer that will assign message priorities.
    - `bCount` - the maximum priority allowed.

Methods

- *createMessageInstance*
  `public Message createMessageInstance( )`
  - Description
    - Returns a generic instance of IndexedQueue.
- *getElements*
  `public Enumeration getElements( )`
  - Description
    - Returns an enumeration of the messages held by the queue.
- *getBucketCount*
  `public int getBucketCount( )`
  - Description
    - returns the maximum allowed priority
- *getSize*
  `public int getSize( )`
  - Description
    - Returns the current number of messages enqueued.
- *isEmpty*
  `public boolean isEmpty( )`
  - Description
    - Returns true if the queue is empty.
- *setPrioritizer*
  `public synchronized void setPrioritizer( plateau.queue.Prioritizer prio )`
  - Description
    - Sets a new prioritizer
  - Parameters
    - `prio` - the new prioritizer
- *enqueue*
  `public synchronized int enqueue( plateau.queue.Message msg )`
  - Description
    - Enqueues message msg into the queue
  - Parameters
    - `msg` - the message that will be enqueued.

- *renqueueAll*
  `public synchronized void renqueueAll( )`
  - Description
    * Reprioritized all the enqueued messages.
- *dequeue*
  `public synchronized Message dequeue( )`
  - Description
    * Returns and extracts the message at the head of the queue
- *makeGuard*
  `public Guard makeGuard( )`
  - Description
    * Creates a guard that can hold messages enqueued in this queue.
- *renqueue*
  `protected synchronized void renqueue( plateau.queue.Message msg )`
  - Description
    * reprioritizes message msg
- *contains*
  `public synchronized boolean contains( plateau.queue.Message msg )`
  - Description
    * Returns true if msg is in the queue
- *toString*
  `public String toString( )`
  - Description
    * Returns a textual representation of the queue.

Methods inherited from class `plateau.queue.MessageQueue`

(in B.2.5, page 114)

- *getPrioritizer*
  `public Prioritizer getPrioritizer( )`
  - Description
    * Returns the prioritizer used by the message queue.
- *setPrioritizer*
  `public synchronized void setPrioritizer( plateau.queue.Prioritizer prio )`

- Description
  - Sets a new prioritizer

- *plugToGraphic*
  `protected void plugToGraphic( plateau.queue.GraphicQueue gqueue )`
  - Description
    - Called by a GraphicQueue if this message qeueu is used to create it.

- *createMessageInstance*
  `public abstract Message createMessageInstance( )`
  - Description
    - Will return the subclass of Message that an implementation of MessageQueue needs to operate correctly.

- *getElements*
  `public abstract Enumeration getElements( )`
  - Description
    - Returns an enumeration of the messages queued

- *renqueueAll*
  `public abstract void renqueueAll( )`
  - Description
    - Reprioritizes all enqueued messages.

- *enqueue*
  `public abstract int enqueue( plateau.queue.Message msg )`
  - Description
    - Enqueues a message

- *dequeue*
  `public abstract Message dequeue( )`
  - Description
    - Returns and extracts the message at the head of the queue

- *isEmpty*
  `public abstract boolean isEmpty( )`
  - Description
    - Returns trute if the queue is empty

- *renqueue*
  `protected abstract void renqueue( plateau.queue.Message msg )`
  - Description
    - Reprioritizes a message

- *contains*
  `public abstract boolean contains( plateau.queue.Message msg )`
  - Description
    - Returns true if the queue contains msg

B.2.7 *Class* IndexedQueue.DefaultPrioritizer

The default prioritizer. Assigns a priority of 1 to all Messages

Declaration

public static class IndexedQueue.DefaultPrioritizer **extends** java.lang.Object **implements** Prioritizer

Constructors

- *IndexedQueue.DefaultPrioritizer*
  `public IndexedQueue.DefaultPrioritizer( )`

Methods

- *getPriority*
  `public int getPriority( plateau.queue.Message msg )`
- *getParent*
  `public MessageQueue getParent( )`
- *setParent*
  `public void setParent( plateau.queue.MessageQueue mq )`

B.2.8 *Class* Guard

Guard keeps a list of Messages that should be reprioritized when its update method gets called.

Declaration

public class Guard **extends** java.util.Observable **implements** java.io.Serializable

Constructors

- *Guard*
  `public Guard( )`
  - Description
    * Constructor.

Methods

- *notifyObservers*
  `public void notifyObservers( )`
- *update*
  `public void update( )`
  - Description
    * Triggers the message queue containing messages that the Guard is holding so that it reprioritizes the messages.

Methods inherited from class `java.util.Observable`

- *addObserver*
  `public synchronized void addObserver( java.util.Observer )`
- *clearChanged*
  `protected synchronized void clearChanged( )`
- *countObservers*
  `public synchronized int countObservers( )`
- *deleteObserver*
  `public synchronized void deleteObserver( java.util.Observer )`
- *deleteObservers*
  `public synchronized void deleteObservers( )`
- *hasChanged*
  `public synchronized boolean hasChanged( )`
- *notifyObservers*
  `public void notifyObservers( )`
- *notifyObservers*
  `public void notifyObservers( java.lang.Object )`
- *setChanged*
  `protected synchronized void setChanged( )`

B.2.9 *Class* Queue

Queue is a FIFO data structure that stores Message objects. Enqueue inserts a Message in the queue, and deQueue extracts an element from the queue. Calling dequeue from an empty Queue returns null. Both operations take O(1) time. An n element queue takes O(n) space. Test program: QueueTest

Declaration

public class Queue **extends** java.lang.Object **implements** java.io.Serializable

Constructors

- *Queue*
  `public Queue( plateau.queue.Message msg )`
  - Description
    - Creates a one element Queue.
- *Queue*
  `public Queue( )`
  - Description
    - Creates an empty queue

Methods

- *isEmpty*
  `public boolean isEmpty( )`
  - Description
    - Returns true if the queue is empty. Returns false otherwise.
- *getSize*
  `public int getSize( )`
  - Description
    - Returns the number of elements queued
- *flush*
  `public synchronized Queue flush( )`
  - Description
    - Returns a copy of the queue and empties this one.
- *append*
  `public synchronized void append( plateau.queue.Queue q )`
  - Description
    - Appends q to the end of this queue. Note that the elements in q are not copied. Operations on the elements of q will also affect the new elements of this queue.
  - Parameters

        * `q` - The queue that will be appended to the end.
- *enQueue*
  `public synchronized void enQueue( plateau.queue.Message msg )`
  - Description
    * Inserts a Message into the Queue.
  - Parameters
    * `msg` - the message to be enqueued.
- *deQueue*
  `public synchronized Message deQueue( )`
  - Description
    * Extracts the next Message in the Queue. If the Queue is empty, returns null
- *seeNext*
  `public MessageNode seeNext( )`
  - Description
    * Returns a node object that represents the element that will be returned from calling the next dequeue method.
- *toString*
  `public String toString( )`
  - Description
    * Returns a string representation of the queue that is based on the string representation of the Messages it holds.

B.2.10 *Class* GraphicQueue

GraphicQueue is a graphic representation of an IndexedQueue. It paints Colorable Messages as a colored circles in a panel.

Declaration

public class GraphicQueue **extends** java.awt.Panel **implements** java.io.Serializable

Fields

- public static int DEFAULT_DIAMETER
  –
- public static final boolean DEBUG
  –

Constructors

- *GraphicQueue*
  `public GraphicQueue( plateau.queue.MessageQueue queue )`
  - Description
    - Constructor.
  - Parameters
    - `queue` - the priority queue to be displayed graphically.

Methods

- *enqueue*
  `public void enqueue( plateau.queue.ColorableMessage msg )`
  - Description
    - Enqueues a colorable message.
  - Parameters
    - `msg` - the colorable message to enqueue.
- *dequeue*
  `public ColorableMessage dequeue( )`
  - Description
    - Extracts the colorable message at the head of the queue
- *renqueueAll*
  `public void renqueueAll( )`
  - Description
    - Reprioritizes and repaints all messages.
- *setMessageDiam*
  `public void setMessageDiam( int diam )`
  - Description
    - Sets the diameter in pixels of each message.
  - Parameters
    - `diam` - the diameter.
- *refresh*
  `public void refresh( )`
  - Description
    - Refreshes the graphic queue to reflect an up to date message order.

- *handleEvent*
  `public boolean handleEvent( java.awt.Event e )`
- *repaint*
  `public void repaint( )`
  - Description
    * Repaints the message queue.
- *paint*
  `public void paint( java.awt.Graphics g )`
  - Description
    * Paints the queue.

Methods inherited from class `java.awt.Panel`

- *addNotify*
  `public void addNotify( )`

B.2.11 *Class* IndexedQueueEnumeration

This enumeration is returned when calling getElements from an IndexedQueue object. It iterates through the elements of an IndexedQueue without modifying the IndexedQueue. However, the Messages returned from calling nextElement are the actual Messages queued, so modifying them will actually modify the elements in the Queue. To iterate through the elements, call nextElement. There is a hasMoreElements method that returns false when there are no more elements in the enumeration. Calling nextElement in an empty enumeration results in a java.util.NoSuchElementsException.

Declaration

public class IndexedQueueEnumeration **extends** java.lang.Object **implements** java.util.Enumeration

Methods

- *hasMoreElements*
  `public boolean hasMoreElements( )`
  - Description
    * Returns true if there are more elements in the enumeration. Returns false if the enumeration is empty.

- *nextElement*
  `public Object nextElement( )`
  - Description
    * Returns the next Message in the enumeration. The returned object is an Object class, so usually it should be casted back to be a Message. Throws java.util.NoSuchElementException is the enumeration is empty.

B.2.12 *Class* QueueEnumeration

This class implements the java.util.Enumeration interface and allows iteration over the Messages in a Queue without modifying the queue structure.

Declaration

public class QueueEnumeration **extends** java.lang.Object **implements** java.util.Enumeration

Constructors

- *QueueEnumeration*
  `public QueueEnumeration( plateau.queue.Queue q )`

Methods

- *hasMoreElements*
  `public boolean hasMoreElements( )`
  - Description
    * Returns true if there are more elements in the enumeration
- *nextElement*
  `public Object nextElement( )`
  - Description
    * Returns the next element in the enumeration. After the last element of the queue has been returned, subsequent calls to nextElement will raise a NoSuchElementException.

Appendix C

Plateau Scribble Source Code

```
package plateau.test;

import plateau.queue.*;

public class ScribbleMessage extends IndexedMessage {

    public ScribbleMessage(Object contents, int type) {
        super(contents,type,true);
    }

    public ScribbleMessage(Object contents) {
        super(contents,true);
    }

    public final static int DRAG = 1;
    public final static int CLEAR = 2;
    public final static int TALK = 3;

}
```

ScribbleMessageopt:pmn

```
package plateau.test;

import plateau.server.*;

public class TestUser extends User {

  String name;
  int count;

  public TestUser(String n) {
    name = n;
    count = 0;
  }
```

```
  public void add() {
    count++;
  }

  public int getCount() {
    return count;
  }

  public String getName() {
    return name;
  }
}
```

TestUser

```
package plateau.test;

import java.awt.*;
import java.util.Vector;

public class DrawCanvas extends Canvas implements java.io.Serializable {

  Vector lines;

  public DrawCanvas() {
    lines = new Vector(100);
  }

  public void drawLine(Line l) {
    lines.addElement(l);
    Graphics g = getGraphics();
    if (g != null) {
            g.drawLine(l.startx,l.starty,l.endx,l.endy);
    } else {
          //System.out.println("Null graphics");
    }
  }

  public Vector getLines() {
    return lines;
  }

  public void clear() {
    lines = new Vector();
    Graphics g = getGraphics();
    if (g != null) update(g);
  }

  public void paint(Graphics g) {
    java.util.Enumeration ls = lines.elements();
    while (ls.hasMoreElements()) {
          Line l = (Line)ls.nextElement();
          g.drawLine(l.startx,l.starty,l.endx,l.endy);
    }
```

```
  }
}
```

DrawCanvas

```
//Title:        ScribbleUpdateHandler class
//Copyright:    Copyright (c) 1999
//Author:       Rodrigo Leroux
//Group:        Sociable Media

/**
* ScribbleUpdateHandler extends the generic UpdateHandler and
* specifies how each client will process ScribbleMessages.
**/

package plateau.test;

// These are the plateau and java packages needed in this class.
import plateau.server.*;
import plateau.queue.*;
import java.awt.*;

public class ScribbleUpdateHandler extends UpdateHandler {

  ScribbleBoard board;   // This is the application specific
                         // CommonState.
  Scribble scrib = null; // A handle to the top-level class used by
                         // clients.

  public ScribbleUpdateHandler(ScribbleBoard b) {
    super(b);
    board = b;
  }

  public void setScribble(Scribble s) {
    scrib = s;
  }

  // This is the main method.  For each type of Message, it specifies
  // how to change board, the common state.
  public void handleUpdate(Message msg) {
    super.handleUpdate(msg); // allows the generic update handler to
                             // do its job
    if (msg.getID() == Message.NEW) {
        // scrib is not null on client side copies of the update
        // handler.
        if (scrib != null) {
          scrib.println("New user:   " +
                        ((TestUser)msg.getContents()).getName());
        }
    } else if (msg.getID() == Message.BYE) {
        if (scrib != null) {
          scrib.println("User " +
```

```
                    ((TestUser)msg.getContents()).getName() + " Has left");
            }
            System.out.println("Bye user");
        } else if (msg.getID() == ScribbleMessage.DRAG) {
            // local drags are handled locally.  If the drag message comes
            // from a remote client, the line contained in the message
            // will be drawn.
          if (!board.isLocal(msg.getSenderID())) {
              Line line = (Line)msg.getContents();
              board.drawLine(line);
          }
        } else if (msg.getID() == ScribbleMessage.CLEAR) {
            board.clear();
        } else if (msg.getID() == ScribbleMessage.TALK) {
            if (scrib != null) {
              String text = (String)msg.getContents();
              String name =
                ((TestUser)board.getUser(msg.getSenderID())).getName();
              scrib.println(name + ":" + text);
            }
        }
    }
}
```

ScribbleUpdateHandler

```
package plateau.test;

import java.awt.*;
import plateau.server.*;
import java.util.Vector;

public class ScribbleBoard extends CommonState {

  DrawCanvas p;
  //Image offImage;
  //Image offGraphics;


  public ScribbleBoard() {
    p = new DrawCanvas();
  }

  public DrawCanvas getCanvas() {
    return p;
  }

  public Vector getLines() {
    return p.getLines();
  }
```

```
  public void drawLine(Line l) {
    p.drawLine(l);
  }

  public void clear() {
    p.clear();
  }
}
```

ScribbleBoard

```
package plateau.test;

import java.awt.event.*;
import java.awt.*;
import java.applet.Applet;
import plateau.server.*;
import plateau.queue.*;

public class Scribble extends Frame implements java.io.Serializable {

  private int last_x = 0, last_y = 0;
  private ScribbleBoard board;
  TextArea field;
  TextField text;
  Client client;
  ScribbleUpdateHandler handler;
  DrawCanvas dc;
  boolean started = true;
  boolean applet = true;
  Button start,clear;


  public static void main(String[] args) {
   try {
    if (args.length == 0) {
      new Scribble("localhost",7777,Math.random() + "",false);
    } else if (args.length == 1) {
      new Scribble(args[0],7777,Math.random() + "",false);
    } else if (args.length == 2) {
      new Scribble(args[0],7777,args[1],false);
    } else throw new RuntimeException();
   } catch (Exception e) {
      System.out.println("Usage:  java TestAdder <host> <name>");
   }
  }


  public Scribble(String host, int port, String name, boolean applet) {
```

```
  this.applet = applet;
  field = new TextArea(5,25);
  board = new ScribbleBoard();
  ViewPrioritizer prio = new ViewPrioritizer(board.getCanvas());
  IndexedQueue queue = new IndexedQueue(prio);
  handler = new ScribbleUpdateHandler(board);
  TestUser usr = new TestUser(name);
  //sender = new ClientSender();
  ClientSender sender = new ClientSender();
  ClientReceiver receiver = new ClientReceiver(handler,queue);
  //if (DEBUG) System.out.println("created sender");
  client = new Client(host,port);
  client.start(usr,sender,receiver);
  ((ScribbleUpdateHandler)client.getUpdateHandler()).setScribble(this);


  board = (ScribbleBoard)client.getCommonState();
  handler = (ScribbleUpdateHandler)client.getUpdateHandler();
  dc = board.getCanvas();
  queue = (IndexedQueue)handler.getMessageQueue();
  //GraphicQueue gq = new GraphicQueue(queue);
  //gq.setMessageDiam(30);
  prio = (ViewPrioritizer)queue.getPrioritizer();
  dc.addComponentListener(prio);
  ScribbleEventListener sel = new ScribbleEventListener();
  dc.addMouseListener(sel);
  dc.addMouseMotionListener(sel);
  prio.setComponent(dc);
  prio.setLocal();
  Panel top = new Panel();
  top.setLayout(new GridLayout(2,1));
    text = new TextField();
    clear = new Button("Clear");
      top.add(clear);
      top.add(text);
      //top.add(gq);
  setLayout(new BorderLayout());
  add("Center",dc);
  add("South",field);
  add("North",top);
  resize(300,300);
  show();
}

//public void resize() {
 //super.resize

public boolean keyDown(Event e,int key) {
  if (key == Event.ENTER) {
      try {
        ScribbleMessage sm = new ScribbleMessage(text.getText(),ScribbleMessage.TALK);
        sm.setState(false);
        client.broadcast(sm);
        text.setText("");
```

```
        } catch (java.io.IOException ex) {
          client.disconnect();
        }
        return true;
    } else return super.keyDown(e,key);
  }

  public void init() {                                                    100
    System.out.println("For later");
  }


  public void println(String text) {
    field.append(text + "\n");
  }

  /*
  public boolean mouseDown(Event e,int x,int y) {                         110
    last_x = x - dc.getLocation().x;
    last_y = y - dc.getLocation().y;
    System.out.println("mouse down: " + last_x + "," + last_y + " from " + e.target);
    return true;
  }*/

  public boolean handleEvent(Event e) {
    if (e.id == Event.WINDOW_DESTROY) {
        client.disconnect();
        if (applet) {                                                     120
          dispose();
        } else {
          System.exit(0);
        }
        return true;
    } else return super.handleEvent(e);
  }

  public boolean action(Event ev,Object arg) {
    if (ev.target == clear) {                                             130
        ScribbleMessage sm = new ScribbleMessage(null,ScribbleMessage.CLEAR);
        try {
          client.broadcast(sm);
        } catch (java.io.IOException ex) {
          client.disconnect();
        }
        return true;
    } else return super.action(ev,arg);
  }
                                                                          140
/*  public boolean mouseDrag(Event e, int x, int y) {
    //Graphics g = board.getraphics();
    x = x - dc.getLocation().x;
    y = y - dc.getLocation().y;
    System.out.println("from: " + last_x + "," + last_y + " to " + x + "," + y);
    Line line = new Line(last_x, last_y, x, y);
```

```
   board.drawLine(line);
   ScribbleMessage sm = new ScribbleMessage(line,ScribbleMessage.DRAG);
   try {
        client.broadcast(sm);
   } catch (java.io.IOException ex) {
        client.disconnect();
        System.out.println("Connection closed");
   }
   last_x = x;
   last_y = y;
   return true;
}*/

private class ScribbleEventListener implements MouseMotionListener, MouseListener {

 ScribbleEventListener() {}

 public void mouseClicked(MouseEvent e) {
  int x = e.getX();
  int y = e.getY();
  last_x = x; //- dc.getLocation().x;
  last_y = y; //- dc.getLocation().y;
  System.out.println("mouse down:   " + last_x + "," + last_y);
 }

 public void mouseEntered(MouseEvent e) {
 }

 public void mouseExited(MouseEvent e) {
 }

 public void mousePressed(MouseEvent e) {
  int x = e.getX();
  int y = e.getY();
  last_x = x; //- dc.getLocation().x;
  last_y = y; //- dc.getLocation().y;
  System.out.println("mouse down:   " + last_x + "," + last_y);
 }

 public void mouseReleased(MouseEvent e) {
 }

 public void mouseDragged(MouseEvent e) {
    //Graphics g = board.getraphics();
  int x = e.getX();//x - dc.getLocation().x;
  int y = e.getY();//y - dc.getLocation().y;
  //System.out.println("from:   " + last_x + "," + last_y + " to " + x + "," + y);
  Line line = new Line(last_x, last_y, x, y);
  board.drawLine(line);
  ScribbleMessage sm = new ScribbleMessage(line,ScribbleMessage.DRAG);
  try {
       client.broadcast(sm);
  } catch (java.io.IOException ex) {
           client.disconnect();
```

```
            System.out.println("Connection closed");
        }
        last_x = x;
        last_y = y;
    }

    public void mouseMoved(MouseEvent e) {
    }
  }
                                                                    210
}
```

Scribble

Bibliography

[1] Henri E. Bal, R. Bhoedjang, R. Hoffman, C. Jacobs, K. Langendoen, T. Ruhl, and M.F. Kaashoek. Orca: a portable user-level shared object system. 1993.

[2] K.P. Birman, A. Schiper, and P.Stephenson. Lightweight casual and atomic group multicast. *ACM Transactions on Computer Systems, 9(3) pp. 272-314*, 1991.

[3] D. Bogia, S. Kaplan, W. Tolone, and C. Bignoli. Flexible, active support for collaborative work with converstaionbuilder. *In Proceedings of the Fourth Conference on Computer-supported Cooperative Work*, 1992.

[4] A. Chabert. Ncsa habanero home page. *http://havefun.ncsa.uiuc.edu/habanero/*, 1996.

[5] J.M. Chang and N.F. Maxemchuck. Reliable broadcast protocols. *ACM Transactions on Computer Systems, 2(3) pp. 251-273*, 1984.

[6] C.A. Ellis and S.J. Gibbs. Concurrency control in groupware systems. *ACM SIGMOID, pp.399-407*, 1989.

[7] C.A. Ellis, S.J. Gibbs, and G.L. Rein. Groupware: Some issues and experiences. *Communications of the ACM, January 1991 pp. 38-51*, 1991.

[8] J.H. Saltzer et al. End-to-end arguments in system design. *ACM Transactions on Computer Systems, Vol. 2, No. 4*, 1984.

[9] Thomas H. Corven et al. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.

[10] D. Ferrari. Client requirements for real-time communication services. *IEEE Communications Magazine. vol.28 no. 11*, 1990.

[11] Andreas Girgensohn and Alison Lee. Developing collaborative applications in the world wide web. *Proceeding of the CHI 98 Conference*, 1998.

[12] Richard Golding. Weak-consistency group communication and membership. *PhD thesis, University of California, Santa Cruz*, 1992.

[13] Leslie Lamport. Time, clock and the ordering of events in a distributed system. *Communications of the ACM, volume 21(7) pp. 558-565*, 1978.

[14] C.M. Neuwirth, R. Chandhok D.S. Kaufer, and J.H.Morris. Issues in the design of computer support for co-authoring and commenting. *In Proceedings of the Third Conference of Computer-Supported Cooperative Work, pp. 183-195*, 1990.

[15] Larry L. Peterson and Bruce S. Davie. *Computer Networks, A Systems Approach.* Morgan Kauffman Publishers Inc., San Francisco, CA, 1996.

[16] Atul Prakash and Hyong Sop Shim. Distview: Support for building efficient collaborative applications using replicated objects. *Proceedings of the conference on Computer supported cooperative work , Page 153*, 1994.

[17] S. Shenker S. Bajaj, L. Breslau. Uniform versus priority drop for layered video. *Proc. SIGCOMM 98, Vancouver, Canada*, 1998.

[18] M. Stefik, G. Foster, D. G. Bobrow, K. Kahn, S. Lanning, and L. Suchman. Beyond the chalkboard: Computer support for collaboration and problem solving in meetings. *Communications of the ACM, 30(1) pp.32-47*, 1978.

[19] Douglas B. Terry. Managing update conflicts in bayou, a weakly connected replicated storage system. *Proceedings of the 15th ACM Symposium in Operating Systems Principles*, 1995.

[20] Fernanda Viegas and Judith Donath. Chat circles. *Proceedings of CHI 99*, 1999.

[21] Huber Zimmermann. Osi reference model- the iso model of architecture for open systems interconnection. *PhD thesis, University of California, Santa Cruz*, 1992.