A Platform for Distributed Learning and Teaching of Algorithmic Concepts

by

Nathan D. T. Boyd

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degrees of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING
AND
MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 23, 1997

Author _____
Department of Electrical Engineering and Computer Science
May 23, 1997

Certified by _____
Seth Teller
Thesis Supervisor

Approved by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

A Platform for Distributed Learning and Teaching of Algorithmic Concepts

by

Nathan D. T. Boyd

Submitted to the
Department of Electrical Engineering and Computer Science on

May 23, 1997

In partial fulfillment of the requirements for the degrees of
Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science

Abstract

The platform for distributed learning and teaching of algorithmic concepts, or Educational Fusion, is a system for developing and interacting with algorithmic visualizations within heterogeneous distributed computing environments. Students and staff access Educational Fusion over the Internet via any World Wide Web browser equipped with a Java interpreter. Educators use this framework to create concept graphs, composed of topics and modules, that visualize the structure of and relationships between algorithm components. Students instantiate private instances of concept graphs, implementing modules at their own pace. The system interprets and visualizes the output of student implementations, providing hidden reference implementations so that students and staff can assess the correctness of their solution.

Educational Fusion envisions unparalleled communication and interaction facilities, which we believe to be critical to the learning process. Concept graphs serve as both coursework visualizations and virtual laboratories, wherein students can see what peers are working on, locate staff for help, and browse through implementation tips left by others. A conferencing system allows clients to broadcast messages to others. Messages can be targeted at specific individuals or to groups, such as other students working on the same topic, and are archived for later review. Finally, the Java applets in which students develop and test their visualizations are to be shared across the Internet. NetEvents allows two or more students to collaborate from physically distant workstations, and gives teaching assistants a virtual hand with which to guide students through tough problems. Network event streams can be recorded for later playback, whether for capturing exceptional output or for demonstration purposes.

Thesis Supervisor: Seth Teller
Title: Assistant Professor of Computer Science and Engineering, MIT Electrical Engineering and Computer Science Department

Acknowledgements

I would like to gratefully acknowledge Professor Seth Teller and Brad Porter – the "rest" of the Educational Fusion development team – for their countless contributions to my research. Seth, thank you for sharing your vision and advice: this project could not be in better hands. Thanks also for all of your counsel as the principal reader of this thesis. Brad, you laid the first bricks for the foundation of the system we have built – best of luck assuming responsibility for administering the NT graphics network. Intel Corporation and Microsoft Corporation must also be recognized for their generous donations of the hardware and software with which Educational Fusion has been crafted.

I would also like to acknowledge my family, for getting me to MIT in the first place and supporting me all the way through. Mom and Dad, you instilled in me from day one a belief that I need never accept failure, and that my life and my mind should know no bounds. Armed with these convictions, I will succeed: for myself, for my fellow man, and always for you both. Special thanks to Aaron and Kasi for faithfully reminding me that Gods are perfect while Man is merely arrogant.

I have been fortunate to make the acquaintance of some very special individuals while at MIT. Noah and Alex, thank you for helping me through many a long night squirreled away in some subterranean Athena cluster – and for never failing to persuade, or be persuaded by me, exhaustion abounding, to make the most of Boston's night life the following day. And Emily, your support is treasured and can never be forgotten. Despite whatever havoc we may have wreaked, you, my dear, are the eye of the hurricane: my sanctuary.

Contents

Figures and Tables

Chapter 1

A New Paradigm for Learning and Teaching

Learning by doing is a tried and true teaching methodology. In this regards computer science coursework is no different from other disciplines: homework often asks students to implement the concepts, or algorithms, which they are studying. Traditionally, students program independently or in small teams, iterating the standard design, code, compile, and test cycle until they have come upon a solution. But today's technology offers radically new ways for students to program, interact, and collaborate. The development platform discussed in this thesis proposal, Educational Fusion, revisits the traditional paradigm, taking advantage of the unique nature of computational science and the World Wide Web.

This work began with an NSF Career Development Proposal submitted by Professor Seth Teller while at the Massachusetts Institute of Technology [Tel94], in which Teller proposes "an integrated collection of research and educational activities that will increase the efficacy and use of collaborative, interactive techniques for design & verification, experimentation, simulation, visualization, and pedagogy." Currently, the Educational Fusion team consists of Professor Teller, Masters candidates Nathan Boyd and Brad Porter, and undergraduate student Nicholas Tornow.

1.1 Overview

Educational Fusion, hereafter designated by ∈f, is a platform in which educators publish computer science coursework via the World Wide Web (Web). Published content is in the form of interactive Java™ applets which students invoke to learn about, develop, and test algorithmic concepts. These applets provide students with a programming environment insulated from whichever architecture, compiler, revision control system, or set of course libraries is in use. Our aim is to focus students upon the problem at hand by abstracting unnecessary details.

But students do not, and should not, work in isolation. Honest collaboration is a cornerstone of university-level coursework and research. This document focuses particularly on the real-time communication systems by which students and educators, regardless of physical location, can share ideas, seek or offer help, and collaborate in their development efforts. We hope to provide facilities that enable remote collaboration to be at least as effective as traditional classroom and computer laboratory interaction.

As we shall see, most previous online education initiatives amount to migrating traditional course material – whether it be lecture notes or textbooks – to an electronic medium. We believe it is time to take the next step and empower students to put their knowledge to work, and to the test, in an environment that encourages exploration and collaboration while generally making learning more fun.

We invite you to visit the Educational Fusion homepage[1] while reading this document, both for comprehensive information as well as to try out online Educational Fusion coursework.

---

[1] The Educational Fusion homepage, hosted by the MIT Computer Graphics Group at `http://graphics.lcs.mit.edu`, is the best resource for our most recent research efforts and demonstrations of the system. Please note, however, that due to the volatile nature of the Web and our local computing environment, this URL may change. Please direct an email inquiry to `edufuse@graphics.lcs.mit.edu`.

It must be made clear that ∈F is a work in progress: while we have implemented many of the techniques described here, there is much remaining to be developed and refined. Moreover, online learning is such an exciting and unexplored field that we have found each completed component to lead to several we had yet to consider. Thus, the reader should treat this thesis as both a design document and a springboard for pioneering thinking about education.

Many of the examples in this thesis are drawn from material that one could expect in an introductory computer graphics course, as this class is the first for which the Educational Fusion system has been implemented. Foley, van Dam, et al., provide an excellent discussion of these topics in "Computer Graphics: Principles and Practice" [FDFH90].

1.2 Educational Fusion Objectives

Before development efforts were begun, the ∈F team laid out a core set of objectives that we felt must be met if ∈F was to become a viable academic tool. Many of these goals have already been realized, and we believe all of them are within our grasp. Below is an outline of these objectives, and the technologies we are developing to meet them:

1. ∈F should simulate a programming laboratory that provides an intuitive visualization of course material. Students should be able to easily identify work completed and remaining, be free to learn at their own pace, and be encouraged to explore the material in as much depth or breadth as they would like. Progress should be monitored without intrusion, interceding only when a student falls behind, fails to complete a task, or requests help.

⇒ ***Concept Graphs.***[2] A Concept Graph is a Java applet[3] that visualizes the principal algorithms students are to implement in the course – it can be thought of as an

[2] New terminology is introduced in *italics.*

interactive course syllabus. The graph's representation is designed so that students can quickly grasp how the various system components fit together: far too often programming courses take a microscopic view of course material, leaving students with little feel for overall structure. Moreover, when presented with the syllabus in this fashion, students can immediately begin exploration of particularly interesting material, and, as we shall see, weave their own path through the course material. Plate 1, page 112, shows a screen capture of the Concept Graph applet; in this case, the student is presented with the concept graph for a computer graphics course.

2. ∈F should enable and encourage efficacious collaboration amongst students in this virtual laboratory: students should be able to easily locate fellow students and annotate system components with hints and suggestions for others. Further, it will be vital that staff members are readily accessible.

⇒ ***Avatars.*** Avatars within a Concept Graph identify each student and staff member, and follow each individual as he moves through the system. Students can identify Avatars with a mouse click, or can look up a particular student through a search facility that not only indicates whether that student is currently in the system, but also locates the project and topic on which he is working. The white Avatar pictured at the right of Plate 1 indicates that the student is currently working on the Bresenham module, within topic Line Drawing; the neighboring green and pink Avatars tells us that the student is not alone.

3. ∈F should abstract the compilation, visualization, and validation of algorithm implementation, so students can focus on semantics. Expression of algorithms within ∈F should be a simple task, yet one that does not limit the flexibility or scope of the computation at hand.

---

[3] An applet describes an application written in Java designed to be executed within a Web browser. This term connotes a small and limited application, belying how Java's language designers foresaw such software. As Java has evolved, however, this designation has become a misnomer. The Concept Graph applet, for example, constitutes many thousand lines of code.

⇒ ***Workbenches.*** A Workbench is a Java applet encapsulating everything a student needs to work on a given algorithm: an editor, a compiler, a revision control system, a debugger, a representative set of inputs for testing, and an output view all rolled into one. The Workbench compiles code when the student is ready, and keeps a record of each version of the code submitted. It provides a visualization of the output of the student's implementation, as well as that of a reference implementation against which the student can compare his output. Educational Fusion content providers are supplied with a set of classes that facilitate creating additional Workbenches. Although it is crucial to understand the role Workbenches play, this document will not examine their inner workings. For additional information see Brad Porter's work on the ∈F homepage. Plate 2, page 113, presents a screen capture of the Bresenham line-drawing algorithm Workbench, a module from the computer graphics Concept Graph shown in Plate 1.

4. ∈F should provide collaboration facilities that allow students to converse and interact within their virtual classroom as effectively as though they were carrying on a spoken conversation. Context-sensitive messaging will be vital to limiting the amount of stimulus to any particular ∈F student: communications should, by default, be filtered according to what the student is currently working on. Without context sensitivity, our system would be no better than throwing an entire student body into a gymnasium but still expecting each class to operate smoothly.

⇒ ***Conferences.*** Hypertext messages can be directed at individual students or staff members, or at groups of people working on the same topic or project as the sender. Incoming messages can be similarly filtered. Conferencing applets provide the interface for this messaging. Moreover, private conferences may be launched from the Concept Graph between select individuals, supporting both group efforts as well as private tutorial sessions. Conferences are also envisioned to be persistent, allowing students to look over old conversations and serving as an invaluable resource when students get stuck or miss a lecture or group interaction.

5. ∈F should allow students to share their work with fellow students and staff, for demonstration or collaboration purposes. Interactive sessions should be recordable, so

they can be reviewed later. We foresee the educator extending a virtual hand over student shoulders, guiding students towards the correct solution and directing them away from spurious hazards.

⇒ ***NetEvents.*** Perhaps the single most compelling vision of the ∈F team, NetEvents will allow Java applets to be easily shared across the Internet. The principal target for implementing NetEvents is the Workbench applet: students will be able to share their code and demonstrate its behavior with classmates and staff. Multicasting will be built into NetEvents, giving rise to an entirely new mechanism for educators to administer demonstrations that an entire class can both observe and participate in. Like the other ∈F components, NetEvents are fundamentally stateful: a demonstration could be recorded live, and then played back later.

6. ∈F should remain platform independent[4] on both the server- and client-side.

⇒ ***Internet Technologies.*** One of the promises of the World Wide Web has been platform-independence. ∈F meets this goal by relying upon components built in cross-platform languages, namely Java and Perl, that interface with cross-platform services such as Web browsers on clients and the Common Gateway Interface (CGI) on Web servers. Although clearly these are not technologies we have developed, we must use them cautiously as it is easy to compromise platform independence.

This thesis describes the design, implementation, and future directions of the technologies introduced above. As we do so, we shall judge how well we meet those objectives set before us. We believe that the answer is very well indeed.

[4] A software component is said to be platform independent, or architecture independent, if it can be executed without regard to the underlying operating system or hardware. We will use these terms interchangeably.

1.3 Why the World Wide Web and Why Java?

As stated, EF is accessed via the World Wide Web, and clients' primary interface is through a collection of Java applets. Before going further, let us explain why these instruments were selected.

1.3.1 The World Wide Web

The World Wide Web was chosen as the principle medium of the Educational Fusion system primarily because of the ease with which distributed client/server systems can be built [WC97]. Clients can access EF from anywhere in the world, so long as their workstation is equipped with Internet access and a World Wide Web browser supporting Java. Furthermore, the Web is perhaps the most architecture-neutral service in existence: the list of platforms with compliant Web browsers is nearly all-encompassing, including Microsoft Windows, Apple Macintosh, and most UNIX variants.

The exponential surge in popularity of the Web over the past few years is a testament to this. Increasingly, businesses, universities, and personal users are dependent upon Web-based services, information, and even entertainment. People are excited about what the Web has to offer, and EF can only benefit from this enthusiasm.

1.3.2 Java

Java was the natural language in which to develop Educational Fusion: seamless integration of Java applets and the Web browser interface makes for the easiest and most powerful method of delivering interactive content available. Moreover, the Java class library's abstraction of network protocols greatly simplifies incorporating network communications into both server-side applications and client applets. Finally, Java is inherently platform independent, as required by our goal of producing a system accessible by a diverse mix of client architectures. In the words of Java distributor Sun Microsystems, "Applications

created in Java can be deployed without modification to any computing platform, thus saving the costs associated with developing software for multiple platforms" [SM97a].

1.4 Why Limit Educational Fusion to Algorithmic Concepts

Educational Fusion departs from previous online education research significantly in that we focus upon algorithmic concepts, including but not limited to those found in computer science curricula, or more broadly, material that can be presented in algorithmic form. At first glance, this may seem quite limiting.

In a sense, that impression is correct. Clearly there are many domains of learning that will not fit into the ∈F paradigm: it is difficult to imagine how ∈F might facilitate a grammar lesson. On the other hand, we have built a framework in principle capable of representing any computational task – and computer science has become an incredibly rich field of study that is perhaps expanding more rapidly than any other. We do not believe that there is any dearth of relevant coursework to be expressed within our system.

Most importantly, computer science coursework is uniquely suited to online learning. Students grasp the principle of refraction by observing how a beam of white light is spread by a prism, and gain an appreciation for civil engineering by constructing miniature bridges out of only toothpicks and glue. Simulation and hands-on experience brings learning to life, both capturing student interest and generating metaphors to which students can more easily relate. We feel, in turn, that the best way for a student to understand an algorithm is to implement it, and programming requires a computer.

Moreover, computer science is an abstract field that depends upon visualization – perhaps no other field's vocabulary is so rich with metaphorical talk of queues and linked lists. Educational Fusion's Concept Graphs and Workbenches provide visualizations that assist students to construct the metaphors and abstractions that are so essential to understanding.

These interactive elements also give immediate feedback, putting the power of trial-and-error debugging and what-if thinking directly onto the student's desktop.

1.5 Outline of This Document

Having explicated the motivation behind ∈F in this chapter, in the next chapter we turn to related research. We describe several other online education initiatives, and how this one is different, and then examine the principal collaboration technologies in use for distributed Internet applications.

In Chapters 3 through 5, we explore the underpinnings of ∈F, beginning with a system overview that clarifies the nature of the principal ∈F components and how they interact, continuing with an in-depth look at the Concept Graph applet and the technologies it embodies, and finishing with an analysis of the Messaging API that supports ∈F networking and collaboration. Chapter 6 explores how we have implemented this messaging class library within the Concept Graph and Collaborator applets; the latter applet is a preliminary version of the conferencing tool described above.

Finally, we wrap up with a review of how well the ∈F tools and technologies described herein meet the ∈F objectives in Chapter 7, and conclude with a few final remarks in Chapter 8.

Chapter 2

Related Work

∈f is certainly not the first online education tool. Before beginning development, we surveyed existing projects and technologies to avoid duplicating others' efforts. We believe you will agree that our objectives are fundamentally different from, and in many ways a step beyond, the ambitions of previous initiatives.

2.1 Learning and Teaching Tools

Several other educational systems have been developed, but none tackle the challenge facing the ∈f team: devising a framework for understanding *by doing* in an interactive, collaborative environment. Existing electronic education initiatives have focused on applying technology to the *presentation* of ideas normally derived from textbooks, lecture notes, and other static media.

Electronic derivatives of static media, often publicized over the Web as multimedia documents or Java applets, are often more persuasive than paper versions. But they fall short where the author feels they are most needed: helping students leap from reading about an idea to actualizing that idea. As a result, they also suffer from the inherent limitations of any presentation: they fail to provide a useful mechanism for evaluation and validation of students' understanding. Simple online forms, the analogue of multiple-choice or short-answer tests, typically serve as the only means for assessment.

2.1.1 WebCT

World Wide Web Course Tools (WebCT) is perhaps the most comprehensive existing system [Gol+96]. WebCT's collaboration facilities include bulletin boards for persistent discussions as well as real-time chatting. WebCT's other strong point is automation tools for creating online content, including facilities for generating textual and graphical content such as problem sets and textbook material, searchable glossaries, indexed image archives, course content indexes, and timed online quizzes. Finally, student progress is tracked by the system for later review by instructors.

While WebCT does a marvelous job of shifting material from paper to an electronic medium, it offers no real innovations for helping students actually learn material other than making course material easier to locate. A student in a computer graphics course, for example, would be forced to rely upon traditional development tools: online material would be limited to course notes and perhaps instructions on where source code is found or how the compiler is invoked. And although the chat areas and threaded discussion forums are useful, they are not sensitive to context as are those envisioned for the ∈F system, nor do they appear to be "real-time".

Finally, the system appears to rely almost exclusively on the JavaScript scripting language [NC96]. JavaScript greatly enhances the interface capabilities of standard Hypertext Markup Language (HTML) documents delivered over the Web, but it is unsuitable for serious applications: it is a proprietary technology, developed by Netscape Corporation and later mimicked by Microsoft Corporation, that can communicate with a server only through additional proprietary technologies (a la Netscape's LiveWire). As a scripting language its performance and fundamental capacities rule out applications requiring serious calculations or a graphical user interface. Finally, JavaScript support in Web browsers is extremely unreliable. A principle objective of ∈F is to remain neutral toward, and independent of, such proprietary Internet technologies.

2.1.2 Interactive Illustrations and WARP

Brown University's *interactive illustrations*, like ∈f, approach online learning by delivering interactive Java applets over the World Wide Web. Interactive illustrations "make use of a variety of media including text, images, and sounds to present a body of knowledge with which a user may interact" [Col97]. These interactive applets allow one to manipulate a visualization of a concept in order to better understand it. For example, the Color Perception series of illustrations[5] demonstrate the perceived color of various spectra, as determined by the user, of incoming light.

Interactive illustrations are perhaps uniquely complementary to the ∈f system, and indeed we have begun dialogue with Brown to find synergies where our research intersects. One could imagine the Color Perceptive illustrations as a segue to a series of Color Perceptive Workbenches, wherein one implements the various perception models involved. Brown's team has not restricted the problem domain to algorithmic concepts, however, and hopes to model almost any body of knowledge. They see entire disciplines being encapsulated within *microworlds*, interactive illustration aggregates of unlimited scope and complexity [Try97].

The Interactive Illustrations Project is part of the Web-based Academic Resource Project (WARP) [GVC95]. WARP is a collection of Java applets that demonstrate principles covered in the Brown computer graphics curriculum. WARP is conducted by the Graphics and Visualization Center at Brown University, one of twenty-four National Science Foundation Science and Technology Centers. One of the fundamental shortcomings of WARP and interactive illustrations as currently implemented is that they lack any higher form of organization: a loose collection of applets is certainly illustrative, but of limited use

[5] http://www.cs.brown.edu/research/graphics/research/illus/spectrum/

to a student seeking greater understanding of larger concepts. System-wide appreciation of course material is a priority of ∈f.

2.1.3 Other Online Education Initiatives

Other principal teaching tools available follow the WebCT paradigm of coursework publication. They include the Yorick Project [Mar96], which provides a Web interface to client-side tools, PLATO [TRO97], offering course material specifically targeted at different types of students, and Hamlet [Zac+97], focusing on tutorial-style delivery of course material and innovative presentation via the Hamlet external viewer. All suffer from the same shortcomings as WebCT: they simply relocate current teaching paradigms to the Web, failing to exploit the broader possibilities of the electronic medium. To our knowledge, Educational Fusion is the only system providing interactive, online development of course material, and is the only system whose collaboration facilities are designed to be closely approximate the interactions of students working together side by side.

2.2 Collaboration Technologies

We now focus our attention on Internet collaboration technologies.

2.2.1 T.120 Series Protocols

The T.120 Standards for Audiographic Teleconferencing is an open, international standard adopted by the International Telecommunications Union (ITU) and many key vendors, including Apple, AT&T, Intel, Microsoft, and PictureTel. T.120 permits sharing multimedia data between heterogeneous, graphically dispersed terminals. An immediate parallel to the collaboration objectives of ∈f can be made: both technologies hope to share information between distributed clients running diverse operating systems. However, the T.120 protocols have focused on solving a much narrower problem: how to very efficiently share multimedia data over various network channels [DC95].

T.120 was designed to enable sharing live audio and video over contemporary networks, including telecommunications services such as Public Switched Telephone Network (PSTN) and Integrated Switched Digital Network (ISDN). The challenge of providing advanced digital service over analog telecommunications networks has resulted in a network stack for T.120 that is several layers thick. Implementation is an entirely nontrivial task, and, despite the platform independence of the protocols themselves, ties the developer to a single vendor and operating system. After perusing the International Telecommunication Union's T.120 protocol specifications[6], which amount to several hundred pages, it became apparent that incorporating T.120 into an application in a timely fashion would require a specialized commercial software development package.

Educational Fusion's collaboration tools, on the other hand, leverage the Java networking API, which provides powerful abstractions for TCP/IP networking. Chapter 5, Messaging, describes how the ∈f messaging model completely abstracts network details from client applications, making it extremely easy to build robust networked Java applets. Our highest priority has been to provide a clean network programming interface, as we require ∈f to be an open and extensible system upon which others can easily add function and content. As such, we have chosen to wait until computer networking is reliable and fast enough such that incorporating audiovisual conferencing into the system will not require sophisticated protocols such as T.120.

2.2.2 Java Networking

Despite the powerful networking primitives packaged with Java, relatively few serious uses have emerged. Most networking applications available now are primitive client-server chat and whiteboarding applications that are unreliable and hurriedly thrown together. Java's

[6] T.120 Protocol Specifications are available at the International Multimedia Teleconferencing Consortium's FTP repository, `ftp://ftp.imtc-files.org/imtc-site`.

infancy is likely primarily responsible for the dearth of substantive research and development, as few university researchers and corporate developers have embraced it. Moreover, the Java core API is still in flux and there has been a scarcity of development tools. This is rapidly changing, however, as Java proves itself in the corporate world and makes inroads in university classrooms [Bow97].

A notable exception is the Shared Object System (SOS) for Java, which defines a "lightweight form of distributed objects" called *shared objects*:

> More precisely, shared objects appear as ordinary Java objects on the client, with the additional property that synchronized methods will be executed – and executed in a consistent order – on all identically-named instances of the same class, across all clients of the same server. Any unsynchronized methods are executed locally on each client. Latency is minimized because clients are only required to obey the ordering constraints for synchronized methods (individually for each client object), not to run in a truly synchronous manner for any call [Bur96].

Although no implementation has been publicized, in theory this Java analogue of Remote Procedure Calls (RPC) would greatly facilitate creating distributed, shareable Java applets.

A similar technology will be bundled with the upcoming release of the Java Developer's Kit v1.1.1, known as Remote Method Invocation (RMI). RMI differs from SOS in that methods can be remotely invoked between clients and the server from which they were spawned (as opposed to peer-to-peer method calls) [Shi97, SM97b].

As these technologies mature, it will be vital that our team assess their applicability to ∈F.

2.3 The MIT Environment

Several Educational Fusion capabilities were inspired by tools currently in use in undergraduate and graduate coursework here at the Massachusetts Institute of Technology. Most notably:

- ***Zephyr.*** Zephyr is a ubiquitous real-time messaging system for clients of Athena, the campus computing network. Zephyr messages, or *zephyrgrams*, can be directed to anyone logged into Athena or to any of many public *zephyr instances*. A zephyr instance is comprised of all clients who have subscribed to that instance; for example, all students in 6.837, the Computer Graphics course, might be automatically subscribed to the "6.837" instance upon registration. Other instances are more abstract, including facilities that notify you when mail has arrived or a friend has logged in. Messages directed to zephyr instances are recorded to a log file which can be read later. Many different zephyr clients have been developed – ranging from simple command line tools to elegant X-windows applications. For many, zephyr has become indispensable, and is depended upon to get help from peers, request problem clarification from Teaching Assistants, or simply share late-night humor [Coh94].

- ***Course Homepages.*** Like those of many universities, MIT's courses increasingly have associated homepages on the World Wide Web. Repositories for lecture notes, problem sets and solutions, exams from previous years, and links to related resources, course homepages have proven to be invaluable both as resources for students and as a means for educators and their assistants to drastically reduce administrative overhead. Another exciting use of these pages is as a showcase for students to demonstrate their achievements in the course, often in the form of a final project or report, encouraging them to explore one another's work and potentially collaborate in the future.

- ***MOTD.*** Certainly no invention of MIT, the Message of the Day has prevailed through the years and remains an important element of computer science courses, especially those in which course software and toolkits may change from day to day. What has become obvious, however, is that a "passive" MOTD is not adequate: students need to be notified when there has been a change to the MOTD, else they will no doubt not bother to examine it.

∈f incorporates elements of all of the above, binding them into a package that makes them readily accessible.

Chapter 3

System Overview

The Educational Fusion system is a complex one, and before examining any one technology we have developed in depth, it is vital to step back and identify the principal components and understand their interactions. We shall impart this understanding by following the intercourse of a hypothetical student with ∈F, at each juncture pausing to examine what is going on behind the scenes.

3.1 Entering Educational Fusion

The ∈F homepage is the starting point for anyone unfamiliar with the system. Here students will find valuable information about the purpose of ∈F, an overview of the fundamental system components, related documents and white papers, as well as references to related research. Of course, the entry point to the system itself is also located here in the form of a hyperlink[7] to the Login page.

[7] A hyperlink is a reference within an HTML document to another HTML document, which may reside anywhere else on the Internet. How that document is delivered is not relevant: a hyperlink gives only the address at which a document may be requested.

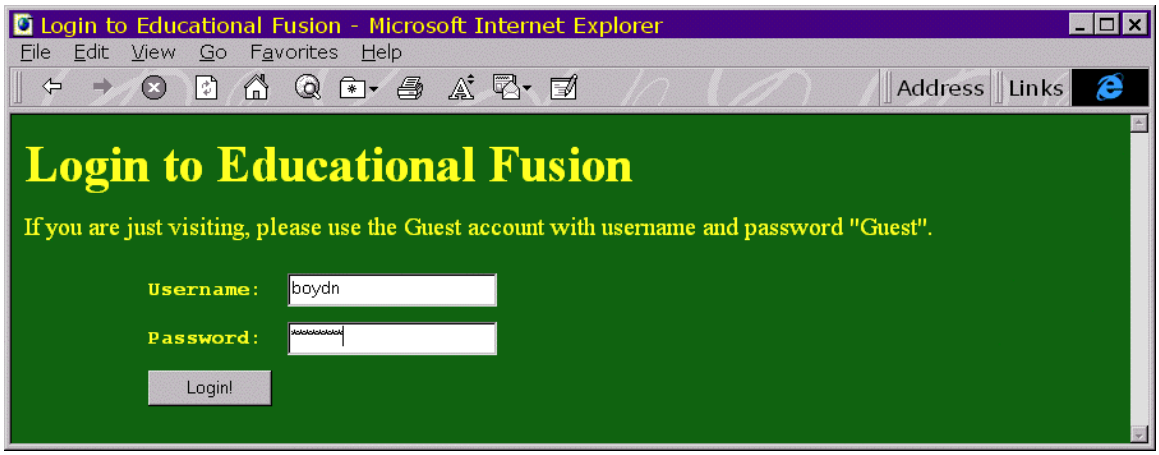

Figure 3-1: Educational Fusion Login

The Login page shown in Figure 3-1 is the entry way into the system, and is the first dynamic document, or dynamic page, dispatched by the Web server. The distinction between *static* and *dynamic pages* is as follows: static pages, such as the homepage itself, are HTML documents that are retrieved from disk and fed over the Internet to the client; dynamic pages, such as the Login page, consist of output from a script invoked by the Web server. The scripts that produce dynamic pages for the ∈f system are collectively referred to as Dispatch Scripts, and are written in Perl [Chr96].

Once the student has successfully logged into the system – a second dynamic page notifies the client if her password is not accepted – she is presented with a list of available *projects*, as shown below in Figure 3-2:

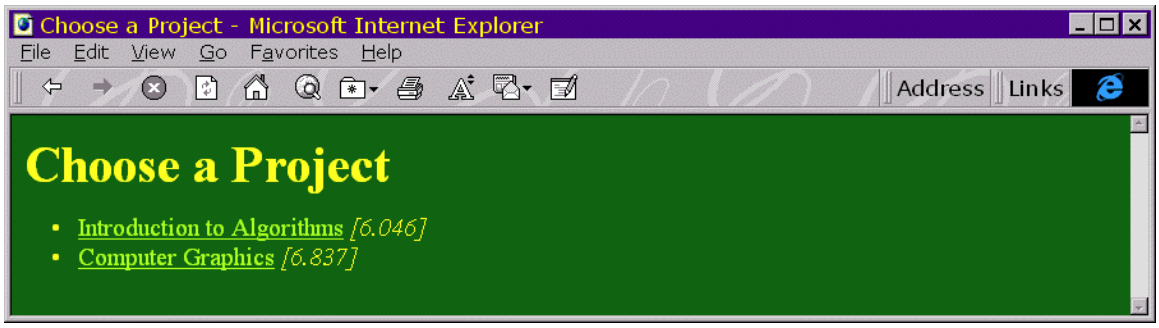

Figure 3-2: Project Choice Page

Each project may represent a single course, as in this case, or alternatively a course might consist of several projects, each project encapsulating a course subject. After selecting the project which she would like to work on, the ∈F Laboratory is opened, as pictured in Plate 4, page 115. The Laboratory page is split into four panes, or frames, including the Concept Graph, the Workbench, the Collaborator, and a frame containing project- or course-related links. The image in Plate 4 shows that the Workbench frame has already been opened to the Bresenham module, but in practice the frame initially supplies directions for the ∈F components.

3.1.1 Educational Fusion Server Configuration

Currently, ∈F is being hosted on a Microsoft Windows NT 4.0 Server machine running the Internet Information System (IIS) 3.0 Web server. Note, however, that these choices are arbitrary: in no way does ∈F rely upon special properties of the NT operating system or IIS. This same machine hosts the ∈F directory hierarchy, containing all documents, scripts, applets, and student state as listed in Table 3-1 below.

Directory	Contents	Virtual Path
Documents	Static HTML Documents	EduFuse
Projects	Project State	
Scripts	Perl Dispatch Scripts	EFScripts
Templates	Dynamic HTML Templates	
Users	Student and Staff State	

Table 3-1. Educational Fusion Directory Organization

The Virtual Path column indicates the path by which Web clients access the Educational Fusion files. Rows with no virtual path contain state that is not meant to be accessible via the EF Web server: access is provided indirectly through the Perl Dispatch Scripts and the EF Registry, as described later. For example, within the Users area there are folders for each registered EF student, and within each student's folder is all work completed and underway by that student. An additional top-level area, Feedback, is under construction to maintain bug reports, comments, and other EF visitor feedback.

3.1.2 Perl Dispatch Scripts

As mentioned above, after a student reaches the Login screen, all documents retrieved from the Web server are the output of the Dispatch Scripts. The role of these scripts is threefold: to customize pages to be delivered to each client, to provide administrative features such as creating new student accounts, and to authenticate and track student sessions.

Upon a request for a dynamic document, the Web server invokes the appropriate Dispatch Script via the Common Gateway Interface (CGI) [NCSA95]. Arguments to the script are passed via the URL of the document requested by the student's Web browser, and optionally the contents of the form being submitted (the Login screen is an example of a form containing two fields: username and password). The script receives these arguments in the form of environment variables. This sequence of requests and responses involved in retrieving a single dynamic page is illustrated in Figure 3-3 below.

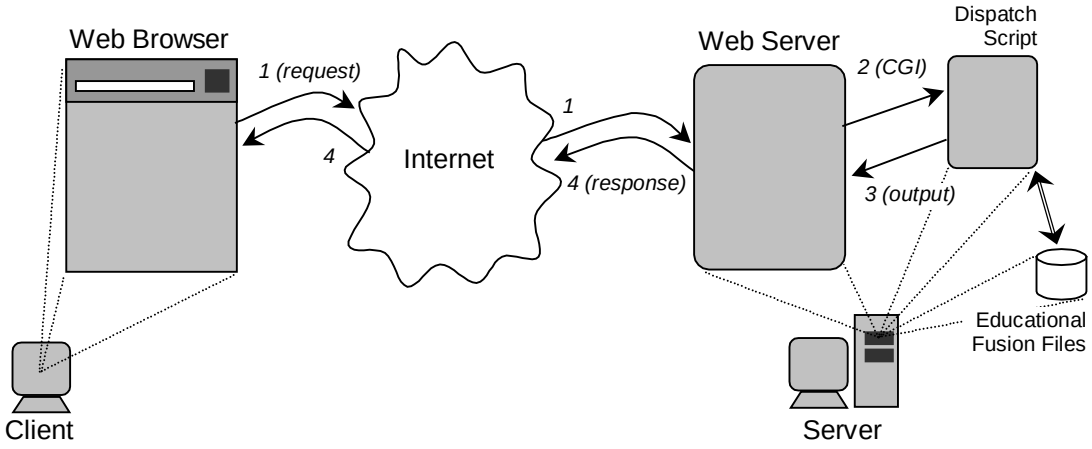

Figure 3-3: CGI Diagram

Dispatch Scripts are so named because their primary function is to process requests from the Web server and dispatch the appropriate response. A typical response is an HTML document that requires missing field to be filled in, which we refer to as a *template*. An example is the Project Choice page: at runtime the Dispatch Scripts determine which projects are available, and for each creates a hyperlink to that project's Laboratory. These links are then inserted into the Project Choice template, and the resulting page is returned to the browser.

While this multi-tiered series of requests and responses is not as efficient as direct calls to Web Server APIs, it meets our requirement of architecture independence. The CGI interface is universally supported, and the Dispatch Scripts are trivially ported to any platform supporting Perl 5, as does almost every modern operating system [Chr96].

3.1.3 Client Persistence

A well-known problem facing Web applications is client persistence: Web browsers and servers communicate via the Hypertext Transfer Protocol, a stateless protocol that requires a reconnect for each request and response [GN97]. HTTP does not provide any means for

the server to identify which client is making a request (other than unsafe means such as the IP address of the machine making the request).

The most popular solution is to identify clients with *cookies*, small stateful objects kept on the client [NC97a]. Yet this method violates a tenet that Web application state should entirely be maintained on the server, for reasons of both client privacy and security; indeed, many people disallow their browser from storing cookies.

Our solution is to maintain client state by means of Universal Resource Locators (URLs) [Ber97] within each hyperlink: the URL identifies which document to retrieve when the hyperlink is selected. A URL has the following structure:

`http://`***`host/document/path_info`***`?`***`query_string`***

`host` identifies the machine on which the Web server is running, either by IP address or name, while `document` is the full path to the requested document or script. `path_info` and `query_string` are optional arguments passed to CGI scripts. In the case of dynamic pages, the `query_string` contains the key-value mappings for a submitted form.

Unlike `query_string`, the `path_info` field is not supplied by the Web browser. Rather, it must be part of the URL when the page is loaded: as the Dispatch Scripts fill HTML templates, they insert information about the requestor into the `path_info` field within each hyperlink on the page. Thus, any hyperlink selected on a returned page will uniquely identify that client. When the Login page is opened there is no information about the client, hence `path_info` is empty – but once the student has logged in, `path_info` will contain at least the client's username and password.

3.1.4 Client Authentication

Client authentication is currently straightforward: upon receiving a Login request, the Dispatch Scripts verify the client's username and password against a password file in the

Users directory (see Table 3-1). Passwords are encrypted with a UNIX-style `crypt`[8] function so that when they are returned to client, within the `path_info` field of hyperlinks, the cleartext password is not transmitted.

Note that this security model is still quite weak: a packet sniffer could easily discover the password when the client initially logs in (recall that the password will be submitted within the `query_string` of the login form submission). Further, a sniffer could steal the crypted password, and thus impersonate that client. To combat the latter, ∈f performs a simple encryption on the entire `path_info` string.

Nonetheless, ∈f remains a system that fundamentally relies upon trusted clients. Clearly this will have to be remedied before the system is put into practice outside of the research environment. The easiest way to do this will be to use a Web server that guarantees channel security: technologies such as the Secure Sockets Layer [NC97b] do this by layering TCP with an encryption protocol that encrypts higher-level network protocols, including HTTP.

3.1.5 Online Administration

A special Administrator account exists for the purpose of maintaining ∈f student accounts and projects. When a client logs in as the Administrator, the Administrator Menu is returned rather than the Project Choice page.

[8] Text in `fixed-width type` denotes variables, methods, and excerpts from ∈f programs and scripts.

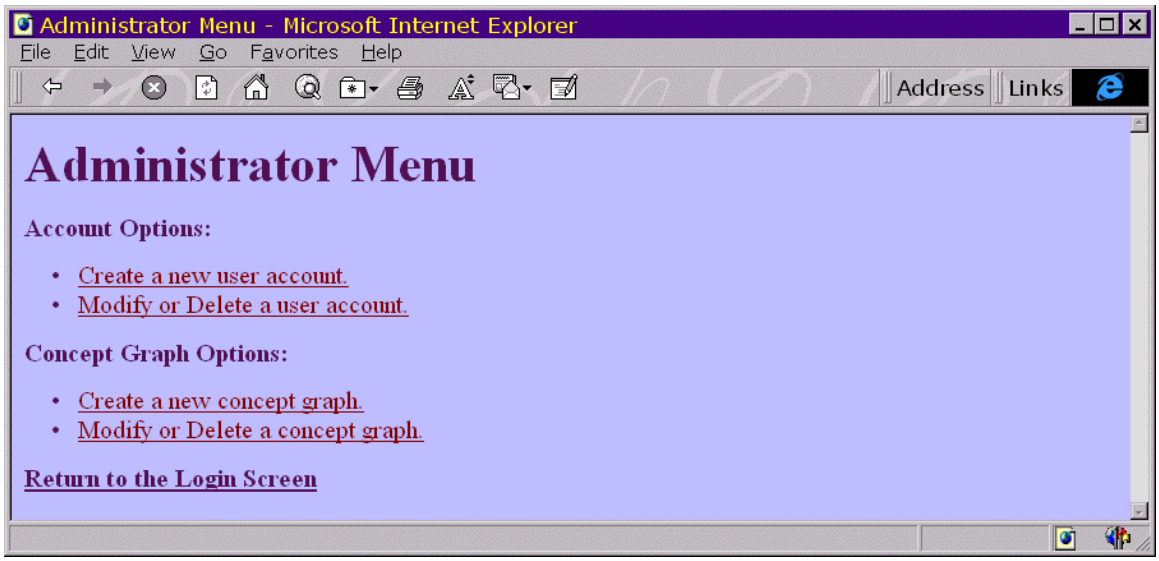

Figure 3-4: Administrator Menu

This facility automatically builds file areas for new students and projects, and is an easy way to maintain their properties, as demonstrated in Figure 3-5.

Figure 3-5: New Account Creation Page

EF records the following properties for each student: `username`, `nickname`, `first` and `last name`, `password`, and `account type`. The `account type` is used to control access privileges, as we shall see later. These properties are stored within the student's file area, which you will recall is a subdirectory within the Users area.

Each project is associated with a `course name`, `course number`, `course homepage URL`, and `directory`. These fields accommodate many-to-one relationships between projects and courses, as aforementioned. As with student accounts, these properties are stored within the project's file area, a subdirectory named by the project's `directory` within the Projects area.

3.2 Loading a Concept Graph

We now return to examining the session of the student who has just loaded the ∈f Laboratory. The Concept Graph frame is loaded with a dynamic page containing a reference to the Concept Graph applet. This reference gives the location of the applet on the Web server as well as supplying applet parameters; these parameters were customized by the Dispatch Scripts, and include student and project identification. When a Java-enabled Web browser loads a page with an applet reference, it immediately makes a new request for the applet code from the server and upon receipt initializes the applet with the reference parameters. In the case of the Concept Graph, initialization consists of the following steps:

1. Opens a socket to, and registers with, the ∈f Registry. As detailed in Section 5.2, the Registry is a Java *servlet*[9] that maintains the location and identification of all students and staff currently logged into the system, provides services such that clients can access this registration information, and provides remote file services (enabling persistent server-side state for clients). Figure 3-6 illustrates this process.

[9] A servlet is a Java server application, traditionally one that specifically supports Java clients. Recently, however, the role of servlets has expanded to include services such as HTTP and FTP.

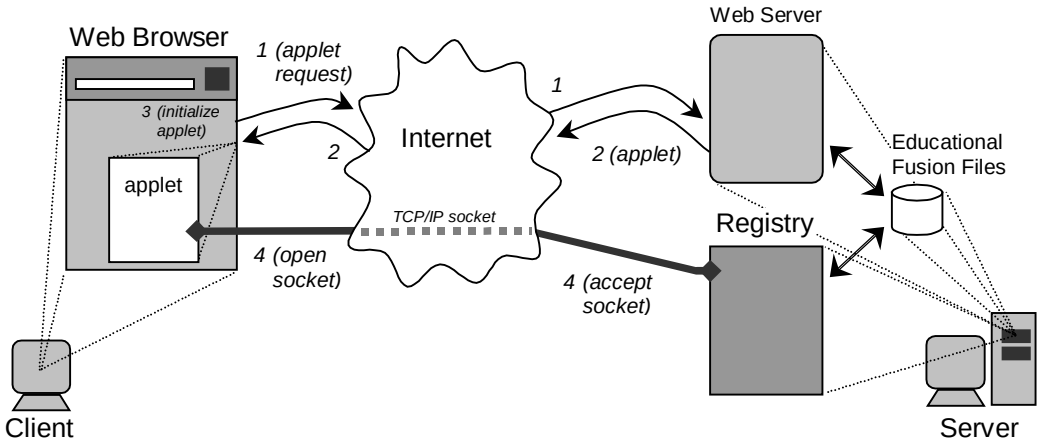

Figure 3-6: Concept Graph Initialization and Registry Connection

2. Attempts to load the student's default concept graph for that project. If none exists, the Registry will supply the default concept graph associated with the project (which is created by the course instructors).

3. Sends a request to the Web Server to load the Collaborator dynamic page into the Collaborator frame. These two steps are depicted in Figure 3-7.

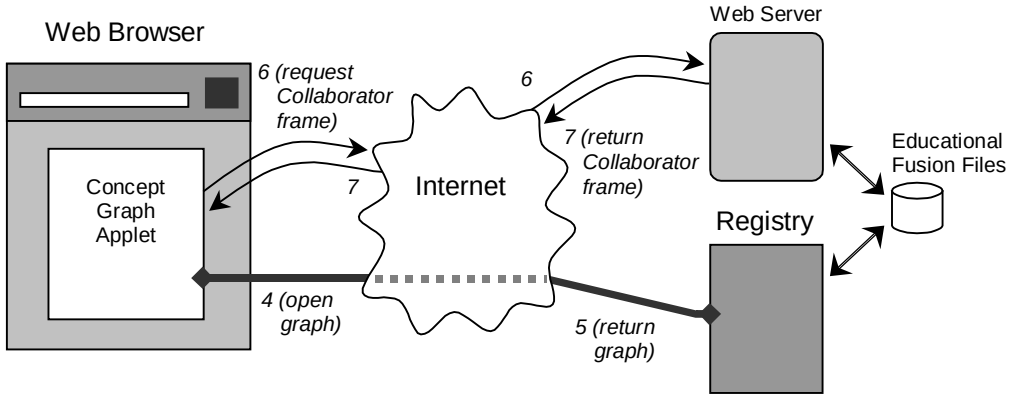

Figure 3-7: Opening the Default Concept Graph and Collaborator

We should note that the Java class files for the EF applets are not maintained within the EF directory hierarchy outlined in Table 3-1, but instead within a Java publishing area of the

Web server. The reason for this is to maintain a unified *codebase* for Java classes: when requesting a Java applet that is composed of many classes, the browser supplies a path, or codebase, where these classes are located. Since we wish to share the various ∈f APIs with other applets, and reciprocally the ∈f applets make use of foreign APIs, we elected to locate all APIs in a common location.

Unlike the stateless HTTP connections to the Web Server, ∈f Java applets maintain an open connection – a TCP/IP socket – to the Registry. Such a persistent connection is required for performance reasons, as shall be elucidated in Chapter 5.

3.3 Loading a Collaborator

As described above, the Concept Graph applet makes an HTTP request of the Web server for a dynamic document containing the Collaborator applet. As with the Concept Graph, this page contains an applet reference, which triggers the same loading and initialization process. Upon initialization, the Collaborator connects both to the Educational Registry and the Message Server. The Message Server handles collaboration-specific networking, such as chatting and whiteboarding.

3.4 Loading a Workbench

Educational Fusion Workbenches are also loaded from the Concept Graph applet, through a process analogous to loading the Collaborator. As described in Section 4.2, when a Workbench module is activated from the Concept Graph, the applet requests that the Workbench dynamic be loaded into the Workbench frame. From there, the Web Browser takes over, requesting the Workbench applet from the Web server and initializing it upon receipt.

Each Workbench supports three modes for visualizing the encapsulated algorithm: *reference*, *student*, and *difference*. Reference mode visualizes the output of the reference implementation

supplied by the module creator, while student mode visualizes the output of the latest student implementation. Difference mode visualizes the difference between the reference and student implementations, making use of a *difference engine*, also supplied by the module creator, to compare the output of the two. The Workbench for the Bresenham line-drawing module shown on Plate 2, page 113, is in difference mode, clearly indicating which pixels are incorrectly drawn by the student's implementation. Plate 3 on page 114 shows this same Workbench, only now with a correct implementation.

3.4.1 Workbench Dynamic Class Loading

As aforementioned, the technology underpinning the Workbench applet will not be examined in detail. However, the system-level workings of the applet require review.

Workbench applets are unique in that they dynamically reinstantiate runtime classes: this is done so that student implementations of the algorithm at hand can be integrated into the functioning of the applet without reloading it entirely. When a client depresses the "Evaluate" button, see Plate 2, page 113, the applet initiates the following process:

1. The student's implementation code is marshaled as a form submission and sent to the Web Server, which forwards the arguments to a Dispatch Script. Upon receipt, the script parses the arguments and retrieves the student implementation code.

2. The Dispatch Script inserts the student code fragment with a student implementation class template, and invokes a Java compiler on the server-side to build the class.

3. Output from the compiler is redirected to a temporary file, which the script returns as its standard output. The Web Server then forwards this to the Workbench applet as a response to its form submission in step 1.

4. If the compile is successful, the applet reloads the class from the server. The applet does this by invoking the same mechanism by which its containing browser initially loaded the applet.

Thus, the applet has re-instantiated one of its classes (or, more precisely, an instance of that class) and the student's implementation will now be visible and operational.

3.5 System Review

There are, in summary, four basic modes of communication employed between the ∈F components:

- Web Browser to Web Server.
- Web Server to Dispatch Scripts (both running on the ∈F server).
- Java Applet to Web Server.
- Java Applet to Java Servlet.

Client persistence is accommodated by encoding client state in the URL of all hyperlinks. This state is passed to Java applets via parameters supplied by the Dispatch Scripts; since ∈F applets maintain stateful connections to the Registry and Message Server, no further wizardry is needed.

Chapter 4

Concept Graphs

We now focus our attention on what is, in many ways, the heart of the Educational Fusion virtual classroom: the Concept Graph applet. The Concept Graph embodies an abstraction of a concept and serves a viewport through which students can explore that abstraction, focusing on the system in its entirety or perhaps a particular sub-problem. Moreover, this applet visualizes the activities of fellow students and course staff, providing for the first time a means for students to instantly locate peers working on the same problems or find a Teaching Assistant of whom to ask a few questions.

After first discussing the abstraction upon which the Concept Graph is based, we shall examine how our implementation realizes this abstraction. We will then turn to the persistence mechanism by which we maintain Concept Graph state across student sessions, and wrap up with implementation details that might not have been explicit in the preceding discussion.

4.1 The Concept Graph Abstraction

Concept abstraction, in the ∈f model, consists of breaking down a concept into a set of *topics*, or sub-problems. This set forms a *concept graph* describing that concept; note the distinction between "Concept Graph" and "concept graph", the former of which constitutes the applet that visualizes of the latter. As discussed in Section 1.4, we restrict the concept domain to

computable problems, implying that each sub-problem can be solved by a computational algorithm. There may be several algorithms that solve a particular problem, just as there are countless ways to sort a list of numbers, and we refer to each potential topic solution as a *module* for that topic. Once an adequate subset of sub-problems is solved the concept as a whole is also solved; the semantics of solving a concept graph are elaborated below.

4.1.1 Topics

An implemented topic, representing one component or sub-problem of the concept, is of limited use in isolation: what good is a Gupta-Sproull anti-aliased line drawing algorithm without any lines to draw? One of the principle lessons of computational science is that abstraction enables reuse, and so it follows that an algorithm's utility is not fully realized until it is in the presence of other algorithms with which it can engage, either by invoking them or being invoked by them. Concepts graph topics are *linked* to codify *invokes* relationships: a direct link from an invoking topic A to an invoked topic B indicates that A makes use of B. More precisely, an algorithm that implements topic A may invoke an algorithm implementing topic B.

Invocation links thus allow us to build directed graphs that embody the structure of the computational system being modeled. Each topic is implicitly associated with an *interface* that defines the arguments it accepts and the nature of its output: a Line-Drawing topic might accept four integers representing the line endpoints and return a set of integers representing the coordinates of each pixel along the line connecting them.

Currently, the concept graph designer assumes responsibility for ensuring that topic links do not violate any interfaces. A subject of future research is to enable topics to publicize their interface, so that the system can do such type-checking automatically. This solution is much more elegant, and frees concept graph designers from needing to anticipate every possible relationship within a graph.

4.1.2 Modules

A module actualizes one implementation of a topic. For example, the Line-Drawing topic in the Computer Graphics concept graph in Plate 1, page 112, includes modules for the Bresenham, Gupta-Sproull, and Incremental algorithms (all well-known methods for drawing lines). While all modules within a topic fundamentally solve the same problem, they may be of varying complexity and efficiency.

The module is more than just a placeholder: as educators construct concept graphs, they are expected to provide a *reference implementation* for each module. This implementation is a correct solution, and while the source code is invisible to students, as described in Section 4.2.2, its output is accessible to serve as a benchmark against which student implementations can be compared.

Like topics, modules are connected by invocation links that denote caller-callee relationships. These links determine the actual topic implementation that each module relies upon, whereas topic links govern the permissibility of such links: module A may be linked to module B only if A's topic is linked to B's topic.

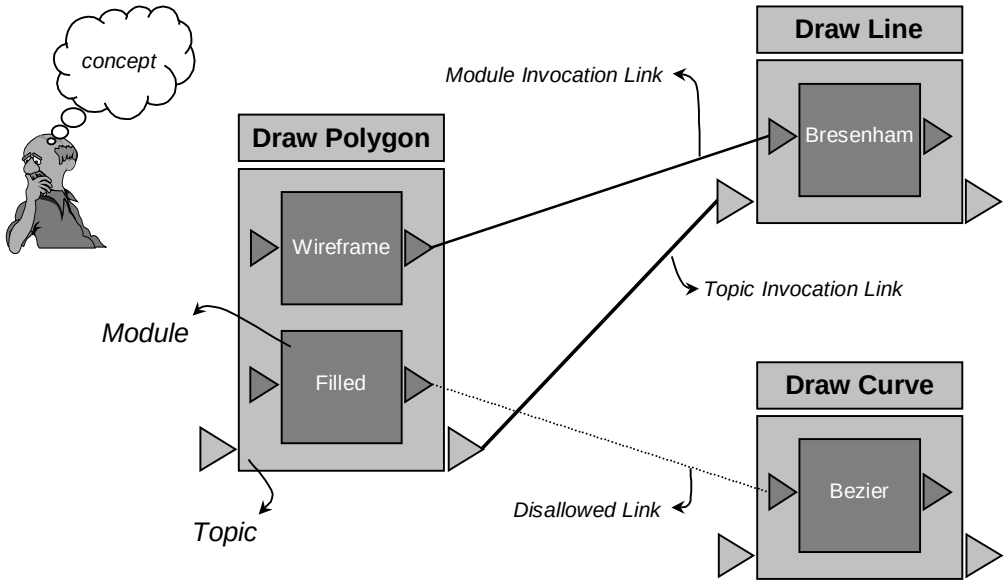

Figure 4-1: Concept Graph Components

4.1.3 Solving a Concept Graph

Solving a concept graph is not a well-defined property: course educators should be free to direct students through whichever path they choose, and, in turn, provide flexibility for students to focus on areas of principal interest. Nevertheless, there are some recognizable properties of a "solved" graph.

Conquering a concept graph involves implementing each module upon a particular *trajectory* through the graph's modules. This trajectory is generally acyclic, and will often visit every node (module), or at the least run from a *root* of the graph to a *leaf*. We define a concept graph's root to be any module within a topic not linked to by any other topic; in other words, a module which no other module in the system relies upon. Conversely, a leaf is any module within a topic not linked to any other topic; a module that invokes no others. Trajectories certainly need not be linear, and indeed this designation may be something of a misnomer as a trajectory may more closely resemble a tree than a path.

Determining if a module has been conquered requires determining whether or not the algorithm it represents has been adequately implemented. This is a question that as of now must be assessed by a member of the technical staff. Automating this process raises many interesting issues, and will be discussed in greater depth in Section 7.6, Future Directions.

4.1.4 Expressibility

Educational Fusion was designed to accommodate virtually any computational concept that can be broken down into specific algorithms or modules, but in many ways we have neglected the most difficult part of doing so: providing content that effectively visualizes these concepts. We believe our framework to be very effective and extensible, and we have discussed extending it to accommodate inheritance, modules that encapsulate entire sub-graphs, and so forth. However, it is only a specification, and implementing a concept graph and the module Workbenches therein is a challenging task. With this in mind, the ∈F team has redirected some of our efforts to soliciting educators to work with our system, and help us discover of what exactly it is capable.

4.2 The Concept Graph Applet

With an understanding of what a concept graph is, we can now turn to the Concept Graph applet in which educators construct and students explore concept graphs.

4.2.1 The View

As depicted in Plate 1, the principal visual feature of the Concept Graph applet is a two dimensional visualization of the graph. This scrollable, zoomable canvas has been implemented such that students can quickly and effortlessly navigate the course concepts.

The represented concept graph is expressed in real coordinates, and mapped to integral screen coordinates only upon display. The bounds of the view can be changed at any time, given that the client has sufficient access privileges (Section 4.3.1). For example, a region of

a concept graph might be hidden from student view while course developers finish implementing topics and modules therein.

We implement double-buffering to smooth animated effects such as scrolling, zooming, and moving graph components. Moreover, the view maintains *dirty* and *refresh* rectangles to minimize recalculation of the buffer and repainting the buffer to the screen. Each time the view is requested to paint itself, it first consults the dirty rectangle and recalculates all visual features that intersect it, and only then refreshes the portion of the buffer contained within the refresh rectangle to the screen. Without these optimizations, Java's graphics API is simply too slow and unwieldy.

Java lacks a unified way to manage colors, so like all ∈f applets, the Concept Graph color scheme references a static ColorMap class that provides a convenient mapping of color names in string form to instances of the Color class. Mostly, this allows concept graph creators to conveniently color graph components.

4.2.2 Topics, Modules, and Invocation Links

Topics are represented as rectangular components with their name appearing in a box directly above. Modules are shown as rectangles within topics, their names centered within. Invocation links are the lines connecting topics and modules via *connectors*, or terminals, appearing on the left and right component borders: the left terminal connects to incoming links, the right terminal to outgoing links. As such, concept graphs are generally laid out such that root topics appear on the left with the graph flowing to their right, ending at the leaf topics.

As aforementioned, each module encapsulates both a reference as well as a student implementation: students can choose which algorithm to visualize within their Workbench, and it is this algorithm that is actually invoked by connected modules. Modules are colored according to which implementation is selected; in Plate 1, green modules expose the student

implementation, blue the reference implementation, and red the difference engine that computes the difference between the two.

The class hierarchy of which topics and modules are members is completely generic: each class is a subclass of a ConceptGraphComponent class. A ConceptGraphComponent, or CGComponent for brevity, defines the fundamental visual and abstract properties of the component (it is likely that this class hierarchy will eventually be split into distinct view, control, and abstraction hierarchies), as well as all methods that act upon them. The abstract properties of a CGComponent include instance variables (variables making up the state of an instantiated class object) for a parent CGComponent, a set of child CGComponents, and a set of incoming and outgoing CGComponents. Thus, the underpinning is the component hierarchy could support a graph of nearly unlimited flexibility; one notable limitation is multiple inheritance, which is also absent from the Java language.

The topic and module subclasses restrict this flexibility accordingly. Each insists that it only be connected to components of like class. Topic children must be modules, while modules are not allowed to have children. Topics do not have parents, whereas modules may or not have one; the latter supports "utility" algorithms that are not part of the required course material but which are to be accessible by students. Each topic is responsible for ensuring that no contained module is linked to a module within a topic to which it is not connected, enforcing the rule that module A can link to module B only if A's topic is linked to B's topic. Modules ensure that they are connected to at most one module in any given topic, else it would not be possible to determine which algorithm they are to invoke.

4.2.3 Status Bar and Debug Output

The final visual features of the applet are the Status Bar, appearing along the top edge of the applet, and the Debug Output appearing along the bottom edge. The Status Bar identifies the mouse cursor's position within the view in real coordinates, notifies the student whether

or not the applet is currently connected to the ∈f Registry, and provides a status area for general notifications.

The Debug Output is mostly an accessory for educators constructing concept graphs, and provides additional feedback and error messages. It can be hidden at runtime, as is the case in the screen shot shown in Plate 1.

4.3 Applet Control

Concept Graph users have two main modes of control, administrator and student, which we describe here. Appendix A.1 includes a reference for controlling the applet, an excerpt from the online directions that appear each time the ∈f Laboratory is opened.

4.3.1 Access Privileges

Many concept graph features are accessible only from administrative mode, which requires special access privileges. Currently, this access mode is based upon the user's account type – recall that this information is supplied to the applet by the Dispatch Scripts. Student and guest accounts are denied administrative privileges, whereas educator, administrator, and teaching assistant accounts are allowed access.

4.3.2 Administrator Features

Administrative features are restricted to those necessary for creating and fundamentally modifying concept graphs. Currently this includes adding and deleting topics and modules, as well as relinking topics: topic linking controls the allowed linking for modules, which must abide by the interface each topic supports, and in no way limits the actual functionality of the module Workbenches that visualize algorithm functioning.

Component properties can also only be modified by an administrator, and include the color and name of each component, as well as the name of the dynamic page to load for each

module's Workbench. Topics can be moved and resized with the mouse in an intuitive fashion that mimics how most popular illustration tools function.

Administrative mode can be toggled on and off. In the latter mode, topic links are less prominently displayed so that it is easier for the student to focus on module interactions.

4.3.3 Student Features

Student mode allows moving and resizing components, so that students can arrange their concept graph as they see fit. This includes sliding component connectors along the component's border.

The principal student mode feature is to *activate* a module: activating a module triggers the Concept Graph applet to load that module's Workbench, as described in Subsection 3.4.1.

Regardless of mode, any student can navigate and zoom the view, enable or disable a visible gridding, and summon the applet directions.

4.4 Persistence

Maintaining modifications to the concept graph and algorithm implementations across client sessions is obviously essential to EF: we envision students interacting with and developing their graphs throughout the class. Moreover, this state cannot be local to the client, rather it must be centralized on the server so that it will be accessible from wherever one chooses to login; this vision is faithful to the client/server architecture of the World Wide Web. Our system accomplishes this by coupling a set of persistence classes with the networking functionality of the Registry.

Two classes, PersistentOutputStream and PersistentInputStream, are the foundation of Educational Fusion persistence. The former defines a protocol for writing object state and class information, and as a subclass of the standard Java FilterOutputStream it can attach to any Java OutputStream to do so. The latter is a subclass of FilterInputStream, giving it the

capacity to attach to any Java InputStream, and defines a protocol for reading the object state and class information written by PersistentOutputStream and instantiating new instances of those objects.

Classes which wish to persist implement the Persistent interface, consisting of `write()` and `read()` methods. These methods are responsible for writing and reading class-specific state information, as we elucidate below. Note that although we discuss these classes in the context of the Concept Graph, their utility is unlimited: any Java developer can make use of them simply by implementing the Persistent interface.

4.4.1 Persistent Output

A PersistentOutputStream (POS), once attached to a stream, is invoked via the `writePersistent()` method, taking the Persistent object as an argument. The first time an object is written to the POS, we must record whatever will be necessary to reconstruct an identical copy later. However, future references to that object should be recorded such that upon reconstruction we only create a pointer to the original object – it would be erroneous to create another copy. To illustrate this, imagine what would happen if a doubly-linked list were recorded with every backward link pointing not to the previous element, but rather to a new copy of that element.

The POS solves this problem[10] with an internal hash table that maps objects to reference numbers: when an object is first written to the POS, it is assigned a unique integral identifier, or reference number, and a mapping is added to the hash table from that object to its identifier. When that object is next encountered it will be present in the hash table, and hence the POS will know it need only record the object reference number.

---

[10] The problem at hand is a classic problem of object-oriented design: how to preserve the relationships between a set of objects when they are translated to flat data structures such as streams, files, or relational databases. Our solution is certainly not unique and derives from the work of Cornell and Horstmann [CH96].

Returning to the `writePersistent()` method, we first lookup the argument in the internal hash table. If the object is present, we know it has already been written, and so simply write to the output stream the reference number to which the object is mapped. Otherwise, the hash table is augmented with a mapping from that object to a new reference number, the object's class name and reference number are written to the output stream, and, finally, the object's `write()` method is invoked with the POS as the sole argument.

The Persistent object's `write()` method is expected to write all object state to the POS. Java primitives such as integers and booleans are written directly to the POS, whereas member objects are written by recursively invoking the POS' `writePersistent()` method. Note the inherent restriction that all object state be a primitive or an object implementing the Persistent interface; special methods are provided, however, for commonly-used Java classes such as Arrays and HashTables.

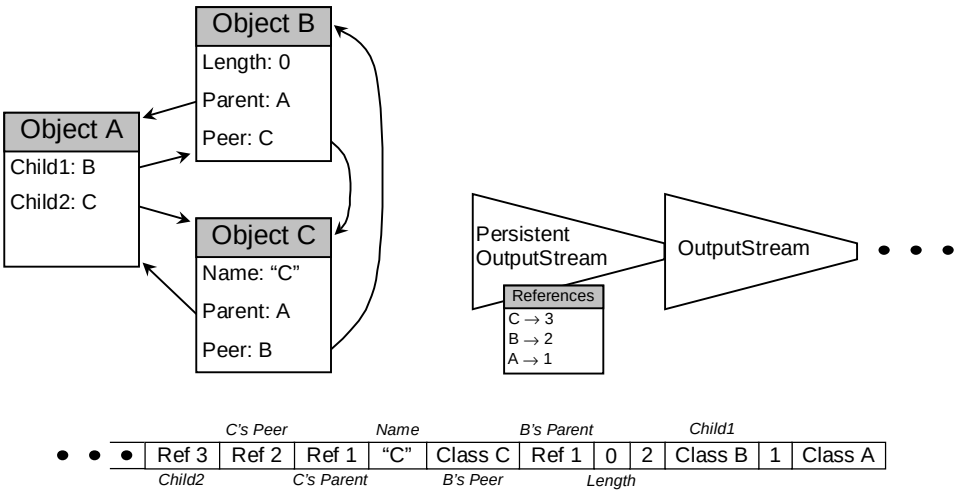

Figure 4-2: A PersistentOutputStream Example

Figure 4-2, above, demonstrates how a PersistentOutputStream saves three mutually-referenced objects A, B, and C; one could easily construct an equally complex concept graph. Notice the reference numbers assigned to each object by the hash table associated with the POS. The sequence of boxes at the bottom of the figure depicts the output of the

OutputStream, with the last written item at the left. Only the first time any of the three objects is encountered is new class information written to the stream – future instances are referenced with "Ref" items.

4.4.2 Persistent Input

A PersistentInputStream (PIS) performs the inverse of a POS: after attaching to an InputStream, its `readPersistent()` method reads object state from the input stream and returns an instantiation of that object. Like the POS, a PIS maintains a hash table of objects that have already been read so that object references can be reconstructed, only now reference numbers are mapped to objects.

The first action of `readPersistent()` is to determine if the object is a new one or a reference: it does so be reading an item from the input stream and ascertaining if it is a "Ref" or "Class" item (see Figure 4-2). In the case of a reference, the reference number is looked up in the hash table and the object to which it is mapped returned.

In the case of a new object, a new instance of the class is created, and the new object's `read()` method is invoked with the PIS as an argument. `read()` is the analogue of `write()`, reading the object's state from the input stream in the same order in which it was written. In some cases, read may also perform initialization of state that was not persisted for class-specific reasons. Finally, the new object is returned.

4.4.3 Saving a Concept Graph

When a student saves her concept graph, the following steps are taken (see Figure 4-3):

1. A PersistentOutputStream is opened onto a new ByteArrayOutputStream. The latter stream, part of the Java API, directs all output to a byte array.
2. The Concept Graph applet's view, an instance of ConceptGraphView, is written to the PersistentOutputStream, through the process detailed above. The entire view is persisted so that the current zoom level, whether or not the gridding is turned on, and

other visual features are saved as well as the concept graph itself (part of the view's state).

3. The RegistryClient is invoked to package the byte array in which this information has been written into a message that is delivered to the Registry.

4. Upon receipt of the message, the Registry writes the byte array to a file in the client's file area.

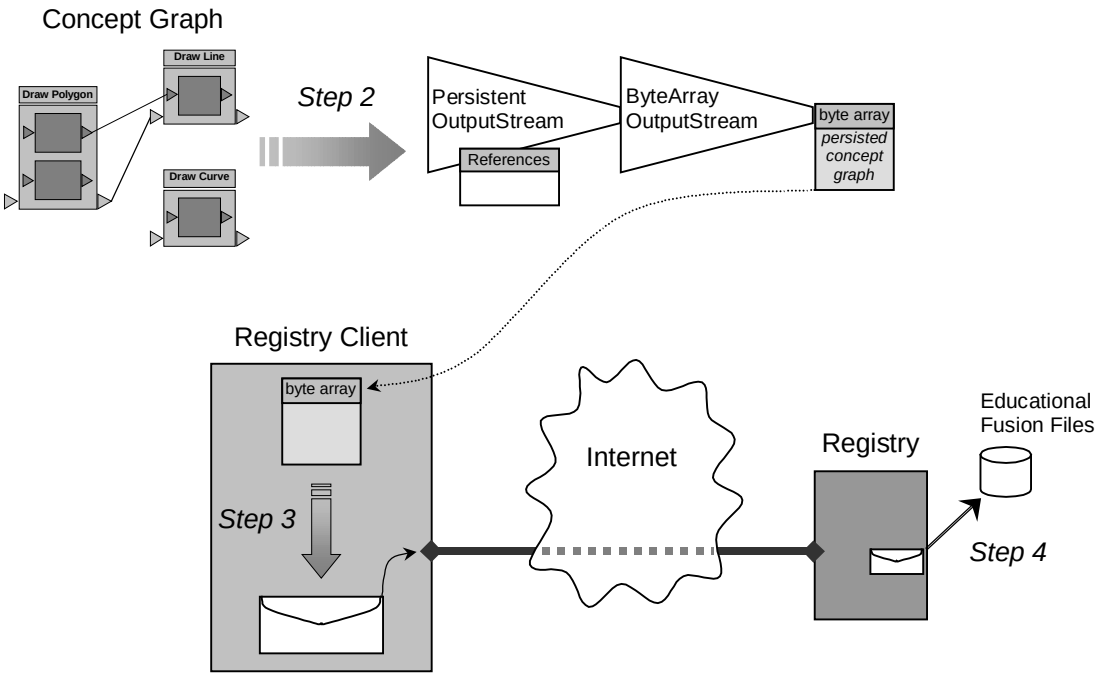

Figure 4-3: Persisting a Concept Graph

Loading a concept graph is basically the reverse of these steps: the RegistryClient requests a concept graph from the Registry, which loads the graph from disk and packages it into a message that is returned to the RegistryClient. The Concept Graph applet then opens a ByteArrayInputStream on the unpackaged byte array, and attaches a PersistentInputStream to it. A new ConceptGraphView is created from the PIS, which replaces the applet's current view.

When a student first opens a project, the Registry copies the default concept graph associated with that project into the student's account. This ensures that everyone starts from the same point.

4.5 Implementation Details

4.5.1 Application vs. Applet

The Concept Graph applet can also be executed outside of a Web browser as a standalone application. As an application, the Concept Graph waives one significant function: module Workbenches are unavailable, as they currently operate only as applets. The ∈F team is already at work to endow Workbenches with the same capacity to operate as applications, so as to remedy this in the immediate future.

4.5.2 Dialog Box Behavior

Dialog boxes opened from applets are unfortunately poorly supported on most Web browsers. Particularly on UNIX platforms, dialog boxes will often appear empty (upon being resized they display properly), far too small or large, or beneath the browser. To ensure that dialogs are recognized by clients before they continue interacting with the applet, when a dialog opens it disables any user interface events from affecting the Concept Graph. Once the dialog has been closed, the user interface is re-enabled.

4.5.3 Third-Party Packages

Other than the standard Java API, Educational Fusion's classes make use of the following third-party packages:

- ***Java Generic Library.*** JGL, a free product of ObjectSpace, Inc., provides a powerful set of Collection classes, modeled after those in the Smalltalk class library, that greatly facilitate and enhance the performance of working with sets, arrays, and hashes [Obj97].

- ***ACME.*** ACME Laboratories' Java package comprises many useful utility classes, including some that the author is incredulous are not in the standard Java API (most notably, formatting numeric string output) [Pos97].

Chapter 5

Messaging

Throughout this document we have alluded to the collaborative power of the ∈f system, and hopefully many questions and curiosities have been triggered as to the nature of the underlying networking technology. ∈f relies upon three closely interacting class hierarchies – message streams, message servers, and message clients – that together form the Messaging API. The message streams provide a means of sending encapsulated, directed messages that can be forwarded by a server that need only inspect the message header: the server is independent of message content. The message servers and clients act together to send and receive messages over message streams, internalizing communication errors to simplify applications built around them. As we shall see, all three hierarchies are designed to be highly extensible, anticipating more powerful and specialized subclasses.

The Messaging API stems from the work of Hughes, et. al., [HHSW97], who detail a multi-layered set of messaging streams and provide the first substantial review of Java's networking and stream classes. The ∈f streams compact Hughes' multi-layered streams (delivering enhanced performance), implement client identification, and are tied to much more complex clients and servers. Of course, the most important differentiation is perhaps is the role the Messaging API plays in the ∈f system: Hughes limits the discussion to networked chat and whiteboarding applications, while we envision much more.

5.1 Message Streams

It is natural that we first describe the transactional unit of ∈f, the message. *Message* has many definitions in modern computer science, ours is mostly a functional description: a message encapsulates data so that only the intended recipient need understand the contents, and prepends a simple header that allows intermediate parties to correctly route the contents to its recipient. A message can be thought of as an envelope, as was depicted in Figure 4-3, with the contents being the enclosed letter and the header being the address printed on the front. As we shall see, the postal service has work to do if it hopes to compete with the facilities enabled by ∈f messages.

5.1.1 Message Headers

The MessageHeader class identifies both the sender and recipient(s) of a message. Information about the sender is encoded in `source` and `component` instance variables: `source` contains the unique integral identifier of the sender, and `component` the byte-sized identifier of the component within the source. Why two identification numbers? Each client applet that registers with the ∈f system receives a *client ID* after logging in, and then individual components of that client are assigned a *component ID*; Section 5.2, below, covers this registration process. While client identifiers allow messages to be targeted at specific individuals, component IDs support communication between particular components of the applets involved.

For example, the Collaborator applet described in Section 6.1 contains both whiteboard and chatboard components, neither of which "understands" the messages of the other. By assigning them differing component IDs, the Collaborator's MessageClient, also discussed below, automatically delivers incoming messages to the appropriate locale. This solution greatly simplifies client implementation as each component need be concerned only with the protocol it defines, avoiding the ugly alternative of requiring the applet to route incoming

messages amongst its components, or, even worse, forcing each component to understand all other components' message protocols.

The remaining state of a MessageHeader identifies the message target(s). A `forwarding` flag indicates how the message is to be handled by the server: *no forwarding* indicates that the message is not intended to be forwarded to a peer of the client, but rather is directed at the service itself, generally to request some service; *broadcast* indicates that the message is to be forwarded to all other clients; *multicast* indicates that the message is to be forwarded only to a named subset of clients. A `destinations` integer array enumerates the client IDs of all intended recipients in the case of a multicast message.

5.1.2 Message Output

Messages are sent via subclasses of the abstract[11] class MessageOutput. MessageOutput subclasses the standard Java stream DataOutputStream, which is in turn a subclass of FilterOutputStream. As such, MessageOutput inherits the capacity to attach to any standard OutputStream and write any Java primitive or a byte array to that stream.

MessageOutput and its derivatives extend this functionality by defining the following methods:

- ***`send()`.*** When invoked, this abstract method should send a message to the OutputStream to which this stream is attached. The message's header is to be set to no forwarding.

- ***`sendRaw()`.*** Another abstract method, sendRaw is to send a message without prepending a header to the message. As we shall see, this superficially unnecessary function simplifies implementing devices to forward messages when we do not care to

[11] In Java, an *abstract* class is one that cannot be instantiated. Generally such classes simply define the protocol which non-abstract subclasses must implement, providing a common root for classes that might not otherwise be related (allowing them to be used interchangeably by way of polymorphism).

know the route along which a message was passed; when the route needs to be identified by the recipient, each forwarder should add a header identifying itself.

- ***`send(int).`*** This method sends a message to the OutputStream whose header indicates it is to be multicast by the server to the client identified by the integral method argument. The only functioning member of MessageOutput, it simply constructs an integer array consisting only of that single integer and invokes the following method.

- ***`send(int[]).`*** Multicast messages are sent by invoking this method, which sets the message header's `destinations` field to the integer array argument and the `forwarding` field to multicast. Again, this method is not implemented. However, it is not abstract as its implementation is not required (one can imagine streams that are not suited to multicasting, such as a private channel). Unless it is overridden by a subclass, the default behavior of this method is to trigger an exception indicating that multicasting is not supported.

- ***`broadcast().`*** This final method is used to broadcast a message, which involves setting the message header to multicast. It's implementation is much like `sent(int[])`, simply triggering an exception when invoked by a sender.

The astute reader will note that MessageOutput defines no protocol for writing the actual message *contents* to the attached output stream! The solution to this paradox might seem to be that one simply invokes the methods inherited by MessageOutput's superclasses. But this would lead to the undesirable situation in which a message's contents are sent out before its header. It would be akin to dumping an envelope's contents into a mailbox, followed by the envelope itself.

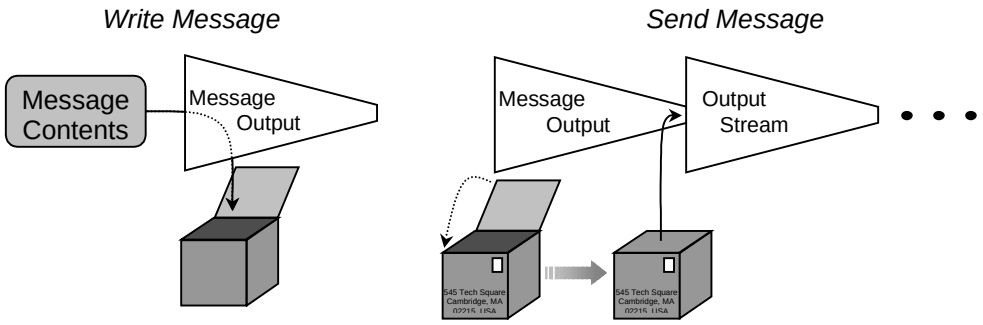

Figure 5-1: Internals of a MessageOutput Stream

To answer the quandary, one must first understand how MessageOutput streams are intended to function. As we have just seen, no message contents should be sent out until the message is complete: a message is an atomic, encapsulated unit of communication. So rather than attaching the MessageOutput stream directly to the OutputStream to which messages are sent, it should be connected to some internal buffer. As such, when a sender invokes the stream's inherited methods, the data will be directed to that buffer and nothing will be sent out, much like filling our envelope with news clippings, photographs, and so forth. This step is depicted in the Write Message portion of Figure 5-1.

Once the message contents are complete, one of the MessageOutput methods listed above can be invoked to package those contents along with a header and send the completed message to the OutputStream. This final step, depicted by the Send Message portion of Figure 5-1, is analogous to sealing, stamping, addressing, and finally delivering the envelope to the post office. After a message has been sent, the internal buffer is cleared and data for the next message can be written to the MessageOutput stream.

5.1.2.1 Message Output Stream

MessageOutputStream (MOS) is the principal subclass of MessageOutput, straightforwardly implementing all of the methods declared by its parent much as one would expect; this includes overriding the exceptional behavior of the multicasting and broadcasting methods.

MOS attaches to a ByteArrayOutputStream, meaning that its internal buffer is in the form of a byte array.

A sent message is prepended with the combined length, in bytes, of its header and contents. Without this information the MessageInputStream, would be unable to discriminate a given message from the next; why the length is in bytes, as well as the rest of the MessageInputStream, is detailed in the following section. A schematic of the stream output of a message sent by a MOS is shown in Figure 5-2. We chose to use this technique rather than make use of a special terminating delimiter for two reasons: for one, that terminator would have to be disallowed from message contents, requiring an encoding scheme to circumvent this limitation; and secondly, out scheme allows message recipients to preallocate adequate storage, which is much more efficient that gradually growing a data structure to accommodate an incoming message.

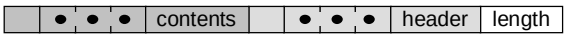

Figure 5-2: Message Format of a MessageOutputStream

Note that the MOS reuses the message header for each sent message. The ∈f design team made a decision that each component within a particular client should use a dedicated message stream for transmitting messages. Hence, when a MessageOutput stream is created, its arguments include the client ID and component ID with which all messages are to be sent. A MessageHeader is created with these identifiers, and its forwarding and destination fields are updated each time a message is sent, according to the whether the message is to be broadcast, multicast, or not forwarded at all. This simplifies sending a message, which is done often, at the expense of making it more complex to construct a message stream, which is done only once, and making it impossible to re-assign a stream to a new sender, for which we have found no use. The analogy we offer is the personalized stamp many use for the return address on envelopes.

MessageOutputStream's one subclass, ForwardingStream, behaves similarly only it presumes that it is attached to another MessageOutput class. When a message is sent via a ForwardingStream, it is forwarded through the MessageOutput stream to which it is attached. To understand how this works, assume that a ForwardingStream `F` has been connected to a MessageOutput stream `M`, which is in turn connected to an OutputStream `O`. Upon receiving a send message call, `F` writes a message to `M` following the protocol of its parent, MOS, described above. However, after this message is delivered, `F` invokes `M`'s `sendRaw()` method. This will cause the message to be sent by `M`, this time to `O`, without a new header being added.

The result is that the message has been forwarded from `F` through `M` to `O`. ForwardingStreams are used in situations where a class does not wish to expose a private MessageOutput stream, but requires another class to write messages to that stream. The solution is for the first class to create a new ForwardingStream attached to the private stream, and expose this new stream to the untrusted class.

5.1.2.2 Queue Output Stream

As its name suggests, a QueueOutputStream (QOS) is used to write messages to a queue. The QOS constructor accepts a BlockingQueue as its argument; BlockingQueue is a derivative of the standard queue data structure whose pop method blocks[12] when the queue is empty. Much like MessageOutputStream, a QOS directs all data written to the stream to an internal byte array. When the QOS' `send` or `sendRaw` methods are invoked, this byte array is pushed onto the BlockingQueue, and a new byte array is initialized. Multicasting and broadcasting are not supported.

---

[12] In Java, when a thread *blocks* it yields control of the processor until the scheduler gives that thread another chance to execute. The thread will continue to block each time it begins executing until some condition is (no longer) met; in this case, the condition is that the queue is empty.

A BlockingQueue is required because QueueOutputStreams and QueueInputStreams are anticipated to function as channel buffers. Imagine a channel carrying bursty but high bandwidth traffic that must be carried over a low bandwidth but always free channel that can, on average, keep up with the traffic stemming from the first channel[13]. A buffer must be interjected to avoid message loss, as visualized in Figure 5-3. During periods of heavy traffic, when the low-bandwidth channel cannot match the output of the high-bandwidth channel, the buffer fills, while during low activity the buffer empties. This is accomplished by attaching the bursty channel to a QOS, which directs all traffic to a BlockingQueue. To this same queue is assigned a QueueInputStream that blocks when the queue is empty, and regularly pop elements from it and forward them to the low bandwidth channel when the queue is not empty.

[13] This scenario anticipates exactly what is encountered with the Educational Fusion client applets, which may generate voluminous traffic when a student is at work, but are generally idle and waiting for input. These applets are attached to stable, lower-bandwidth channels – most often TCP/IP sockets over Ethernet.

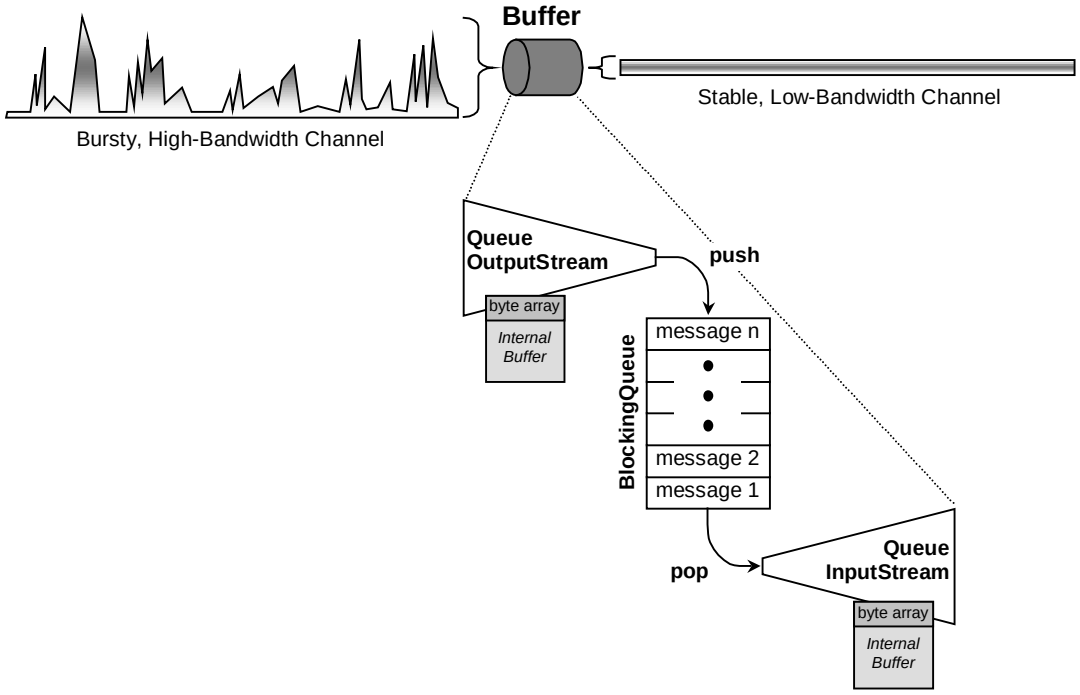

Figure 5-3: Channel Buffering with Queue Streams

QOS does not attach message headers or content length: the former is not relevant to the purpose fulfilled by the queuing streams, and the latter is not needed as each message's contents are readily extracted as one queue element. Of course, if a MessageOutputStream is attached to a QOS, the QOS will faithfully communicate the message header and length provided by the MOS. To understand this behavior, consider that when two MessageOutput streams are chained, the second stream sees nothing written to its message until the first stream sends its message. So the entire message contents, header, and length of the first stream are dumped into the message contents of the second stream; when its send message, in turn, is invoked, it will package and forward its message accordingly, potentially prepending a second header and length field.

5.1.2.3 Passthrough Stream

PassthroughStreams are the simplest MessageOutput streams: they simply pass data written to them directly on to the OutputStream to which they are connected. Their role is also trivial: when a program would like to write directly to an OutputStream, but calls for a MessageOutput stream, attach a PassthroughStream to the OutputStream and use it in lieu thereof. In effect, PassthroughStreams discard all of the overhead that distinguishes messaging streams from other streams; `send()` and `sendRaw()` are overridden to do nothing at all, as, in a sense, there is never a pending message to send.

5.1.2.4 Message Delivery Stream

Also an extremely simple class, MessageDeliveryStream delivers its messages to a single MessageRecipient. A MessageRecipient is a class implementing an interface of the same name whose sole method is `receive(DataInputStream)`. As the signature suggests, delivery is accomplished by creating a DataInputStream on the message and invoking the `receive()` method.

A MessageDeliveryStream directs all data written to the stream to an internal byte array. When a message is sent, a ByteArrayInputStream is attached to this array, to which a DataInputStream is attached. This last stream is the argument to `receive()`. Note that when a MessageDeliveryStream sends a message, the caller is forced to wait until the MessageRecipient has handled the message. This limits the applicability of this means of communication, but also means that there is extremely little overhead when compared to asynchronous methods (e.g., using a BlockingQueue).

As one would suspect, this stream does not add message headers, expecting previous streams to have been responsible for doing so. And of course, multicasting and broadcasting do not make sense, so the methods for doing so will generate exceptions as declared in MessageOutput.

5.1.3 Message Input

Each message written by a MessageOutput stream will eventually be read by a MessageInput stream. MessageInput is an abstract class derived from DataInputStream, inheriting the ability to connect to any Java InputStream and read primitives and byte arrays from that stream. MessageInput streams function similarly to MessageOutput streams, internally buffering the contents of individual messages: each time a MessageInput stream receives a message from the InputStream, the message's contents are transferred into an internal buffer. The stream connects itself to this buffer, so that invoking the MessageInput's inherited methods read data from the buffer. Just as MessageOutput does not allow direct access to the OutputStream, MessageInput disallows reading directly from the InputStream.

MessageInput defines the following abstract methods that must be implemented by subclasses:

- ***`receive()`.*** Receive the next message from the InputStream, transferring the message's contents into an internal buffer. The message's header should be assigned to a publicly accessible instance variable, so that the object making use of the MessageInput stream can determine the source and destination of each message received.

- ***`receiveRaw()`.*** Receive the next message from the InputStream, transferring the message in its entirety into an internal buffer. The message is not parsed for a header.

5.1.3.1 Message Input Stream

MessageInputStream (MIS), the primary subclass of MessageInput, is constructed with an InputStream from which messages are to be received. When MIS is directed to receive a message, it reads the message contents and header into a byte array and attaches a ByteArrayInputStream to that array, to which it then attaches itself. Consequently, reading from the MIS stream returns data from the last message received.

In the case of the `receive()` method, MIS also reads the MessageHeader from the internal byte array: reading from the MIS thus extracts data from the start of the message contents, whereas `receiveRaw()` forces data retrieval to begin with the message header. The extracted MessageHeader is assigned to a public instance variable of the MessageInputStream object.

5.1.3.2 Queue Input Stream

QueueInputStream (QIS) is the analog of QueueOutputStream, and its functionality should already be clear: it retrieves messages from a BlockingQueue, blocking when a message is requested but the queue is empty. Like MessageInputStream, when a message is received it is placed in an internal byte array, to which the QIS connects itself by means of a ByteArrayInputStream.

5.1.4 Message Copiers

Message copiers function as the name suggests, copying messages from a MessageInput stream to a MessageOutput stream as rapidly as possible. The fundamental message copier, MessageCopier, extends the Java Thread class so that it runs as an independent thread: a MessageCopier is instantiated by connecting it to the relevant input and output streams, and then its `start()` method is invoked to begin the message copying process. A running thread can be killed, and the message copying terminated, by passing the MessageCopier a `stop()` message.

Once a MessageCopier is started, it sits in a simple copying loop where it:

1. Directs its MessageInput stream to receive a message.
2. Reads the entire message into an internal byte array.
3. Writes that byte array to its MessageOutput stream.
4. Directs its MessageOutput stream to send the message.

An additional argument to the MessageCopier constructor dictates whether message headers should be preserved: if this flag is true, messages' original headers should be preserved and no new header added, and so the input stream's `receiveRaw()` and output stream's `sendRaw()` methods are invoked in steps 1 and 4; otherwise, we invoke the `receive()` and `send()` methods, which means that messages' original headers are replaced by that of the copier's MessageOutput stream. The latter method is used only when message headers are not of interest.

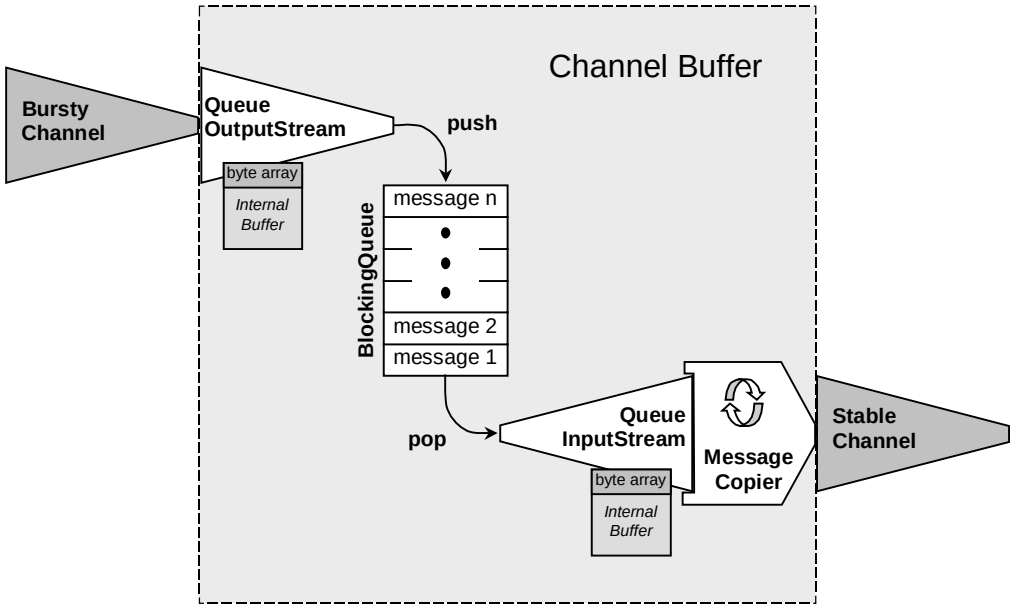

Figure 5-4: A Complete Channel Buffer

A prototypical application of a message copier is to retrieve messages from a queue as soon as they are available. In fact, implementing the aforementioned channel buffer requires a MessageCopier to deliver messages to the output channel as they become available in the queue. Figure 5-4 gives a more complete picture of the buffer described in Part 5.1.2.2.

There are two specializations of the standard MessageCopier, HaltingMessageCopier and NotifyingMessageCopier. HaltingMessageCopier, when constructed, accepts a *sibling* thread:

this thread's execution depends upon the copier. If the message copier should fail, before terminating it will notify its sibling by invoking the sibling's `stop()` method. NotifyingMessageCopier is a subclass of HaltingMessageCopier that also accepts a MessageRecipient object (see Part 5.1.2.4) upon construction: each message copied is also delivered to the MessageRecipient. This is useful when a third party wishes to observe the message stream passing through a copier.

5.1.5 Stream Errors

Many methods of these message streams will throw IOExceptions when an error is encountered, whether it be the result of a low-level stream error, which is especially plausible when network streams are being used, or a protocol failure, for example when a recipient tries to read from an empty stream. As such, programs that use these streams must be careful to catch any thrown exceptions.

∈f isolates developers from stream or network failure, however, by providing developers with the Messaging Servers and Clients described next. A principal benefit of these classes is their encapsulation of all error conditions: when an error is encountered, applications are notified via a predefined method responsible for error-handling and cleanup.

5.2 Message Servers

Message Servers are server-side applications, or servlets, that process requests from clients and route messages between them. Currently, Educational Fusion relies upon two servlets, a MessageServer and a RegistryServer, which are active at all times. Our discussion now focuses on these classes, as well as a servlet to accommodate NetEvents, our applet-sharing technology.

5.2.1 Message Server

The MessageServer class is responsible for accepting new client connections, but once this connection is made it starts a new MessageServerHandler to handle messages to and from that client.

5.2.1.1 Startup

MessageServer is a subclass of Thread that is invoked from the command line. Java Virtual Machines startup applications from the command line by invoking the named class' `main()` method with arguments determined by the command line parameters. MessageServer's `main()` method initializes and starts the server thread, then exits with the server running in the background.

MessageServer requires one command-line argument, the port number to listen to for new connections. The port number corresponds to a TCP/IP port number, which can range from 1 to 65,535, although on many platforms ports 1 through 1000 are reserved for system services. A second, optional argument gives a filename to open for logging. Status messages and errors encountered while dealing with any client will be output to this logfile; if no filename is provided, this information is simply directed to standard error.

5.2.1.2 Server Handlers

A running MessageServer executes within a loop whose body accepts a new client connection, opens a bi-directional TCP/IP socket to that client, and starts a new MessageHandlerServer to handle that client. A ServerSocket, part of the standard Java API, is instantiated by the MessageServer constructor and is responsible for creating new sockets: invoking a ServerSocket object's `accept()` method blocks until a client requests a new connection, at which point a socket is created and returned.

The MessageServerHandler constructor accepts input and output streams attached to the newly created socket, as well as the *routing table* for the MessageServer. The routing table is a

hash table that maps integral client IDs, defined in Subsection 5.1.1, to MessageOutput streams used to send messages to the corresponding clients. Every MessageServerHandler shares the same routing table, so modifications to it are carefully synchronized.

Real work is done by the MessageServerHandler only after it is started – like the MessageServer itself, the handler is a subclass of Thread. The steps taken by a MessageServerHandler are detailed below:

1. The integral ID of the client is read from the socket input stream.
2. If this key is found in the routing table, this client is already registered with the MessageServer and the connection is refused by writing a boolean "false" to the socket output stream. The handler is then gracefully terminated.
3. Otherwise, the connection is accepted by writing a boolean "true" to the socket output stream.
4. A new BlockingQueue is created to which all messages for this client are sent. A QueueOutputStream is attached to the queue, and a new mapping is added to the routing table from the client's ID to this QOS. This queue stream is used by other clients' MessageServerHandlers to send this client messages. As the handlers are running as synchronous threads, we must ensure that they properly synchronize their access to this QueueOutputStream.
5. A MessageOutputStream is attached to the socket output stream for our new client. We connect a QueueInputStream to the BlockingQueue created in step 3, and connect this QIS to the MOS with a HaltingMessageCopier. We then start this message copier, which amounts to forwarding messages from the queue to client via its TCP/IP socket. The sibling (Subsection 5.1.4) of the HaltingMessageCopier is the MessageServerHandler itself: any errors copying messages to the client will lead to the handler itself shutting down.
6. Finally, we enter a routing loop where messages are received from a MessageInputStream attached to this client's socket and routed to other clients.

Multicast messages are sent to each client whose ID is in the MessageHeader's list of destinations and which is found in the routing table, while broadcast messages are simply sent to every client in the routing table. As noted in step 3, we take care to synchronize on the MessageOutput stream (more precisely a QueueOutputStream) for each client to which a message is sent. Regardless of the message's forwarding, we also invoke the MessageServerHandler's `receive()` method, providing the message's contents and header as arguments, anticipating that the message may be a request of or response to the server.

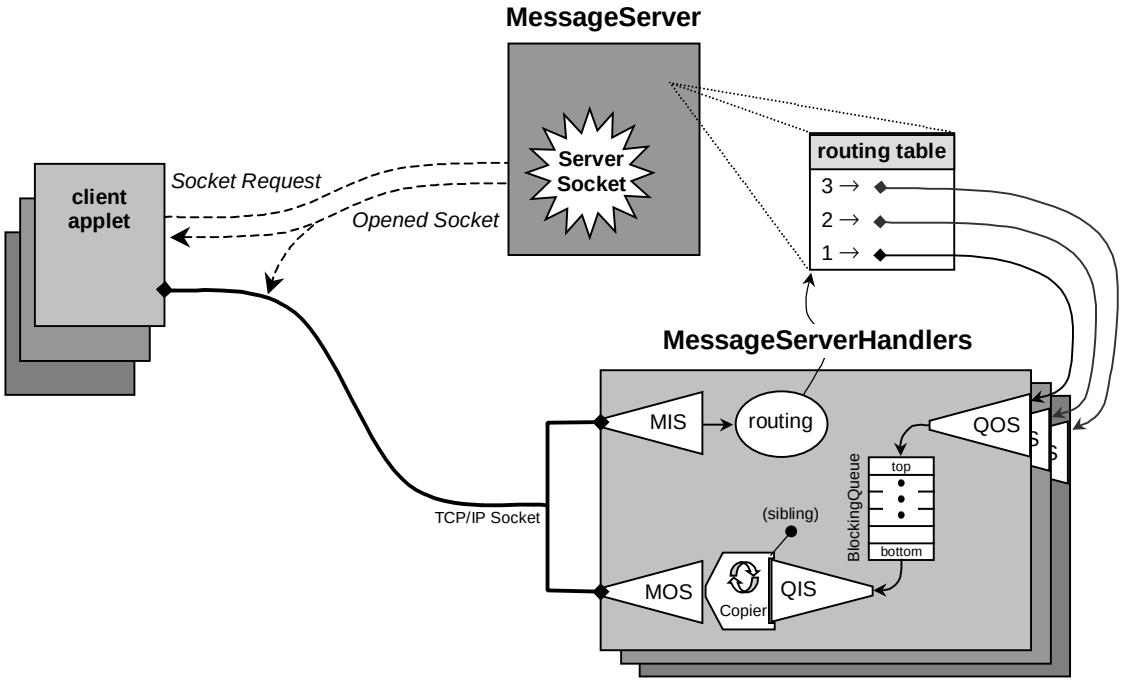

Figure 5-5: Message Server and Handlers

The results of this rather lengthy sequence are summarized by the above figure, which depicts the state of a message server and its handlers after three clients have connected. Note that the TCP/IP sockets are shown only for Client 1, and we assume that all three clients registered successfully. The oval labeled "routing" embodies control for the routing loop described in step 6.

Careful observation of the diagram shows that a MessageServerHandler acts as a channel buffer for outgoing messages. The volume of outgoing traffic may be quite high, given that an unlimited number of clients may be asynchronously sending messages to a particular client, and at times may exceed the maximum capacity of the outgoing TCP/IP channel – this is exactly the circumstances under which channel buffers were introduced. Exacerbating the situation is the fact that a TCP/IP socket does not guarantee availability, and hence throughput may be quite unstable. Buffering outgoing messages ensures that we do not lose messages due to network problems or exceptionally high traffic.

As discussed in step 6, the handler is given the opportunity to examine each incoming message via its `receive()` method. However, the message server handler does not offer special services, and hence this method does nothing.

5.2.1.3 Server Termination

A running MessageServerHandler will terminate when its thread is interrupted or when a network failure causes the TCP/IP socket to its client to break. If the thread has been interrupted, it is likely because the MessageServer that invoked it is terminating or the HaltingMessageCopier delivering outgoing messages failed and sent its sibling, the handler, a `stop()` message. As the handler unwinds, it stops the message copier, removes the mapping for its client from the routing table, and finally closes its MessageOutputStream (which, in turn, closes the TCP/IP socket to which it is attached, leading to closure of the MessageInputStream reading from the socket). Although it may be redundant to stop the copier or close the input/output streams, we must make sure all resources are freed. Any errors triggered during cleanup are captured and discarded.

When a MessageServer terminates, either due to an internal error or when it is shut down by the operating system, it simply closes the logfile (if one was specified). All handlers invoked

by the server receive `stop()` messages, causing them, too, to terminate and free all allocated resources.

5.2.2 Registry Server

We now focus our attention on the RegistryServer, which has been mentioned throughout this document. RegistryServer subclasses MessageServer, specializing the registration process and providing ∈f services by making use of RegistryServerHandlers, which extend MessageServerHandlers. A distinction that the reader should keep in mind is that while the Registry is intimately tied to ∈f, Message Servers are completely generic: RegistryServer expects its clients to be Concept Graph clients and provides services tailored expressly for them, whereas we have seen that MessageServer makes no such assumptions of its clients. The ease with which generic classes such as MessageServer can be extended for specific functions is a central design goal of the ∈f Messaging API.

A RegistryServer requires an additional command-line argument: the path to the ∈f file area discussed in Subsection 3.1.1. This argument allows the server to access student and project files. The RegistryServer functions identically to the MessageServer, opening new socket connections and spawning a RegistryServerHandler for each.

5.2.2.1 Registration

Registration with a RegistryServerHandler differs from logging into a MessageServerHandler primarily in that while the latter assumes the client already has an assigned ID, the former assigns a new ID to the client. The client is expected to send a single boolean value to the server, indicating whether the client is a *listener* or a *participant*. Listeners are clients that the ∈f Registry does not track through the system: their location is not known, nor is their presence reported to others. Listeners are, in effect, unobservable by other clients, although listeners do receive information about participants. Participants are Registry clients that are actively working on a concept graph, meaning that their location – in terms of project, topic,

and module within that project's concept graph – is known. When a participant changes modules or enters or leaves ∈F, listeners and participants alike are notified.

The handler replies to this boolean flag by writing a new, unique ID to the socket, which becomes the client's identification number. This client ID is synonymous to the session ID employed by many contemporaneous authentication systems: both serve to uniquely identify a client, and the human represented thereby, but only for the duration of the current session. The recipient includes this client ID in all outgoing message headers to identify itself as the sender. An entry is added to the RegistryServer's routing table that maps the ID to the handler's QueueOutputStream: a RegistryServerHandler does not modify the internal streams and buffers it inherits from MessageServerHandler. Two new data structures are also modified in the case of a participant – like the routing table, these structures are common to the RegistryServer and all spawned handlers.

The first is the *registry*, a hash table that maps client IDs to ConceptGraphClient objects. The ConceptGraphClient (CGClient) class is a simple container for all public state about the student each client represents, including username, nickname, first and last name, account type, location, and client ID. The registry allows fast identification and location of students based upon ID. When a participant logs in a new CGClient is created and an entry is added to the registry.

The second new data structure of the Registry is the *usernames* hash table, which maps client usernames to IDs. This structure provides a fast way to determine the ID currently assigned to a client with a given username (and of course, once the ID is determined the registry can be used to locate and provide more information about that client). Again, a new mapping is added to the usernames hash when a participant is logged in. Recall that listeners are not visible to others, hence there is no need to add them to these new tables, which serve only to expedite client lookup.

Once this registration process is complete, the client is provided with a list of all participants: each CGClient in the usernames table writes itself to the new client's QOS, where it will eventually be forwarded to the client. Like all RegistryServer communications, the client's RegistryClient handles parsing this list and notifies the client when it is received; this will be covered in more detail in the following section on Message Clients. Finally, if the client is a participant, a message is broadcast to all other clients of the participant's entrance into ∈f.

5.2.2.2 Services

The handler next enters the routing loop inherited from MessageServerHandler, receiving messages from its client via the MIS attached to the TCP/IP socket and forwarding them as necessary. The registry handler, however, offers several services via its `receive()` method:

- ***Move.*** When a student activates a module or otherwise changes her location within ∈f, her client notifies the Registry via a move message. The RegistryServerHandler updates its CGClient accordingly, and then announces the new location to all others currently logged in.

- ***Leave.*** When a students logs out of ∈f this message is sent to the handler to remove information about the client from the routing table, registry, and usernames table. However, the handler remains open and the socket connection alive.

- ***List Users.*** A client makes this request when it wishes to receive a listing of all participants in the Registry. The handler's response is identical to when a client first logs in: every CGClient in the usernames table is written to the client.

- ***Locate User.*** This service request includes a string containing the username of a student that the client wishes to locate. If the requested username is found in the usernames table, the corresponding client ID is used to lookup the student in the registry. The server responds with the CGClient for that student, or indicates that no one with the supplied username could be found.

- ***Disconnect.*** This handler service is similar to Leave, only the TCP/IP socket to the client is closed. The handler then closes down gracefully.

- ***Save File.*** Save File requests (see Subsection 4.4.3) include a byte array and a string: the string represents the filename of a file to create containing the byte array. The new file is created in the ∈f file area of the student represented by this client. Server response is limited to a boolean that is true if the file was successfully created, false otherwise.

- ***Load File.*** A Load File request includes the filename of the file to load from the student's file area. Upon success, the file contents are packaged in a byte array and returned to the client, otherwise the client is notified that the operation could not be carried out.

Having already reviewed the Concept Graph applet, it should be obvious how most of these Registry services correspond to applet functioning. These relatively simple services are all that is needed to support even a sophisticated client such as the Concept Graph. Moreover, we hope to impress upon the reader the ease with which server-side services can be built atop the fundamental MessageServer class. It was our intention to provide a set of core classes that could easily be extended to support any imaginable client applet.

5.2.3 NetEvent Server

Supporting NetEvents, the ∈f applet-sharing technology, is the NetEventServer. At heart, NetEvents involves the distribution of Java *events* local to an applet to its peers across the network. In the Java programming model, an event is any user interface action that the runtime environment delivers to an applet or application. Examples include a keypress, a mouse click or movement, and notification that the containing window has regained focus[14].

[14] In the graphical windowed environments common to modern operating system, the window encapsulating the application with which the user is interacting is said to have *focus*. Only one application can have the focus at a given time. An application *gains* focus when, for example, the mouse is clicked within the bounds of its containing window; in turn, the previously active application *loses* focus.

As such, NetEventServer's primarily responsibility is the distribution of Java UI events, encapsulated within ∈F messages, in an intelligent manner. What do we mean by "intelligent manner"? The answer is that unlike the servers we have described above, NetEventServer must be concerned with the order in which messages are received and routed. In particular, without proper *serialization* of NetEvents, the server is ineffectual.

To best understand serialization, consider two students logged into ∈F using NetEvents technology to share a Concept Graph applet. Imagine the behavior that will result if the students simultaneously, or nearly so, relocate two modules within the displayed concept graph by dragging it with the mouse. Student A broadcasts the selection of the module, via a mouse button press, and the movement thereof, via a stream of mouse movements with the button still depressed. Student B does the same with a second module. Now consider what happens if the NetEvents from B reach A while the latter is still dragging: A's Concept Graph will suddenly unselect the module chosen by A and select the module B is dragging, and a confusing series of relocations of this module will follow as the Concept Graph interleaves the NetEvent streams derived from A (locally) and B (over the network).

An intelligent NetEventServer must thus serialize NetEvent streams so as to make every student's interaction with a shared applet appear atomic. There are several techniques under investigation by the ∈F research staff for doing so:

- ***Tokens.*** The most practical approach is to associate a *token* with each collaborative session: only the owner of a token is allowed to control the shared applet, hence a client may broadcast NetEvents only when in possession of such a token. In effect, tokens provide a means by which one student can lock out others. Tokens could be passed on request: the applet being shared has an associated dialog box by which students can request and relinquish ownership of the token. This approach also makes it straightforward to enable demonstrations, wherein, for example, a professor might conduct a collaborative session in which no other participant is allowed to obtain the control token.

Reconsider the above example of two students simultaneously dragging two modules. Before attempting to drag a module, each student will request the control token from the NetEventServer, which will pass the control token to the student whose request is received first. The other student will be notified that he cannot attain control until the token is relinquished by the other student. In this way, one student will be assured of uninterrupted control of the shared Concept Graph applet.

- ***Activity-Based Synchronization.*** A second, much more sophisticated approach requires the NetEventServer to impose an artificial synchronization of NetEvent streams. The idea behind this synchronization is for the server to programmatically determine the atomicity of client actions based upon the frequency with which incoming NetEvents are received. A high frequency of incoming NetEvents render a client "active". In this view, a continuous series of events are considered atomic. For example, in the case of a student dragging a module component, as we discussed earlier, the rapid sequence of mouse events that result in the module drag would be considered atomic. Hence, an atomic event is terminated when the incoming message stream becomes idle.

  A NetEventServer could actually make use of tokens to implement this scheme, only token arbitration would be handled automatically. Assume no client is in possession of the token, when the server's handler synchronously receive NetEvents from several clients. The server randomly chooses one client (a more intelligent schedule could be implemented if starvation becomes an issue) and passes it the control token. That client remains in possession of the token as long as its NetEvents stream is active, during which time events from the client are broadcast while events from others are buffered. Once the token is relinquished, the server picks the next active client and the process continues.

  Consider the two students simultaneously trying to dragging modules. A tokenized NetEventServer will give the control token to whichever student's module selection message it receives first: the other student's messages are queued for later retrieval. Once the student has finished dragging, as indicated by a pause in the incoming NetEvents stream from that student, the token will become available.

Clearly, the method we describe here is challenging, and potentially problematic. For example, we have chosen to define atomicity in terms of idleness, but humans are inherently slow when placed in a computational framework. Just because a student pauses while dragging a component does not mean her control over the applet should suddenly be relinquished to another. Yet the idea of computationally determining "active" clients is an extremely interesting and potentially valuable area of research.

- ***Intelligent Clients.*** The third principal synchronization technique we have considered is to rely upon intelligent NetEvents clients, which should broadcast only atomic events. This frees NetEventServer from having to worry about synchronicity, but obviously we have only displaced the necessary intelligence into the client application.

  From this perspective, a NetEvent is a semantic, application-level entity, rather than an operating system level event such as a mouse click or keypress. In the example examined earlier, each student attempting to drag a module would broadcast the complete drag event within a single NetEvent message. Whether the message contains simply the starting and ending points, or the set of points along the path which the component was dragged, is left to the client developer. Obvious advantages of this approach are reduced network traffic and the ease with which it is implemented.

  A potential shortcoming of this technique is that applet sharing might not feel as interactive: events that occur over a period of time on a client will be received as an instantaneous happening by others. One way around this could be to encode timing information within the transferred NetEvent. For instance, the set of points along which a component is dragged might be annotated with the time delay between each point. Then, recipients could replay NetEvents at the same speed at which they were recorded.

  Another solution is to construct semantic events at the recipient rather than the sender. This model foresees clients continuing to broadcast low-level NetEvent messages, while queuing incoming messages into buffers (each peer having been allocated a dedicated buffer). Messages are processed in a manner prescribed by the application: a complex Concept Graph client could simultaneously drag several modules by incrementally reading from several buffers at once, while a simple client might wait until a drag has

been completed to read from a buffer. In this way, the developer balances complexity and responsiveness versus simplicity and delayed response.

Other than implementing one of these serialization methods, we anticipate that NetEventServer will be a straightforward specialization of the MessageServer class. If registration services are to be provided to identify the individuals sharing an applet – which could depend upon the number of students involved – NetEventServer should subclass the RegistryServer.

5.3 Message Clients

Some readers may have noticed that our discussion of the message servers neglected the specific protocol of the various client-server communications, such as the order of service request arguments. The reason for this is that ∈F developers need not be concerned with such details, just as they should not be concerned with network errors. MessageClient and its subclasses make this possible, providing powerful utilities for handling server registration, requests, and responses.

5.3.1 Message Client

MessageClient (MC) is a subclass of Thread that handles all communications between the various components of an application and a MessageServer. MessageClient's constructor accepts the hostname of the machine running the server, a port number to which the server is listening for new connection requests, and the ID number with which the client wishes to register with the server. The constructor first attempts to open a TCP/IP socket to the MessageServer, and if successful logs on by following the registration procedure outlined in the previous section. Finally, the constructor initializes its *component routing table* as well as a new BlockingQueue, and QueueOutputStream attached thereto, for buffering outgoing messages.

5.3.1.1 Components

Recall from the discussion of MessageHeader that EF messages are identified with both the client ID as well as the component ID of the sender: messages from a particular component are intended to be directed to the same component of the recipient. The component routing table makes this possible by providing a mapping from each component ID to the corresponding component.

After a MessageClient is constructed, each component of the client application registers by invoking the MC's `register()` method with a unique component ID. Component IDs are byte-sized and defined by the applet's designer, and thus are determined at compile-time. Each component that wishes to communicate via a MessageClient must be a ClientComponent, implementing the ClientComponent interface:

- ***setMessageOutput(MessageOutput).*** This method notifies the component of the MessageOutput stream to which it should send outgoing messages.

- ***receive(DataInputStream).*** MessageClient invokes this method when the component has received a message – reading from the passed DataInputStream extracts the message's contents.

- ***messageDisconnect().*** When, for whatever reason, the MessageClient disconnects from the MessageServer, it notifies each component via this message. The component is expected to perform any relevant cleanup.

- ***byte getID().*** `getID()` returns the unique byte ID of this component.

Returning to the `register()` method, the MC first validates that the component ID is unique, and then adds a mapping to the component routing table from the ID to the ClientComponent. The ClientComponent's `setMessageOutput()` method is then invoked with a new ForwardingStream attached to the MC's QOS. The ForwardingStream is constructed with a MessageHeader whose component ID corresponds to this

ClientComponent. When a component sends a message to this stream, it will automatically be forwarded through the QOS to the output channel buffer, where it is queued until the MC sends it to the MessageServer.

Note that the MessageServer itself ignores component IDs. Servlets are concerned only with client IDs, as they only route messages between clients. Component ID routing is relegated to the MessageClient instantiated by each client.

5.3.1.2 Startup

Once the MessageClient has been constructed and all applet components registered, the applet starts the MC by invoking its inherited `start()` method. The first runtime task is to begin copying messages queued for output in the BlockingQueue to the TCP/IP output socket stream. As usual, a QueueInputStream is attached to the queue and a HaltingMessageCopier created to copy messages from it. The message copier is attached to a PassthroughStream, that is, in turn, connected to the TCP/IP output stream.

As you recall, the PassthroughStream directly writes all data to the stream to which it is connected. We can use such a simple stream because the outgoing messages' headers are already complete: the ForwardingStream used by each ClientComponent added the header before the message was queued. The MessageClient itself never initiates a message, in contrast to the MessageServerHandler and its subclasses, which send responses to client requests. Accommodating server responses forces us to use a MessageOutputStream: server responses are written directly to the MOS, bypassing the channel buffer. This stream adds a MessageHeader identifying the server as the origin of the response. When forwarding messages from other clients, the handler's message copier directs the MOS to preserve the original message header by invoking the MOS' `sendRaw()` method.

5.3.1.3 Routing

The figure below depicts a MessageClient after it has connected to a MessageServer. The applet starting the MC has registered three ClientComponents, each with an entry in the component routing table and its own ForwardingStream for outgoing messages. A principal role of the MessageClient is to provide a buffer for outgoing messages, just as MessageServerHandlers do on the server. In this case, the buffer provides a convenient means of synchronizing the output from several ClientComponents, while ensuring that messages are not lost due to high traffic or network packet delay.

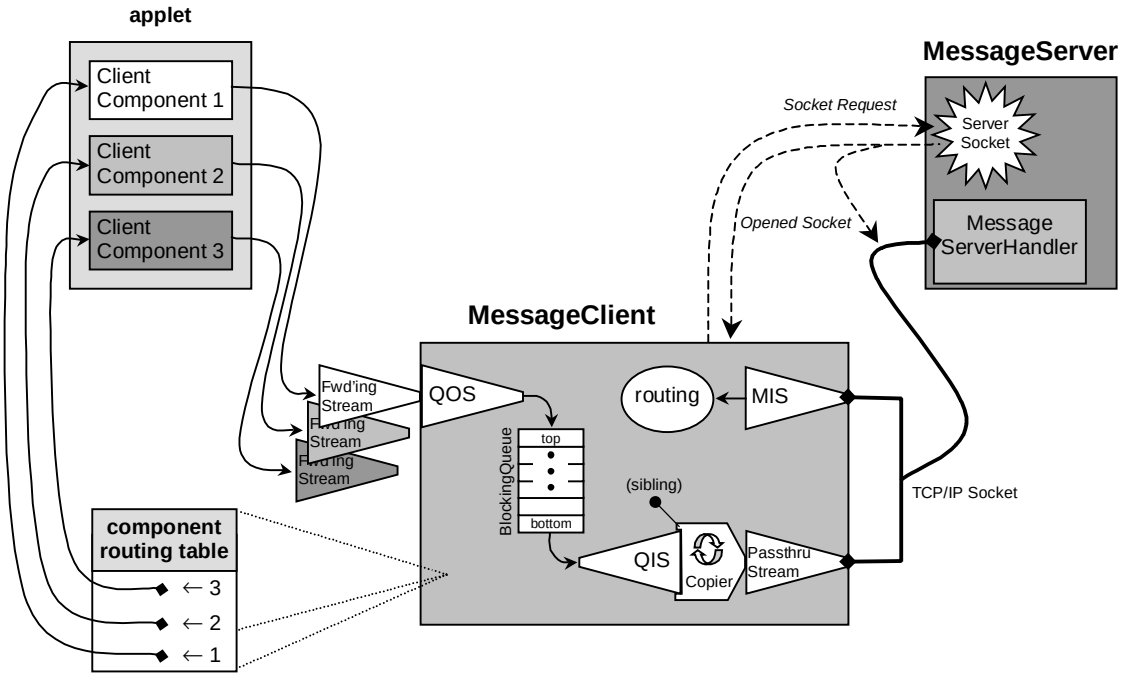

Figure 5-6: Message Client

With the channel for outgoing client messages in place, the MessageClient starts the message copier and enters a routing loop for handling incoming messages. The routing loop body retrieves a message from a MessageInputStream attached to the incoming TCP/IP socket stream, and parses the message header. The component ID of the message source is looked up in the component routing table, and if found, the corresponding component's

`receive()` method is called with a new DataInputStream attached to the MessageInputStream: reading from the DataInputStream extracts data from the latter stream, which at this point contains the newly received message.

5.3.1.4 Termination

A running MessageClient will terminate when it is interrupted or when a network failure causes the TCP/IP socket to the server to break. If the thread has been interrupted, it is likely because the applet that invoked it or the HaltingMessageCopier delivering outgoing messages has sent a `stop()` message. As the MC unwinds, it stops the message copier, closes its TCP/IP socket output stream (which leads to the closure of all streams connected to the socket), and finally deregisters all ClientComponents. Component deregistration removes the component's mapping from the component routing table and sends the component a `messageDisconnect()` message. Thus, all resources taken by the MessageClient are freed; any errors triggered during cleanup are captured and discarded.

5.3.2 Registry Client

Communication with the ∈F Registry is handled by a RegistryClient, a subclass of MessageClient. The RegistryClient (RC) constructor is similar to that of the MC, accepting the hostname and port number of the server, a flag indicating whether the client is a participant or a listener, and also a ConceptGraphClient object. The CGClient is initialized by the Concept Graph applet to reflect the state of the student logging into ∈F, except that the client ID is unknown. The RC proceeds with the logon process detailed when we discussed the Registry Server (see Part 5.2.2.1), receiving a new client identifier for the student with which the CGClient object is updated accordingly.

RegistryClient maintains the component routing table it inherits, as well as a *registry* that maps client IDs to CGClients. Recall that after a client logs into the Registry, the server responds with a list of the CGClients for every other client currently in the system. This listing is used

to initialize the registry contents, and in effect ensures that the client's registry is identical to the registry maintained on the server.

5.3.2.1 Components

The RegistryClient expects its components to implement a much richer interface, RegistryClientComponent. The principal addition to the RC component interface is callback support for the many special services provided by the ∈f Registry: a component instigates a service request by directing the RegistryClient to send the server a request, and once the response has been received the RegistryClient notifies the correct component by invoking one of the methods constituting the RegistryClientComponent interface. As such, components need only implement those methods that correspond to services they expect to use. The interface consists of the following methods:

- ***registryConnect().*** This method is invoked after a component has been registered with a connected RegistryClient.

- ***registryDisconnect().*** This method is invoked when a component is deregistered from a RC, and is called automatically after the RC disconnects from the server.

- ***reportClientLocation(ConceptGraphClient).*** The component requested the location of a student, and the server has responded with the CGClient for that student; the argument will be null if the student could not be found.

- ***reportClientList(Array).*** The component requested a list of all clients logged into the Registry, and the server responded with the list of CGClients contained within the Array argument.

- ***clientEntered(ConceptGraphClient).*** When a participant enters ∈f, every other listener and participant is notified thereof. The RegistryClient forwards this notification to each component by sending it the `clientEntered()` message.

- ***clientLeft(ConceptGraphClient).*** This method is received when the participant determined by the argument has left ∈F.

- ***clientMoved(ConceptGraphClient).*** This method is received when a participant has moved within ∈F; the new location is contained within the new CGClient.

- ***loadFile(boolean, byte[]).*** This message is received in response to a load file request. The first argument is true if the load was successful, and the byte array contains the file contents; otherwise, the load was unsuccessful and the array is empty.

- ***reportSaveFile(boolean).*** This message is received in response to an attempt to save a file. The argument is true if the load succeeded, false otherwise.

Note the one-to-one mapping between the service responses supported by the RegistryClientComponent interface, and the RegistryServerHandler services detailed in Part 5.2.2.2.

5.3.3 NetEvents Client

NetEventClient is responsible for communicating with the NetEventServer described in Subsection 5.2.3. The interface to be supported by NetEventClient components is extremely simply. Like RegistryClientComponent, NetEventClientComponent will include methods that notify components that a server connection has either been established or broken. The only other method of significance, `handleNetEvent()`, will serve to handle received NetEvents, a networked version of the `handleEvent()` handler Java applications invoke for local UI events.

Most likely, NetEvent client components will implement `handleNetEvent()` to dispatch the message to `handleEvent()`, but with a flag set to indicate that this message is a NetEvent. `handleEvent()` will not distinguish between local and network events except to ensure that only local events are broadcast, by way of NetEventClient, to others. This, of course, prevents NetEvents from being continually retransmitted.

NetEventClient should otherwise prove to be a straightforward extension of MessageClient or RegistryClient. However, this will depend upon the implementation chosen for NetEventServer. As discussed in Section 5.2.3, the serialization technique employed by NetEventServer may require logic for NetEvent synchronization to be built into NetEventClient or its components.

Chapter 6

Messaging in Practice

We have examined in great detail the Messaging API upon which Educational Fusion networking and collaboration is built, and now it is time to step back and examine these features from a student's perspective. First, we shall examine how the Concept Graph applet handles logging in and out of EF, and how it links students to one another as they work on a project. Next, we turn to the Collaborator applet by which students actually communicate.

6.1 Concept Graph Networking

In our previous discussion of the Concept Graph, the applet was described in the context of one client working in isolation – clearly this is not the scenario under which we envision it being used in practice. In fact, the Concept Graph applet is the principal means of locating fellow students and staff, features we describe here.

6.1.1 Client Registration

When the applet is initialized, it immediately creates a RegistryClient and attempts to register with the Educational Fusion RegistryServer. The RegistryClient logs on as a participant, rather than a listener, as the student's movements throughout the concept graph are to be made public.

Upon a successful connection, a Logged On dialog box appears. This dialog informs the student that registration was successful, and is to remain open as long as the student wishes to remain connected: clicking the Logoff button within the dialog terminates the connection. In the case of a networking failure, the student is notified that registration failed and that the applet will continue to operate in *disconnected mode*. In this mode, the client can still access local concept graphs and Workbenches, but all networking features are unavailable. If the student should choose to logout, the applet switches to this disconnected mode.

Requiring that the Logoff dialog remain open may seem awkward, and indeed it is. However it is a necessary compromise for running a networked application within a Web browser. When the browser in which an applet is running loads a new page, replacing the Educational Fusion Laboratory – note that this is distinct from loading a new page into one of the other frames within the laboratory page – the browser calls the applet's `stop()` method, but leaves the applet suspended in memory. If the student returns to the Laboratory page, for example via the "Back" facility common to most browsers, the applet's `start()` method is invoked. The Logoff dialog lets the student know that she is still logged into the system, and allows her to logoff without actually backing up to the ∈f Laboratory page.

As currently implemented, `stop()` and `start()` have no effect in the Concept Graph. Logging in/out upon every start/stop invocation would not only increase network overhead, but would require us to confirm whether or not the student's open concept graph should be saved. We felt that such an interface would be disruptive.

The Logoff dialog's purpose further manifests when examining Web browser behavior when a client reloads the Educational Fusion Laboratory while still logged in: the browser will invoke the running applet's stop method, and then create and initialize a new instance of the applet. Having no means of detecting that the initialization is occurring while a session is open or replacing the new instance with the already running instance, the Concept Graph

applet proceeds to establish a second Registry connection. The second Logoff dialog for the new session makes this rather confusing behavior apparent to an end-user.

6.1.2 Avatars

Once logged in, the Concept Graph's RegistryClient receives a list of all students currently in the system, which is forwarded to the ConceptGraphView. Avatars for each student are positioned on the view to reflect the module at which that student is currently located. A student's Avatar is located at the last module he activated. Avatars for students who are working on other projects or have not begun work on a module are not visually represented. A special Avatar is also created at startup for the student who opened the applet. Each Avatar is associated with the client ID of the student it represents.

Left-clicking the mouse on an Avatar identifies the student's username and nickname in the Status Bar; this facility consults the RegistryClient's registry, retrieving the CGClient that corresponds to that Avatar's identification number. At any time, a student can determine the whereabouts – via the ∈F Registry's Locate User service – of any other student by supplying his username: if the student is found, his location is given in a dialog box; otherwise, the student is notified that he is not currently logged in.

As a student moves through the system, the RegistryClient invokes the Registry's Move service, notifying everyone else of the move. Upon receipt of such a message, the Concept Graph applet will relocate that student's Avatar appropriately so as to correctly represent the system's current state. Similar messages are sent between clients as students enter and leave ∈F.

6.1.3 Persistence

While the earlier discussion of Concept Graph persistence was fairly complete, we now better understand the Registry services, Load File and Save File, that the RegistryClient uses to retrieve and store concept graphs. Note that these services are completely generic,

capable of retrieving any file from the Educational Fusion file area into a byte array or storing any byte array as a file. In no way are they tied to specifically persisting concept graphs.

6.2 The Collaborator

The Collaborator applet provides a chatboard and whiteboard for communications between ∈f clients, and is depicted in Figure 6-1, below. Normally, the Collaborator applet occupies a single pane of the Educational Fusion Laboratory as depicted in Plate 4 on page 115. Here it is depicted running within its own page. The Collaborator, like the Concept Graph, can also be run as a standalone application.

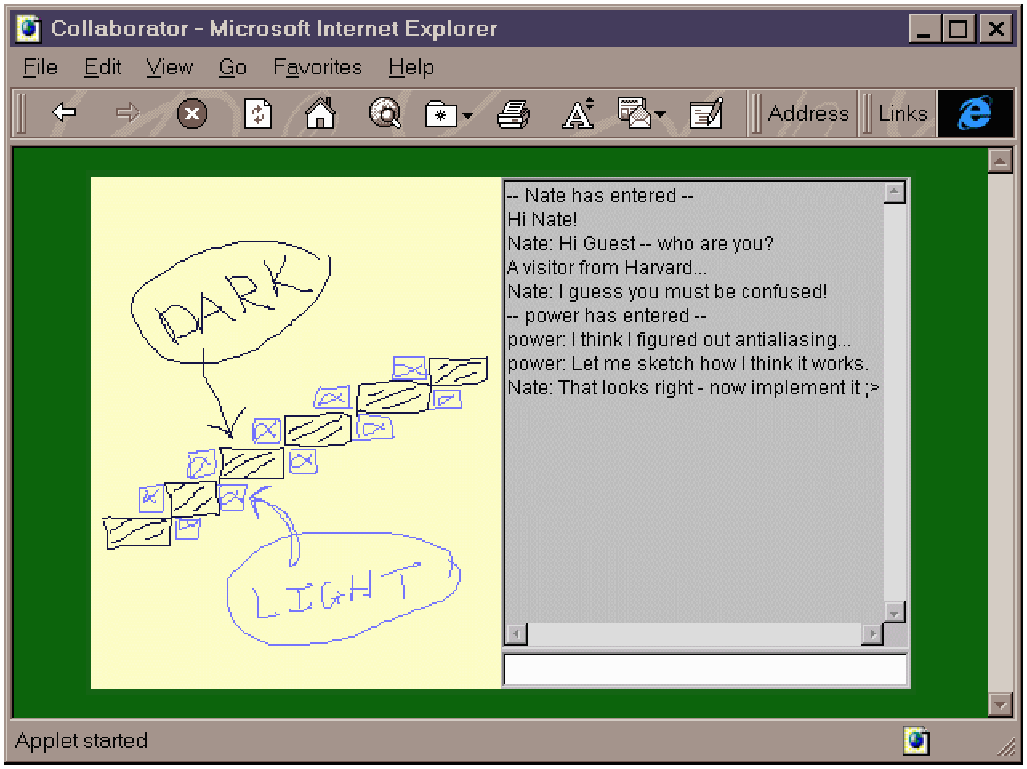

Figure 6-1: The Collaborator Applet

6.2.1 Registration

When a Collaborator is started, the first thing it does is parse its parameters; recall that these parameters are supplied to the applet via an applet reference within the HTML page loaded by the browser, or as command-line arguments when it is running as an application. These parameters identify the student that has launched this instance of the Collaborator, the *tracking ID* for that student, and the hostnames and port numbers for connecting to the ∈F RegistryServer and MessageServer.

Just as an instance of the RegistryServer is running at all times to support ∈F Registry services, an instance of the MessageServer is always alive to support Collaborators. After a Collaborator has parsed its parameters, it initializes and displays its visual components, and then creates a RegistryClient that connects to the Registry as a listener. Collaborators connect to the ∈F Registry so that they are notified of the arrival and departure of ∈F participants, however as listeners their presence is unknown to others. This methodology makes sense because when a student logs into ∈F, the Concept Graph should have already registered him as a participant. We also wanted students to be able to be involved in Collaborator discussions without actually having to open a concept graph.

Assuming successful registration, the Collaborator attempts to connect to the ∈F MessageServer. After the connection has been made, the Chatboard and Whiteboard components are registered with its MessageClient; the Collaborator applet itself is the only object registered with its RegistryClient.

The Collaborator applet maintains a ConceptGraphClient representing the state of the student by whom it was invoked. This CGClient is constructed based upon the applet parameters as well as the tracking ID, mentioned above. If a tracking ID is supplied to the client, then this student is also registered with the ∈F Registry as a participant: the client ID of the participant is given by the tracking ID. Thus, whenever the applet receives

information from the Registry about a client whose ID matches the tracking ID – such as that the client has moved or left the system – the CGClient for this student is updated. This mechanism provides a means for synchronizing the Collaborator and Concept Graph applets running within an ∈F Laboratory for a single student. If the client being tracked logs out of ∈F, tracking is disabled: client IDs are randomly-assigned and hence good only for one session, hence it would only be by sheer coincidence if the same student received the same client ID when she next logged in.

6.2.2 The Chatboard

Students send textual messages to one another via the Chatboard, which occupies the right half of the Collaborator depicted above. Text entered in the text-entry field at the bottom of the Chatboard is broadcast to every client connected to the ∈F MessageServer. Messages are displayed in the scrolling text area above the text-entry field: received messages are preceded with the nickname of the student that sent the message, while sent messages are echoed without any leading nickname.

As we noted above, the Collaborator registers with the ∈F Registry so that it is notified of special events, such as clients entering or leaving. Upon receiving such notification from the MessageClient, the Collaborator writes a descriptive message to the Chatboard's text area. These event messages are preceded and followed by "--" to discriminate them from standard messages. In the example shown above, event messages indicate that clients entered representing students with nicknames "Nate" and "power".

6.2.3 The Whiteboard

The Collaborator includes a Whiteboard for accommodating graphical communication. The Whiteboard, as implemented, is quite simple – yet also easily extended to include more powerful functionality. Students sketch within the whiteboard by dragging their mouse cursor, producing a line in a color that they specify. Figure 6-1 pictures one student

demonstrating the idea behind anti-aliased line drawing to the rest of the class. The Whiteboard also serves as a feasibility test case for real-time collaboration: even with many clients participating, sketches appear across distributed machines with negligible delay.

To optimize network communications, each line sketched undergoes three distinct phases:

1. ***Start.*** As soon as the mouse button is depressed within the Whiteboard canvas, a line is begun by broadcasting the coordinate at which the mouse was depressed (the line's starting point) and the color of the line. Each recipient adds this information to an internal hash table, using the drawer's ID as the hash key.

2. ***Continue.*** As the mouse is dragged around the canvas, each intermediate coordinate of the line being sketched is broadcast. Upon receipt, the recipient looks up the sender's ID in the hash table. A line is drawn in the specified color on the recipient's Whiteboard canvas from the coordinate in the hash table to the newly received coordinate, and then the latter replaces the former in the hash table.

3. ***Finish.*** Once the mouse button is depressed, the final line coordinate is broadcast. Again, the drawer's ID is found in the hash table, and a final line segment is drawn. Finally, the mapping for this line is removed from the hash table.

This mechanism, in effect, means that each client asynchronously follows the progress of all lines being drawn by every other client. We tradeoff client sophistication for lower network bandwidth A simpler solution would have clients separately broadcast each line segment within a sketched curve, so that recipients could simply draw the segment locally and discard the message. However, this would double network bandwidth: each delivered message would be over twice as large, containing starting and ending coordinates as well as the segment's line color.

Chapter 7

Assessing Educational Fusion

Over the course of the last four chapters we opened the hood and dived into the inner workings of the Educational Fusion system. First, we explored the interactions between the many ∈F components, understanding the various modes of communication employed between client browsers and applets and the ∈F server. Second, we examined the Concept Graph applet and the abstract concept graph which it visualizes. Third, we introduced the Messaging API and described how the message streams, servers, and clients function. Fourth, in the preceding chapter we saw how the ∈F Collaborator and Concept Graph employ the Messaging API to provide collaborative facilities.

We now return to the ∈F objectives proposed in Chapter 1, applying these objectives as measures with which to assess the ∈F components. The objectives will be assessed in the order in which they were presented, each evaluation including suggestions as to how any shortcomings might be overcome. Once our appraisal of the ∈F system has been concluded, we offer suggested directions for future research and development. These latter directives should be viewed as long-term goals that could evolve into research wholly independent of Educational Fusion.

7.1 Simulating A Virtual Programming Laboratory

The first goal we laid out was to simulate a programming laboratory that provides an intuitive visualization of course material, while allowing students to easily identify both work completed and work in progress. Our second, complementary goal was that students within this virtual laboratory should be able to easily locate fellow students and staff, and annotate system components with hints and suggestions for those that follow.

The ∈F Concept Graph applet is a big step towards realizing these objectives. The concept graph abstraction has been shown to be a powerful one, capable of expressing virtually any computational subject matter. Moreover, when we add in the capabilities of the ∈F Registry, suddenly students are connected as never before, with Avatars and search mechanisms making it effortless to locate others.

However, there are shortcomings. Perhaps the principal shortcoming of the Concept Graph is the lack of non-intrusive progress-tracking. Currently, there is no means by which staff can virtually "check-off" students' work to indicate that modules have been completed satisfactorily. We feel that an interface for doing so could fit within the administrative mode of the Concept Graph, or perhaps deserves a separate mode requiring only “educator privileges”. Our vision is of a Teaching Assistant opening a student's concept graph from within her own ∈F laboratory space, proceeding to module workbenches to examine student code and behavior thereof, assigning a numeric or letter assessment to each. When the student next opens his concept graph, the assessments would be visible for his review, along with any comments the TA might have added. Moreover, the Concept Graph applet would visualize finished work, adding a small check mark to each completed module, for instance.

A second shortcoming of the Concept Graph is the lack of a MOTD feature, whereby students and staff can annotate concept graphs to provide helpful information for others. Such tips might clarify the assignment or algorithm at hand, provide pointers to additional

information about the topic, or expand upon the data structures and functions available to the student from within the module being implemented. These comments could be added by simply selecting a concept graph module or topic and opening a dialog box that accepts a text message. Perhaps these comments would enter a queue, pending staff approval, before actually being made public. Implementing this interface would be relatively easy, and would be supported by a few simple Registry services.

Finally, the searching facilities offered by ∈F should be extended to include the ability to locate others by account type. This would give students the ability to quickly locate, for example, every teaching assistant or recitation instructor currently logged in.

7.2 Abstracting Algorithm Implementation

Third on the list of ∈F objectives was to abstract the compilation, visualization, and validation of algorithm implementation. We want to focus students on the semantics of the task at hand, rather than getting embroiled in the messy details of a compiler or debugger.

Our answer, the ∈F Workbench, does just that, giving rise to an entire new methodology for iterating the code-compile-test process. Never before have students been given a truly complete development environment, as previous attempts left out one critical ingredient: a framework allowing them to quickly evaluate their work. The reference implementation provided with each Workbench module make it all the easier to compare an algorithm implementation with the correct solution. Moreover, the Java classes upon which Workbenches are build make it straightforward for developers to build new modules, as they need only provide the components that are unique to each module: a visualization of algorithm output, a reference implementation of the algorithm, and a difference engine for computing the difference between the reference algorithm's output and that of the student's implementation.

The only deficiency of Workbenches is a lack of integration with the Concept Graph. For example, when a student toggles the output view of a Workbench between the reference, student, and difference engine, the Concept Graph is not notified. To solve this problem, the Workbench must begin to make use of the ∈f messaging and registry classes, to be able to exchange information with the Concept Graph.

7.3 Providing Face-to-Face Collaboration Facilities

The fourth intent of the ∈f team was to provide collaboration facilities that allow students to interact as effectively as though standing face-to-face. Filtering communications according to context was believed to be vital to limiting incoming stimulus to relevant information.

Perplexingly, in this regard the ∈f team has both made the most progress and yet is farthest from reaching our goal. No doubt because this is an extremely difficult problem. The Messaging API described in Chapter 5 gives developers a powerful set of tools for creating collaborative Java applets, having already solved many issues including: message identification and routing, channel buffering to guarantee lossless communications, insulation of applications from network and protocol errors, and automatic connection and registration. Moreover, we have demonstrated the power and performance of these core classes with the Collaborator applet and the use of the ∈f Registry by the Concept Graph.

It is imperative that a truly robust Conferencing applet soon be built to support communication amongst Educational Fusion students. This client should accept all incoming messages, but filter which messages are displayed so that only relevant information is presented. Students should be able to control the granularity of this filtering, limiting, for example, messages to the current module, topic, project, or perhaps disabling filtering altogether.

This Conferencing tool should combine the best of the two traditional Internet communications devices: threaded forums and real-time chats. Threaded forums, such as Usenet, provide a stateful bulletin board for communications, wherein participants follow *threads* of discussion: a thread consists of an originating post about some new topic, and all replies thereto. Participants can revisit the discussion at any time, and contribute asynchronously. Real-time chat, however, is a synchronous communication device wherein participants post messages that appear immediately to everyone else. Generally, there is no order to chat messages, as they simply scroll by (similarly to how the simple chatboard in the Collaborator operates).

We envision the ∈f Conferencing tool as a real-time threaded discussion, where threads are transmitted in real-time to all students. Moreover, these threads should be stateful: when a student enters ∈f, the Conferencing applet will automatically retrieve a history of ongoing discussions. So not only will students be involved in interactive discussions, but they will be able to review previous discussions – with which they may have never been involved – as well. The ∈f team believes this will prove to be an invaluable resource: attach a search engine to the Conference history and suddenly students can quickly determine if anyone else has been talking about any issue they are facing in their own concept graph development.

Although most of the functionality needed for building such a conferencing tool simply requires building the client applet, the MessageServer will need to be extended to persist messages. Of course, this kind of functionality is exactly why the messaging server classes were designed with a means of processing each message being routed: the MessageServerHandler's `receive()` method.

Moreover, ∈f must allow private conferences to be initiated. Supporting private conferences could be handled either by starting a new servlet on the server to support the conference, or by properly routing and filtering messages through the original conference server. This

design decision is a difficult one, and will likely be decided only after some trial-and-error programming. The author's intuition is that given the potentially high volume of traffic, it will make most sense to manage private conferences with dedicated servlets that establish connections via private TCP/IP ports. Implementing private conferences will demand constructing an interface for setting up a conference and selectively permitting new members to join, as well as persisting the conversation to a designated location for later review (most likely, the ∈f file area for the originator of the conference).

7.4 Enabling Real-Time Work-Sharing

Enabling students to share their work with their peers and course staff, whether for collaboration, instruction, or demonstration, was the fifth ∈f objective. To this end we have developed the NetEvents technology, a network layer that builds upon the ∈f Messaging API.

NetEvents is the most premature of the technologies described in this document, and constitutes one of the principal areas of active research. We believe that the NetEventServer and NetEventClient we have outlined will enable peer-to-peer applet sharing at relatively little implementation cost to developers. Moreover, based on the performance of the Collaborator applet, the ∈f teams feels that NetEvents should make interactions appear real-time, without which application sharing is cumbersome at best.

Some extensions to the client and server technology discussed earlier are still needed, however. Two of these extensions mirror proposals for the Conference tool: persisting message streams and initialization of private sessions. When applied to NetEvents, persistence will allow client sessions to be recorded and later played back. And clearly a private session is required by the group of individuals sharing a particular applet.

Beyond this, there are many application-specific extensions that might be added to NetEvents clients. A primary concern is that different members of a session be assigned different access privileges: for example, perhaps only the initiator of a Concept Graph sharing session should be allowed to save or load a concept graph. Another issue is how to make each client aware of the others. In the instance of a shared Concept Graph applet, perhaps only the Avatars for participants would be shown. Alternatively, displaying the position of all participants' mouse cursors, possibly as crosshairs, within each running applet would provide much more detail as to how others are interacting and the actions for which each is responsible.

7.5 Maintaining Platform-Independence

The system described in the document is remarkably platform independent given its complexity. The Java applets and servlets and Perl Dispatch Scripts contain no code that is in any way tied to a specific platform, with the exception of the way in which files are accessed on the server. This distinction is unavoidable, as every operating system has its own file- and path-naming conventions.

Wherever possible these distinctions have been relegated to configuration files, and as a result nothing needs to be recompiled when ∈F is shifted to a new platform. Moreover, these parameters must be updated during any ∈F installation, regardless of platform, as they provide the location at which the ∈F files are located. ∈F does not hard-code path or file names.

The veracity of our claims are validated by the fact that midway through the development process, the ∈F team decided to shift our efforts from the Unix platform to Windows NT. The entire effort took only a day, and forced us to isolate and document all operating-system dependencies. Moreover, to this day team members concurrently develop the constituent Java and Perl software on both NT and Unix workstations.

7.6 Future Directions

Having seen how Educational Fusion meets its objectives and addressing any shortcomings, we direct the reader to look ahead to its future.

7.6.1 Content Provision

Educational Fusion defines a power framework for interactive learning and teaching, yet a framework is of little use in and of itself, just as the skeletal frame of any creature is guaranteed to be inanimate once the surrounding tissues are stripped away. We believe that in order to become a viable resource with the university environment, ∈F must begin actively seeking content provision in the form of concept graphs, topics, and modules. Once a critical volume of development has been accomplished – likely defined by the point at which sustaining ∈F is no longer an option, but rather a necessity – soliciting new developers and educators will become unnecessary, as people will actively seek to move coursework to our platform.

The ∈F team has already taken the first step, and is piloting the system for the introductory Computer Graphics course here at MIT, taught by Professor Teller. Moreover, we are in contact with researchers at Brown and U.C. Berkeley, who have expressed interest in both developing and deploying concept graphs for computer science courses.

Another form of content provision that is being actively pursued is to integrate ∈F with online instructional tools such as Brown's Interactive Illustrations. As we have shown, ∈F provides excellent resources for learning by doing and collaborating, but the system omits the first phase of the educational process: introducing students to the material at hand. Interactive Illustrations, as well as many of the other online learning initiatives discussed in Chapter 2, provide exactly this kind of material in an interactive, electronic format. Seamlessly fusing these phases into a complete package would result in an even more complete system for online learning.

7.6.2 Concept Graphs

The expressibility of the concept graph abstraction remains an open research question. Perhaps the best approach to understanding this issue has already been mentioned: providing content. As ∈F concept graphs are built of increasing sophistication, the limits and capabilities of the abstraction should become better understood.

Nonetheless, yet more powerful means of abstraction are already under review. Primarily, we are interested in the possibility of extending the concept graph to support *inheritance*. Inheritance is the a for the object-oriented way in which one object specializes or extends the behavior and state of another: the specialized subclass is said to *inherit* from its more general parent. When applied to concepts graphs, if module A inherits from module B, then module A is expected to extend the functionality provided by module B. Many modules currently grouped within a single topic could be related in this fashion: returning to the Computer Graphics concept graph illustrated in Plate 1, page 112, the Gupta-Sproull anti-aliased line drawing module could be seen as a specialization, or inheritor of, the standard Bresenham module. Yet is inheritance really the correct relationship, given that it would be very awkward to actually make use of the Bresenham's algorithm when implementing Gupta-Sproull?

Another avenue of research also stems from object-oriented techniques: we would like to see ∈F topics as independent entities, separable from concept graphs. Developers would add each new topic and its modules to a global pool of topics, from which educators would draw when constructing concept graphs. Topics become modular components that can be "plugged-in" wherever they are needed.

An immediate benefit of this approach is that topics become reusable components that can easily be shared between projects. Secondly, it will be much easier for educators to tailor a concept graph from year to year or course to course, mixing and matching topics with the

course material as they see fit. Thirdly, this metaphor allows ∈f to be easily extended to support topics and modules added dynamically by students. As the system is currently implemented, a concept graph is a static template that students fill-in. We would like it to evolve into a dynamic document where anyone involved in the course can add new topics and modules: the final project in the Computer Graphics course, for example, might be for students to extend the supplied graphics pipeline in some novel way, say by adding a Shadows topic or a Bump-Mapped module to the Lighting topic.

Finally, while educators certainly want to encourage independent exploration of the topics covered by a course concept graph, they may also like to enforce the order in which topics are actually implemented. We envision a set of rules that define deadlines by which module implementations must be submitted, accompanied by another set of rules that restrict which modules Workbenches are available based upon which have been completed. Entirely optional, such rules could provide some structure to prevent students from getting lost.

7.6.3 Validation and Tracking

Validation and tracking of student progress is another ∈f technology that is in its infancy. Perhaps one of the most interesting possibilities is automation of the validation process. Each module implementation would be accompanied by a set of scripts that exercise the algorithm being implemented; these scripts could potentially be created by recording NetEvents sessions. Once a students submits an implementation and declares it complete, ∈f would replay the test scripts with the module in difference mode: any anomalies between the reference and student implementation output detected by the difference engine would be reported to the student (or perhaps be privy only to course staff).

The difference engine itself holds many interesting prospects. As currently implemented, module difference engines simply compare the output of an algorithm; output is defined as the sequence of calls an algorithm makes to a given method, e.g., a line-drawing module's

output might consist of all calls to a pixel-drawing procedure. For some algorithms the order in which calls are made is not relevant to algorithm correctness – such as any randomized algorithm – while for others it is essential. Currently, no implemented difference engine considers output ordering, nor have we devised a visualization of difference engine output that intelligently conveys this ordering. Moreover, there are several other dimensions along which output could be measured, namely time and resource consumption.

The ∈F Registry also could be modified to record detailed statistics about the progress and effort of every student in the class. Some parameters for which progress might be recorded include the time spent on particular modules and topics, the number of revisions of code submitted before a student implementation is complete (and how many more until it is correct), and how actively the student participates in ∈F conferences and NetEvents sessions. Aggregating this data could provide fascinating insights into how a class as a whole spends its time and effort. This kind of information is an educator's dream, but also raises numerous ethical questions concerning student privacy.

Student progress tracking also suggests integrating ∈F with a revision control system. Revision control would provide students with a means of retrieving past versions of concept graphs and module implementations. Traditionally, students fail to take advantage of revision control systems due to the steep learning curve and awkward interfaces of the tools commonly available.

7.6.4 Collaboration

A principal long-term collaboration goal of the ∈F team is to complement our messaging services with real-time audio and video conferencing. But we do not see these technologies as being built atop the ∈F Messaging API: adequate performance requires tying real-time audiovisual streams to lower-level network protocols. We view such technology as a

throwback to more traditional forms of communication, forms of communication that are actually less powerful for sharing complicated algorithmic concepts than a simple text-based chatboard or a networked Java applet.

While we do believe that the existing Messaging API provides a powerful foundation for EF networking, we have not yet examined the system's behavior under extremely heavy loads. By no choice of our own, this behavior may be revealed when EF is piloted in the MIT Computer Graphics course. Preferably, a set of benchmarks can be established to measure response time and other performance parameters under an artificially high load, with which a feasibility analysis can be performed to ascertain how many students EF can support. Of course, this will greatly depend upon the resources available to the EF server, both in terms of local computing power and network bandwidth.

7.6.5 Server Performance and Network Topology

This brings us to yet another possible area of research: addressing the server-side scalability of Educational Fusion. As any Web surfer knows, trying to access an extremely popular Web site can be an extremely slow, frustrating affair. Far too often, a Web server cannot cope with the demand for the pages it serves. The best solution to this problem has proven to be redundancy: the data served by a Web site is mirrored across several servers, across which incoming requests are automatically distributed.

However, the dynamic nature of EF state greatly complicates the situation. If several servers handle EF networking, each running the appropriate messaging servlets, we must devise a way to ensure that each server has an identical view of the EF file area. One means of doing so is to extend the EF servlets with synchronization mechanisms, so that any change in local state is reflected across every other server. An alternative solution separates the EF data from the actually network processing, providing a common file area for all servers.

If the sheer volume of network traffic between ∈F clients and the server is the limiting factor, this latter method should suffice. This is the preferable solution primarily because of the simplicity with which it can be implemented: in effect, the ∈F network topology would be arranged into a three-tier system, with the bottom tier consisting of a data server that handles all requests for system state from the network servers actually communicating with clients.

Unfortunately, if the performance bottleneck stems from excessive CPU or disk use this method may fail, as all data accesses are still concentrated on a single machine, the data server. In this case, it will be necessary to distribute the data processing itself across several machines, as per the first solution suggested. Implementation of server synchronization will of course be difficult – indeed, without proper buffering synchronization itself could prove intractable, generating large volumes of network and disk activity. Perhaps a hybrid solution would best serve in this case: maintain a three-tiered topology, but distribute the bottom tier across several data-only servers. The data tier would be responsible for automatic synchronization (as supported by many modern databases), allowing us to blindly distribute ∈F state requests from the network tier.

A common attribute of all of these possibilities is use of a database to manage Educational Fusion state. The current method of relying upon the host operating system's file system is fine for a developmental system, but will not be adequate if ∈F must scale to hundreds or thousands of students. Luckily, integrating ∈F with a third-party database should not prove to be a daunting task: the next version of the Java API, the Java Development Kit (JDK) 1.1.1, includes Java Database Connectivity (JDBC). JDBC provides a standard SQL interface to most popular relational databases [SM97a].

Moving to a database-driven architecture could also help resolve a related shortcoming of Educational Fusion: when a student logs into the system multiple times, no synchronization

mechanisms are in place to guarantee that a student does not lose work. For instance, a student connects to ∈F twice, and opens the same concept graph within each of her two virtual laboratory spaces. After several modules have been completed within one version of the graph, she saves her work in both and logs out. If the unaltered graph is saved last, her work will be lost. Of course, simply disallowing multiple logins is the most straightforward solution.

Subtler defects may arise during NetEvent sessions, wherein many distributed clients share access to one participant's data. If ∈F state were managed with a database, one could easily leverage the built-in locking mechanisms to preclude such problems. This solution is far more appealing than implementing mutexing directly within ∈F.

7.6.6 Security and Academic Honesty

As discussed in Subsection 3.1.4, the ∈F security model replies upon trusted clients: students will never accidentally infringe upon one other, but a willful saboteur could readily impersonate another student. The first step to rectifying this, as aforementioned, is to rely upon a secure Web server, which will automatically encrypt all communications between browser and server.

However, we still require a means of encrypting applet to servlet communications. One possible solution is to overlay the Message API upon an encrypted message stream. The encryption technology could rely upon a Kerberos-style authentication, whereby each client obtains a secure *session key*, or ticket, with which to encrypt messages to the server. Moreover, the client ID issued upon registration would not be valid indefinitely – rather, it should expire after a fixed period of time, at which point the student is required to re-

authenticate. Further information about Kerberos authentication is available from the MIT Kerberos Team homepage[15].

The centralized nature of ∈f state also makes it much more risky for students to cheat. Student implementations of modules could be compared automatically, with educators notified via email of suspicious submissions. The persistent nature of the ∈f communication facilities make it very dangerous for students to use them for dishonest activities.

Despite the many safeguards that ∈f might offer, we have anticipated that it will initially be deployed in environments where students can be trusted to responsibly make use of the powerful tools available to them. As such, we must acknowledge that internal security has not been a top priority up to this point. On the other hand, denying foreign agents access to ∈f system – other than through Guest accounts – has always been an imperative.

7.6.7 New Technologies

Finally, many exciting technologies are nearing fruition whose adoption could enhance Educational Fusion. Perhaps the most significant of these is the upcoming release of JDK 1.1.1, as mentioned above. Besides JDBC, the database interconnect library, JDK 1.1.1 will offer Remote Method Invocation (RMI), Object Serialization, Reflection, and enhancements to the Abstract Window Toolkit (AWT) [SM97a].

RMI provides a means for Java objects to invoke methods of objects running in remote Java virtual machines, even on different hosts. Basically a Java-specific implementation of RPC, RMI could drastically simplify the service request and response protocols defined by ∈f clients and servers.

[15] `http://web.mit.edu/kerberos/www/`

Object Serialization provides a generic version of the ∈f Persistence classes, encoding any Java object into a byte stream and providing complementary reconstruction from that stream. Object serialization is much more sophisticated than the Persistent streams, programmatically persisting all object state and determining object relationships so that developers need not implement the `read()` and `write()` methods of the Persistent interface. Object Serialization supports any Java object, whereas the Persistence classes require special support for classes not created by the application developer.

Reflection allows Java objects to discover information about the fields, methods, and constructors of any other object or class. This technology could enable automating the publication of concept graph topic interfaces, so that topic and module linking can be verified at runtime rather than requiring an educator to define all relationships in advance.

Principal enhancements to the Java AWT include new graphical primitives such as pop-up menus and hierarchical lists. Pop-up menus are direly needed to simplify the confusing interface of the Concept Graph applet, whose many key commands listed in Appendix A.1 are far from user-friendly. Hierarchical lists are needed to support the threaded Conference applet envisioned for ∈f: incoming messages would be organized into a tree that could be quickly browsed with such a list. AWT 1.1 also supports a new event model that is much more extensible that the current implementation. This model could be used to implement the intelligent, semantic NetEvents proposed in Chapter 5. Regardless, it is recommended that all applets be migrated to the new event model, which is to replace the old standard.

Finally, the author recommends that the ∈f team continue to follow the development of the Distributed Component Object Model (DCOM) [BK96] and Common Object Request Broker Architecture (CORBA) [OMG97] protocols. These protocols are competing specifications of Microsoft Corporation and the Object Management Group, respectively, for building distributed, platform-independent, object-oriented applications. They include

techniques for publishing and locating interfaces, sharing and registering objects, global naming, and many other technologies in development for the ∈F system.

Chapter 8

Conclusions

Educational Fusion is a powerful tool for enabling learning and teaching of algorithmic concepts across a distributed, heterogeneous computing environment. Concept Graphs express concepts, and the relationship of constituent subject matter, in a structured way that allows students to quickly absorb course structure and content, while providing a convenient laboratory in which students can locate help and their peers. Workbenches act as algorithm construction toolkits that abstract the details of compilation, debugging, and testing so that students can focus on semantic issues. Tying together all of this are Conferencing tools, including real-time chatting, whiteboarding, and application sharing for both collaboration and teacher assisted guidance.

Currently, the ∈F team is readying the system for a selection of eager students in the introductory Computer Graphics course here at MIT. No doubt much about the current strengths and weakness of ∈F will be gleaned from this test run. Already in place is the initial framework for this course, accessible from the ∈F homepage.

We have verified the cross-platform capabilities of ∈F by installing the server components on both Windows NT and UNIX servers, while accessing ∈F from both Microsoft and Netscape Web browsers running on Windows 95, Windows NT, Macintosh, Silicon Graphics, Sun Microsystems, Athena, and Linux workstations. We anticipate that other UNIX platforms

should demonstrate equivalent compatibility. Moreover, EF clients have connected successfully while running directly on the machine serving EF, as well as by way of our local area network, distant T1 connections, and even modem-based dialup accounts.

Nevertheless, Educational Fusion's potential is far from fully realized. We have seen that these are emerging technologies, functional yet still in their infancy. Moreover, as the EF components have fallen in to place, each opens a new world of possibilities, whether it be automated validation of student work, audiovisual conferencing, or programmatic type-checking of concept graph topic interfaces. Our exciting work has just begun.

We must keep in mind that like so many other sciences, this field's technology has begun to outstrip the political and ethical status quo. It is imperative that we examine how EF will integrate with traditional educational practices, so that our perspective is not of the technologist alone. This process should include an address of ethical issues such as dishonest use of EF collaboration facilities and the implications of Internet security breaches.

Moreover, academicians must investigate the tradeoffs of a structured environment such as EF. Some, for example, will no doubt raise objections to Educational Fusion's concealment of the compiler and debugger, both of which are vital tools of any practicing computer engineer. And what of the student who is most comfortable working within the confines of a favorite development environment such as Microsoft's Visual Studio?

Finally, feedback from educators not focused on computer science curricula should lend a fresh perspective to this work, and perhaps even lead to techniques for extending EF outside of the algorithmic problem domain. It is easy for like-minded and educated individuals to overlook the obvious: maximizing exposure to Educational Fusion by distributing it as widely as possible may be the best countermeasure, as well as the best proof of its capabilities.

Plates

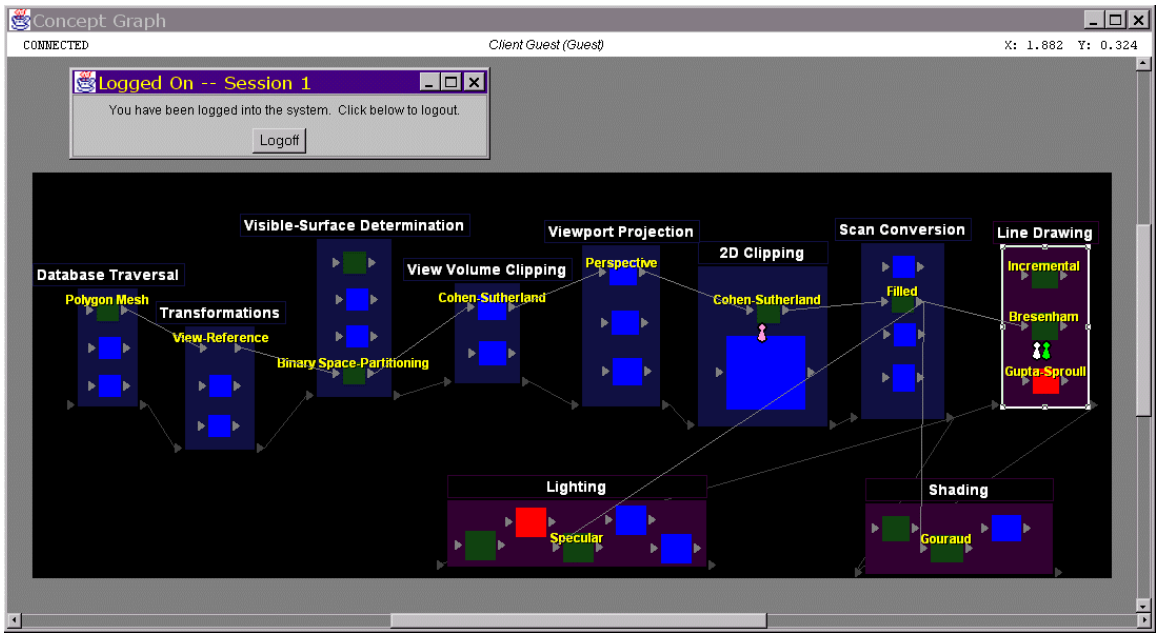

Plate 1. Concept Graph of a 3D Rendering Pipeline

The Concept Graph depicted here is running as an application rather than as an applet. The dialog box entitled `Logged On - Session 1` opens when the Concept Graph is started, and indicates that the student has successfully logged into the Educational Fusion Registry. This student's Avatar (white) indicates that he is currently working on the Bresenham module, within topic Line Drawing, which is selected. The status bar tells us that he last clicked on the Avatar for client Guest (green).

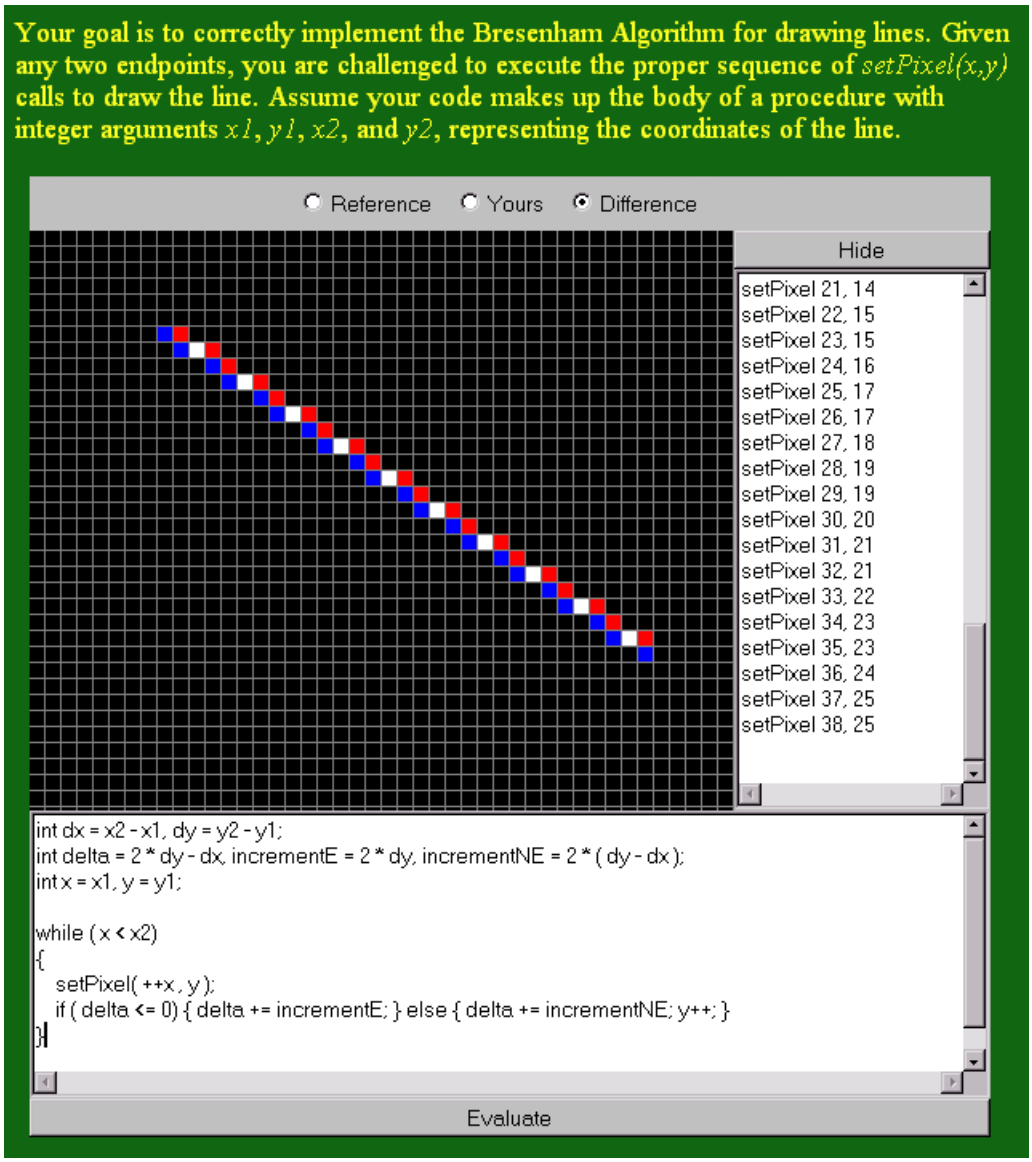

Plate 2. Bresenham Workbench with Incorrect Implementation

Shown here is the Bresenham Workbench, a module from the Concept Graph shown in Plate 1. The student's code is displayed in the lower text area, while the gridded region above is set to visualize the differential output between the student and the reference implementations: white pixels are correct, blue are missed, and red are extraneous. The scrolling region along the right edge displays debug information, comprising all calls to `setPixel()` by either implementation. The student's code is nearly correct when the slope of the line is less than one (the y axis is inverted) and the algorithm steps from left to right: it misses the right endpoint and is off by one x coordinate.

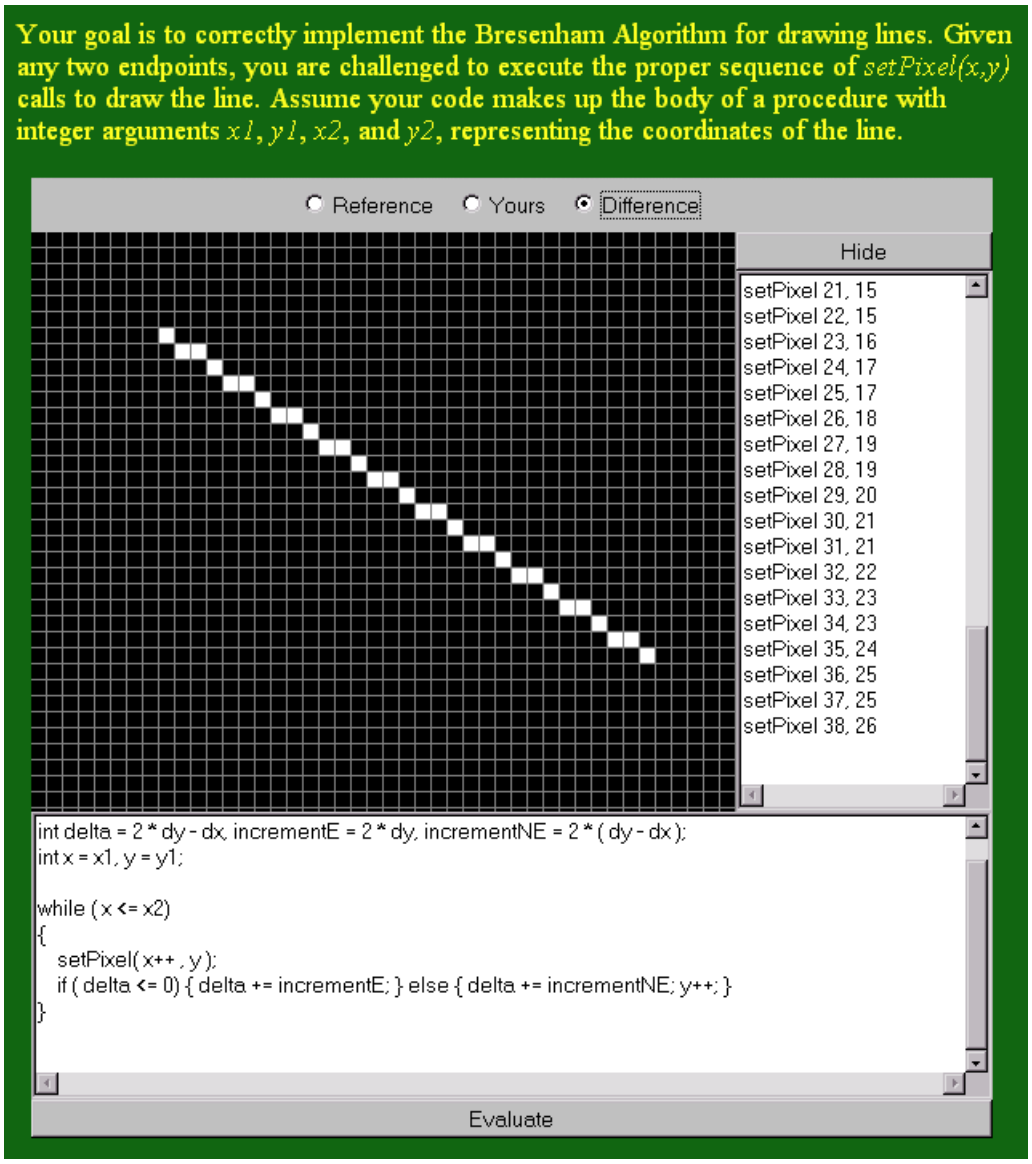

Plate 3. Correct Bresenham Implementation

The Bresenham Workbench is now visualizing the differential output between the reference implementation and a correct (in the case of a line with slope less than one that steps from left to right) student implementation: every pixel is correctly colored white. The student has fixed the endpoint and off-by-one errors present in the implementation of Plate 2.

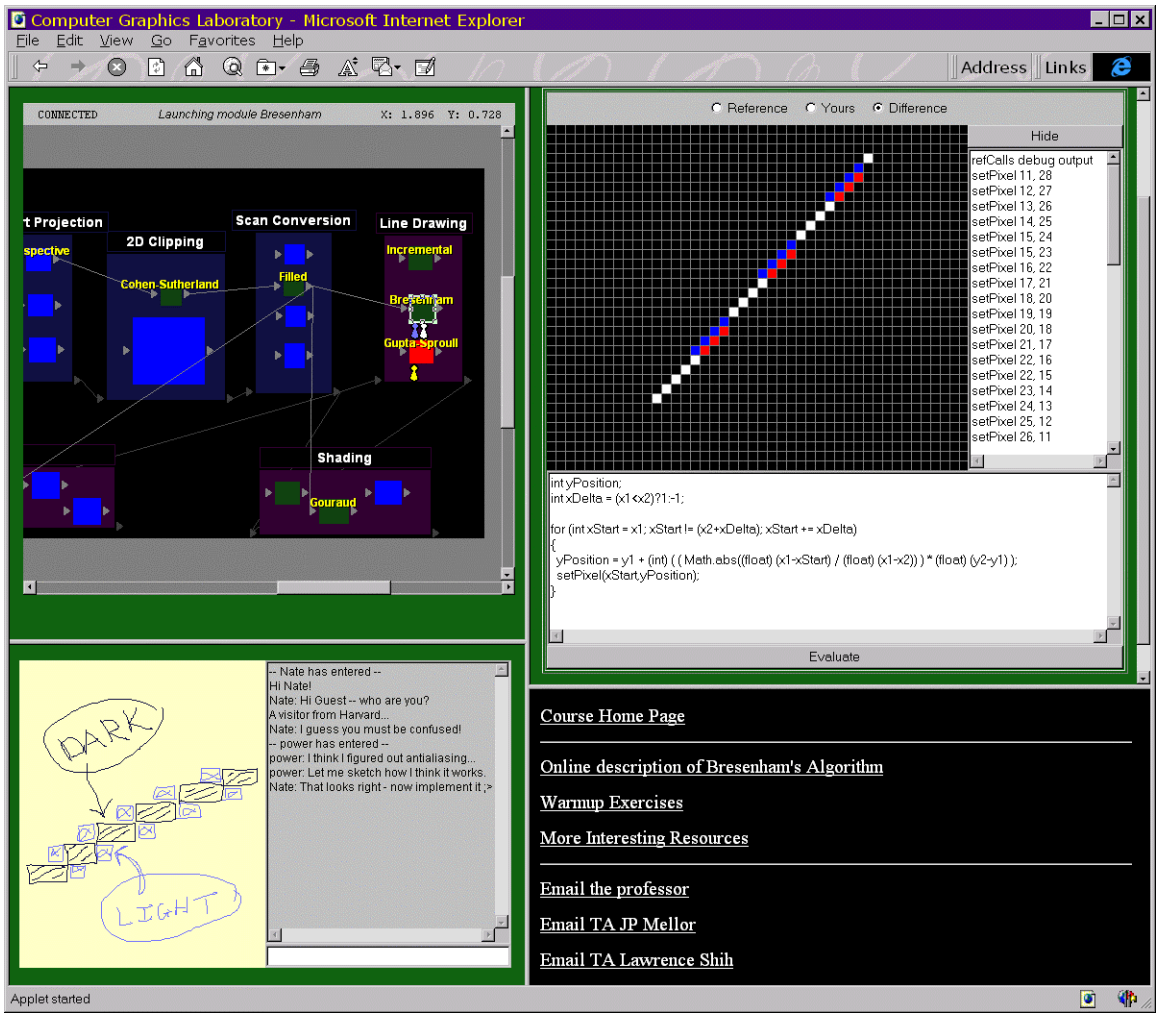

Plate 4. Computer Graphics Pipeline Virtual Laboratory

The complete computer graphics laboratory is split into four quadrants: the upper-left consists of the Concept Graph, the upper-right the current module workbench, the lower-left the Collaborator, and the lower-right hyperlinks to useful resources. The client, a Guest, has opened the Bresenham Workbench, and is actively engaged in a discussion with others via the Collaborator.

Appendix

A.1 Concept Graph Directions

General Instructions

Double-click	On a module to load its workbench here
Double right-click	In any unoccupied space to return to this page

Advanced Features

Right-drag	Connectors to relocate them
Drag	Module connectors to relink them
Drag	Components to relocate/resize
+	Zoom In
-	Zoom Out
l	Load a graph
s	Save a graph
u	Update user list/avatars
f	Find a student (by username)
F1	Reconnect to the Registry (after logging out)
F12	Open the About box
CTRL-D	Toggle the debug view on/off
CTRL-G	Toggle gridding on/off

Administrator Features

Drag	Topic connectors to relink
V	Set view properties
[Enter]	Edit selected component properties
M	Add a module
CTRL-E	Toggle execute/edit mode
CTRL-X	Quit

Bibliography

[Ber97] Tim Berners-Lee. "WWW Addressing Overview." *http://www.w3.org/pub/WWW/Addressing/*. World Wide Web Consortium, 1997.

[Bow97] Barry D. Bowen. "Educators Embrace Java." *http://www.javaworld.com/javaworld/jw-01-1997/jw-01-education.html*. Web Publishing Inc., 1997.

[BK96] Nat Brown and Charlie Kindel. "Distributed Component Object Model Protocol 1.0." *Internet Engineering Task Force Internet Draft*. Microsoft Corporation, 1996.

[Bur96] Paul Burchard. "Shared Objects System (SOS) for Java." *http://www.cs.princeton.edu/~burchard/www/interactive/sos/*. Princeton University Computer Science Department, 1996.

[Coh94] Abbe Cohen. "Inessential Zephyr." *http://www.mit.edu:8001/afs/sipb/project/doc/izephyr/html/izephyr.html*. The Student Information Processing Board, 1994.

[CH96] Gary Cornell and Cay S. Horstmann. *CoreJava*. Sun Microsystems, Inc., 1996.

[Chr96] Tom Christiansen. "The Perl Language Home Page." *http://www.perl.com/perl/index.html*. Tom Christiansen, 1996.

[DC95] Databeam Corporation. "T.120 Whitepaper." *http://www.dtic.mil/ieb_cctwg/contrib-docs/T.120/T.120-WP.html*. Databeam Corporation, 1995.

[Dol97] Steven C. Dollins. "Interactive Illustrations." *http://www.cs.brown.edu/research/graphics/research/illus/*. Brown University Department of Computer Science, 1997.

[FDFH90] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Huhes. *Computer Graphics: Principles and Practice.* 2nd ed., Addison-Wesley Publishing Company, 1990.

[GN97] Jim Gettys and Henrik F. Nielsen. "Hypertext Transfer Protocol Overview." *http://www.w3.org/pub/WWW/Protocols/*. World Wide Web Consortium, 1997.

[Gol+96] Murray W. Goldberg, et. al. "WebCT – World Wide Web Course Tools." *http://homebrew.cs.ubc.ca/webct/*. University of British Columbia Department of Computer Science, 1996.

[GVC95] Graphics and Visualization Center. "WARP." *http://www.cs.brown.edu/stc/edu/warp/*. Brown University Department of Computer Science, 1995.

[HHSW97] Merlin and Conrad Hughes, Michael Shoffner, and Marie Winslow. *Java Network Programming.* Manning Publications Co., 1997.

[Mar96] Thomas L. Marchioro II. "The Yorick Project." *http://uces.ameslab.gov/uces/authoring/*. Undergraduate Computational Engineering and Science Project, 1996.

[NC96] Netscape Communications Corp. "JavaScript Guide." *http://home.netscape.com/eng/mozilla/3.0/handbook/javascript/index.html.* Netscape Communications Corp., 1996.

[NC97a] Netscape Communications Corp. "Persistent Client State: HTTP Cookies." *http://cgi.netscape.com/newsref/std/cookie_spec.html.* Netscape Communications Corp., 1997.

[NC97b] Netscape Communications Corp. "The SSL Protocol." *http://home.netscape.com/newsref/std/SSL.html.* Netscape Communications Corp., 1997.

[NCSA95] National Center for Supercomputing Systems HTTPd Development Team. "The Common Gateway Interface." *http://hoohoo.ncsa.uiuc.edu/cgi/. NCSA, 1995.*

[OMG97] Object Management Group. "What is Corba?" *http://www.omg.org/omg00/wicorba.htm.* Object Management Group, 1997.

[Obj97] ObjectSpace, Inc. "JGL, The Generic Collection Library for Java." *http://www.objectspace.com/jgl/*. ObjectSpace, Inc., 1997.

[Pos97] Jef Poskanzer. "Package ACME." *http://www.acme.com/java/software/*. ACME Laboratories, 1997.

[Shi97] Hank Shiffman. "Java Grows Up: JNI, RMI, Beans, and More." *Developer News.* Silicon Graphics, Inc., 1997.

[SM97a] Sun Microsystems, Inc. "JavaSoft Home Page." *http://java.sun.com/*. Sun Microsystems, Inc., 1997.

[SM97b] Sun Microsytems, Inc. "RMI Documentation." *http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/*. Sun Microsystems, Inc., 1997.

[Tel94] Seth Teller. "NSF Career Development Plan." *http://graphics.lcs.mit.edu/~seth/proposals/cdp.ps.* MIT Department of Electrical Engineering and Computer Science, 1994.

[Try97] Samuel Trychin. *Interactive Illustration Design.* Brown University Computer Graphics Lab, 1997.

[TRO97] TRO Learning, Inc. "Welcome to TRO's PLATO® Info Center." *http://platobeta.ibsys.com/*. TRO Learning, Inc., 1997.

[WC97] World Wide Web Consortium. "The World Wide Web Consortium." *http://www.w3.org/pub/WWW/*. World Wide Web Consortium, 1997.

[Zac+97] Joe Zachary, et. al. "Hamlet Project." *http://www.cs.utah.edu/~hamlet/*. University of Utah Department of Computer Science, 1997.