A Framework for the Global Cooperative Computing System

by

Amit R. Patel

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1997


The author hereby grants to M.I.T. permission to reproduce and distribute publicly paper and electronic copies of this thesis and to grant others the right to do so.

Author .................................................................
Department of Electrical Engineering and Computer Science

May 23, 1997

Certified by ..............................................................
Thomas F. Knight, Jr.
Senior Research Scientist
Thesis Supervisor

Accepted by ..............................................................
Arthur C. Smith
Chairman, Departmental Committee on Graduate Theses

OCT 29 1997
A Framework for the Global Cooperative Computing System

by

Amit R. Patel

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 1997, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Over the last few years we have seen a dramatic increase in the prevalence of network connected computers. The Global Cooperative Computing project aims to leverage this connectivity to provide an environment which facilitates the cooperative and distributed development and use of software. The Collaborative Application Programming Environment, CAPE, provides the basic framework for this environment. CAPE allows its users to develop Java programs whose software modules’ definitions can be distributed over multiple network connected computers. In addition, CAPE lays down the basic feedback mechanism by which users can communicate thoughts and ideas about each software module. And finally, CAPE scales to a distributed setting where a large number of users can post new versions and access and use existing versions of modules.

Thesis Supervisor: Thomas F. Knight, Jr.
Title: Senior Research Scientist
A Framework for the Global Cooperative Computing System

by

Amit R. Patel

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 1997, in partial fulfillment of the requirements for the degree of Master of Engineering in Electrical Engineering and Computer Science

Abstract

Over the last few years we have seen a dramatic increase in the prevalence of network connected computers. The Global Cooperative Computing project aims to leverage this connectivity to provide an environment which facilitates the cooperative and distributed development and use of software. The Collaborative Application Programming Environment, CAPE, provides the basic framework for this environment. CAPE allows its users to develop Java programs whose software modules’ definitions can be distributed over multiple network connected computers. In addition, CAPE lays down the basic feedback mechanism by which users can communicate thoughts and ideas about each software module. And finally, CAPE scales to a distributed setting where a large number of users can post new versions and access and use existing versions of modules.

Thesis Supervisor: Thomas F. Knight, Jr.
Title: Senior Research Scientist
# Contents

Acknowledgments ................................................................................................................ 4

1 Introduction ................................................................................................................... 7

2 Background ................................................................................................................... 9
2.1 Java ........................................................................................................................ 9
2.2 Web......................................................................................................................... 10
2.3 Version Control ........................................................................................................ 10
   2.3.1 Version Control Issues.................................................................................... 11
   2.3.2 SCCS ................................................................................................................. 11
   2.3.3 RCS, CVS, Distributed RCS, and Distributed CVS ..................................... 12
   2.3.4 Versioning and Programming Granularity Mismatch.............................. 12
   2.3.5 POEM – Programmable Object-centered Environment .......................... 12
2.4 Feedback Information ................................................................................................. 13
2.5 Annotation Systems .................................................................................................... 13
   2.5.1 CoNote .............................................................................................................. 13
   2.5.2 ComMentor ....................................................................................................... 14
2.6 Lotus Notes ................................................................................................................. 14
2.7 File Systems ................................................................................................................. 15

3 The CAPE Java Development Environment .................................................................. 17
3.1 Overview.................................................................................................................... 17
3.2 Description of the CAPE Environment ..................................................................... 18
3.3 An Environment for Java Development .................................................................. 20
3.4 The Document Data Structure ............................................................................... 21
3.5 The Java Document .................................................................................................... 22
   3.5.1 The Definition of a Java Module ..................................................................... 22
   3.5.2 Dependency Graphs ......................................................................................... 24
   3.5.3 Comments ........................................................................................................ 25
   3.5.4 The Version Graph ......................................................................................... 25
3.6 Distributed Database .................................................................................................. 27
3.7 Version Control .......................................................................................................... 27
3.8 The Organization of Java Documents ....................................................................... 28
   3.8.1 Version Documents ......................................................................................... 28
   3.8.2 Group Documents .......................................................................................... 28
3.9 Naming ....................................................................................................................... 28
   3.9.1 CAPE's Naming Scheme ............................................................................... 29
   3.9.2 Naming needs for documents ........................................................................ 30
3.10 How to use Java Documents to build Programs .................................................... 31
3.11 CAPE Features ........................................................................................................ 32
3.12 Extensions to the Environment ............................................................................... 33
   3.12.1 Search Engines ............................................................................................. 34
   3.12.2 Automatic Recompilation ............................................................................ 34
   3.12.3 Naming with Version Commands ............................................................... 34
   3.12.4 An RCS Back-end ....................................................................................... 34
   3.12.5 Feedback Gathering .................................................................................... 35
   3.12.6 Access Control ............................................................................................ 35
4 The CAPE System: Design and Implementation

4.1 Overview

4.2 Design Goals

4.2.1 Simplicity

4.2.2 Availability

4.2.3 Modularity

4.3 Network Architecture

4.3.1 A Client/Server Topology

4.3.2 A Peer-to-Peer Topology

4.4 The CAPE Design and Architecture

4.5 The Database Layer

4.5.1 The Local Database

4.5.2 The Control and Maintenance of the Document Space

4.5.3 Connecting a Distributed Database

4.5.4 Future Advances in the Database Layer

4.6 The Control Layer

4.6.1 Submitting a Java Document to the CAPE Environment

4.6.2 Getting a Document from the CAPE Database

4.6.3 Adding a Comment to a Java Document

4.6.4 Compiling a Java Document

4.6.5 Switching a Java Document’s Dependence

4.6.6 Synchronizing a set of Java Documents

4.6.7 Reading Source Code from a File

4.6.8 Saving Source Code to a File

4.7 The User Interface Layer

4.7.1 Overview of the Architecture

4.7.2 The URL format used in CAPE

5 Evaluation

5.1 Evaluation Strategy

5.2 The Software Program

5.2.1 Overview

5.2.2 The Program Architecture

5.3 Demonstration of a Successful System

5.3.1 Demonstration of the development features of CAPE

5.3.2 Demonstration of a program built from distributed software modules

5.3.3 Demonstration of the communication features of CAPE

6 Conclusions

References
Chapter 1

Introduction

Over the last few years we have seen a dramatic increase in the prevalence of network connected computers. Computers no longer exist in isolation; local area networks, wide area networks, and telephone networks link computers together into a global network called the Internet. This network connectivity allows computers and computer users to collaborate and communicate by sharing information and resources. The World Wide Web (WWW) leverages this connectivity to enable people to share, disseminate, and gather information. The Global Cooperative Computing (GCC) [Eslick] project aims to leverage this connectivity to facilitate the cooperative and distributed development and use of software.

As society’s reliance on computers increases, software projects are becoming larger and more complex. These projects demand a larger number of developers who are not necessarily near each other. A proper framework based on the Internet infrastructure can unite these large teams together. It can also unite the large number of independent software developers and development teams whose computers are connected by the Internet. These developers could benefit from cooperation and collaboration with each other. Also with this connectivity, developers can distribute software modules and programs faster, and thus, decrease the turn-around time for patch releases and bug fixes. Finally, this connectivity allows developers to form a large, well-documented repository of code which can be accessed by a large number of people.

Imagine a system where each software module has a universal identifier which identifies the location of the module anywhere on the network. Imagine using these global names in our source code. Now, instead of only being able to refer to modules which are located on the local file system, we can refer to a module located anywhere on the network. Just as the WWW allows users to share information, a programming environment like this allows developers to share source code easily and efficiently.

Imagine a system which can collect comments, bug reports, performance hints, and other feedback information and store them along side the source code. Future users of the module can benefit from this feedback information. This feedback information can also be used to alert the authors of a bug or security hole. They can then release a new version. Users seeing this new version can then recompile their code. We can even imagine a system which automatically signals for the recompile or sends an email to notify the user of the new software module.
The goal of our research is to design and implement the basic framework for the GCC system. We call this framework system CAPE for Collaborative Application Programming Environment. CAPE provides a system which allows its users to design and write Java programs whose software modules' definitions can be distributed over multiple network connected computers. In addition, CAPE lays down a basic feedback mechanism by which users can communicate ideas and opinions about each software module. And finally, CAPE scales to a distributed setting where a large number of users can post new versions and access and use existing versions of modules.
Chapter 2

Background
This chapter describes some relevant issues and technologies to the CAPE system. Chapters three and four will refer back to these issues and technologies to help explain CAPE's environment and system design.

2.1 Java
Sun Microsystems, Inc. has developed a fairly new object oriented programming language called Java [Java]. Java programs consist of two types of software modules: classes and interfaces. Sun designed Java for portability, its main advantage over traditional languages like C and C++. The Java compiler compiles Java source code to Java byte-code. A computing platform's Java interpreter then interprets this byte-code. Since Sun Microsystems provides a Java interpreter for many of the major computing platforms, the Java byte-code is very portable.

Java pays for this portability through a performance hit. There are several ways to increase this performance. Instead of compiling to portable Java byte-code, a compiler can compile directly to machine code. This machine specific code will perform better, but the code will not be portable. A Just-in-time (JIT) compiler, on the other hand, retains the portability of Java byte-code and increases program performance. A JIT compiler replaces the Java interpreter. As the JIT compiler interprets the Java byte-code, it saves the machine code it generates. If the program comes to a section of code which has already been interpreted once, the JIT compiler simply uses the already generated machine code, thus saving time by not interpreting that section of code again.

The latest version of Java Development Kit (JDK), version 1.1, supports Remote Method Invocation (RMI). With RMI, developers can build systems in which the objects in one Virtual Machine can call on the remote methods of remote Java objects in another Virtual Machine. To enable this type of communication, the Java object which wants to allow access to its methods via RMI needs to implement a Java remote interface.

JDK 1.1 also adds Object Serialization to its feature set. Using Object Serialization, developers can encode a Java object as a stream of bytes and can retrieve the object from that stream. This extremely useful feature facilitates the transportation and storage of objects. For example, a developer, who wants to store an object as a file, can convert the object to a stream of bytes and
write those bytes to a file. When the object is needed, he can produce that object from the bytes in the file.

Sun distributes Java with a wide ranging set of useful classes. These classes are called the core Java classes. They are available on most Java run-time environments, and so, they provide programmers with a useful base from which to start developing their programs and eliminate duplication in effort to build commonly used and desired classes.

Most web browsers can also interpret Java byte-code. Small Java programs called applets, can be embedded into web pages. This allows web page developers to significantly increase the functionality of their web pages. A browser downloads the byte-code for an applet from the web server and then interprets the byte-code directly on the browser’s machine. As we shall see in the next section, the web is becoming increasingly popular, and Java’s ability to increase web page functionality offers a tremendous advantage.

Because applet byte-code is being downloaded and run on client machines, security becomes an important concern. The developers of Java have addressed this issue. Applet code can only be run in environments with a security manager. This security manager controls the applet’s ability to access system resources. The byte-code must also pass through a byte-code verifier. The verifier checks to ensure that the bytes which claim to be Java code, really are Java code. Once the code has been verified as Java, the security manager limits its action during run-time.

2.2 Web
The World Wide Web has become an important and popular tool for the publication and distribution of information. It has proven to be a very scaleable system which allows a distributed set of documents to be integrated and linked together. The WWW completely eliminates the concept of distance. Information from many, possibly distant, sources can be linked together into a single WWW document. This creates a medium where there is less need for duplication.

2.3 Version Control
Designers of a cooperative software development system must deal with the issue of version control. The version control system handles the storage of versions of software modules and the policies for creating, changing, sharing, relating, and building these versions.
2.3.1 Version Control Issues

There are three major issues which must be dealt with. First, in any development environment, whether it be cooperative or not, developers are constantly creating and modifying modules. The version control tool manages the evolution of these modules. The version control system must provide two features to the developers: backtracking and stability. By backtracking we mean that the developers should be able to go back to previous versions of modules, so they can correct mistakes or redirect their efforts. By stability we mean that the system needs to provide a point at which a module can be frozen and broken off from the development track. This will allow others to continue to use the stable module while developers go onto make changes and additions.

In a cooperative environment, a version control system also needs to control access. In a setting where there are multiple developers, we need a mechanism to manage who has access to particular modules, how multiple people can interact with these modules, and how all these people can agree on the evolutionary path of these modules.

Finally, a version control system in a cooperative environment needs to scale well. In a cooperative system there are many developers who want to work on the same project. As the size of the project and the development team increases, the number of requests on the version control system will also increase. A system that does not scale well will not perform well.

2.3.2 SCCS

Over the years, the power and sophistication of version control tools have increased. One of the earliest version control systems is the Source Code Control System (SCCS) \[\text{Roc}\]. SCCS deals with the issue of backtracking and stability by storing all the changes made to a file. It uses the following model. It stores the original file, and then every time the file is changed, the system records a “delta” for that file. The “delta” describes the changes that have been made relative to the previous version. To reconstruct a version, we start at the beginning and “apply” successive “deltas” until we get to the version we want. This model provides both stability and the ability to backtrack. SCCS provides only a simple access control model with the use of locks and reliance on the underlying operating system. Only the person who holds the lock on a file can add a “delta” to the file. This forces the evolutionary path to be linear because only one person can “check-out” the file at a time. For the same reason, the locks also hinder the scalability of the system. As the number of users grows, the contention for these locks will increase and the system will not perform.
2.3.3 RCS, CVS, Distributed RCS, and Distributed CVS

The Revision Control System (RCS) [Tic] extends the SCCS model by allowing branches in the evolutionary path of a file. It can also merge these branches back together. This helps the scaleability problem slightly because multiple people can check-out a file if they are saving their changes in different branches. One of the major drawbacks of both SCCS and RCS is the limitation that they can only version individual files. Concurrent Version System (CVS) [Ber] addresses this issue by allowing directories to be versioned. This allows developers to group and version a set of files. Distributed RCS [ODG] and Distributed CVS [HK] attempt to alleviate some of the scaling problem by extending the RCS and CVS models respectively into the client-server environment. They both have a central server which can service local client requests. These two systems will probably work well in a local area network (LAN), but due to its single central repository, they won’t scale well in a more distributed environment [VHW].

2.3.4 Versioning and Programming Granularity Mismatch

A flaw that all of these version control systems suffer from is that they are not versioning software modules but instead the text in files. This is clearly not what we want. The functionality of a module is defined not only by its text, but by its dependencies as well. Therefore, a change to a sub-module could change the functionality of the module. In all the systems that we have described so far, the versioning information does not control for these types of changes.

2.3.5 POEM – Programmable Object-centered EnvironMent

Researchers Lin and Reiss addressed this problem in their paper “Configuration Management with Logical Structures.” [LR] They note that developers program using entities like functions, classes, and procedures, but that version control systems version based on entities like files, directories, and makefiles. They propose that the two processes, software design and version control, should both be based on programming entities. They have developed an environment for C/C++ development called POEM (Programmable Object-centered EnvironMent) which does just this. Unlike previous systems, POEM creates and stores version information at the procedure and class level. Versioning at this level provides a more natural fit to the problem of version control. In reality a programmer does not want to import a version of a file instead he wants to import specific versions of functions and classes. POEM’s versioning scheme also provides better stability compared to earlier systems. POEM keeps track of each module’s dependencies. If the user wants to capture the functionality of a particular version of a module, he can take a “snapshot” of it. This will tell POEM to save the current module and all its dependencies. One drawback in this scheme
is that the snapshot must be explicitly initiated by the user. An advantage that POEM provides is automatic building. Since POEM keeps track of all of the dependencies of a module, there is no need for a makefile. POEM can traverse the dependency tree and perform the build on its own. A limitation that POEM imposes on software developers is that it only allows one version of a module to be used in a workspace. This means that different parts of a project cannot use different versions of the same module.

2.4 Feedback Information
Collecting feedback information about a module and how it works is important so that we can improve that module, make better use of it in other modules, and design future modules better. We can classify feedback into one of two categories. Comments and information from developers and users fall into the first category. And, information from the run-time environment or system comprise the second category.

Users often find that they have a comment, complaint, or suggestion that they would like to convey to the author or to the user base of a software package or program. This type of feedback can be very useful. It can help identify bugs, security holes, and performance hits. Relaying this information to the authors or users quickly, decreases the turn around time that is necessary to take the appropriate action. Also, by making this information available to a large number of users, we can help people learn from the mistakes and successes of others.

Information taken directly from the application can also help in the evaluation of a module. Examples of this type of information are usage information and profiling information. Usage information can be compiled to tell us interesting statistics on how the program is being used, when it is being used, and why it is being used. Profiling information can help us capture run-time statistics on our program so that we may try to eliminate any performance bottlenecks.

2.5 Annotation Systems
An annotation system is a system which allows users to add information to existing data without disrupting that data.

2.5.1 CoNote
CoNote [Dav], an annotation system developed at Cornell University, allows users to engage in a discussion on a set of web documents by adding comment annotations to them. These annotations can be thought of as Post-it® notes. Users can make comments about the document and then attach them to the document. The only restriction is that the comments can only be made at certain places
on the document. This restriction limits the freedom of the users, but it also forces their attention to specific areas of the document which the author feels is worthy of discussion. A neat feature of this system is that the annotations can be organized into conversations.

2.5.2 ComMentor

ComMentor [RM] is another annotation system which was developed at Stanford University. It is similar to CoNote but has a slightly different focus. Whereas CoNote is trying to provide an arena where users can collaborate and discuss a set of documents, ComMentor is trying to provide a framework which will allow third-party sources to provide useful information to the users of the document. An example would be a rating agency which would add a web page rating annotation to web pages which people could use to restrict access.

2.6 Lotus Notes

Lotus Development Corporation developed the Lotus Notes® [Lotus] technology because they saw the need for a tool which allowed the people in an organization to communicate with each other, to collaborate together, and to coordinate their various activities. Lotus examined the way businesses and business processes worked. They recognized that a central resource in most businesses was the company’s data which was most likely stored in a computer database. They also recognized that the company stored this data not only for record keeping, but also as an aid in business decision making. They recognized that many people might have an interest in one piece of data. Some of them might need to send messages about this piece of data to each other. Some might also need to work together to act on this information. And finally, some of these people might need to coordinate their different responsibilities and functions with each other so that the business as a whole performs smoothly. Lotus developed Notes to help data users perform these functions.

Lotus Notes is primarily a document database. That is, all information stored in the Notes environment is in the form of a data structure called a document. Documents can accommodate any type and amount of data which makes them a very general data structure. This generality allows Notes to be used in all areas of a business, and therefore, to form a central point of access for all of the company’s information.

Because Lotus designed Notes to act as an agent to facilitate the communication, collaboration, and coordination of many people, they knew that the Notes environment would have to be distributed in terms of people and resources. They, therefore, have built-in to Notes a powerful and configurable version control system, an efficient and reliable replication mechanism, and flexible
and rigorous security scheme. The version control system allows multiple people to access and modify a document without having to worry about who else is accessing and modifying that document. All changes can be versioned. Note's version control system is also very configurable. Thus, allowing developers and administrators to customize the version control system for specific needs. Recognizing that the users of the Notes environment might be spread geographically, Notes enables the servers in these various geographies to share documents by an efficient and reliable replication scheme. Users on one Notes server can get documents belonging to another Notes server. A very powerful synchronization mechanism allows changes made to the data to be propagated to the home server and to the other remote servers using that data. Finally, though Notes aims to facilitate sharing of data between users, it also needs to restrict access to data. Not all users have the need or the right to view or modify certain information. Lotus has taken security very seriously in Notes. They implement security at four levels: authentication of users, access control lists on documents and document fields, encryption of document fields, and digital signatures for verification.

Besides providing this powerful distributed database environment, Notes goes a step further. It provides mechanisms for communication. Often, while working on a set of data, a user will find the need to send a message to an individual or a group. Notes allows its users to do this easily. They can send email messages to these people. These email messages can contain links to documents stored in the Notes database. The recipient of the email message can read the message and then view the corresponding data by simply clicking on the link. Notes can also be configured to send emails automatically when triggered by an event. For example, Notes can automatically send email to a manager when the stock of a certain good in a warehouse falls below a certain quantity threshold.

Finally, Notes allows application developers to use these Notes features when designing applications. The application can worry about its application specific details while leaving the details of data handling and coordination to Notes.

2.7 File Systems

File systems are databases that allow computers to store data in data structures called files. Most computers rely heavily on a file system. They use the file system to store the operating system and program files that they use and also to store user and system data. The file systems of most of the popular computing platforms are designed to serve a single host or a small collection of hosts. They can not scale to a larger environments. First, they are not designed to handle the demand
imposed by a large environment. Second, they do not enforce the necessary security measures needed in larger, more distributed environments. And finally, they don't provide for any version control which could be important in an environment where many people are working on the same set of data.
Chapter 3

The CAPE Java Development Environment

3.1 Overview

Software modules are the building blocks for software programs, which are themselves software modules. Depending on the programming language, the types of these software modules can vary from functions and procedures to classes and interfaces. These software modules are built using other software modules and a language’s primitives. The language specifies how these modules and primitives can be put together. The software modules of a program and the way they are assembled define the program.

In traditional programming environments, the definition, or the source code, of a software module is stored as a text file on the host computer’s file system. The development environment is built directly on top of this file system. This introduces very significant limitations in terms of the accessibility of software modules, the versioning of these software modules, and the collection and distribution of feedback information concerning these software modules.

First, the nature of the local file system limits access to and use of its files, and therefore, the software modules. Most file systems are primarily designed to serve either a local host or a set of hosts in a local area network. Since most file systems are not designed to perform in a global or a very distributed setting, it is difficult for the users of these file systems to make their files and, in effect, their software modules accessible to others that are not on the local network.

Second, most file systems in use today do not provide any versioning mechanism and neither do most readily available development environments. Therefore, when necessary or desired, the users of these environments must install a separate version control package. This package acts as an agent between the file system, where versions of software modules are stored, and the user who requests, possibly through the development environment, specific versions of the software modules. These separate version control systems not only inconvenience the user due to their manual requisition process, but they also impose restrictions on the user due to their poor scaling and due to the mismatch between the programming level and the versioning level.

Finally, most development environments in use today do not provide a mechanism by which feedback information about a software module can be easily collected and stored with that software module without disrupting the source code of that module. At best, additional non-
programming information can be stored in the form of comments in the source code. But this is inconvenient and simply a hack that gets around the problem without addressing it.

Though systems exist which try to address the problems in each of these areas, we know of no system which addresses them all. CAPE tries to address these issues in a straightforward and simple manner. CAPE’s goal is to establish a system which allows software developers to cooperate and collaborate on software development by providing a framework which allows them to share software modules and information about these software modules.

### 3.2 Description of the CAPE Environment

Under the CAPE environment, software developers, working on any distributed set of network connected computers, can collaborate and cooperate on software development. CAPE enables these developers to very easily share their software modules and to communicate their thoughts and ideas about specific software modules. In traditional environments, developers manipulate and use software modules stored locally on machines and databases within their control. But in a setting where a development team is split geographically or in a setting, such as the Internet, where many independent and highly distributed programmers might want to cooperate on software development projects, developers can benefit from an environment which facilitates the use of distributed software modules. Using CAPE, developers can develop and build Java programs whose software modules are possibly distributed over multiple network connected computers. Developers in a collaborative software project also need to communicate with each other. Under the CAPE environment, developers can attach comments to the software modules. These comments can be viewed by others, and so, they form a communication channel between users.

The CAPE database is a distributed database of documents. Documents are a very flexible data structure, and they come in a variety of types. Each CAPE host has its own local database which is built on top of its local file system. Most file systems are not designed to operate in a highly distributed and possibly global network of hosts. They, therefore, impose a serious limitation in terms of data accessibility. For this reason, CAPE implements its sharing mechanism in the database layer, a layer above the file system, and it uses the file system for storage only (see Figure 3.1). The separation of sharing and storage allows CAPE to implement a very scaleable protocol for sharing. This protocol connects the local databases together to form an accessible and scaleable distributed repository of documents.
In CAPE, data structures called Java Documents hold the source code for Java modules. Each Java Document in the CAPE database contains at most one definition of a public Java module. A Java Document’s source code can use and refer to any Java module which is stored in an accessible Java Document in the CAPE distributed database. So, developers can easily use remote software modules. Java Documents are like the source code files used in the traditional Java development environment. At compile time in the traditional Java environment, the necessary files are retrieved from the file system and the source code in these files is compiled. At compile time in CAPE, the necessary Java Documents are retrieved from the CAPE database and their source code is compiled.

A Java Document can have multiple versions. When we want to use or refer to the software module contained in that Java Document, we need to specify the version we want. When writing source code, however, we use a generic name which does not specify a version. It is the responsibility of the dependency list attached to a Java Document to identify the exact version to use for each generic reference in the source code. By keeping the version information separate from the source code, the system is more easily able to change and manipulate the dependency graph of a Java Document. Whereas in the traditional Java environment, the definition of a Java module is contained solely in the source code, this is not the case in CAPE. The source code in a Java Document does provide most of the definition, but a list of dependencies is still needed. The source code plus the list of dependencies together define the Java module contained in a Java Document.

Because CAPE has to deal with versions of Java Documents, it needs to provide a version control mechanism to control access to these Java Documents. Once a user enters a Java Document into the database, CAPE freezes the definition of its Java module. By not allowing the definition of a
Java module to change, we ensure both stability and the ability to backtrack. Static definitions also eliminate the need for locking which allows the system to scale better. Finally, in CAPE versioning and programming are performed at the same level. Programming in Java is done at the class and interface level, and versioning in CAPE is done at the Java Document level which is the same as the class and interface level.

Java Documents in CAPE are more flexible than the files in a traditional development environment. Besides source code, a Java Document can also hold other information such as comments. Users can add comments to a Java Document without disrupting the source code contained in that Java Document. These comments provide a mechanism by which users can communicate thoughts and ideas about the Java Document. CAPE bundles this information which is relevant to a Java Document’s source code with the source code in a non-intrusive manner. Examples of other information that can be stored in a Java Document include: the Java Document’s dependencies, the Java Document’s parent versions, the Java Document’s child versions, and the Java Document’s creation date.

3.3 An Environment for Java Development

In designing a programming environment, one of the first issues that needs to be addressed is the choice of supported languages. In order to maintain flexibility, CAPE does not want to limit the range of supported languages. But implementing a system that handles a large number of languages is very complicated and time consuming because the differences in language syntax and semantics vary greatly across the range of languages. This additional complexity is not justified by our project goal. For this reason, we have decided to design the system with one language in mind while at the same time keeping in mind that, in the future, we might want to extend support to other languages.

We choose the Java language for several reasons. The most important reason is Java’s portability. Since Java is an interpreted language, its byte-code can be run unmodified on any supported platform. This makes the language very attractive for a large, globally accessible database. It means that developers for many platforms will be able to use the software modules in the CAPE system. We also like Java’s language design. In the Java language, there are only two types of software modules: classes and interfaces. Also, by the language’s own rules, only one public class or interface can be declared in any compilation unit. In the traditional Java environment, a compilation unit is stored as a file. In CAPE’s environment, the compilation unit is stored in an analogous data structure, called a Java Document. In both environments, the level of storage, the
file or the Java Document, provides a nice level of granularity for versioning. That is, the level of granularity for versioning, the file or the Java Document, is the same as the level of granularity used by the programmer, the class or interface. Finally, Java was chosen due to its current popularity. Its popularity will hopefully attract a lot people to try and use CAPE.

3.4 The Document Data Structure

Since one of our goals is to create a large repository of software modules, it becomes obvious that CAPE must define a data structure that it and its users can use to store software definitions and other CAPE related information. We call these data structures documents. Fundamentally, they are a collection of another data type called annotations (see Figure 3.2). All data in a document is stored in annotations. Annotations come in different types. Each annotation type can store differing amounts and types of data. Since documents are simply a collection of annotations and since different annotation types can be created to hold any and all sorts of data, documents are very flexible in the types and amounts of data they can contain. Different types of documents have different restrictions on the types and number of annotations they can hold. CAPE supports a number of different types of document, but they must all have the basic features defined in the basic document type, the Document.

The Document allocates space for four annotations: a Document Name Annotation, a Parent Name Annotation, a Document Submitted Date Annotation, and a Document Modified Date Annotation (see Figure 3.3). All except the Parent Name Annotation are required. The Document Name Annotation identifies the name of this document. The Parent Name Annotation identifies the name of the document which is the parent of this document in CAPE's document space. The Document Submitted Date Annotation identifies the date and time that this document was submitted to

![Figure 3.2: A Schematic for a basic document](image-url)
CAPE’s database. And, finally, the Document Modified Date Annotation identifies the date and time that this document was last modified. All document types must support these annotations. Once a document is submitted to CAPE’s database, its Document Name Annotation, its Parent Name Annotation and its Document Submitted Date Annotation are locked. The Document Modified Date Annotation is kept up-to-date as the document is modified.

![Figure 3.3: A Schematic for a Document](image)

### 3.5 The Java Document

A Java Document is a specialized type of document which holds the definition of a Java module. In addition to the annotations described by the basic document structure, the Document, a Java Document can hold a Source Code Annotation, a set of Comment Annotations, a set of Parent Version Name Annotations, a set of Child Version Name Annotations, and a set of Dependency Name Annotations (see Figure 3.4). The CAPE system versions Java Documents and, consequently, the Java modules that they define. The definition of a Java module in a Java Document is, as we shall see in section 3.5.1, specified by the Source Code Annotation and the set of Dependency Name Annotations. When a Java Document is versioned these annotations, which define the Java module are frozen, but the sets containing the other Java Document annotations can still be modified. Versions of a Java Document share a generic name. A generic name is simply a Java Document's name without the version number. The version number is used to differentiate between versions of a Java Document. Naming is described in more detail in section 3.9

#### 3.5.1 The Definition of a Java Module

A Java Document holds its source code in a Source Code Annotation. Unlike, the traditional Java environment, the source code does not completely define the software module represented by a Java Document. References in the source code to classes and interfaces stored in the CAPE system are made using generic names, names which do not specify the exact Java Document version that they are referencing. To completely define a Java module, we need to specify which version of a
Java Document to use for each generic reference. A Dependency Name Annotation serves this purpose. Each different generic reference to a class or interface in the source code should have a Dependency Name Annotation identifying the exact Java Document version to use for that reference. If a complete dependency listing is not specified by the Dependency Name Annotations, that is if every different generic reference does not have a Dependency Name Annotation, then the Java Document is not completely defined and it can not be compiled. Figure 3.5 shows how the references in the source code and the Dependency Name Annotations work together to completely define a Java module.

The Java Document replaces the file of the traditional Java environment as the basic storage unit for source code. The source code in a Java Document must, therefore, obey the same restrictions that are enforced when source code is stored in files in the traditional Java development environment. For example, only one public class or interface can be defined in a Java Document. Also, similar to how the traditional environment requires that the simple name of the public module defined in a file must match the filename of that file, CAPE requires that the Java
Figure 3.5: Source Code and Dependency Annotation Usage

Document's name be related to the fully qualified name of its public module (see Figure 3.9 and Section 3.9.2). This allows the system to find the appropriate Java Document when its software module is referenced.

3.5.2 Dependency Graphs

When a Java Document, through its source code and Dependency Name Annotations, uses or refers to a Java module stored in an other Java Document, it is said to be dependent on that Java Document. A dependency graph shows the dependency relations of a Java Document. Each node in the dependency graph represents a Java Document. A directed edge from one Java Document to another shows that that Java Document is dependent on the other. For a complete dependency graph every Java Document in the connected graph must have a complete dependency listing.
Figure 3.6: Dependency Graph Example

Figure 3.6 shows the dependency graph of a Java Document called A. In this figure, Java Documents are represented by box with their names written inside the box and a dependency is depicted by an arrow. The Java Document at the base of the arrow is dependent on the Java Document at the point of the arrow. In this example, A and E are dependent on each other. A also depends on B and C. E depends on C and D. And finally, B depends on D. If we wanted to compile A, we would need to create and traverse this graph to assemble all of these Java Documents.

3.5.3 Comments

Comment Annotations provide a neat way to bundle user feedback information with the appropriate source code without disturbing it. Comment Annotations consist of user supplied strings. There is no limit on the number of Comment Annotations that can be added to a Java Document. Comment Annotations can be added to a Java Document, but they can not be deleted or modified. Users of a Java Document can use these comments to communicate thoughts and ideas about the Java Document.

3.5.4 The Version Graph

Parent Version Name Annotations allow the system and user to identify the names of Java Documents from which this Java Document is derived. A Java Document can have more than one parent version. More than one parent version shows the merging of two or more evolutionary lines in the Java Document's version graph. Parent versions of a Java Document are not limited to the versions of the Java Document. A Java Document can list any Java Document as a parent version.
The set of Parent Version Name Annotations is frozen when the Java Document is entered into the database.

Child Version Name Annotations are the opposite of Parent Version Name Annotations. They identify Java Documents which are derived from this Java Document. A Java Document can have more than one Child Version Name Annotation. More than one Child Version Name Annotation shows the splitting of a line of work into many different development tracks. Like parent versions, child versions of a Java Document are not limited to versions of the Java Document. A Java Document can list any Java Document as a child version. The set of Child Version Name Annotations can be augmented throughout a Java Document's life.

Parent Version Name Annotations and Child Version Name Annotations help us create a version graph for a Java Document. A version graph shows how a set of Java Documents has evolved. Since parent and child versions of a Java Document are not limited to the versions of that Java Document, the nodes in a version graph are not necessarily all versions of a single Java Document. They are, however, all related in their evolution. Also, the version graph produced by a Java Document does not necessarily include all the versions of that Java Document.

![Figure 3.7: Example of a Version Graph](image-url)

Figure 3.7 shows an example of a version graph for a set of Java Documents. Java Documents are depicted by the circles. The name of each Java Document is written inside each the circle. Parent
and Child Version Name Annotations are depicted by the dotted and solid arrows respectively. The base of each arrow is at the Java Document which holds that Annotation, while the point of each arrow is at the Java Document which is either the parent version or child version of that Java Document. JD1 is the original version in this version graph. It has no parent versions, and therefore, no Parent Version Name Annotations (no dotted arrows). JD2 and JD3 are both Java Documents which are derived from JD1. They represent independent work and evolution based on a common heritage, namely JD1. JD4 is derived from both JD2 and JD3. It is the merging of two independent lines of work. JD5 is derived directly from JD3. Finally, JD6 is also a merging of different lines of work. In this case, the evolutionary lines are not completely independent, but they are distinct.

3.6 Distributed Database

A document belongs to a particular CAPE host. The CAPE host is responsible for the storage, update, maintenance, and retrieval of its documents. The host relies on its file system for storage only. Sharing and access control are implemented in a layer above the file system (see Figure 3.1). This allows CAPE to establish its own protocol for sharing. The protocol is designed to scale better than most file systems which allows CAPE hosts to form an accessible and distributed repository of software modules.

The distributed nature of the CAPE system eliminates a central authority. Anyone on a network can setup a CAPE host. Documents on this host are now available to other CAPE users. The beauty in this is that no central authority needs to approve the establishment of a new host. The CAPE system is similar to the WWW, where web servers can come and go without any central administrative control. Because of this distribution, we hope that the CAPE system will be as scaleable as the web has proven to be. Hosts communicate only to pass documents. Most computation and work is done on a local CAPE host. The control and access to a local host can be monitored by a local authority that can manage the load of a host.

3.7 Version Control

Let us examine the version control strategy used by CAPE. Versioning is done on the Java Document level. And since a Java Document can contain only one Java module definition, CAPE, like POEM, versions at the programming entity level too. Since submission to the database freezes the definition of the Java module, backtracking and stability are easily obtained.
Our initial system will have no access control. Anyone will be able to add new Java Documents to the system and anyone will be able to add comments to existing Java Documents. Though this is an area that needs to be dealt with, it is not critical to building an initial framework.

CAPE is a very scaleable system even though the there is a home repository for each Java Document. First, since the Java module definition of a submitted Java Document is static there is no reason to lock it. Therefore, numerous developers can be working simultaneously on child versions of the same document in different branches of the version graph.

3.8 The Organization of Java Documents

3.8.1 Version Documents
The CAPE system versions all Java Documents. Versions of a Java Document share a generic name (Generic names will be described in Section 3.9). These versions are grouped together by Version Documents. A Version Document, in addition to the standard annotations, contains a set of Version Name Annotations. A Version Name Annotation points to a Java Document that belongs to this Version Document. Each Java Document belongs to a Version Document. A Version Document also maintains the next available version number for the group of Java Documents. When a new Java Document is created that belongs to this set of Java Document versions, the Version Document supplies a new version number for it to use. See Section 3.9.2 and Figure 3.10.

3.8.2 Group Documents
A Group Document groups together related Version Documents and other Group Documents. A Group Document is similar to a package in the traditional Java environment. It groups together documents which are similar in purpose or function. Each CAPE host has a Group Document which is the root of the document space. All other documents in this local database are placed under this node in the document tree. See Section 3.9.2 and Figure 3.10. It uses Sub-Version Document Name Annotations and Sub-Group Document Name Annotations to specify the names of Version Documents and Group Documents that are directly under it in the document space.

3.9 Naming
CAPE needs a naming scheme to identify documents in the CAPE database. The documents are, of course, the three document types that we have identified: the Java Document, the Version Document, and the Group Document. The naming scheme needs to be flexible enough to handle the identification of all these documents.
The naming scheme also needs to comply with the Java naming scheme. The Java compiler has a specified naming scheme. When we design a program using Java Documents, we need to identify these Java Documents as we use them in the source code. Unless the Java Document identifiers fit the Java naming scheme, the compiler will not be able to parse the source code. The names of Java Documents must therefore be compliant with the traditional Java naming scheme.

Finally, names must be human-readable. That is, they must be represented by a string which humans can use, remember, and manipulate.

3.9.1 CAPE's Naming Scheme

There are three pieces of information that a name must be able to specify. First, there is the host name. Documents in the CAPE system belong to a host. In order to identify a document, we must specify the host name of the document. CAPE also allows Java Documents to be versioned. In order to identify a Java Document, a name must be able to convey a version number. Finally, the name must carry a local name which is used to identify a document once the host name has been established.

A name which specifies a host name is called a full name; a name which does not specify a host name is called a local name. A name which does not specify a version number is called a generic name, while those that do are called non-generic names.

Since the name may be used in source code, all names follow the traditional Java naming scheme. In this scheme, names are a sequence of alpha-numeric characters separated into parts by periods. Each part must begin with a letter, never a number.

Since all CAPE names fit the Java naming scheme, the system needs a way to differentiate between identifiers which refer to documents and identifiers which refer to a Java module found in the standard Java distribution. All names which point to CAPE documents must have a part that looks like this: “GCC”. This is a reserved part.

The name must be able to convey a host name. Since all document names must have a “GCC” part, we use this as a marker for the end of the host name parts. All the parts preceding the “GCC” part are used to specify the host. How do we identify a host? We could use its IP address which is broken into parts by periods too. The problem is that the parts which make it up consist of numbers only. This is bad because the first character in each part must be a letter. The host name is the other alternative. A Domain Name Server (DNS) maps host names to IP addresses. The host name is also demarcated by periods. The issue is that the DNS does not enforce the same restrictions on its
parts as Java does. It allows characters other than letters and numbers, and it does not demand that the first character in each part begin with a letter. Though it does not enforce the proper restrictions, most host names comply to the Java naming scheme anyway. And, for those that don’t, an additional entry in its DNS server can be added. This new entry can be Java compliant and be mapped to the host’s IP address.

The parts after the “GCC” part form the local name. They are used to locate a document once the host has been identified.

The version number is added onto the last part of the local name. The sequence of numbers forming the version number are separated from the characters comprising the last part by the character “v”. This imposes an additional restriction on naming. If the characters following the last “v” in the last part of a name are numbers, then those numbers represent a version number. The characters before the “v” form the last part of the name, and the character “v” is ignored. “v” is only a marker.

Figure 3.8 shows some examples of names.

3.9.2 Naming needs for documents

All documents have full names. Group and Version Documents have generic names. Java Documents have non-generic names. The full generic name of a Java Document must match the
fully qualified name of the Java module it holds (see Figure 3.9). The Java Documents belonging

to a Version Document all have the same generic name which match the name of the Version
Document. Each host has a Group Document which is the root of the host’s document space. Its
name is of the following form: “hostname.GCC”. Except for this root Group Document, all
Version Documents and a Group Documents have parent Group Documents. The names of all
Version and Group Documents, except for the root Group Document, is simply the name of the
parent Group Document extended by one “part” in the local name. An example of a first level
Group Document name is “hostname.GCC.onePart”. Figure 3.10 shows an example of the CAPE
document space and their naming.

3.10 How to use Java Documents to build Programs
It is very easy to use Java Documents to write programs. Almost nothing changes from the
traditional Java programming environment. Each Java Document contains exactly one public class
or interface. If we want to refer to this class or interface in another Java Document’s source code,
we simply treat the name of this class (or the generic name of the Java Document) as the fully
qualified name of the class (which it is). The developer specifies exactly which version of the Java
Figure 3.10: picture of document space and names

Document to use by setting the Dependency Name Annotations of the Java Document (see Figure 3.5). A Dependency Name Annotation holds a full, non-generic name which tells the system which Java Document to use for a particular generic name found in the source code. By keeping the version information outside the code, it is easier for a user or the system to change a Java Document's dependencies. The system does not need to parse the code to find all occurrences of the reference and then change them all to the new name. It is a simple matter of changing the dependency annotations. No parsing is required.

3.11 CAPE Features

How can a user interact with the system? What features are available for him to use? A user:

- can submit a new Java Document to the system, with source code, names of its parent version Java Documents, and the names of the Java Documents on which this new Java Document depends. The names of the parent versions and the dependencies need
to be valid CAPE names which refer to existing Java Documents, otherwise an error is sent back to the user. If the source code is parseable, CAPE first checks to see if the source code that is provided is valid for a Java Document. Valid source code describes exactly one Java module, class or interface, and has a fully qualified name which is full and generic. If the source code is valid, then the fully qualified name is used as the full, generic name of this new Java Document. A new version number is obtained to form a full, non-generic name. If the source code is not valid, then CAPE tries to use the full, generic name supplied by the user to form a full, non-generic name. If this name is not a valid Java Document name, then CAPE assigns the Java Document a default name.

• can view a document: Java Document, Group Document, or Version Document. Each is displayed differently. They each have links to other documents to allow for navigation within the document space and to other information which is not displayed in this view.
• can attach a comment to a Java Document.
• view the comments attached to a Java Document.
• can view the dependencies that are attached to a Java Document.
• can view the parent versions of a Java Document.
• can view the child versions of a Java Document.
• can call up an editor which allows us to submit a new Java Document possibly based on existing one.
• can read source code into the editor from a file on the CAPE host.
• can save the source code in the editor to a file on the CAPE host.
• can create a new Java Document based on an existing one, such that, CAPE replaces each occurrence of a particular Java Document in that Java Document’s dependency graph with another. This function might create a new versions for some or all of the Java Documents in the dependency graph.
• can synchronize a set of Java Documents. This means that if any of the Java Documents in the set depends on a version of any of the other Java Document in the set, CAPE changes this dependency to point to the version of that Java Document which is in the set. This function might create a whole new set of Java Documents.

3.12 Extensions to the Environment
The current system is a framework for future work. Some possible extensions to the CAPE environment are outlined in this section.
3.12.1 Search Engines

One possible extension we can make to the environment is to add a search engine that can help users find useful Java Documents. In the current system, programmers have to know of the existence of and the name of the Java Document that they want to use. With search engines users can find Java Documents that they didn’t know existed. Java Documents could have a Key Word Annotations that describe it. Users could use a search engine to search for Java Documents with the desired attributes. The engine could also use other information to aid its search such as author’s name, submission date, and parent versions. The engine would help make a possibly huge distributed database of Java Documents more manageable.

Search engines are research areas in themselves. An example of a search engine that searches through distributed space is Digital Equipment Corporation’s Alta Vista Engine. This and other web search engines are similar in nature to the type of engine that would be needed in the CAPE environment.

3.12.2 Automatic Recompilation

Imagine a system which would recompile your software automatically when triggered by an event. For example, let say that you were using a program that was developed using the CAPE system and that the program depended on a specific Java Document which turned out to have a bug in it. When that bug is fixed it would be nice if CAPE could automatically detect the presence of this new bug fixed version and signal a recompile to all the users of the original Java Document. This would save developers from continuously checking for updates and manually initiating a recompile.

3.12.3 Naming with Version Commands

In the current version of CAPE, Java Documents are named using specific version numbers. Imagine a system where instead of a number specifying the Java Document, we use a command. An example of a command of this form would be one which tells the CAPE to always use the most recently submitted version of a Java Document. Another example would be a command which says to use the most recent version that is a descendent of a particular Java Document. Yet another example would be a command which tells CAPE to find the most recent version at submission time and to set the dependency list to use that version.

3.12.4 An RCS Back-end

Currently in CAPE, versions are merged manually. We could add an RCS back-end to CAPE which could merge the source code of two Java Documents automatically. RCS and other version
control systems merge source code by comparing the texts. If the changes are made in different areas of the text, then RCS automatically commits the changes. If the changes are made near each other, then RCS asks the user to resolve the ambiguity. This could be a helpful feature to future users.

3.12.5 Feedback Gathering
In this version of CAPE, the only type of feedback that we gather is comments. We could extend this by collecting other feedback information such as usage statistics, bug reports, and performance statistics.

3.12.6 Access Control
Access control could also be added to the system. Currently, any user from anywhere can access the Java Documents on a CAPE host. We could implement access control on a user level which would restrict access to a host or particular documents based on the identity of the user. We could also restrict access at the host level. In this scheme, each CAPE host would essentially have a list of trusted hosts whose users could access the documents in its database. We could also use a combination of these two access control schemes.

3.12.7 HyperJava
We could also extend the system by displaying the Java code in a Java Document in a hypertext format [Brown]. CAPE would markup the source code so that we could navigate through it very easily using a web browser. We could for instance click the identifier specifying the type of a variable and be taken to that type's definition. Or we could click on a variable and be taken to where the variable is declared. We could also click on the name of a method that we are using and be taken to that method's definition. These are just some of the ideas that could be implemented in HyperJava.
Chapter 4

The CAPE System: Design and Implementation

4.1 Overview

This chapter describes the design and implementation of the CAPE system. The CAPE system implements the CAPE Java development environment described in Chapter 3. There are three major parts that implement the CAPE environment. The first is a host application that we have written which provides most of the CAPE features and functionality. The second is a web browser which provides the user interface to CAPE. And finally, there is a network which connects the CAPE hosts and the web browsers. Figure 4.1 shows how these three pieces fit together.

![Figure 4.1: General Architecture of the CAPE system](image)

4.2 Design Goals

4.2.1 Simplicity

Simplicity is a major driving force in this system. Our goal is to get the system designed, implemented, and running as quickly as possible. A quick creation and installation allows people to start using and testing the system earlier. They are then in a better position to comment about new features and improvements. Simplicity comes at a cost of inefficiency, but these inefficiencies can be eliminated later based on priority.
4.2.2 Availability

One of the attractive features of CAPE is its ability to allow developers to share software modules. And, as the number of developers using the CAPE system increases, so does the number of software modules that are in the CAPE database. And a large database attracts more developers. This is a self-feeding cycle. It is, therefore, in our interest to make CAPE available to as many people as possible.

4.2.3 Modularity

We design CAPE to be very modular. Modularizing a system breaks it into smaller, more manageable, and independent sub-systems. Modularity also allows us to replace a part of the system with a newer, improved part with relative ease.

4.3 Network Architecture

A network connects CAPE hosts to other CAPE hosts and to users via their web browsers. Currently, CAPE has no access control, and so a web browser or a CAPE host can communicate with any CAPE host that is accessible through its network. This forms a very loose and flexible arrangement which forms an available and accessible network architecture (see Figure 4.1). Access, however, might want to be restricted to manage load and accessibility. Depending on the way that access control is implemented, a number of different network topologies can be created. A client/server topology and a peer-to-peer topology are described in the following sections, but these are not the only topologies available.

4.3.1 A Client/Server Topology

We could create a client/server arrangement where one host or a group of hosts serve a community of users. This arrangement is depicted in Figure 4.2. Basically, we establish access control lists on these hosts which allow access only from web browsers and CAPE hosts running on certain trusted machines. These web browsers and these trusted hosts can obtain documents from other hosts on the network, but these outside hosts can not access the documents stored on this set of hosts.

An advantage of a client/server topology is that there is a central repository which can be carefully maintained and administered. Administrators can better enforce access control, more easily back up the database, and can more carefully maintain the host. A disadvantage of this topology is that there is a central resource which is shared. If the host or set of hosts crashes, then the users of these host(s) are left without a CAPE environment.
Outside requests are not serviced by local CAPE hosts.

But requests from local browsers and hosts can still reach and be serviced by outside CAPE hosts.

4.3.2 A Peer-to-Peer Topology

We could also create an arrangement where only a web browser running on the CAPE host could access the documents stored on that host. The user of this system can now work in the CAPE environment and can use Java Document on both this host and on others. This is basically a client/server arrangement where the number of clients is limited to one (see Figure 4.3).

The advantage of this topology is that there is no central resource. Each user maintains his own CAPE host. If one host crashes, the other users are affected less because they are running their own CAPE host. Only the software modules belonging to the crashed host become unavailable. Also, this distributed setting allows users to easily join the CAPE system. All they have to do is start
their own CAPE host. To use documents stored on other hosts, they may have to ask the administrators of those hosts to set the appropriate permissions. But to simply make available a set of documents, they do not need to get permissions from a central authority. A disadvantage with this topology is that hosts in a distributed arrangement are harder to maintain. A normal user of CAPE might not have the time, energy, or resources to maintain the CAPE host as well as a dedicated administrator could. But it is hard for a dedicated administrator to work in such a distributed environment, it therefore, becomes the responsibility of each user to maintain his or her host.

4.4 The CAPE Design and Architecture

We broke the system into three major layers (see Figure 4.4). A Database Layer controls and provides access to the documents stored in the CAPE database. It stores and retrieves documents from a local database, maintains the local document space, and communicates with the Database Layer of other hosts to form a connected database of documents. The Database Layer exports an interface, which hides the distributed nature of the database and the details of the underlying storage and versioning mechanisms, to the Control Layer. The heart of the CAPE system is the Control Layer. The Control Layer implements CAPE’s features and functionality. The User Interface Layer provides a communications channel between the system and its users. In addition to this high-level modularity, we design each of the layers themselves to be modular.

The Database Layer is made up of three sub-modules (see Figure 4.5). The Document Control Module enforces and maintains the document space and controls the access to documents in the CAPE database. The Storage Module implements the local database. And the Remote Document Control Module works with the Document Control Module to provide a communication channel between CAPE hosts.

The Control Layer, implemented primarily by the Control Center Module, provides and defines all of the CAPE features. It relies on the Database Layer for storage and retrieval of documents and on
the User Interface Layer for access to its users. Within this layer, the Control Center relies on the Java compiler and a Code Processor (see Figure 4.6). CAPE uses the standard Java compiler issued by Sun Microsystems, Inc. as its Java compiler. The Code Processor retrieves important information, such as the fully qualified name of a class, from a given source code. The Code Processor relies heavily on a modified Java parser built with the help of the Java Compiler Compiler (JavaCC) from Sun Microsystems, Inc.

The User Interface Layer provides communications channel between the CAPE system and its users. The User Interface Layer consists of four sub-modules (see Figure 4.7). A web browser, using HTTP, provides the vehicle through which the user can communicate with CAPE. At the CAPE host end, the HTTP Interface module listens on the CAPE user interface port for HTTP requests. Once a connection is established with a web browser, CAPE creates a new thread of type HTTP Request Handler to process the request. The HTTP Request Handler interprets the request and makes the appropriate calls to the Control Center and Code Processor in the Control Layer and
to the Web Page Factory. The Web Page Factory produces the appropriate web pages that the HTTP Request Handler then sends back to the user’s web browser.

![Diagram of the User Interface Layer](image)

**Figure 4.7:** The User Interface Layer

Figure 4.8 shows a schematic of the whole system.

## 4.5 The Database Layer

The Database Layer provides an interface to the CAPE distributed repository of documents. It has three major functions: safely storing documents which belong to the local host, maintaining the local document space which includes providing version control, and communicating with the Database Layer of other CAPE hosts to provide access to remote documents. Correspondingly, there are three major modules which work to satisfy these functions: the Storage Module, the Document Control Module, and the Remote Document Control Module. Modularizing the Database Layer like this affords us the flexibility to replace any of the modules easily. For example, currently we implement a simple storage module which in the future we could replace with a more sophisticated one. Also, in the current system, the Database Layer uses RMI for communication, in the future we could implement a new protocol simply by slightly modifying the Document Control Module and replacing the Remote Document Control Module.

### 4.5.1 The Local Database

If we want to establish a database of documents, we need store these documents in a some form of stable memory. The Storage Module is responsible for the safe storage and retrieval of these modules from stable memory.

The Storage Module relies on the host’s file system for its stable memory. This module converts a document to a file when storing a document in the database, and it converts this file back to that
document when retrieving the document from the database. It can write any document, with a full name, to the file system. It relies on Java’s Object Serialization feature to produce streams for each document object. It then writes the bytes in these streams to files. At initialization time, the Storage Module needs to have a pointer to a directory on the host’s file system where it can store and manipulate all these files.

The Storage Module can save, retrieve, and remove documents from the local database. The remove feature must be used carefully. It can in no way be exported to the users. The Document Control Module removes documents only when it aborts an operation and needs to clean up the local database.

Database technology is a research area on its own. We use a very simple database in CAPE. We can improve it tremendously in terms of reliability, efficiency, and performance. For example, we
can add a database log improve reliability. A log records all transaction requests. During a system crash, there are some requests which have been requested, but have not yet been committed to the database, without a log these requests would be lost. A database with a log can reinitiate these requests once the system has been brought back up. To improve the efficiency of the Storage Module, we can redesign the actual storage mechanisms. Instead of relying on the host's file system, we can possibly use a dedicated storage medium which we can optimize for CAPE. The general nature of the file system precludes our ability to optimize its storage mechanism for CAPE.

4.5.2 The Control and Maintenance of the Document Space

The Document Control Module controls access to the CAPE database of documents, provides version control for Java Documents, and maintains the document space which consists of Java Documents, Version Documents, and Group Documents. It relies on the Storage Module for the implementation of a local database and works with the Remote Document Control Module for a communications channel to remote databases on other CAPE hosts. It provides the database interface to the Control Layer. The following sections describes the exported interface.

Adding a Java Document to the CAPE Database

Given a Java Document, the Document Control can add it to the CAPE database. The name of the Java Document must be a full, non-generic name. If the host name of the Java Document does not match the host name of this CAPE host, then the Document Control asks the appropriate CAPE host's Remote Document Control Module to fulfill the request. Otherwise, the Document Control asks the Storage Module to add the Java Document to the local database. It assumes that the Java Document's version number was obtained by asking this module to return the next available version number for the given name (see below). This assumption allows the Document Control to further assume that the necessary support structure of Version and Group Documents exists because it created them when the version number was obtained. Before the Java Document is entered into the database, its Parent Name Annotation is set to the name of the Group Document which holds its Version Document. If anything goes wrong, the Java Document is not entered into the database.

Getting a Document from the CAPE Database

Given a document's name, the Document Control can try to retrieve that document from the database. If the name identifies the requested document as belonging to another CAPE host, then the Document Control asks the appropriate CAPE host's Remote Document Control Module to
fulfill the request. If the document belongs to this host, then the Document Control asks the Storage Module to retrieve the document.

Checking the Existence of a Document in the CAPE Database

Given a document’s name, the Document Control informs the caller whether or not a document bearing that name exists in the CAPE distributed database. If the name identifies the document as possibly belonging to another CAPE host, then the Document Control asks the appropriate CAPE host’s Remote Document Control Module to fulfill the request. If the document belongs to this host, then we poll the Storage Module to verify whether or not the document exists.

Deleting a Java Document from the Local Database

Given a Java Document’s name, the Document Control can try to delete that Java Document from the local database. If the name identifies the document as belonging to another CAPE host, then the Document Control can not delete it. If the name is not a full, non-generic name, then again the Document Control can not delete it because that name does not belong to a Java Document. If the Java Document exists in this host’s local database, then the Document Control removes it and its entry in its Version Document. The Java Document is returned to the caller. The Control Layer has to be very careful with its use of this feature.

Modifying a Java Document in the CAPE Database

Given a Java Document’s name, a Java Document method, and the arguments for that method, the Document Control can modify that Java Document. It retrieves the Java Document from the CAPE database, performs the given method on that Java Document using the arguments given, and stores the Java Document back in the database. If the host name of the supplied name does not match the name of this host, then the Document Control asks the appropriate CAPE host’s Remote Document Control Module to fulfill the request. Otherwise, this method modifies the Java Document found in the local database.

Getting the Next Available Version Number for a Generic Name

Given a generic name, the Document Control can get the next available version number for this name. The name must be a generic name. If the host specified in the name does not match the name of this CAPE host, then the Document Control asks the appropriate CAPE host’s Remote Document Control Module to fulfill the request. Otherwise, the generic name is used to try to retrieve a Version Document from the local database. If one does not exist, the Document Control creates it and its supporting documents, the necessary Group Documents, and enters them into the local database. From this Version Document, the Document Control obtains the next available
version number. This number can be used by the Control Center which can assign it to a new Java Document.

4.5.3 Connecting a Distributed Database

The CAPE system is a collection of hosts each with their own local database. CAPE unites these local databases into one accessible database by using the Java RMI feature as the communications protocol between hosts (see Figure 4.9). This allows the Database Layer of a host to retrieve documents from other hosts. By performing this communication in the Database Layer, we provide the abstraction of a single database to the Control Layer; the Control Layer does not need to concern itself with the details of a distributed database.

![Database Layer of a CAPE Host](image)

![Database Layer of a CAPE Host](image)

**Figure 4.9**: Schematic of the Database Layer Communications Channel

The Remote Document Control Module works with the Document Control Module to provide the communications channel between hosts. The Remote Document Control Module accepts RMI calls from the Document Control Module of other CAPE hosts. When a Document Control Module discovers that it is trying to access a document which does not belong to its local database, it initiates an RMI call to the Remote Document Control Module of the appropriate CAPE host. This Remote Document Control Module then performs some necessary checks to ensure that this is, in fact, the right host to handle the call. It then asks its Document Control Module to handle the request and passes the results back to the requesting Document Control Module.
4.5.4 Future Advances in the Database Layer

There are two extensions that we could make to the Database Layer that are worth mentioning: caching and Lotus Notes.

**Caching**

Caching of remote documents aids in both performance and reliability. In the current version of CAPE, every time a remote document is needed it is retrieved from the remote host's local database via the network. This network access has a substantial cost. A better strategy would be to cache the remote document. Then, the next time that document is requested by this host, it can be supplied from the cache instead of via the network. There is, however, a concern that the cached copy will become stale. So, we can either limit the use of the cached copy to functions which only use the static parts, for example the definition of a Java module, of the document, or we can establish a cache coherency protocol which ensures that the cached copy is always up-to-date. Caching documents would also improve reliability. If the home host of a document goes down, hosts with cached copies can still use the static information on those copies.

**Lotus Notes**

An even better extension might be to replace the Database Layer with Lotus Notes. Lotus Notes provides many of the features we require in a Database Layer. It can store documents, it can communicate with other Notes servers, and it can provide version control. Not only can it do all these functions, it can do them better than our simple Database Layer implementation. Lotus Notes would provide an excellent database foundation for CAPE. We did not implement a Notes database in the current version of CAPE because Notes is very complex. Dealing with this complexity would slow down our implementation time.

4.6 The Control Layer

The Control Layer coordinates all of the activities of a CAPE host. It provides all of the "CAPE" functionality. The Control Center Module is the main module of this layer. The best way to describe the Control Layer is to describe the interface the Control Layer exports to the User Interface Layer through the Control Center.

4.6.1 Submitting a Java Document to the CAPE Environment

To submit a Java Document to the database, the user must supply, through the User Interface Layer, the following information: source code, a list of dependencies, a list of parent version Java
Documents, and a suggested local name. Using this information, the Control Center produces a Java Document object and enters it into the CAPE database.

First, we need to name the Java Document. CAPE first tries to assign the Java Document's name based on the fully qualified name of the Java module defined in the source code. If successful, then we are assured that the Java Document name matches the Java module's name. The Code Processor tries to parse the source code and retrieve the fully qualified name of the module. If the source code is unparsable, then we must try another manner by which we can name this Java Document. If the code is parseable, then the method checks to see if the source code is valid source code for a Java Document: there must be exactly one Java module defined in the source code and the fully qualified name of that module must be a full, generic name. If the source code is valid then the fully qualified name of the Java module is used as the generic name of the Java Document. For this generic name, the Control Center asks the Document Control Module for the next available version number, and using that number and the fully qualified name of the Java module, the Control Center forms a full, non-generic name for the Java Document.

If the source code is not parseable or if the conditions listed in the preceding paragraph are not met, then the Control Center attempts to assign the Java Document based on the suggested local name provided by the user. It attempts to create a full, generic name from the suggested local name and the name of this host. If successful, the Control Center retrieves a version number for this name from the Document Control Module, and uses this version number to form a full, non-generic name for the Java Document. If this does not work, then the Control Center assigns the Java Document a default name. The generic default name is of the form: “hostname.GCC.default”.

With this name and the source code provided by the user, the Control Center creates a Java Document. Next, the Control Center tries to add the dependencies supplied by the user to the Java Document. Each dependency needs to meet the following conditions before it can become a dependency of the Java Document: the dependency must be a full, non-generic name, the dependency must refer to a Java Document that exists in the CAPE database, and the dependency must not have the same generic name as the Java Document. If all the dependencies meet these conditions, then the Control Center adds them to the Java Document.

Finally, the Control Center tries to add the names of the parent versions of this Java Document to the Java Document. Each parent version name must meet the following conditions: the name must be a full, non-generic name, and the name must refer to a Java Document that exists in the CAPE database. If all the parent version names supplied by the user meet these conditions then the
Control Center adds them to the Java Document. Also, to each parent version of this Java Document, the Control Center adds the name of this Java Document as a child version.

4.6.2 Getting a Document from the CAPE Database
The Control Center can retrieve a document from the CAPE database. Given the name of a document, it asks the Document Control Module to retrieve that document.

4.6.3 Adding a Comment to a Java Document
The Control Center can add a comment to a Java Document. A comment in this system is simply a string of characters. To attach a comment to a Java Document, the Control Center needs two pieces of information: the comment and the name of the Java Document. The Control Center uses the modify feature of the Document Control Module. It passes to the Document Control Module the following: the name of the Java Document, the Java Document add comment method, and the comment. If successful, the Control Center returns the modified Java Document to the User Interface Layer.

4.6.4 Compiling a Java Document
The Control Center can compile a Java module stored in Java Document. To do this, it needs to know the name of the Java Document and the name of a directory on the host where the compiled class files are to be placed. CAPE uses the standard Java compiler with no modifications. The Java compiler expects the source code to be in text files not in Java Documents. So, the Control Center needs to store the source code for each Java Document that needs to be compiled in a text file. It then feeds these text files to the compiler, and the compiler writes the compiled classes to files in the specified output directory.

First, the Control Center obtains the Java Document that we wish to compile from the CAPE database. Having this Java Document is not enough, to compile the Control Center needs all the dependencies of this Java Document as well. Once the Control Center has all the Java Documents, it saves the source code of these Java Documents to files. CAPE has a special directory that it uses to store these temporary files. To keep the files from different compilations separate, the Control Center creates a sub-directory for each compilation attempt. Finally, the Control Center calls the compiler with the names of all the files and the output directory.

4.6.5 Switching a Java Document's Dependence
The Control Center can change a Java Document's, and all its dependencies, dependence on a particular Java Document to another Java Document. For example, in Figure 4.10a, we see the dependency graph for a Java Document called A. This graph shows that many of the Java
Documents in A’s dependency graph are directly or indirectly dependent upon H (A, B, C, and G). The feature described in this section allows the Control Center to produce a partially new set of Java Documents which are identical to the original set except that they depend on H' instead of H. This new dependency graph is shown in Figure 4.10b. We will use this example, the shift from A to A', to illustrate the algorithm.

To do this the Control Center implements a fairly simple algorithm. First, it separates the Java Documents in a dependency graph into two groups. One group contains Java Documents which directly or indirectly depend on the particular Java Document, and the other group does not. Then, for every Java Document that depends on the particular Java Document, it creates a new Java Document which has the same source code as the original but a new set of dependencies which shift the Java Document’s dependence from the particular Java Document to a different version of that Java Document.

To facilitate the description of the algorithm, let us define a few names first. We want to change the dependency graph of the given document. The dependency, which we want to replace, we call the original dependency. We replace the original dependency with the updated dependency. To describe the algorithm we will also use the concept of sets. We will use four sets. Sets A and C will contain Java Documents which have been identified as Java Documents which depend on the original dependency. Set B will contain the remaining Java Documents. Set D will contain the new Java Documents.

![Figure 4.10: Dependency Switch](image)
First, we get all the Java Documents in the dependency graph of the given document. If the original dependency is in this graph, then we place it in set A. The rest we place in set B. We then start the separation process. For every Java Document in set B, we obtain a list of its direct dependencies. We compare every element in this list to every element in set A. If there is a match, then this Java Document must depend on the original dependency. We mark this Java Document. After going through all of set B, we move all the elements in set A to set C. Then we move all the marked Java Documents in set B to set A. And this process repeats until there are no elements in set A. At this point, all the Java Documents in set C depend on the original dependency, and all the Java Documents in set B do not. Figure 4.11 shows this separation process for our example problem.
Then, for every Java Document in set C, except the original dependence, we clone it, assign it a new version number, and place it in set D. Now, for every element in set D, we obtain a list of its dependencies. If a member of this list points to a Java Document in set C, which is not the original dependence, we change that dependency to point to the corresponding Java Document in set D. And if a member of the list points to the original dependency, then we change it to point to the updated dependency. Once, this is done, all Java Documents in set D should be independent of all Java Documents in set C. Figure 4.12 shows this cloning and dependency shifting process.

The elements in set B were not changed at all. The elements in set D however are new, and they need to be stored in the GCC system.

This completes the algorithm.

4.6.6 Synchronizing a set of Java Documents

As the development of a program proceeds, the dependencies of the Java Documents comprising the program might get skewed. That is, all the Java Documents in the program might not be using the same versions of the Java Documents. We illustrate this with an example. Supposed we have a program which consists of three Java Documents. Let’s call these Java Documents A, B, and C. Originally, Av1 depends on Bv1 and Cv1, and Bv1 depends on Cv1 (see Figure 4.13). But as we continue to modify and develop these Java Documents, we produce newer versions of these Java Documents. So, after some time we have Av5 which still depends on Bv1 and Cv1, we have Bv5 which depends on Cv4, and we have Cv6, where Av5, Bv5 and Cv6 are the most current versions of each of the Java Documents (see Figure 4.13). What we would like to do is to synchronize these three Java Documents, so that new versions are produced which all depend on each other rather than on other versions. So, we want Av6 to depend on Bv6 and Cv7, and we want Bv6 to depend on Cv7, where Av6, Bv6, and Cv7 are just like Av5, Bv5, and Cv6 except for these dependency changes (see Figure 4.13).

Given the names of Java Documents that need to be synchronized, the Control Center first retrieves these Java Documents from the CAPE database. These Java Documents comprise our synchronizing set. For each document in the set, the Control Center obtains and assigns a new version number from the Document Control Module. Next, the Control Center obtains a list of each Java Document’s dependencies. For each name in the list, we see if that name has the same generic name as a Java Document in the set. If it does, then we change that dependency to point to the Java Document in the set. Finally, these Java Documents with new version numbers are added back to the local database.
Clone every Java Document in Set C except for H

(1) Shift the dependencies for A'
(2) Shift the dependencies for B'
(3) Shift the dependencies for C'
(4) Shift the dependencies for G' -- done

Figure 4.12: Cloning and Dependency Shifting

4.6.7 Reading Source Code from a File

The Control Center returns to the User Interface Layer the source code that is stored in a particular file on the host’s file system.
4.6.8 Saving Source Code to a File

The Control Center writes a given source code text to a particular file on the host’s file system.

4.7 The User Interface Layer

The User Interface Layer provides the mechanism by which users of CAPE can interact with the system.

4.7.1 Overview of the Architecture

We want to make the CAPE system very accessible. It is for this reason that we choose the web browser as our interface agent. A web browser is available for almost all platforms, and so the users of almost all platforms have access to CAPE. Another benefit afforded by web browsers is their support of the HTTP. This is the protocol used by the World Wide Web; it has proven to be very scaleable in a globally distributed setting. A very scaleable protocol is exactly what CAPE needs. The web browser is also a very flexible interface agent because it can incorporate small programs called applets into the pages it displays. These applets can tremendously increase the
functionality of web pages. The choice of a web browser as our interface agent buys us accessibility, scaleability, and flexibility.

The web browser and CAPE communicate exclusively using HTTP. Web browsers initiate HTTP requests to CAPE. Serviceable HTTP requests come in two flavors: a CAPE request and a web request. A CAPE request asks CAPE to perform a CAPE function such as: submit this new document, view a document, or synchronize a set of document. A web request asks CAPE to return the contents of a file, specified by a given path. This, for instance, can be used by the browser to request an applet's class file. All the information in a request is encoded into a URL. The HTTP Interface Module listens to CAPE's user interface port for HTTP connection requests. When a browser requests a connection, this module accepts the connection and opens a socket for communication between the web browser and this CAPE host. It then passes this socket, a pointer to the Control Center, a pointer to the Web Page Factory, the name of the local root document, a pointer to the Code Processor, and a pointer to the directory where the files for web documents are stored to a new HTTP Request Handler thread for interpretation and processing of the HTTP request. From the socket, the HTTP Request Handler reads the HTTP request, and interprets the request. For CAPE requests, it calls upon the appropriate methods in the Control Center, the Code Processor, and the Web Page Factory to service that request, and then returns the web page produced by the Web Page Factory to the web browser via the socket. For Web requests, it reads the appropriate web file from the web directory and returns that file to the web browser via the socket.

The Web Page Factory dynamically creates the different web pages that CAPE uses to communicate with the user. Each page presents the requested information or options. These options are in the forms of links and applet commands which allow the user to interact with CAPE.

4.7.2 The URL format used in CAPE

All information traveling in the browser to CAPE direction is done via a Uniform Resource Locator (URL) [Ber]. The CAPE URL is a restricted version of this general URL.

A CAPE URL has one of the two following forms:

CAPE request URL:


Web request URL:

http://<hostname>:8000/<web-document>
The value of <hostname> tells the browser the name of the CAPE host to which we wish to connect. “8000” tells the browser to try to connect on port 8000, the user interface port, on the CAPE host. <local-name-of-document> specifies the local name of the document we wish to access. A local name of a document is made up of alpha-numeric parts which are separated by periods, and so, this field is also made up of alpha-numeric characters and periods. Since the local name of a document specifies a CAPE document, it must begin with “GCC”. The <hostname> and <local-name-of-document> together identify a document in the CAPE database. They are put together in the following fashion to produce a CAPE name:

<hostname>.<local-name-of-document>

The “?” separates the <local-name-of-document> field from the rest of the URL. The string “GccCommand=” identifies that the value of <command> is the command that the user is requesting be performed on the specified document. If the URL ends at the <local-name-of-document> field, that is, if there is no “?GccCommand=<command>[arguments]”, then the HTTP Request Handler will assume that the URL is requesting to view the specified document, and the URL is treated as if it had “?GccCommand=ViewDocument” appended to it. [arguments] is an optional field that can be used to supply addition information to the CAPE host, if needed. This field has the following format:

&<argumentName>=<argumentValue>[arguments]

The <argumentName> field tells us how to interpret the value of the <argumentValue> field. The <argumentValue> field contains the information that we wish to send to the CAPE host along with the command. There can be as many arguments as needed. The type and number of arguments varies depending on the command. The ordering of the arguments does not matter.

The <web-document> field tells us the path of a web file which is stored on this host.

The field <hostname> should contain only alpha-numeric characters and periods. The fields <local-name-of-document> should also only contain alpha-numeric characters and periods. Sometimes, the periods in this field will be replaced with their equivalent URL encoding value, “%2e”. The <command> field should only contain alphabet characters. In the [arguments] field, the <argumentName> field should only contain alphabet characters. The <argumentValue> field can consist of any characters, but only alpha-numeric characters can be used directly. All other characters need to be encoded in the URL encoding.
Chapter 5

Evaluation

5.1 Evaluation Strategy
We believe that CAPE does provide a collaborative and distributed software development environment. We believe that if we show that developers in CAPE can develop software, that programs can be built from software modules which are distributed over network connected host, and that developers can communicate with each other via comments attached to software modules then we have shown that CAPE has met its intended goals.

Section 5.2 describes the program that we will use in our demonstrations. And section 5.3 outlines the demonstrations.

5.2 The Software Program
5.2.1 Overview
The program that we use to demonstrate CAPE’s features is a highway simulation program. It is very simple. It consists cars moving along a highway. The highway has three lanes and cars which can have two different types of drivers. Aggressive drivers and non-aggressive drivers. Non-aggressive drivers drive slowly and never change lanes. Aggressive drivers drive fast. If there is a car blocking their lane, they try to pass that car either on the left side or on the right side. If they can’t then they are forced to slow down. These are very simple rules for drivers. They seem to work, but they also cause occasional accidents on the roads.

5.2.2 The Program Architecture
This program tries to simulate driving. The program has MotorVehicle objects which move along Road objects. Road objects are made up of Lane objects which are in turn comprised of LaneSegment Objects. A LaneSegment can be empty, have a MotorVehicle, or have an accident (more than one MotorVehicle). Driver objects control these MotorVehicle objects. Drivers look at the Road and the position of other MotroVehicles on that Road. With this information they decide on an action. Drivers come in two flavors: aggressive and non-aggressive. Non-aggressive Driver objects move their MotorVehicles forward one LaneSegment in each time interval. Aggressive Driver objects try to move forward two LaneSegments in each time interval. If they can’t do this because of other MotorVehicles or collisions in their Lane, then they try to pass that obstacle on the left or on the right. Otherwise, they slow down and simply move forward one LaneSegment. A
Simulator object is the top level object. It produces new MotorVehicle objects with Driver objects and places them at the beginning of the Road. It also runs the simulation. For every iteration, it asks each Driver for that Driver’s action. Then, it moves each MotorVehicle accordingly. Finally, it writes the state of the Road object to a file. This file can be read at the end of the simulation. These six Java classes make up the simulation program. The dependency relations among these classes can be seen in the dependency graph shown in Figure 5.1.

![Dependency Graph for the Highway Simulator](image)

**Figure 5.1:** Dependency Graph for the Highway Simulator

### 5.3 Demonstration of a Successful System

#### 5.3.1 Demonstration of the development features of CAPE

CAPE claims to be a development environment. For this, we need to demonstrate that developers can use CAPE to develop Java modules. We need to show that CAPE provides the necessary tools that are needed to modify and produce new Java Documents. There are four actions that allow developers to develop Java Documents: submitting a new Java module definition to a CAPE host to form a Java Document, creating new Java Documents by modifying existing ones, creating a new Java Document by asking CAPE to switch a Java Document’s dependence on one Java Document to another, and finally, synchronizing a set of Java Documents.

We use the highway simulator program to illustrate these features. In this example we will place all of the Java classes for this program on one CAPE host, qt.ai.mit.edu. The generic names for these classes will be:

- qt.ai.mit.edu.GCC.simulators.driving.Road
- qt.ai.mit.edu.GCC.simulators.driving.Lane
- qt.ai.mit.edu.GCC.simulators.driving.LaneSegment
Using the Java Document editor, we submit to qt.ai.mit.edu new Java Documents based on the source code for each of the original classes. We have neither set the Dependency Annotations for these documents (see Figure 5.2), nor have we modified the source code to reflect the new class names (listed above). And so, these Java Documents are not ready to be compiled.

All Java Document Names should be prefixed with: "qt.ai.mit.edu.GCC.driving."

Figure 5.2: Dependency Graph of the Simulation Program just after submission

Using the Java Document editor, we create new Java Documents based on these existing ones. In each of these new Java Documents, we have modified the source code so that the Java module’s fully qualified name matches the full, generic name of the Java Document, and we have added Dependency Annotations (see Figure 5.3). But, we still cannot compile the simulation program. Though each new Java Document has Dependency Annotations, these Dependency Annotations do not always point to the other members of this new set. This is due to the restriction, that a Dependency Annotation can only point to an existing Java Document. So, as we formed Java Documents in the new set, we could not always point their Dependency Annotations to other members of the new set because they did not exist yet, and so we consistently set the dependencies to point to Java Documents in the first set.

This problem can be fixed by synchronizing the Java Documents. We ask CAPE to synchronize all the Java Documents in the second set, the ones with Dependency Annotations and a modified source code. The dependency graph of the simulator program after the synchronization can be seen
All Java Document Names should be prefixed with：“qt.ai.mit.edu.GCC.driving.”

Figure 5.3: Dependency Graph of the Simulation Program just after modification

in Figure 5.4. The program is now ready to be compiled. We compile and run it. And we note that it does in fact work correctly.

All Java Document Names should be prefixed with：“qt.ai.mit.edu.GCC.driving.”

Figure 5.4: Dependency Graph of the Simulation Program after Synchronization

So, far in this demonstration, we have shown that we can submit new Java Documents to CAPE, that we can modify the source code and the dependency list of existing Java Documents to produce new ones, and that we ask CAPE to synchronize a set of Java Documents. In the remaining part of
In the simulation program, aggressive drivers can pass other cars on both the left and the right. We now want to restrict these drivers so that they can only pass on the left. We, therefore, create a new Driver Java Document, qt.ai.mit.edu.GCC.mit.driving.Driverv3, with this new restriction. To incorporate this new Java Document into the simulation program, we ask CAPE to completely switch the Simulator’s dependence to this new class. The resulting dependency graph is shown in Figure 5.5. Driverv3 is still dependent on the older version of MotorVehicle and Road. To fix this, we ask CAPE to synchronize the program. It does and the resulting dependency graph is shown in Figure 5.6. We compile this program and notice that the aggressive drivers only pass on the left-hand side of cars.

In this section, we have successfully displayed CAPE’s development environment features.

5.3.2 Demonstration of a program built from distributed software modules

To demonstrate that a program can be built from a distributed set of Java modules, we take the modules that comprise the highway simulator and store them on two different CAPE hosts: jumpgate.ai.mit.edu and qt.ai.mit.edu.

We place the Road, Lane, and LaneSegment classes on jumpgate.ai.mit.edu, and we place MotorVehicle, Driver, and Simulator on qt.ai.mit.edu. The following list shows the fully qualified names of the new Java classes which are the generic names of the Java Documents which hold these classes:

Figure 5.5: Dependency Graph of the Simulation Program after switching the dependence
All Java Document Names should be prefixed with: "qt.ai.mit.edu.GCC.driving."

Figure 5.6: Dependency Graph of the Simulation Program after Synchronization

- jumpgate.ai.mit.edu.GCC.simulators.driving.Road
- jumpgate.ai.mit.edu.GCC.simulators.driving.Lane
- jumpgate.ai.mit.edu.GCC.simulators.driving.LaneSegment
- qt.ai.mit.edu.GCC.simulators.driving.MotorVehicle
- qt.ai.mit.edu.GCC.simulators.driving.Driver
- qt.ai.mit.edu.GCC.simulators.driving.Simulator

We then change the source code of each of these Java Documents to use these new fully qualified names and then submit each of these Java Documents. We then add the appropriate Dependency Annotations to the Java Documents to create the correct dependency graph for the Simulator module, which is the top level class in this program. Finally, we ask the CAPE host, qt.ai.mit.edu, to compile the Java Document, qt.ai.mit.edu.GCC.simulators.driving.Simulator. It did, and so we have demonstrated that a program can be built from a distributed set of software modules.

5.3.3 Demonstration of the communication features of CAPE

CAPE also claims to have the ability to allow users to communicate via comments attached to Java Documents. To demonstrate this we first attach a comment to a Java Document, for instance qt.ai.mit.edu.GCC.simulators.driving.Driver. We then try to view that comment using web browsers on different hosts. We tried this and it worked, and so we have demonstrated that users can, in fact, communicate via comments attached to Java Documents.
Chapter 6

Conclusions
The GCC project aims to leverage the prevalence of network computers to aid software development, distribution, and use. CAPE establishes a solid foundation for GCC. Under the CAPE environment, developers can build programs from a distributed set of Java modules. An environment such as this allows developers to easily share and use software modules developed by others. Developers save time and energy because they do not have to duplicate someone else’s development effort. Developers who post their modules in the CAPE environment also benefit. By allowing others to use their modules, these developers can more quickly obtain feedback about their modules. Users will surely want to inform the authors of bugs, security holes, and inefficiencies because they have an interest in obtaining an improved module. The feedback channel provided by CAPE allows users to not only inform the authors, but the other users as well. Because both developers who submit modules to the environment and those that use modules in the environment benefit from CAPE, we think the CAPE environment would be very popular in the programming community.

To assess its popularity, we need to test it in real settings. We never had a chance to perform this sort of testing, but it is something that needs to be done. As people use the system, we the developers of CAPE can get real feedback on how to improve it.

We think CAPE establishes a solid framework for future work. Many extensions have been outlined throughout this thesis. Improvements to the Database Layer are crucial. The current implementation is very simple. Its design allows for a quick implementation, but in a real setting reliability and performance will determine the success of CAPE. Also, currently CAPE is primarily a development environment. We could extend CAPE to perform distribution functions as well. Companies or developers could use CAPE to distribute finished programs and upgrades more easily.

CAPE unites distributed and possibly independent developers together. It provides an easy way to share and distribute software modules. Based on this framework, many improvements can be made to the software development and distribution processes.
References


