VISUALIZING THE EXECUTION OF OBJECT-ORIENTED CODE MOBILITY APPLICATIONS

by

Yi Wang

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements of the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario
CANADA, K1S 5B6

June, 2000

The undersigned recommend to the Faculty of Graduate Studies and Research the acceptance of the thesis

VISUALIZING THE EXECUTION OF OBJECT-ORIENTED CODE MOBILITY APPLICATIONS

submitted by Yi Wang, M. Eng in partial fulfillment of the requirements for the degree of Master of Engineering

Thesis supervisor

Chair, Department of Systems and Computer Engineering

ABSTRACT

Code mobility has potential to provide more flexible and efficient solutions to some traditional client/server applications, especially in a large scale, dynamic and heterogeneous network environment. However, object-oriented distributed applications employing code mobility are intrinsically complex and hard to understand. Event-based visualization can be an invaluable tool for understanding the dynamic behavior of such applications by providing graphical views of the program execution.

In this thesis, interesting events that help understand the execution of object-oriented code mobility applications are identified by carrying out a study of the mobile code technology. An innovative approach to visually present code mobility is proposed. An information tracing and visualization infrastructure (CMVS) for understanding the execution of object-oriented code mobility applications was developed. CMVS supports both on-line monitoring and postmortem visualization. It facilitates view zooming and information query to allow users to focus on a particular area of the view, and to get more detailed information regarding a specific event. Challenging issues with respect to program visualization are addressed, including program instrumentation, preservation of event causality, scalability, and quick focus on particular concerns.

ACKNOWLEDGEMENT

I would like to thank my supervisor, Professor T. Kunz, for his guidance, advice and support. Working with him is intellectually stimulating and this thesis profited enormously from discussions with him.

I would like to take this opportunity to thank all my friends who always give me encouragement and support no matter where I am.

I want to express my gratitude to my parents, who have encouraged me to strive for excellence and always supported me in pursuing my goals. They have given me also wise advice and unconditional love.

This research work was supported by Natural Sciences and Engineering Research Council (NSERC) and Bell Mobility Co.. I am also grateful to them for providing financial support to allow me to concentrate on my thesis.

Table Of Contents

Chapter 1 Introduction

1.1 Research Motivation 1

1.1.1 Distributed Object-Oriented Applications with Code Mobility 2

1.1.2 Complexity Of Distributed Object-Oriented Applications with Code Mobility 4

1.1.3 Event-Based Program Visualization 5

1.2 Thesis Contributions 6

1.3 Thesis Organization 7

Chapter 2 Related Work

2.1 Automated Instrumentation and Monitoring System (AIMS) 11

2.2 Pablo Performance Analysis Environment 14

2.3 XPVM 18

2.4 ParaGraph 21

2.5 PVanim 24

2.6 Summary 28

Chapter 3 Mobile Code Technology and an Agent-Enhanced ORB-Voyager

3.1 Mobile Code Technology 30

3.1.1 Mobile Code Paradigms 31

3.1.2 Mobile Objects 34

3.2 Java and An Agent-Enhanced ORB-Voyager 36

3.2.1 Java Architecture 37

3.2.2 Code Mobility Supported by Voyager 39

Chapter 4 Visualizing the Execution of Object-Oriented Mobile Code Applications

4.1 Process-Time Diagrams 42

4.2 Graphical Representations of Interesting Events 44

4.2.1 Object Creation and Disposal 44

4.2.2 Code Mobility 45

4.3 Visualization Objectives 49

Chapter 5 Event Data Collection Subsystem of CMVS

5.1 Program Monitoring 53

5.1.1 Program Instrumentation 54

5.1.2 Event Records 61

5.2 The Architecture of Event Data Collection Subsystem 63

5.3 Design and Implementation Issues 68

5.4 Functionality 74

Chapter 6 Event Processing and Graphical Display Subsystem of CMVS

6.1 The Architecture of Event Processing and Graphical Display Subsystem 76

6.2 On-line Event Reordering 81

6.3 Facilities 83

6.4 Summary 86

6.4.1 An Overview of CMVS 86

6.4.2 System Evaluation 89

Chapter 7 Summary and Future Work

7.1 Summary 95

7.2 Future Work 97

References

List of Figures

Fig. 3.1: Code on demand. 31

Fig. 3.2: Remote evaluation 32

Fig. 3.3: Mobile agent. 32

Fig. 3.4: Service migration 34

Fig. 3.5: Java virtual machine. 38

Fig. 4.1: Process-time diagram 42

Fig. 4.2: Object creation/destruction 45

Fig. 4.3: Code mobility depiction-approach 1 46

Fig. 4.4: Code mobility depiction-approach 2 47

Fig. 4.5: Code mobility depiction-approach 3 48

Fig. 5.1: The visualization process 52

Fig. 5.2: Instrumentation by inheritance. 56

Fig. 5.3: Program instrumentation by applying wrapper pattern 57

Fig. 5.4: Schematic overview of event records. 62

Fig. 5.5: Distributed event data collection 64

Fig. 5.6: Centralized event data collection 66

Fig. 5.7: Relationship of host, server and mobile objects. 69

Fig. 5.8: Services provided by the trace event collection subsystem 75

Fig. 6.1: The architecture of the event processing and graphical display subsystem. 77

Fig. 6.2: Application structure 80

Fig. 6.3: Snapshot of CMVS graphical view and user interface 83

Fig. 6.4: Snapshot of the process-time view query. 84

Fig. 6.5: Snapshot of the process-time view zooming 85

Fig. 6.6: A skeleton application modified using class inheritance to enable visualization 90

Fig. 6.7: A skeleton application modified using wrapper pattern to enable visualization 92

Chapter 1 Introduction

1.1 Research Motivation

Distributed computing refers to spreading out the processing and data over more than one computer, usually over a network, to cooperatively solve a single large problem. Numerous advantages, including a better cost/performance ratio, incremental scalability, better availability and robust programming mode, have brought distributed computing into the mainstream [1]. However, developing a distributed application is quite a challenge and so is the understanding of its dynamic behavior. This is due to the fact that distributed applications are intrinsically more complex and non-deterministic. Increasingly, event-based visualizations are becoming useful and powerful tools for understanding the behavior of distributed executions [2] [3] [4] [5] [6]. Such tools can assist in reconstructing and analyzing information about program execution to help software developers debug, test, maintain, and optimize their codes [7] [8]. This thesis discusses approaches and presents toolkits to facilitate program understanding for one specific class of applications, distributed object-oriented applications with code mobility.

1.1.1 Distributed Object-Oriented Applications with Code Mobility

Over the last decade, distributed computing has evolved significantly. With the growing merger of computers and communication networks, as well as the introduction of object-oriented programming paradigm and new programming languages, several new technologies have been developed for distributed object-oriented applications.

A promising one is the so-called distributed object technology and many frameworks (generally referred to as Object Request Broker, or ORB) are currently available, such as CORBA [9], RMI [10] and Voyager [11]. This technology integrates remote procedure calls (RPC) with the object-oriented paradigm. It provides facilities to hide the underlying communication mechanism and supports location transparency. In distributed object computing, an object reference is created locally and bound to a server object. The local program can then invoke a method on this local reference as if it were a regular local object. The ORB transparently intercepts the method invocation and transmits the method request and its arguments to the server object, where the work is actually performed. The return value is then transmitted back to the local object.

Another innovative technology emerging recently is called mobile code. This technology exploits the notion of code mobility, which can be defined informally as the capability to dynamically change the bindings between code fragments and the location where they are executed [12]. Unlike other distributed technology, in this paradigm, the

code segment and the state of an execution unit (EU), which is the run-time view of a program such as a process or an individual thread, are not bound to a single computational environment (CE) for their entire lifetime, but rather they can move independently from one to another over the network.

Code mobility has potential to provide more flexible and efficient solutions to some traditional client/server applications, especially in a large scale, dynamic and heterogeneous network environment. The advantages expected from the introduction of code mobility are particularly appealing for some specific distributed applications, such as distributed information retrieval, remote device control and configuration, network management and electronic commerce [13]. With a perspective to enable new ways of building distributed applications and even of creating brand new applications, mobile code technology has aroused increasing interest from both academy and industry. More recently, several languages and frameworks have evolved to support code mobility, such as Java from Sun Microsystems [14], Voyager from Object Space, Aglet from IBM [15], Obliq from DEC [16], Telescript and Odyssey from General Magic [17] [18], Agent Tcl from University of Dartmouth [19], Concordia from Mitsubishi [20], to name but a few.

However, distributed object-oriented applications with code mobility involve all the complexity of distributed programs plus problems specific to object migration. This is due to the fact that those programs integrate distributed object architecture with mobile code technologies, and thus problems arise from both areas, each producing complexity.

1.1.2 Complexity Of Distributed Object-Oriented Applications with Code Mobility

Distributed programs are by nature large and complex [21] [22]. Multiple processes or threads residing in different locations interact with each in complex and possible unforeseen ways. Execution may be non-deterministic. Data collection is more difficult since there are multiple streams of events and these events must be ordered and merged by some means of ordering such as time stamp. Communication delays between processors may be large, and may vary greatly. It is not possible to have a global clock or synchronized system clocks running at the same rate and with sufficient resolution across the multiple processors involved in the computation.

Further complexity is introduced by mobile objects that can be shared by processes and moved to remote locations, execution flows that can migrate from site to site, and instances that can be dynamically created and destroyed in various places.

The complex nature of distributed object-oriented applications with code mobility makes it difficult to gain an understanding of those applications. It is evident that software development and maintenance each relies on program understanding. In order to cope with those complexities and take full advantage of the adopted technology, software developers much be provided with tools that help understand the execution of their programs as well as the collaboration and distribution of the objects.

1.1.3 Event-Based Program Visualization

Tools and modeling languages used to understand the static code structure are helpful but they have no explicit way for expressing distribution of objects over the network. It is believed that event-based visualization, the use of graphics and animation to visually describe and analyze the program execution, is a promising candidate to help gain insight into how a distributed object-oriented application with code mobility works. Event visualization tools provide the user with a graphical view of the execution. This view presents the occurrence of different types of events over time [23] [24]. An event is a conceptual entity that causes a change in the state of the execution. Each event occurs at the instant at which some predefined computation is completed. One supporting argument for choosing graphics over text is that textual presentations of data describing the execution of distributed application are inherently sequential while graphics may convey far more meaningful information than text. Another reason is that humans possess highly developed image processing system, which allow us to track multiple complex visual patterns and to easily spot anomalies in them.

However, a search of recent literature reveals that no visualization toolkits currently available provide facility for understanding code mobility. With the increasing application of code mobility in distributed network environment, it is very important to have a means to visually describe and illustrate the cooperation, distribution and migration of objects. Software developers can benefit from this during software

development and maintenance. It also helps network administrators to monitor those mobile objects if they allow them to float inside their network.

1.2 Thesis Contributions

Much of the research to date has focused on the visualization of parallel/distributed programs, no work seems to have been done on code mobility. This thesis discusses approaches and presents toolkits to facilitate understanding the behavior of distributed object-oriented applications with code mobility. Two major issues are addressed: what needs to be visualized and how do we visualize it. The thesis has accomplished the following achievements:

1. By examining mobile code technology and distributed applications that exploit code mobility, interesting events that need to be portrayed visually to help understand the program execution are identified.

2. Several existing visualization systems for distributed/parallel applications are surveyed. This survey gives an overview of the current research in the area of program visualization and can help visualization tool developers in designing their own system.

3. Several approaches to depict those interesting events are analyzed. An innovative way to present code mobility on a process-time diagram is proposed. This solution explicitly indicates the location change of the mobile objects without increasing display space needed. It is inherently scalable and correctly reflects concurrency.

4. An infrastructure that provides tracing facilities and supports both on-line and post-mortem visualization is designed and implemented. A solution to insert instrumentation with few modifications of source code is provided. An approach to uniquely identify the mobile object is proposed. Time adjustment strategy is developed to help preserve the causality of trace events.

5. A graphical display toolkit is developed to provide on-line and postmortem visualization. This toolkit facilitates trace play control, view zooming, information query, customization of the display order and space.

6. Challenging issues with respect to program visualization are addressed, including little user intervention, minimal probe perturbation, on-line event reordering, scalability, quickly focusing on particular concerns, and so on.

1.3 Thesis Organization

The thesis is organized as follows:

Chapter 2 reviews several existing visualization tools for parallel/distributed programs. The contributions of a particular system are discussed and a comparison to our work is given.

Chapter 3 examines mobile code technology in a greater detail in order to help understand why a particular event is chosen for display. The thesis is

based on programs using Object Space's Voyager as target environment. A brief overview of Voyager is provided as well.

Chapter 4 discusses what entities, relationships and actions existing in the program might be visualized. A detailed explanation of how to graphically present them is also given.

Chapter 5 and Chapter 6 present the code mobility visualization system (CMVS) that supports both on-line and postmortem visualization.

Chapter 5 introduces the event collection subsystem of CMVS. Its architecture and major components are described. Several design issues are addressed, including program instrumentation, trace file format, unique object identifier and the preservation of causality.

Chapter 6 presents the event processing and graphical display subsystem of CMVS. Its architecture and facilities are introduced. The issue of on-line event reordering is discussed and a brief evaluation of CMVS is given.

Chapter 7 gives a summary of this work and recommends some future research directions.

Chapter 2 Related Work

In this chapter, a number of visualization tools for distributed/parallel programs are reviewed. Examples of other tools that perform parallel/distributed program visualization are described in [25] [26] [27]. The most prevalent approach taken by these tools is to collect data during program execution, then provide postmortem or on-line analysis and display the different aspects of the program behavior. Some tools do both steps in an integrated manner, while others provide just one of these functions. The following tools were investigated:

- AIMS - instrumentor, monitoring library, and analysis tools
- Pablo - monitoring library and analysis tools
- SvPablo - integrated instrumentor, monitoring library, and analysis tool
- XPVM - graphical console and monitor for PVM (Parallel Virtual Machine) [34]
- Paragraph - graphical display system
- PVanim - monitoring library and animated visualization tool
- Polka - software animation and visualization tool

These tools are publicly available with source code. They are maintained and supported by their developers. For each tool listed above, we obtained the source code and compiled it. After the software was installed successfully, we went through any

tutorials or examples available to get familiar with the tool. The goal of this evaluation is to review the achievements that a particular tool has made to perceive how they will help build up our own visualization. Therefore, we place our emphasis on the following criteria, rather that carry out a side-by-side quantitative comparison.

- Functionality
- Usability
- Extensibility

For functionality, we focus on those facilities that are relevant to our visualization rather than give an overall discussion.

To be useful, we expect a tool with intuitive easy-to-use interface. The tool should have adequate and clear instruction on how to install and use it, in addition to other necessary documentation and support.

A tool is extensible if it provides facilities for users to add new displays of their own that can be viewed along with existing views. This capability is extremely useful to support special-purpose displays for particular applications.

Although some other criteria such as portability, scalability, and robustness are also very important, we do not focus on them for two reasons. One is that such information is unavailable for most of the toolkits. The other is that we did not develop a typical

application to examine those toolkits with respect to these criteria, since they are not as relevant as functionality, usability and extensibility to our evaluation goal.

2.1 Automated Instrumentation and Monitoring System (AIMS)

URL	http://science.nas.nasa.gov/Software/AIMS/
Version	3.7.2
Languages	C, Fortran 77
Platforms	IBM SP2, Sun, SGI, and HP work stations, SGI Power Challenge

Functionality

AIMS [28] is a software toolkit for measurement and analysis of Fortran 77 and C message-passing programs using the NX, PVM, or MPI communication libraries [29].

AIMS consists of three major components:

- A source code instrumentor
- A run time performance monitoring library
- A set of tools that process and display the performance data.

1) Source code instrumentation

AIMS is quite flexible in allowing the user to specify what constructs should be instrumented, and the instrumentation is done automatically without manually changing the

source code. However, its instrumentation is rather limited with respect to the source language, which must be standard Fortran 77 or ANSI C.

2) Trace file generation

After the instrumented source files are compiled and linked with the AIMS monitor library, they can be run in the usual manner to produce trace files. AIMS generates separate files for each process and then automatically collects and merges the per-process files to create a single trace file. Two important files generated by the instrumentor, an application database and an instrument-enabling profile, are used during run-time monitoring.

3) Trace file analysis

AIMS provides a set of software tools that process and display execution data.

- View Kernel (VK)
- Tally statistics generator
- Sysconfig

Tally and Sysconfig are used to display performance statistics and the network topology, respectively. VK provides graphical animations of the trace file. It has a view called OverView, which is similar to the process-time diagram, a visual presentation of the program execution in a two-dimensional display with time on one axis and individual entities (e.g. processes, threads, objects) on the other [51] [57]. In the OverView, each process is represented by a horizontal bar, with different colors for different instrumented

subroutines and white space to indicate blocking due to a send or receive. Messages between processes are represented by lines between bars. Both bars and message lines can be clicked on for additional information, including source code click back to the line that generated the event. The playback of the trace file can be controlled with VCR-like controls that allow the user to return to the beginning of the current trace file, step through it, or pause the animation. The time range of the time-line can be adjusted. However, it does not appear to have any way to scroll backward or to zoom on this view.

Usability

AIMS documentation is well-written. The Users' Guide gives step-by-step instructions on how to instrument, run, and analyze a user's application program; it also clearly explains the various features and options for each of the AIMS components.

Installation instructions are included in the software distribution. The major installation task consists of editing the top-level Makefile with system specific definitions. Although instructions are given on how to do this with sample Makefiles for different platforms, it is not a trivial effort to modify the Makefile. In addition, it is not easy to instrument, compile and run the examples.

Extensibility

AIMS does not appear to provide support for users to seamlessly add new displays of their own to the existing views, nor does it facilitate modifications of a particular displays.

2.2 Pablo Performance Analysis Environment

Compoents Item	TraceLibrary	SvPablo	Analysis GUI
Languages	Language-inde-pendent	ANSI C, HPF	
Platforms	MPICH on Convex Exemplar, Intel Paragon, Unix work stations, IBM SP	SGI running IRIX 6 Sun running Solaris	Built and tested on Sun Solaris 2.5.1
Version			
URL	http://www-pablo.cs.uiuc.edu/Project/ResearchProjects.htm		

Functionality

Pablo [30], developed at the University of Illinois, is the most complex toolkit reviewed here. It is designed for constructing the performance analysis environment of parallel programs, and therefore, most of its functionality is provided to help the user address the problem of bottleneck identification, performance evaluation and tuning.

Pablo 5.0 consists of several separate components for instrumentation, event tracing and performance data analysis.

- Trace library
- SvPablo (Source View Pablo)
- Analysis GUI

Pablo also provides facilities to convert trace files produced by other trace libraries, such as PICL (Portable Instrumentation Communication Library) [31] and AIMS, to SDDF (Self-Defining Data Format) [32], so that these trace files can be analyzed using the Pablo analysis tools.

1) Source code instrumentation

Instrumentation can be done by manually inserting calls to TraceLibrary routines into the application source code, or interactively by using SvPablo, a graphical interface for instrumenting application source and browsing dynamic performance data. SvPablo only supports interactive instrumentation of C, Fortran 77, and Fortran 90, and automatic instrumentation of HPF programs.

The instrumentation library consists of a basic trace library with extensions for procedure tracing, loop tracing, I/O tracing, as well as NX and MPI message-passing tracing. The basic trace library supports counting, interval timing and event tracing, among which the first two are only used to capture data for performance measurements like the counting of a specific event or the time spent in particular code fragments.

2) Trace file generation

After compiling and linking with the TraceLibrary routines, the application executable can be run in the usual manner. Each process outputs a trace file, which can be merged later using an SDDF utility.

What really justifies a mention of Pablo is its self-describing data format (SDDF) used for the trace files. In addition, interoperation between the various Pablo components is also based on it. SDDF is a trace description language that specifies both the structures of data records and data record instances. Because SDDF does not define a fixed set of event types, nor does it specify the size, data types or semantics of a particular event, new event types relevant to specific applications or architecture can be added without modifying the Pablo trace library.

3) Trace file analysis

SDDF trace files can be analyzed using the Pablo Analysis GUI. The Pablo Analysis GUI supports so-called graphical programming models. The user can specify the desired data transformations and presentations by graphically connecting analysis and data display modules and then selecting which trace data should be processed by each data analysis module. The developers of the Pablo Analysis GUI claim that this design provides the desired flexibility and extensibility. However, it introduces greater complexity in configuring the graphs. The Analysis GUI provides performance views using typical graphics such as bar graphs, bubble charts, strip charts, kiviat diagrams and matrix displays. Most of these displays represent statistical information such as processor utilization and message traffic. Although these types of views and displays are helpful for performance evaluation and tuning, they are very limited to illustrate the semantics of a program in its application domain, on which our visualization will focus.

The SvPablo GUI only allows the user to view performance summary statistics for each instrumented routine and loop, such as number of calls made to the routine and the cumulative time for the routine. Detailed statistical information about a routine can be seen by clicking on it with the mouse buttons.

Usability

The Pablo documentation is thorough. However, it would be helpful to have an up-to-date overview document that describes how the various components relate to one another and how they can be used together.

Brief installation instructions are provided in the README file for each component. The major installation task consists of editing the top-level Makefile with system-specific definitions. All the components except the TraceLibrary require GNU Make 3.75. For Analysis GUI, we had to edit some source code files. The Analysis GUI and SvPablo include a tutorial in their distribution that leads the user through the steps needed to build an analysis graph with a series of examples.

SvPablo is not very difficult to use. The Pablo Analysis GUI, on the other hand, has a steep learning curve for learning how to construct the analysis graphs. It would be helpful if this tool provided example graphs for a particular application area or programming model to get the user started.

Extensibility

Pablo provides possibilities for the user to develop and add new data analysis modules and theoretically, various types of view could be constructed using its graphical programming model. However, adding application-specific displays to Pablo is definitely a time-consuming task because it requires X-window System programming, which is rather complex and has a steep learning curve. Furthermore, most of the display facilities provided by Pablo are for presentation of performance data, and they are not very useful to build our visualization.

2.3 XPVM

URL	http://www.netlib.org/pvm3/xpvm/
Version	1.2.5
Platforms	Intel hypercubes Sun, SGI, and HP work stations SGI Challenge

Functionality

XPVM [33] is an X-window based graphical console and monitor for PVM. It provides a graphical interface to the PVM console commands and information, along with several animated views to monitor the execution of PVM programs. XPVM is written in C and Tcl/Tk [35]. It runs like another PVM task and has to execute on one of the machines comprising the virtual machine.

We restrict ourselves here only to the functionality of XPVM as a monitor.

1) Trace file generation

XPVM itself does not provide instrumentation functionality, nor does it provide a trace library. All the views and functionality in XPVM are driven by information captured via the PVM tracing facility. No annotation of the user program is necessary to use XPVM, as the PVM distributions version 3.3.0 or higher include built-in instrumentation for tracing user applications.

Any tasks spawned from XPVM automatically send back trace events. These trace events can be used either for postmortem trace visualization or for on-line performance monitoring. The trace files for XPVM are generated in self-defining data format, or SDDF, as designed for representing trace events in the Pablo system. The XPVM group argues the reason for choosing SDDF over the PICL (Portable Instrumentation Communication Library) format is that PICL was not sufficiently flexible to represent all the trace information generated by PVM.

2) Trace file analysis

XPVM provides several different views for monitoring program execution, among which the Space-Time view provides a similar visualization to the process-time diagram.

The Space-Time view shows the status of individual tasks as they execute across all hosts. Each task is represented by a horizontal bar along a common time axis, where the

color of the bar indicates the state of the task, namely computing, overhead, and waiting. Communication activity among tasks is shown using lines drawn between the task bars at the corresponding message send and receive times.

The Space-Time view supports several nice facilities. It has a VCR-like trace play control, which provides "Play", "Stop", "Pause" and "Single Step" buttons to control the display. It also facilitates multiple views, Space-Time view queries and Space-Time view zooming.

Usability

XPVM is well documented. The User's Guide explains the various features available. Each distribution of XPVM includes a README file that describes how to install that version of the XPVM software and a general installation procedure is provided in the User's Guide. In general, the source distribution can be compiled on any Unix platform which supports X-window and Tcl/Tk. In order to build XPVM, PVM 3.3.0 or later is also needed. The major installation task consists of editing the top-level Makefile.aimk with system specific definitions and of modifying the user's .cshrc file to set up the XPVM environment. The installation is easier than the two tools discussed above due to fewer modifications needed for the Makefile. XPVM has an intuitive interface that is easy to use.

Extensibility

XPVM is written using Tcl/Tk, allowing extensibility to include a variety of views. Although modifying an existing display or adding an application-specific view will not be an easy task and requires Tcl/Tk programming, at least the programmers can concentrate on only those portions of the graphics programming that are relevant to their application, taking advantage of all the other necessary facilities supported by XPVM. A further consideration of the size and complexity of XPVM and Tcl/Tk will find that XPVM is easier to be adapted than either AIMS or Pablo.

2.4 ParaGraph

URL	http://www.netlib.org/paragraph/index.html
Version	6/8/94
Platforms	Sun, IBM, SGI, HP and DEC work stations

Functionality

ParaGraph [36] is a graphical display system for visualizing the behavior and performance of message-passing parallel programs. ParaGraph opted for a dynamic animation approach to provide postmortem trace visualization. It is written in C and based on the X-window system.

1) Trace file generation

Similar to XPVM, ParaGraph only supports data analysis and graphical display. It takes input data from execution trace files produced by PICL [31], a Portable Instrumentation Communication Library, which allows the user to produce trace data automatically. This approach minimizes the need for explicit instrumentation by having the distributed system automatically generate the annotation whenever possible. However, PICL only provides instrumentation for those generic communication primitives that it supports. Since ParaGraph relies on PICL only for its input data, it could work well with other data sources in the same format.

2) Trace file analysis

ParaGraph provides a variety of views. Its Spacetime Diagram provides similar display and functionality to the OverView of AIMS.

In the Spacetime Diagram, the processor number is displayed on the vertical axis and time on the horizontal, which scrolls as time proceeds. The processor status, namely busy, doing overhead and idle, is indicated by horizontal lines. The line is blank if the corresponding processor is idle; it will be solid in the other cases. Messages between processors are depicted by slanted lines between the sending and receiving processor status lines. Each message line is drawn when the called process starts to receive. The communication lines are color-coded according to message size or message type. Like AIMS, although the time range of the timeline can be adjusted, the current version does not support scrolling backward or zooming.

ParaGraph has some nice features, including multiple views, trace file playback control, and display motion control. The user can view as many of the displays as will fit on the screen simultaneously, each giving a distinct perspective based on the same trace data. The graphical animation can be interrupted for detailed study by use of the pause/ resume and single-step button. The speed with which the data are viewed can be slowed down dynamically.

Usability

ParaGraph's major documentation includes a README file and a User's Guide. Although the installation is not complex, it would be helpful, however, to include a brief instruction on installation in either of these two files. When building ParaGraph, we had to figure out the installation procedure and modify a top-level Makefile.

The User's Guide explains the various features available. It also points out some limitations of ParaGraph. According to the User's Guide, the most fundamental restriction in the parallel-programming model supported by ParaGraph is the assumption that there is only one user process per processor. In addition, ParaGraph does not explicitly support fully synchronous or full asynchronous communication. Currently, the inter-processor communication model supported by ParaGraph is that the sending process never blocks, but the receiving process will block if the message is not available when it attempts to receive.

The interactive user interface of ParaGraph is easy to use and several sample trace files come with the source code for demonstrating the functionality of ParaGraph.

Extensibility

Paragraph allows the users to add application-specific displays that can be selected from a sub-menu and viewed along with the usual generic displays. ParaGraph has calls at appropriate points to routines that provide initialization, data input, event handling and drawing for application-specific displays. In addition, the tracemarks events of PICL can be used to supply additional events for an application-specific display.

However, writing the necessary routines to support application-specific displays is still a nontrivial task that requires a general knowledge of X-window System programming. To help the users who may wish to develop application-specific displays, several example routines are distributed along with the source code for ParaGraph.

2.5 PVanim

URL	http://www.cc.gatech.edu/gvu/softviz/parviz/pvanim/pvanim.html
Version	1.1
Platforms	Sun, IBM, SGI, HP and DEC work stations

Functionality

The PVanim [37], developed by the GVU center of Georgia Institute of Technology, is a toolkit for producing animated program visualizations of the executions of PVM

applications. A prominent feature of PVanim is its support for application-specific visualization. Unlike performance visualization toolkits, PVanim focuses on visualizations of the actual execution and correctness of a program. The major limitation of the current version is that they do not work well for visualizing extremely long program traces. However, it is not very clear from the available documentation what data volume would be manageable.

PVanim has two main components: tracing library and Polka visualization library

1) Source code instrumentation and trace file generation

PVanim provides its own tracing library to create trace records rather than utilizing the PVM tracing facility. The PVanim tracing library uses macro wrappers to add its tracing to the PVM communication primitives for C or C++ programs. The user adds a header file that redefines communications primitives to be PVanim stubs, which perform the tracing first and then call the appropriate communication primitive. In addition to providing instrumentation for generic communication primitives, the tracing library also supports user-defined event tracing by providing a pvanim_print () routine that prepends the user-written string with an event identifier, a process identifier and time stamps. The trace file format is developed by GUV and used by all other components with visualization built on Polka.

Two modifications are required for instrumenting the PVM programs. Modified applications should be recompiled and linked with the PVanim tracing library. The trace

file generated by the processes when the program is run will be converted to standard PVanim trace file format by the PVanimSort command.

2) Trace file analysis

The PVanim visualization library, based on the using Polka animation toolkit [38] [39], provides a set of views that show different perspectives of an application. Its Causality view is an adaptation of the process-time diagram.

In the Causality view, the Y-axis is labelled with process identifiers and the X-axis labelled with Lamport logic clock value. When a message is sent, a circle representing the message appears at the appropriate time coordinate. Varying circle radius is used to denote message size and the color of the circle is to indicate the message type. During the delivery of the message, an arrow grows from the coordinates of the message sender to the receiver. Simultaneously, the circle moves along this path and then disappears. The advantage of this approach is that it provides a sense of motion and change other than that reflected by the temporal relationship. Different from XPVM, AIMS and ParaGraph, PVanim does not use bar or trace line to depict the process status.

Despite the difference in the principles of building the display, PVanim is quite similar to XPVM in functionality. The Polka Control Panel has VCR-like controls that allow the user to control playback of a trace file. The control panel allows the user to change the speed of the animation, step through it, or pause the animation. Graphical objects in a PVanim view can be queried by clicking on them to obtain more information.

Furthermore, the user can scroll along the vertical and horizontal direction, as well as zoom in or out. It would be convenient, however, if PvAnim provides a reset feature that clears all displays and returns to the beginning of the current trace file, rather than having to reload a trace to run it again.

Usability

The installation of PVanim is easy and includes two parts: the tracing library and the display library. A brief installation guide can be found in the top level README file. The PVanim tracing library includes documentation with its distribution that describes how to install the library, what modification should be made for instrumentation and how to correctly compile the source code to produce the trace file.

The documentation of Polka includes a README file, Polka Animation Designer's Package, and a paper that explains clearly the methodology adopted by Polka to provide an application specific visualization. The README file introduces briefly how to view the trace file with Polka. The Designer's Package gives a detailed description how to develop a visualization of a user's own design with Polka. The usage of Polka is self-explanatory and easy to learn.

Extensibility

Polka is a software toolkit designed to facilitate application-specific animated visualization. It provides the developers with an object-oriented design model by

supporting a set of primary classes that developers can instantiate, subclass and manipulate to design their own visualization. Polka provides two critical features:

- It provides primitives for creating animations and visualizations
- It provides high-level graphical objects, such as lines, rectangles, circles, polygons, and text whose color, location and size can be changed

Therefore, it is very easy to change the color of the trace lines using Polka. Moreover, it will not take much effort to add some symbols like triangles, diamonds, circles or rectangles since Polka provides a Polygon class, Circle class and Rectangle Class to represent them.

Polka is implemented in C++ on top of UNIX with X11 system or on Windows 95 using MS Visual C++5.0. Although it requires some effort to learn the methodology of its animation paradigm, it is much easier to learn than X-window programming.

In addition to the extensibility of Polka, as discussed above, the PVanim tracing library also facilitates user-defined event tracing.

2.6 Summary

Although existing visualization tools do not directly facilitate the understanding of code mobility and most of them are used for PVM or other distributed/parallel systems

based on message-passing paradigm, the principles and mechanisms developed by them can help build up our own visualization.

Among the tools that we reviewed, XPVM, ParaGraph and Polka provide their equivalent process-time diagram, which can be adapted as our graphical representation of the program behavior. The reason for choosing a process-time diagram is that it gives a compact view of the event history and places more emphasis on temporal relationships. Process-time diagrams will be explained in a greater detail in the following section.

The facilities provided by those tools, such as trace file playback control, information query as well as view scrolling and zooming, can also be adopted to allow the user to quickly focus on particular concerns or get more detailed information of a specific view.

SDDF, self-describing data format, developed by the Pablo group is very flexible to represent trace information and can be used for our trace file as well. The instrumentation approach used by ParaGraph and PVanim can inspire us to provide a way to instrument the program with little source code modification. Although XPVM is tightly bound with PVM, the mechanism used to drive the on-line analysis and display can be applied to our visualization.

Chapter 3 Mobile Code Technology and an Agent-Enhanced ORB-Voyager

Despite the widespread interest in mobile code technology and applications, this research area is new and still immature. There are confusion and disagreement about some concepts, abstractions, and terms in the domain of code mobility. This chapter gives a brief review of mobile code technology and an introduction of our target environment, Voyager. It serves two purposes: one is to clarify what a particular term means in the context of this thesis; another is to help identify the characteristics of mobile code technology so that we can understand what needs to be visualized.

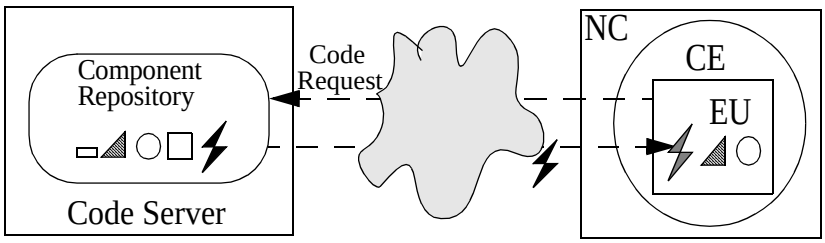
3.1 Mobile Code Technology

Code Mobility is not a new concept. In the past, several mechanisms and facilities have been designed and implemented in order to move code among the nodes of a network, such as remote batch job submission [40] and file downloading. A more structured approach that has been followed is the so-called process migration, the transfer of an active process from the machine where it is running to a different one [41]. Migration facilities, most of which support transparent process migration, are introduced at the operating system level to achieve load balancing.

More recently, an innovative technology called mobile code technology has been developed to provide enhanced code mobility, fostered by a new generation of programming languages, such as Java and telescript, and systems, often referred to as mobile code systems (MCSs). In MCSs, mobile code does not bind statically to one host, rather it can migrate to other hosts to provide services.

3.1.1 Mobile Code Paradigms

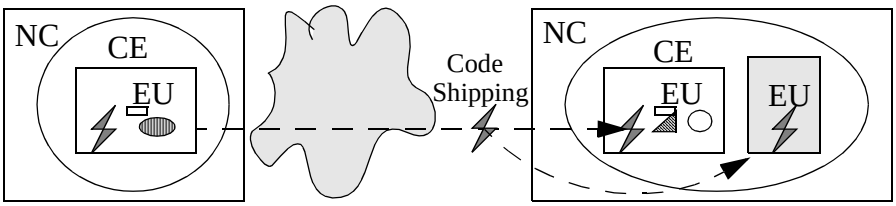
Existing MCSs supports code mobility in different ways. According to [42], the approaches followed to build MCSs can be classified into three main design paradigms, namely code on demand (COD), remote evaluation (REV), and mobile agent (MA). They are depicted in Fig. 3.1, Fig. 3.2 and Fig. 3.3, respectively. These paradigms are characterized by the location of components before and after the execution of the service, by the computational component which is responsible for execution of code, and by the location where the computation actually takes place.



CE: Computational Environment EU: Execution Unit NC: Network component

Fig. 3.1: Code on demand

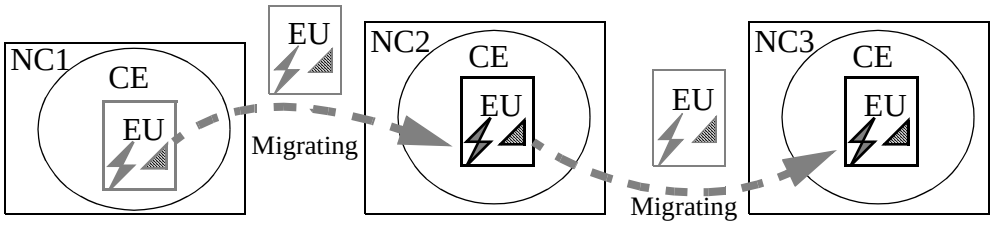
Code on Demand: As shown in Fig. 3.1, an execution unit, which is the run-time view of a program such as a process or an individual thread, can fetch the code from a remote node to be dynamically linked and eventually executed using the local resources. A particular application of this paradigm is the use of Java applets in web browsers.



CE: Computational Environment EU: Execution Unite NC: Network component

Fig. 3.2: Remote evaluation

Remote Evaluation: As shown in Fig. 3.2, the service code can be shipped to a remote computational environment where it will be executed. This encompasses two cases: either the EU in the destination node is created from scratch to run the incoming code or a pre-existing EU links the incoming code dynamically and executes it.



CE: Computational Environment EU: Execution Unite NC: Network component

Fig. 3.3: Mobile agent

Mobile Agent: The term "agent" has many different meanings according to the research domains where it is used. In the scope of this thesis, a mobile agent is basically an execution unit (EU) that, while in execution at a given network node, is able to migrate to a different node and resume its execution seamlessly there. The time and destination for migration can be determined autonomously by the migrating agent itself or by a different EU.

Among these paradigms, the REV and MA paradigms allow the execution of code on a remote node, using the resources there. On the other hand, the COD paradigm enables computational components to retrieve code from other remote components, providing a flexible way to extend dynamically their behavior and the types of interaction they support.

Code mobility affords new opportunities for the distribution of processing and control in the network. There are several areas that may benefit from appropriate use of code mobility:

- If a software component needs to exchange a large number of messages with another component in a remote program, they can move closer to each other to reduce network traffic and increase throughput.
- Migrating software components from their working nodes that have experienced a partial failure can improve the fault tolerance of the application system.

- A server program can move service code to a consumer device so that the device can be served even after it is disconnected from the network.
- A software component can be delegated a task and sent to a remote device to perform its operation. This can be used for network management applications.
- Code mobility can be used for on-line extension of application functionality or software upgrades [43].
- As shown in Fig. 3.4, server programs can move their services to a better execution environment to balance the load or to avoid bottlenecks and long latency.

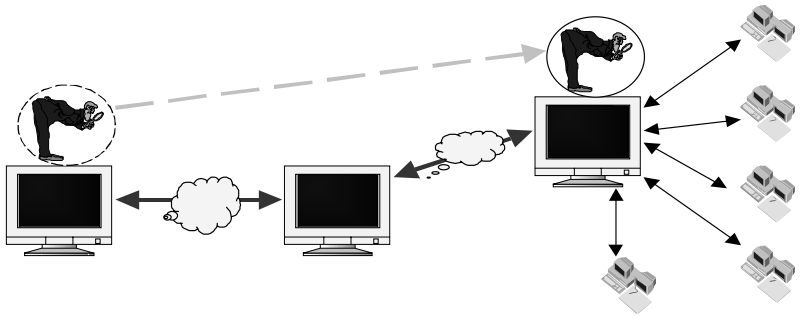


Fig. 3.4: Service migration

3.1.2 Mobile Objects

When we restrict our discussion about code mobility to the context of the object-oriented paradigm, “code” intuitively refers to classes or objects. Since objects are actually the active entities, we will not place our attention on issues related to class mobility, such as class downloading or shipping, but rather to mobile objects. Although there is no com-

monly agreed definition of what is a mobile object, in the context of this thesis, a mobile object has the following characteristics:

- It is a distributed object that not only has behavior and state but also has location and an unique identity

  From the perspective of programmers, a mobile object can be uniquely identified even after its migration. The location of a mobile object is not hidden from the programmers, but rather applicants are location-aware and may take actions based on such knowledge.

- It has the capability to move independently from one network component to another during its lifetime.

  The programmers can determine objects locations and may request explicitly the migration of a object to a particular node

According to those characteristics that distinguish a mobile object from other regular objects, we can define a set of events that are of interest to the mobile object during its life time, and that are helpful in understanding its dynamic behavior.

Creation: Analogous to the constructor of an object, creation is the starting point of an object's life. It can be initiated either by a mobile object or a stationary object.

Disposal: Analogous to the destruction of an object.

Arrival: Signals that the mobile object has successfully arrived at its new location.

Dispatch: Signals the mobile object to prepare for departure to a new location. This event can be generated explicitly by the mobile object itself upon requesting to move, or it can be triggered by another object that has initiated the migration.

There are some other likely interesting events, such as those related to interaction with other objects. These events are not specific to code mobility and such events are typically covered by previous visualization systems [44] [45]. As the goal of this thesis is to provide a sensible visualization scheme for the purpose of understanding code mobility, our efforts will not focus on these events, but rather on those associated with creation, disposal, arrival and dispatch.

3.2 Java and An Agent-Enhanced ORB-Voyager

The thesis is based on programs using Voyager as target environment. The reason for choosing Voyager is that it supports rich code mobility. Another reason is its popularity. Voyager is an agent-enhanced ORB, which merges distributed object technology and mobile code technology together. Therefore, in addition to those capabilities found in most other ORBs, it provides a host of other features for supporting code mobility. Voyager is 100% Java, and is designed to use the Java language object model [11]. In the rest

of this chapter, we will give a brief introduction of the mechanisms and features that Java and Voyager support for code mobility.

3.2.1 Java Architecture

Developed by Sun Microsystems, Java technology [14] has triggered most of the attention and expectations on code mobility. We refer to Java as a technology, rather than only as another programming language. Java architecture consists of four parts [46]:

1) The Java programming language

2) The Java class file

   The Java class file contains byte codes, the machine language of the Java Virtual Machine. It is very compact and can be transmitted over networks quickly

3) The Java application programming interface (Java API)

   The Java API is a set of run-time libraries that provide a standard way to access the system resources of a host computer.

4) The Java virtual machine (JVM)

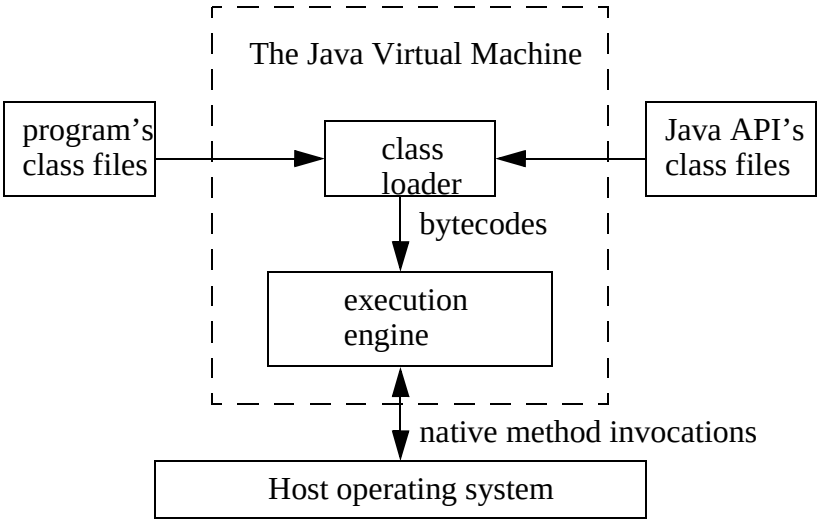


Fig. 3.5: Java virtual machine

As shown in Fig. 3.5, the JVM contains a class loader and an execution engine. The class loader is responsible for loading class files that are actually needed by a running program.

The JVM and Java API form the Java run-time system on which all Java programs are compiled and executed. Java provides many features, but here we restrict our discussion only to the Java mechanisms that supports code mobility.

The Java compiler translates Java source programs into an intermediate, platform independent language called Java Byte Code. The byte code is interpreted by the JVM. Java provides a programmable mechanism, the class loader, to retrieve and link dynamically classes in a running JVM. The class loader is invoked by the JVM run-time

when the code currently in execution contains an unresolved class name. The class loader actually retrieves the corresponding class, possibly from a remote host, and then loads the class into the JVM. At this point, the corresponding code is executed. In addition, class downloading and linking may be triggered explicitly by the application, independently of the need to execute the class code. Therefore Java supports code mobility using mechanisms for fetching code fragments. The code loaded is always executed from scratch and has neither execution state nor bindings to resources at the remote host.

3.2.2 Code Mobility Supported by Voyager

Voyager takes a different strategy to facilitate code mobility. It provides a middleware layer supporting code mobility for higher level layers. It extends Java by adding capabilities to support REV paradigm. In Voyager, almost all serializable objects have the potential to move between different locations. Voyager implements mobility in the following ways:

- Shipping an object to a remote location

  An object can be moved explicitly to a new location by obtaining access to the Mobility facet and invoking the MoveTo () operation on that facet.

- Mobile agent

  Voyager claims that it supports the notion of mobile agent. As mentioned before, there is no commonly agreed definition for this term. The mobile agent approach in Voyager is different from the one we discussed earlier. An object can be treated

as an agent and moved explicitly to a new location by obtaining access to the Agent facet and invoking the MoveTo () operation on it. In this way, the code of an agent, including its non-transient parts, is shipped to the remote location to instantiate a new thread, which means that the agent is re-executed from scratch after migration, although it retains the value of its object attributes which are used to provide initial state for its computation. This approach is similar to the first one except that a mobile agent is an active thread of Java.

- Passed explicitly as parameter

  An object can be serialized and passed to the remote virtual machine as a parameter of remote invocations on a Voyager object. We would rather refer to this approach as passing an object by value. It is a form of class mobility, since the shipped code is actually used as data resource for a remote invocation like any other data types.

- Remotely construct objects

  Voyager enables an object to be constructed on different hosts through a remote invocation. Most other infrastructures do not allow objects to be remotely constructed. A proxy object reference is returned from the construction process so that the local process can use the newly constructed object right away.

Chapter 4 Visualizing the Execution of Object-Oriented Mobile Code Applications

As discussed in the previous chapters, when the execution of applications are of concern, an event-based approach is typically employed. Events represent some activities that cause changes in the state of the execution and are considered to occur at an instance in time. They are defined according to a particular aspect in which a specific observer is interested. This thesis focuses on the understanding of code mobility, consequently, those aforementioned events of a mobile object in its lifecycle, namely creation, disposal, arrival and dispatch, are our interesting events.

In this chapter, we attempt to address the question of how to visualize. In other words, ideally, how those interesting events might be graphically presented. We first introduce an adaptation of process-time diagram, which is adopted in the Poet visualization system [47] [48] [49], then we explain, in more detail, how to apply a similar graphical representation in our visualization.

4.1 Process-Time Diagrams

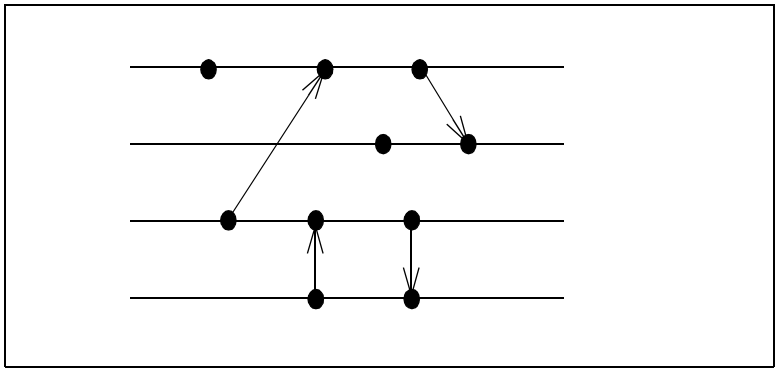


Fig. 4.1: Process-time diagram

The graphical representation of the behavior of programs can be carried out either by animation or by process-time diagrams. McDowell and Helmbold once suggested that the ideal debugging or monitoring tool for complex parallel programs should support both animation and process-time displays [50].

Animation is good at capturing a sense of motion and change, and thus may provide an intuitive feeling for the dynamic behavior of the program. However, temporal relationships are difficult to analyze using animation, because the granularity of temporal relationships shown in a single frame of animation is limited. An alternative to animation is the process-time diagram [51] [57], which gives a compact view of the event history, and places much more emphasis on the temporal relationships between entities.

As shown in Fig. 4.1, in the progress-time diagram, each entity, such as process, task, thread, object and semaphore, is viewed as a sequence of atomic events and represented by horizontal lines, called traces, with time progressing from left to right. Events constitute the lowest level of observable behavior in a distributed execution, such as sending/receiving a message, or the creation/termination of a process.

In reality, no global clock exists and events can only be ordered partially. The basic relation between primitive events is the happened-before relation introduced by Lamport [51]. The precedence relation ($\rightarrow$) is determined according to the following rules:

1) If a and b are two events in the same entity, and a comes before b, then $a \rightarrow b$
2) If a is an asynchronous send event and b the corresponding receive, then $a \rightarrow b$
3) If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
4) If $a \not\rightarrow b$ and $b \not\rightarrow a$, then a and b are concurrent

The visualization focuses on interactions between entities instead of the internal activities of each entity. A symbol, such as an open/solid circle/square, represents important events in each entity. Interactions, such as communication, between entities are shown by arrows, which are drawn from the appropriate event in the sending entity to the one in the receiving entity.

4.2 Graphical Representations of Interesting Events

In our display, we use symbols to represent events.

- Solid squares represent object creation events.
- Open squares depict destruction events.
- Solid circles are applied to object-dispatch events.
- Solid diamonds represent object-arrival events.

The trace lines can be invisible or drawn as solid or dashed according to the state of the entities they represent. Before the creation of a entity and after its destruction, the line is invisible. During the entity's blocked period, the trace line will be dashed. Except for these two cases, the trace line will be solid to depict an active entity.

Arrows that depict interactions between entities are vertical for synchronous interaction and sloping for asynchronous one.

4.2.1 Object Creation and Disposal

As shown in Fig. 4.2, an object is drawn as a trace line. Since the object that invokes the creation will be suspended until the creation operation returns, object creation involves two synchronized events from two different trace lines: a creating event on the

creator trace and an object-created event on the new trace line. Object destruction is a unary event with no partner.

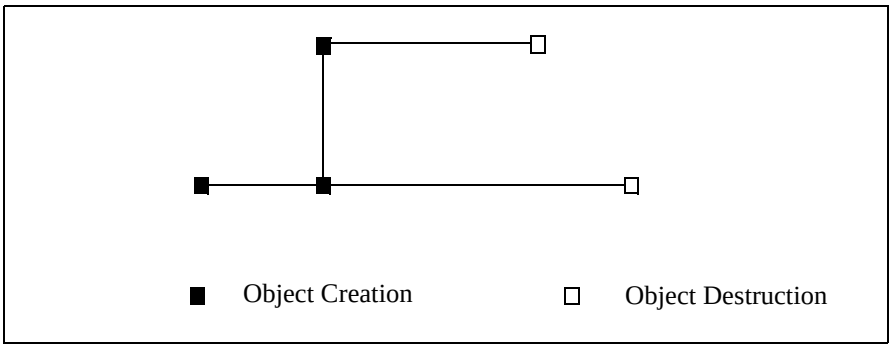


Fig. 4.2: Object creation/destruction

4.2.2 Code Mobility

A search of recent literature on program visualization reveals that no research explores the area of code mobility, which means that no existing toolkits provide visualization facilities for such information as when the object migrates to which node, when it is deleted, and how it interacts with other objects. In this thesis, a solution to depict code mobility in the process-time diagram is proposed.

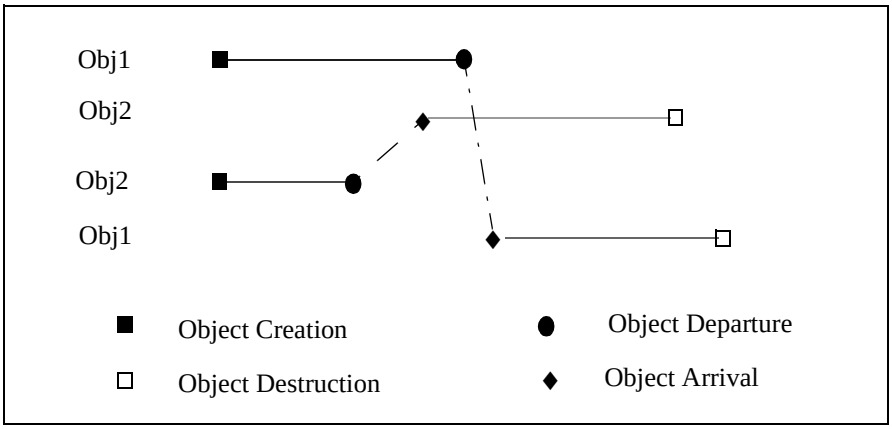


Fig. 4.3: Code mobility depiction-approach 1

There are some likely ways to represent an object move from node to node. A first possibility, shown in Fig. 4.3, is to move the trace line when the corresponding object migrates. This sort of naive display has some obvious problems. First, it dramatically increases the display space needed. As a new trace line has to be drawn each time the object moves, the vertical space needed is in proportion to the number of migrations. This type of display also needs to adjust the layout when a new line is placed and it makes the diagram rather cluttered even if we only attempt to depict a small number of objects. Therefore, the notion of changing the position of the trace line in this type of display is unpractical.

Since the object that invokes the migration of another will not be suspended after the invocation in Voyager, the migration invocation is represented by a sloping line hereafter.

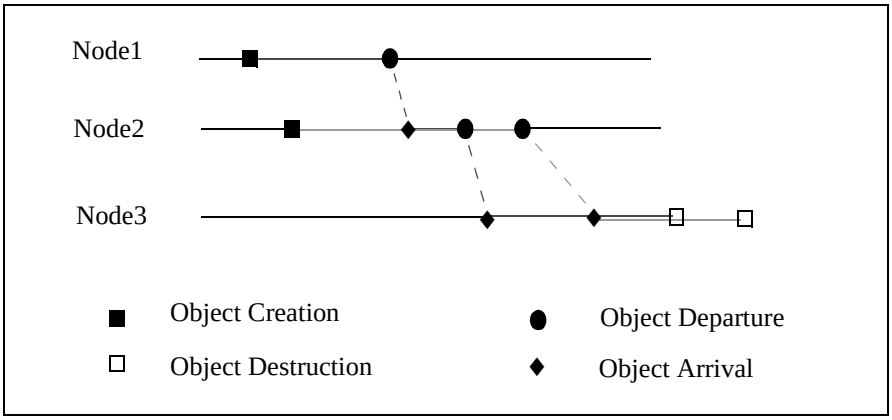


Fig. 4.4: Code mobility depiction-approach 2

Another way, as shown in Fig. 4.4, is to take advantage of color. Color is an important graphical attribute among others like size, shape, gray level, orientation, and texture. It has the potential to communicate a great deal of information efficiently. One potential drawback of using color is that it requires color monitor. However, computers that only support monochrome are rarely used nowdays and they have problem anyways in running other applicaitons that require color. In this approach, color is used to identify different objects with mobility. In this display, the trace lines represent servers on different or the same nodes that support mobile objects. Each mobile object is colored differently. More precisely speaking, each path from the object creation to the object destruction, following the migration message arrows and trace lines in the time direction represents the life cycle of a mobile object and is depicted with one color. The advantage of this approach is that the mobile object occupies the same vertical display space as other objects without mobility. However, this simple solution can not serve very large

numbers of objects because of limitations of the color dimension. Another problem is that some potential for confusion exists when more than one mobile object appear on the same trace line. In this type of display, the trace line represents mobile object servers that can be considered as a cluster of mobile objects. If the behavior the cluster involves visible concurrency, the actual precedence is likely distorted by placing concurrent events as if one preceded the other.

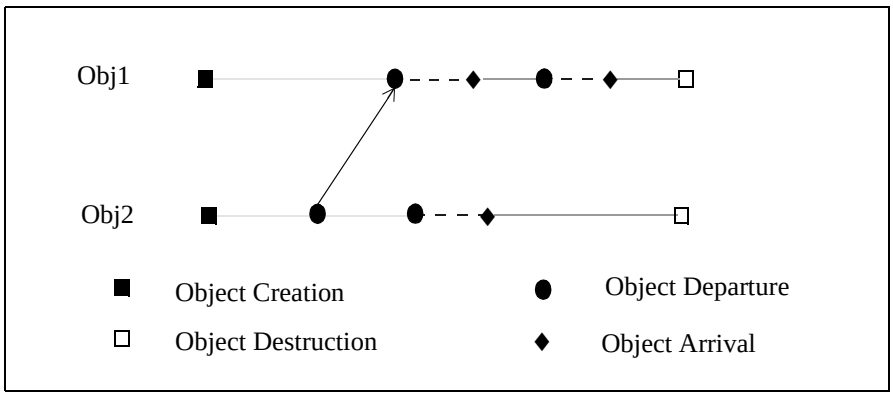


Fig. 4.5: Code mobility depiction-approach 3

To resolve the problems inherent in those two types of display, we propose a third one shown in Fig. 4.5. This solution meets the requirements that the display explicitly indicates the mobility of the objects, minimizes the display space needed, is inherently scalable, and could correctly reflect concurrency. In this display, the trace lines represent objects. It also adopts color to encode information about object mobility. However, the color used here is to depict the change of location, not to identify the object itself. In other words, the trace line will consist of segments with different colors that depict

different nodes where the mobile object locates. The color changes when the object arrival event occurs. In this way, the mobile objects occupy the same display space as those without mobility and the object migration is explicitly depicted without much cost. Since the trace lines represent objects, this display has no extra problem in depicting concurrency. Furthermore, we use colors to indicate the change of location, not necessarily to identify the location; the number of objects that this display could serve will not be restricted by the color dimension. In general, if the number of nodes that mobile objects move to is no more than the types of color that the display supports, we can also represent the nodes using color-coding. If this is not the case, we just change the color of the segment on the trace line alternately to show the migration of the object. We can also provide a facility to allow the user to get location information by clicking on the individual line segment. Therefore, this type of display will not impose extra limits on the number of objects it could depict.

4.3 Visualization Objectives

Based on the challenges that exist in developing and understanding distributed object-oriented programs with code mobility, this section discusses the objectives that a framework for the visualization of such programs may need to achieve. These objectives, actually, have for quite a while been recognized as some of the most important open problems within all software visualization research.

1) Minimal visualization effort

The efforts on the software developer for visualizing an application should be minimal. This implies that the developer shouldn’t have to add much code to the application for performing the visualization.

2) Visualization is desirable to be performed both on-line and postmortem.

3) Little user intervention

The visualization generating should require little or no programmer intervention. It is suggested that a good visualization should "guide", which means that the visualization leads to discover things still unknown, not "rationalize", which means that it illustrates things that people have already known [52]. If the visualization tool can provide interesting and informative displays without detailed control by the user, it has the potential to guide the user.

4) Presentation of the "right" things

The visualization should not show all the low-level details that a user is unaware of and would have to work hard to understand. Rather, the visualization should present a high-level abstraction of the most important aspects of a program.

5) Scalability

The visualization should not be restricted to present only small, laboratory programs and systems. Rather, the visualization must be applicable to large systems. [52].

6) Minimal probe effect

Efforts should be made to minimize the perturbation incurred by the visualization instrumentation as much as possible.

7) Quick focus on particular concerns

The visualization should provide users with some "rewinding" and navigation functionality to allow users to focus on their particular concerns quickly. Detailed information must be displayed when requested.

Chapter 5 Event Data Collection Subsystem of CMVS

As shown in Fig. 5.1, most visualization tools divide the visualization process into the following steps:

1) Event data collection: Collect the interesting events from the application that is visualized.

2) Event data processing and storage: The collected events are processed for visualization and stored in a queue, or a log file.

3) Display: The visualization events are translated into graphical primitives (i.e. shape, color, position, etc.) and displayed.

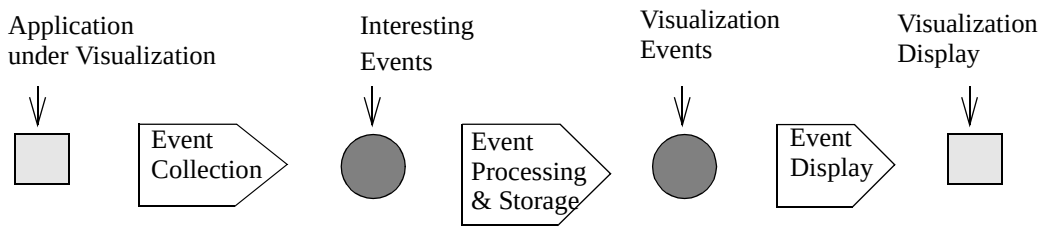


Fig. 5.1: The visualization process

We use these steps in the design and implementation of our tool, which is called CMVS (Code Mobility Visualization System). Conceptually, CMVS consists of two major components. One is for the event data collection and the other is for the event data processing and graphical display. Event data collection and visualization depend deeply on each other. Without proper data collection mechanism, no information exists to drive a visualization. On the other hand, without visualization, understanding the data derived can be tedious and complex.

In this chapter, the event data collection subsystem will be introduced. We first present an approach to event collection, including program instrumentation. Then we describe its architecture. Some design and implementation issues are also addressed. The chapter ends with a description of its functionality.

5.1 Program Monitoring

There are two different mechanisms for collecting event data:

1) Sampling

In sampling, a monitor task running concurrently with the program periodically observers the state of the program by accessing global data

2) Tracing

In tracing, all occurrences of interesting events are stored for a certain interval of time, typically for the duration of the application, and a sequence of event records, which are encoded instance of the action and its attributes, is generated. Each record typically includes the following:

- What action occurred (i.e. an event identifier)
- The time when the event occurred
- The location where the event occurred
- Any additional data that defines the event circumstances

These two approaches have their own advantages and disadvantages. Generally, sampling is less intrusive [53], but it is useful when only cumulative statistics is needed. Tracing potentially introduces more perturbation to the application. However, it provides more detailed information and hence provides more support for visualization. Consequently, this thesis solely focuses on tracing.

Event tracing relies on program instrumentation, which will be described in more detail in the following section.

5.1.1 Program Instrumentation

Program instrumentation involves the placement of small pieces of code into the operating system, the run-time system, or in the user's source program. The function of this code, referred to as probes, is to report some value to the component responsible for

aggregating the event data. In most cases, operating system and run-time environment are almost inaccessible in terms of code insertion. Besides, instrumentation at either of these levels is most unlikely to provide information about abstract, high-level events in the application program. In contrast, source code is easy to access and instrumentation at this level provides a reliable information about events at different abstract levels. Therefore, source program instrumentation is preferred over the other two and chosen for our system.

In order to gather the expected trace events we need to specify where in the source code the appropriate events are generated, and in most cases, it is necessary to modify the program, either by:

- Modifying source code to generate explicit calls to event log library routines
- Using wrapper class or method overriding to add tracing to the API of the run-time system

These two approaches have their own advantages and disadvantages. The first approach is straightforward and simple. But manual code insertion is error-prone and time consuming, especially when the source program is complicated and we have a number of events to trace. The second one is well-established and achievable with relatively less source code modifications. It is adapted from the macro wrappers of library primitives [31] [37] [54], which first perform the tracing operation and then call the desired routines. The disadvantage of this approach is that programmers have to follow a

set of rules in their programming and get used to these wrapper classes or overridden methods. In the rest of this section we will explain the approach applied in our system.

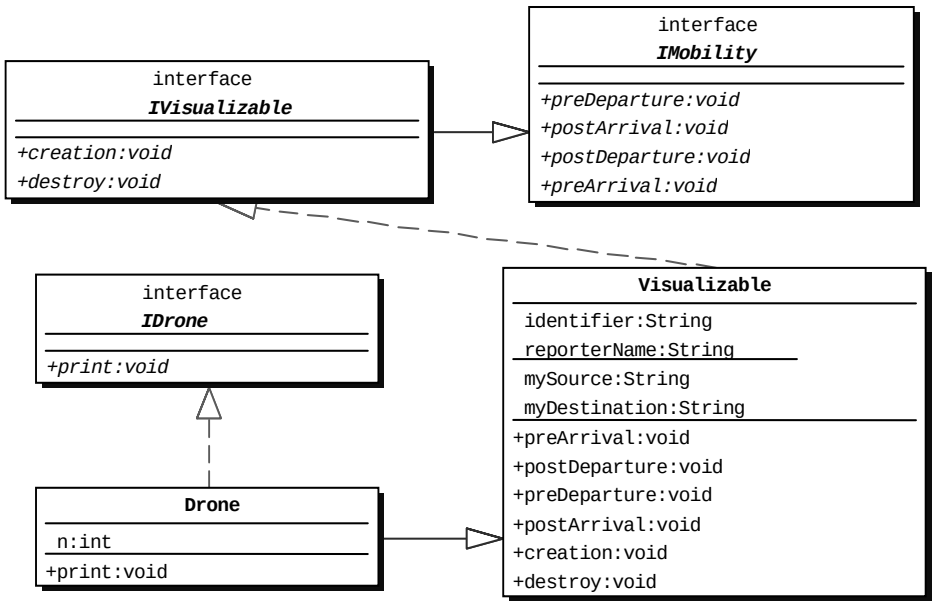


Fig. 5.2: Instrumentation by inheritance

Obviously, it is desirable to instrument the source code without or with minimal source code modifications. With this goal in mind, we provide an interface IVisualizable and a class Visualizable that implements the interface. Fig 5.2. depicts their relationship with other application classes in UML notion. In this class diagram, interface IMobility is provided by Voyager for object move notification. Interface IVisualizable extends this interface with two other methods, creation() and destroy(), which generate object creation and disposal events respectively. Interface IDrone and class Drone are provided by the application program that is visualized. The instances of class Drone will move from one

node to another over the network. Class Visualizable implements all the methods that will be called when an associated event happens. These methods generate corresponding trace events and report them to a local component that is responsible for collecting the data. In the user's program, any mobile objects that want to be visualized must inherit from class Visualizable. This approach provides a neat solution for both trace events generation and reporting. It allows instrumentation with little source code modification and the programmers do not need to know much detail about the event tracing.

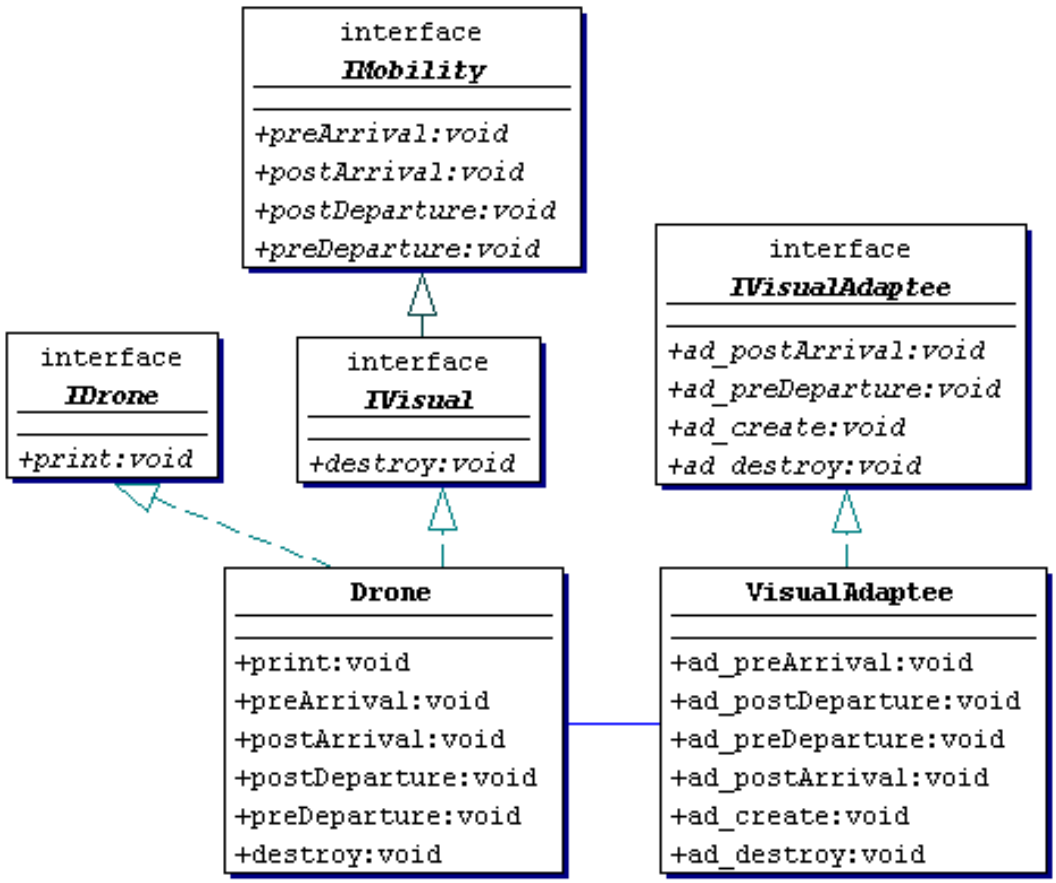


Fig. 5.3: Program instrumentation by applying the wrapper pattern

Since Java does not allow multiple inheritance, this approach has a potential problem when the mobile object class has to inherit from another class. Another limitation of Java is that every method should be written within a class, which means that we can not provide the user with a stand-alone method. The Adapter (Wrapper) pattern can be used to solve these problems. As shown in Fig 5.3, a UML class diagram, we provide interface IVisualAdaptee and class VisualAdaptee that implements the interface. This class defines methods with the same implementation as those defined in class Visualizble. It serves as an adaptee. Interface IVisual extends IMobility with one method: destroy(). The mobile object has to implement the IVisual interface and compose an instance of VisualAdaptee class. It acts as an adapter. A client invokes an operation on an adapter instance. In turn, the adapter calls the adaptee's corresponding operation that actually carries out the request.

The program instrumentation schemes introduced above are currently applied in CMVS by manually annotating the source code. Manual instrumentation is well established and adopted in several other visualization tools. To further relieve the user from the burden of modifying the source code, especially for rather large and complicated applications, automatic instrumentation can be considered in the future. However, it is by no means a easy job since we have to carefully analyze the program structure and class hierarchy to determine where to annotate the program. Moreover, there is no generic way to achieve this goal since automatic instrumentation will be source language dependent.

The approach that we use to instrument the program allows the user to select the class for annotation. If a class has quite a lot of objects but we are interested in some of them, we can consider to filter out those objects not of interest by setting a flag that indicates wheter the object is to be visualized.

Voyager facilitates move notification, which is a delegation-base event-handling model. If an object implements the IMobile interface, it will receive callbacks during the move. The IMobile interface declared the following callback methods in the order of being invoked:

- preDeparture(String source, String destination)

  This method is invoked on the original object at the source when it is about to move.

- preArrival()

  preArrival() is executed on the copy of the object at the destination when the migrating object arrives at its destination, but not yet ready for operations.

- postArrival()

  At the time when this method is invoked, the copy of the object at the destination becomes the real object, the object at the source becomes the stale object, and the move is deemed successful and cannot be aborted.

- postDeparture()

This method is invoked on the original stale object at the source to signal that the object has just been moved.

IVisualizable extends IMobile interface with two public methods:

- creation ()

  creation () is used to generate an object creation event. It has to be placed inside the constructor of the mobile object class or its superclass.

- destroy ()

  destroy () method is intended to generate mobile object disposal events. Unlike C and C++ that require programmers to explicitly release the memory allocated to an object, Java provides garbage-collection facility to do it automatically. There is no way for programmers to explicitly free an object, all they can do is to release all references to the object. Nevertheless, we have no way to know when any particular object will be garbage collected because we do not generally know how garbage collection will be performed inside a JVM. In some cases, the garbage collector will not run at all before the program exits if JVM does not lack memory. At first glance, it seems to be a solution to get destroy() invoked inside the finalizer, which is defined as a regular Java instance method named finalize(). Java specifications make the promise about finalizers that before reclaiming the memory occupied by an object that has a finalizer, the garbage collector will invoke the object's finalizer first [14]. However, we still can not predict when the finalizer

will run or if it will ever run. We tried this approach and got a surprising picture. Several disposal events belonging to the same mobile object were reported. The reason is that in Voyager, a mobile object leaves its stale copy to be garbage collected at the source after it successfully migrated. This kind of display is potentially misleading because we can not guarantee that all the copies distributed in the network nodes will be garbage collected during the interval of the visualization or the duration of the program execution. Since we lack tracing facility from both Java and Voyager run-time to know when an object will be freed or at least has no references to it, we have to address this issue from the application layer. In this thesis, we are mainly interested in the mobility of an object. As a result, if an object will not move anymore, from the perspective of visualization, we assume that this object is finished. In this way, the programmers will be responsible for signalling when a mobile object will stop moving through the network.

The other four methods are inherited from IMobility interface. Two of them are of particular interest here: preDeparture (String source, String destination) and postArrival (). They perfectly match the purpose for generating object dispatch and arrival events.

5.1.2 Event Records

An important part of the design of the event data collection subsystem is to create a set of event records that gives a full description of the actions that are performed in the application that is visualized. We have constructed four types of event records corresponding to

different types of events. The common characteristics of these event records are defined in the superclass MobileObjectEvent. Fig. 5.4 depicts their structure in UML notion.

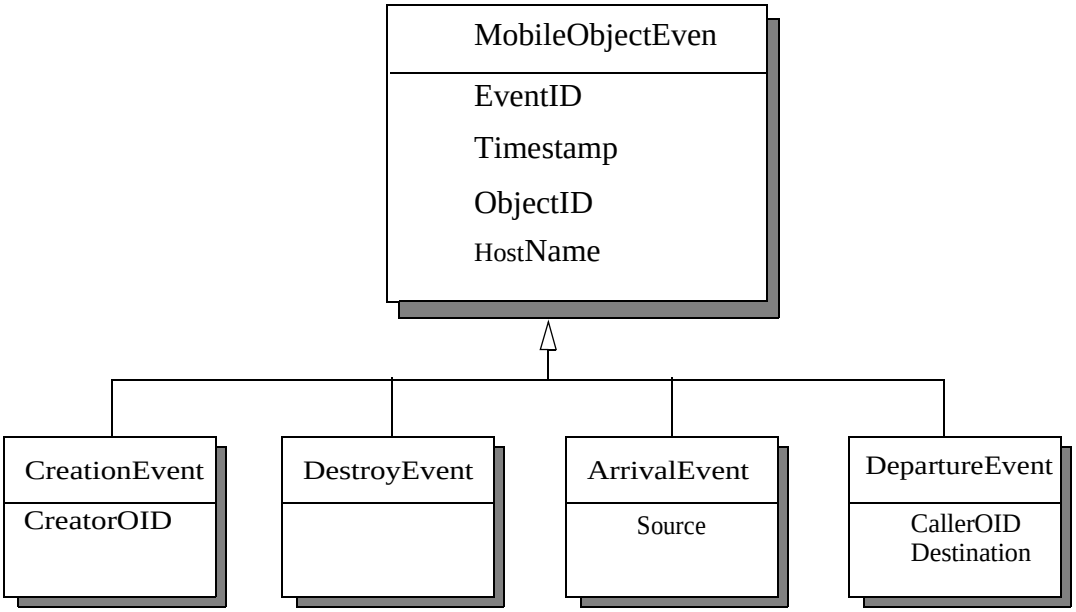


Fig. 5.4: Schematic overview of event records

Every MobileObjectEvent has four basic attributes:

- EventID: identifies the event type
- Timestamp: the time at which an event happens
- ObjectID: determines the identity of the object that the event belongs to
- HostName: the host where an event happens

In addition to these attributes, the CreationEvent has a CreatorOID attribute that identifies the object which instantiates a specific mobile object. The ArrivalEvent has an

attribute to determine the source, i.e. where the object comes from. This kind of event is generated when an object successfully migrates to another node. The DepartureEvent has attributes for the CallerOID and Destination, i.e. where the object will go. The CallerOID is used to identify the object that will initiate the migration. This attribute is not necessary for the migration initiated by the migrating object itself.

This set of event records are created to meet our current visualization needs. It is by no means complete, and can be extended in the future, based on new requirement.

Event Records will be saved into a trace file. There are several candidates for the format of the trace file: SDDF designed for the Pablo project [32], PICL for the ParaGraph system [31] or some kind of self-defined data format specific to our system. SDDF is chosen over the others because it is sufficiently flexible to represent the trace information. SDDF allows arbitrary data structuring without defining a fixed set of event types, nor the size or data types of a particular event. Therefore, we can change the contents of existing event records or add new event types with less code modification.

5.2 The Architecture of Event Data Collection Subsystem

An important consideration in event data collection is whether the information gathered is utilized on-line or in a postmortem fashion, in other words, whether the derived data will be displayed as the program runs with some relative time delay or the

program produces a collection of trace event records which is post-processed at a later time. Correspondingly, there are two different ways for event data collection:

1) Distributed event data collection

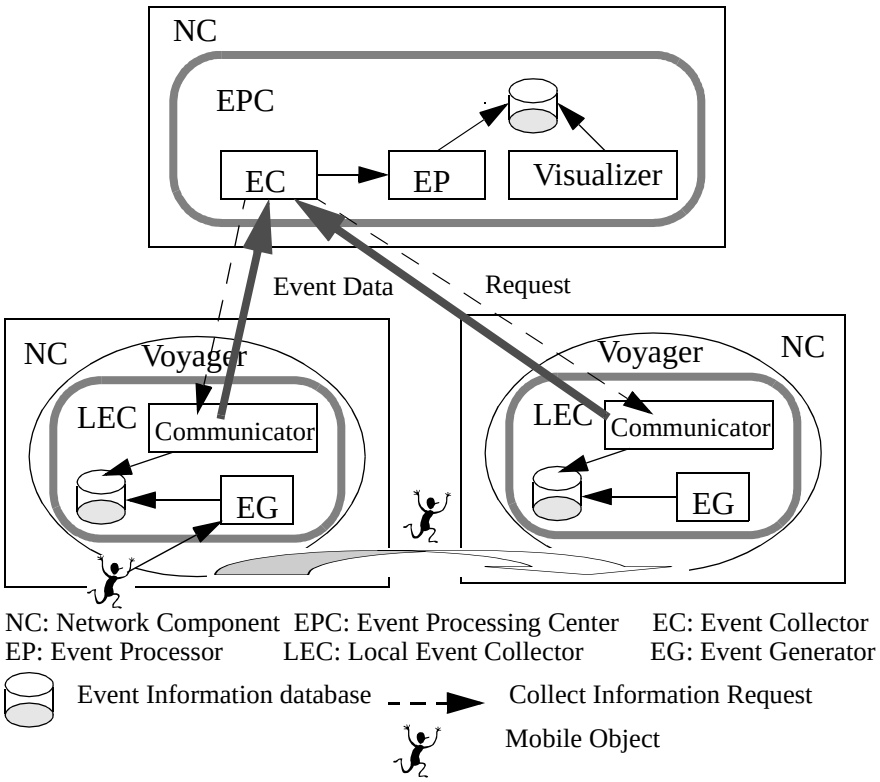


Fig. 5.5: Distributed event data collection

As shown in Fig. 5.5, one local event collector (LEC) resides on each host to which mobile objects will migrate. The LEC has two components: the event generator (EG), which collects event data generated by the objects on that host, and the communicator, which is responsible for the local trace file transfer. There is one event processing center

(EPC). It has two components related to event data collection and processing: the event collector (EC) and the event processor (EP). The EC request the communicatior of the LEC to send back the trace file it has collected. The EP is used for event data processing. Another component, the visualizer, is for event data display. The EC, EP and Visualizer are most likely distinct components for postmortem visualization.

During the program execution, interesting events emitted by probes are sent to the LEC, where they are buffered and then stored into a file. After the application has finished executing, the individual log files are collected automatically by the EPC, where they are post-processed into a single file. In most cases, data processing is carried out on these events to preserve the causality relationships. After such processing, the file is ready for visualization.

2) Centralized event data collection

As shown in Fig. 5.6, there is one event processing center (EPC) typically located on a host that provides visualization functionality. The EPC has four components, of which the EC, EP and visualizer have similar functionality to those described in the distributed event data collection case. The new component is the coordinatior, which is responsible for coordinating the task of the EP and the Visualizer. This conceptual component is needed in the case of on-line visualization. There is one local event collector (LEC) residing on each host. It has two components, event generator (EG) and monitoring controller (MC). The EG has similar responsibility to the one introduced in the last

section. The MC primarily facilitates event data transfer in an on-line fashion. It may also provides some other facilities, such as event data buffering and timestamp adjustment. When the application is running, the probes capture trace events of interest. They are collected and partially processed by the LECs. These trace events are then fed to the EPC where they are further processed and then either read by the visualization tool for on-line display or stored in a trace file for postmortem analysis.

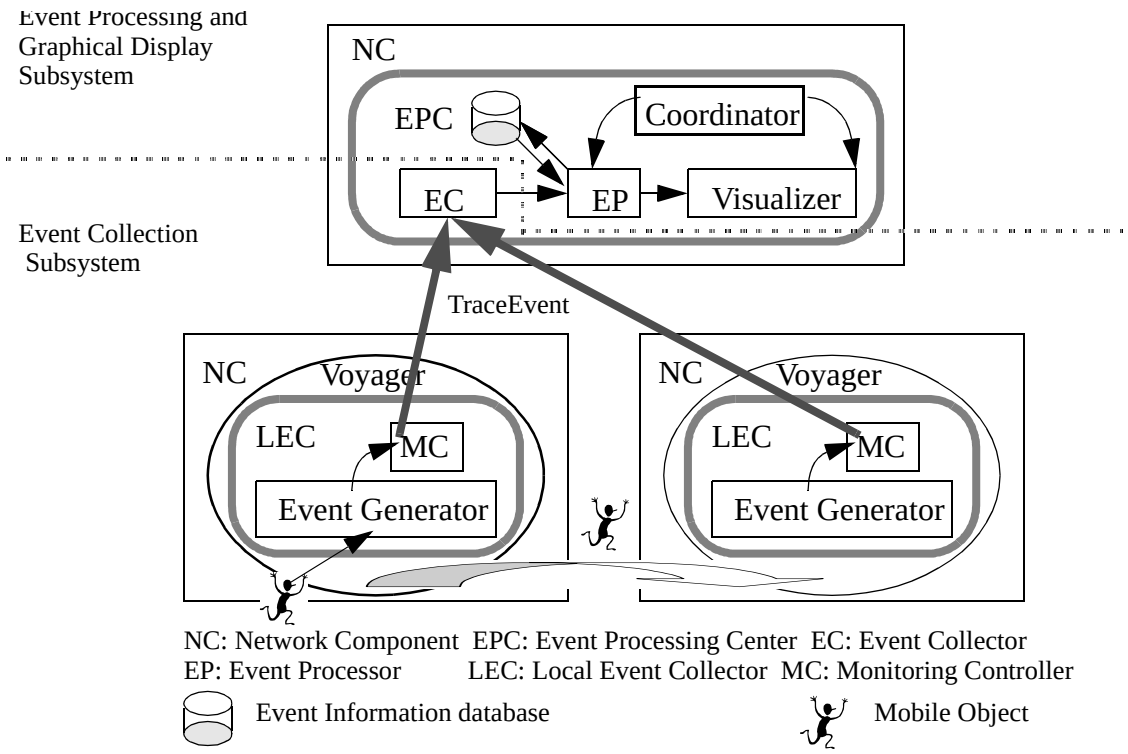


Fig. 5.6: Centralized event data collection

These two different event collection schemes both have advantages and disadvantages. On one hand, distributed event data collection is simpler and more

efficient. Moreover, extensive buffering of the recorded data is possible since analysis is usually deferred until after the application has completed. As a result, it is potentially less intrusive. But it only supports postmortem visualization. Another drawback of this approach is that the EPC has to be coupled with the other LECs, which participate in the tracing, because it has to ask them to send back individual trace file after all. On the other hand, centralized event data collection potentially facilitates both on-line and postmortem visualization. However, this scheme is more complicated to implement and it potentially impedes the performance of network computing applications for two reasons. First, it consumes extra network bandwidth. Second, if used for on-line visualization, the events must be processed as soon as possible and extensive buffering would not be applicable. Moreover, centralized architecture has potential problems with scalability. Despite its complexity and potential performance drawbacks, we adopt this approach in our system for the following reasons:

- Our system supports both on-line and postmortem visualization since such systems have a wider usability. Some visualization activities, due to their nature, must be performed on-line to be effective, such as environment monitoring and interaction/ steering. Some other activities also benefit from need the support for both on-line and postmortem visualization, such as performance evaluation and debugging.
- Comparing the network bandwidth available with the amount of data that will be generated by the tracing system, we believe that trace events will not consume a substantial network bandwidth and thus will not introduce undue overhead.

- Although the centralized architecture may not scale up as well as distributed one, we can compensate this by providing more than one central event collectors. However, in order to support on-line visualization, all the events collected still have to be sent to the node on which the visualizer resides on. Therefore, we would not consider it as a problem of scalability but rather a trade-off for on-line visualization.

5.3 Design and Implementation Issues

In designing and implementing collection subsystem, several considerations should be taken into accounts:

Programming Language

The fact that the application program will be targeted for Voyager, a 100% Java distributed computing platform, requires that application programs will be written in Java. It is straightforward that two components of the data collection subsystem, the LEC and the central EPC are also implemented in Java. However, C/C++ is chose for the latter because Java is an interpreted language that runs substantially slower compared with C/C++. Consequently, it may cause a potential performance bottleneck if the central EPC interacts with other reporters frequently, especially in some cases, such as on-line visualization, when the EPC has to do data processing and displaying concurrently with data receiving.

Object Identifier

Obviously, a mobile object needs an identifier to get identified and located. Moreover, this identifier should be globally unique during its lifetime. Basically, there are two ways to assign an identifier, a lexical name or a serial number. A serial number is more likely to be unique but it is less meaningful to the user. In contrast, a lexical name is easier to be understood but it is hard to guarantee its uniqueness. A conjunction of the name and a serial number can be an obvious solution.

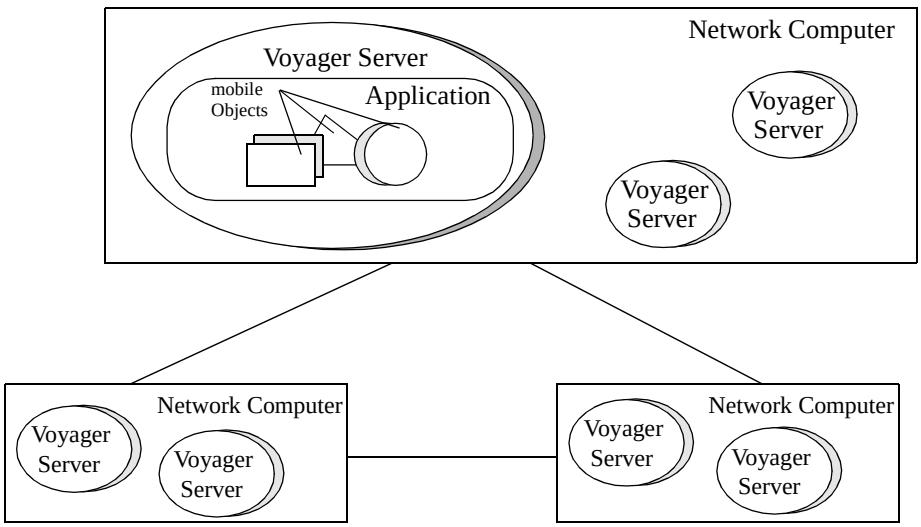


Fig. 5.7: Relationship of host, server and mobile objects

As shown in Fig. 5.7, Voyager servers introduce a hierarchical structure. A given computer in a network can host multiple Voyager servers; each Voyager server holds a number of mobile objects. The fact that a host can contain more that one server requires that servers have unique names within a host. It is relatively easy to ensure that server

names are unique. One reason is that the number of servers within a host is relatively small. Another reason is that a server generally has to bind with its name in Voyager's namespace, which disallows two objects bound with the same name. Therefore, a Voyager server can be uniquely identified by its name combined by the domain name of the host on which it resides. Typically, a domain name can be expressed as a URL. Then we can use a fully qualified class name to distinguish different classes and an integer number for distinguishing the instances of a given class. Generating identifiers in this hierarchical way, an object could be identified by the conjunction of a URL, its server's name, its class name and an integer number. The server here refers to the application in which an mobile object is originally created, while the URL refers to the host on which this server runs.

A potential problem with this approach is that the resulting object identifier can be very long. One solution is to allow the user to assign a nickname to a mobile object so that an object can be identified by the URL, the server name and the nickname. In this way, it is the user's responsibility to guarantee that the nickname is unique. This solution gives the user a chance to name the object of interest in their own way, most likely in a shorter form. If the nickname is not set, the object identifier will still be based on the longer default approach.

Timestamp Creation

The timestamps that have been recorded with the program events are used to order the events for visualization. The creation of timestamps is crucial to tracing. Lamport provides algorithms for logical timestamps that require the piggybacking of time information into messages sent in the distributed system. [51]. This approach is applied in our system with timestamp adjustment.

In distributed network environments, it is most likely that there is no global clock. As shown in Fig. 5.6, event data are collected from individual local event collectors (LEC), which send events to the central event processing center (EPC) at their own speed. Poor clock synchronization among different network computer may be one reason that leads to event timestamps that do not accurately reflect the actual order of program execution. Consequently, some events will appear to occur "back in time". We provide a timestamp adjustment strategy to address this problem.

After a LEC is created, it attempts to connect with the EPC and asks it to send back its current local time. The LEC compares this time with its own local time (LEC_lt) and produces a delta time. All future timestamps are produced as the difference of the current time and this delta time. In this way, individual LECs adjust their local time with that of the central EPC. The timestamp adjustment will be done periodically.

We also take into account the communication delays between the LECs and EPC when calculating the delta time. The LEC gets its local time before and after it sends the

adjust time requirement to the EPC. The average of these two time values is taken as the local time of LEC (LEC_lt).

Although the time adjustment strategy by no means can guarantee the causality of the trace event, according to our experience, it is very effective and can be used with other mechanisms to preserve the causality, such as the views of visualization toolkit compensating for the out-of-time ordering and rearranging themselves to correctly reflect new information. A drawback of this strategy is that it consumes extra network bandwidth and processor time, and hence potentially imposes extra perturbation to the applications. Therefore, we should select an appropriate adjustment frequency when using it.

How to Locate the Local Event Collector Object

For security reasons, mobile objects are almost unable to open a network connection when they get to a host other than their home where they are created. Therefore, our system has to facilitate an interface between the mobile object and the alien host it moves to. The local event collector (LEC) servers this purpose. Residing on each Voyager-enabled host that participates in the tracing, it partially processes the trace events and sends them to the remote event processing center via sockets. Since the mobile objects move between hosts, how to locate the local LEC is a design consideration. Our solution is to take advantage of Voyager's naming service. Every LEC has a well known identical name. After being created, it is bound with that name in its local namespace by invoking

NameSpace.bind (String name, Object object). The mobile object is able to know the URL of its destination from the move notification. Therefore it can find a LEC, which resides on a given host, soon after its arrival by invoking NameSpace.lookup (String name). If there is no LEC object alive on that host, mobile objects will catch an exception after invoking methods on the LEC. The mobile object may then decide to cancel the request.

Tracing Facility

Tracing facility refers to the functionality provided by the data collection subsystem for the trace event generation and transfer.

- Buffered tracing

  As mentioned previously, on-line collection of trace events from individual hosts may impose more intrusion on applications. Buffered tracing is a possible solution to this problem because it allows a number of trace events to be combined into a single message to be transferred. The disadvantage of this approach is that it introduces buffering delay, which is undesirable for on-line visualization. It is a trade-off between the probe perturbation and the display update speed. Therefore, we should select an appropriate buffer size and flush the buffer either after it is full or a time-out occurs.

- Trace event structure

Designed with flexibility and extensibility in mind, we will not hard-wire the contents of trace events, but rather explicitly define each trace event record at run time before its appearance in an event trace. This closely models the SDDF trace file format by including trace descriptors in the event stream. These descriptors precisely define the contents of each trace record and specify the data types and unpacking order of the trace information. Using this trace event structure, new trace events can be added and trace record contents rearranged without modifying our trace file generation code.

5.4 Functionality

As shown in Fig. 5.8, the information collection subsystem provides a number of services to facilitate event collection:

- Communication service

  The local event collector (LEC) and the central event processing center (EPC) communicate with each other using TCP sockets. They operate in a typical server/client model.

- Encoding and decoding services

  Since the LEC is implemented using Java, while the EPC uses C/C++, encoding and decoding services are provided to enable client and server in different languages the communicate with each other by using sockets.

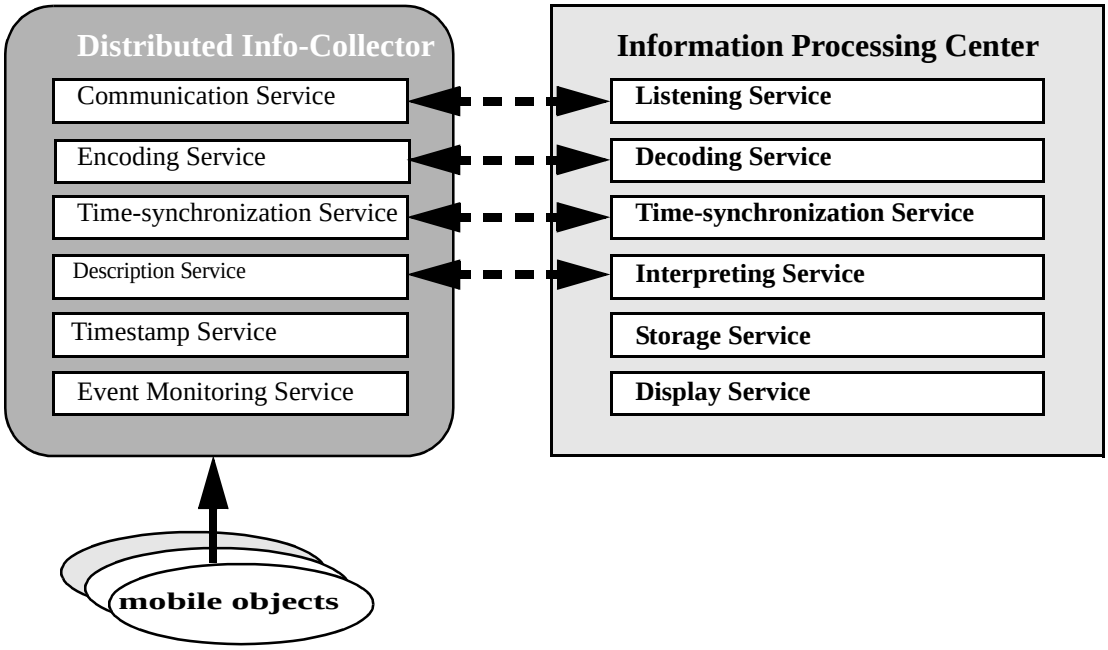


Fig. 5.8: Services provided by the trace event collection subsystem

- Description and interoperation service

  As mentioned earlier, before sending a trace record to the EPC, the LEC should send its descriptors first. The EPC interprets individual trace records according to their descriptors.

- Time synchronization service

  This service allows the LEC to adjust its time according to the local time of the EPC.

- Event monitor service and timestamp service

  These services are used to generate timestamped trace events.

Chapter 6 Event Processing and Graphical Display Subsystem of CMVS

Once the events have been generated and collected, event processing and visualization can be performed based on these events. This chapter introduces the design of the event processing and graphical display subsystem. We first present its architecture. Then we discuss the issue of on-line event reordering and describe the facilities provided by this subsystem. Finally, a summary of CMVS is given.

6.1 The Architecture of Event Processing and Graphical Display Subsystem

The event processing and graphical display subsystem facilitates both on-line and postmortem visualization. As shown in Fig. 6.1, it has three functional modules, the event processor (EP), the visualizer and the coordinator. The first two are responsible for event management and event visualization respectively. The last one is for coordinating their activities. This section describes the functionality these modules provide and how they interact with one another.

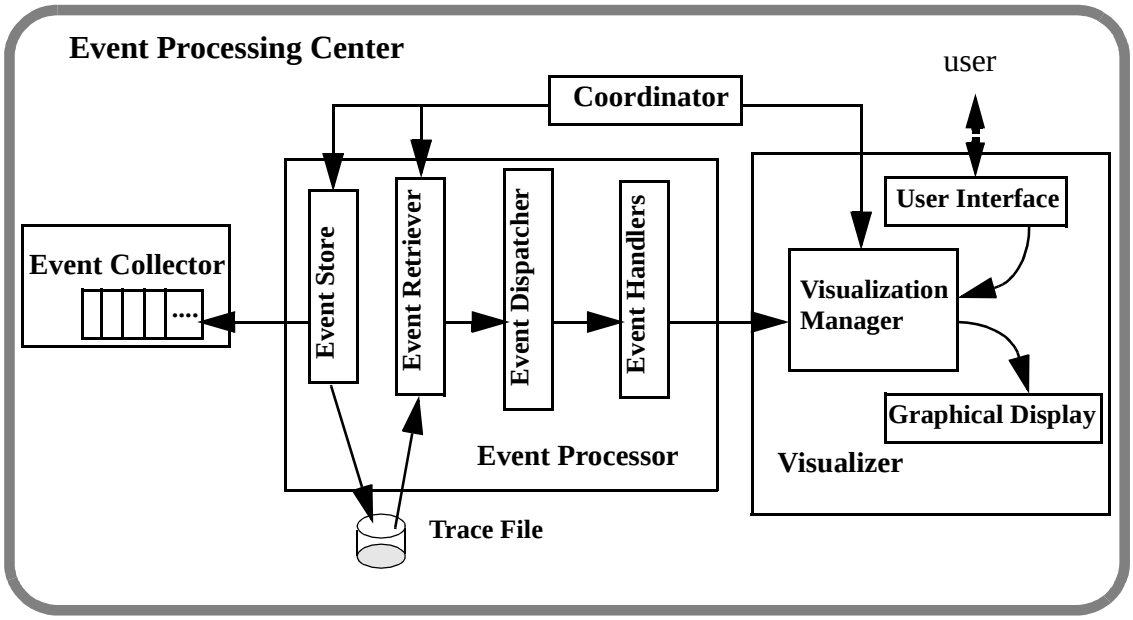


Fig. 6.1: The architecture of the event processing and graphical display subsystem

The Event Processor

The event processor (EP) is essentially an event server for its client, the visualizer. It consists of several components for storing, retrieving, and handling the collected events. According to whether the visualization runs on-line or postmortem, the EP performs different tasks.

On-line Mode: Basically, there are two tasks scheduled by the coordinator. One is handled by the event store (ES), which reads the collected event data from the buffer of the event collector and stores them into a trace file in SDDF format. The other is performed by the event retriever (ER), the event dispatcher and the event handlers. The

ER reads the events out of the trace file and feeds them to the event dispatcher, which processes and dispatches them to different event handlers according to their event identifiers. The individual event handlers retrieve the program information associated with the events and push them to the visualization manager of the visualizer, where they will be further processed and shown in the graphical display.

Postmortem Mode: In this mode, no event data will be saved into the trace file. Only the latter task will be performed, i.e. read the events out of the trace file, process them and feed them to the visualizer.

The Visualizer

The visualizer has three functional components, the user interface, the graphical display and the visualization manager. The first two can also be depicted as one module. We separate them here to emphasize their different functions. The user interface provides the user with a means to control the entire visualization, while the graphical display is used to present the graphical representation of the events. The visualization manager is the core of the visualizer. By interacting with both the user and the EP, the events that the user wishes to be visualized can be input from the EP and passed on to the graphical display.

In order to translate the generated program events into a meaningful visualization of the executing program, the visualization manager maintains two kinds of information, the program state information and the visualization information. The program state information is used to recreate the program state as the visualization executes. This kind

of information is necessary to support information query and view zooming. It consists of the structured information retrieved from the collected events, such as the object identifier, the timestamps and other event specific items. The visualization information is used to present the visualization of the executing program. It stores the information that is required to maintain the state of the graphical elements. In our implementation, these two kinds of information are stored in a linked list according to different object identifiers. In addition to the above functionality, the visualization manager may also perform on-line event reordering, which will be introduced in more detail in the next section. The model-view-controller (MVC) pattern [55] is applied in the design of the visualizer, with the information kept by the visualization manager as the model, the graphical display as the view and the user interface as the controller.

The visualizer is written using the Tcl/Tk toolkit [35] with application extended C commands. Tcl/Tk provides a programming environment for developing GUI applications for the X-window system. It supports a set of rich built-in widgets, such as lines, rectangles, circles, canvas, buttons, to name but a few, and provides convenient facilities to identify a specific widget instance and change its attributes such as the position, size, and color. It also facilitates view scrolling and a general-purpose binding mechanism that can be used to create additional event handlers for widgets.

Another argument to justify the use of Tcl/Tk is its extensibility. Each application can extend the core Tcl/Tk with additional commands written in C/C++ that are specific to

that application. This is the major reason that Tcl/Tk is chosen for our system. In this way, the application would be largely application specific C/C++ code and include a small amount of Tcl script for configuration and the graphical interface. This capability is extremely useful when we hope that the GUI can run more efficiently and when it is impossible to provide the same functionality purely in Tcl script. Fig. 6.2 depicts a general relationship between the Tcl interpreter and the rest of the application.

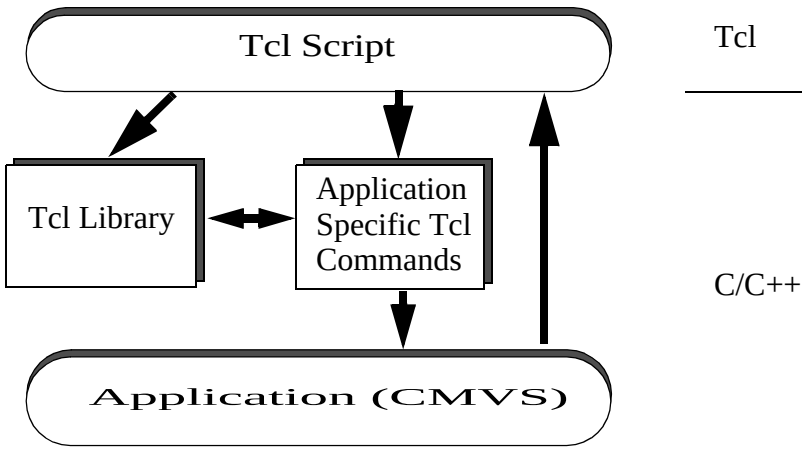


Fig. 6.2: Application structure

The Tcl library implements the interpreter and the core Tcl commands. Application specific Tcl commands are implemented in C/C++ and registered as commands in the interpreter. The interpreter calls these command procedures when the script uses the application specific Tcl commands.

The Coordinator

After initialization, the Tcl/Tk interpreter will normally stay in an event loop waiting for X-window events. If the application is idle, the coordinator will schedule some tasks to execute alternatively according to the visualization mode and the schedule policy. These tasks include the two tasks performed by the event processor and the task that is handled by the visualization manager to update the view.

6.2 On-line Event Reordering

As mentioned in the previous chapter, the program events collected at the central event collector (EC) may be not in order. Although the timestamp adjustment strategy can compensate to some extent the poor clock synchronization among different hosts, it, used alone, can not guarantee the causality of the events. In order to preserve the behavior of the original program when presenting such information to the user, reordering has to be performed so that the causal order of events exhibited by the executing program is preserved and enforced.

Basically, there are two forms of misordering. One is that the timestamps associated with the events do not reflect the actual or causal order in which the application executes. The other is that an happen-before event may be received later due to buffering, transmission latency or some other delays. For the first form of misordering, a possible solution is to increment the timestamp that is out-of-order by the minimum amount necessary for the partial order to hold. For our system, due to the use of timestamp the

adjustment strategy and the fact that object migration takes some time, this form of misordering occurs rarely and at this moment we ignore this problem and assume that the timestamp associated with an event correctly reflects the actual order of the program execution.

For the second form of misordering, some existing systems (e.g. Paragraph [36], AIMS [28]) rely on a sort by timestamp value to impose a total order on all events stored in trace files. The on-line nature of our system precludes using such a solution, and sorting by timestamp order does not entirely eliminate this problem [56]. A possible solution is to examine each event collected, checking whether it complies with the pre-specified ordering rule. If no rule is violated, it will be forwarded to display; otherwise, it will be held back for a while until the rule is satisfied. In our system, this form of misordering only appears occasionally. Moreover, an out-of-order event appears immediately after the one which should have occurred after it. This greatly simplifies our solution to address this issue. Before the visualization manager presents an event, it first checks if this event is in order. If it appears back in time, it will be placed right before the event whose timestamp is newer. Otherwise, it is displayed after the current last event.

How to diagnose and correct the misordering events with suitable efficiency is a challenging issue in on-line visualization systems. Although our approach deals with a relatively simpler situation, it effectively solves our problem and can be extended by applying some ordering rules for more complicated situations.

6.3 Facilities

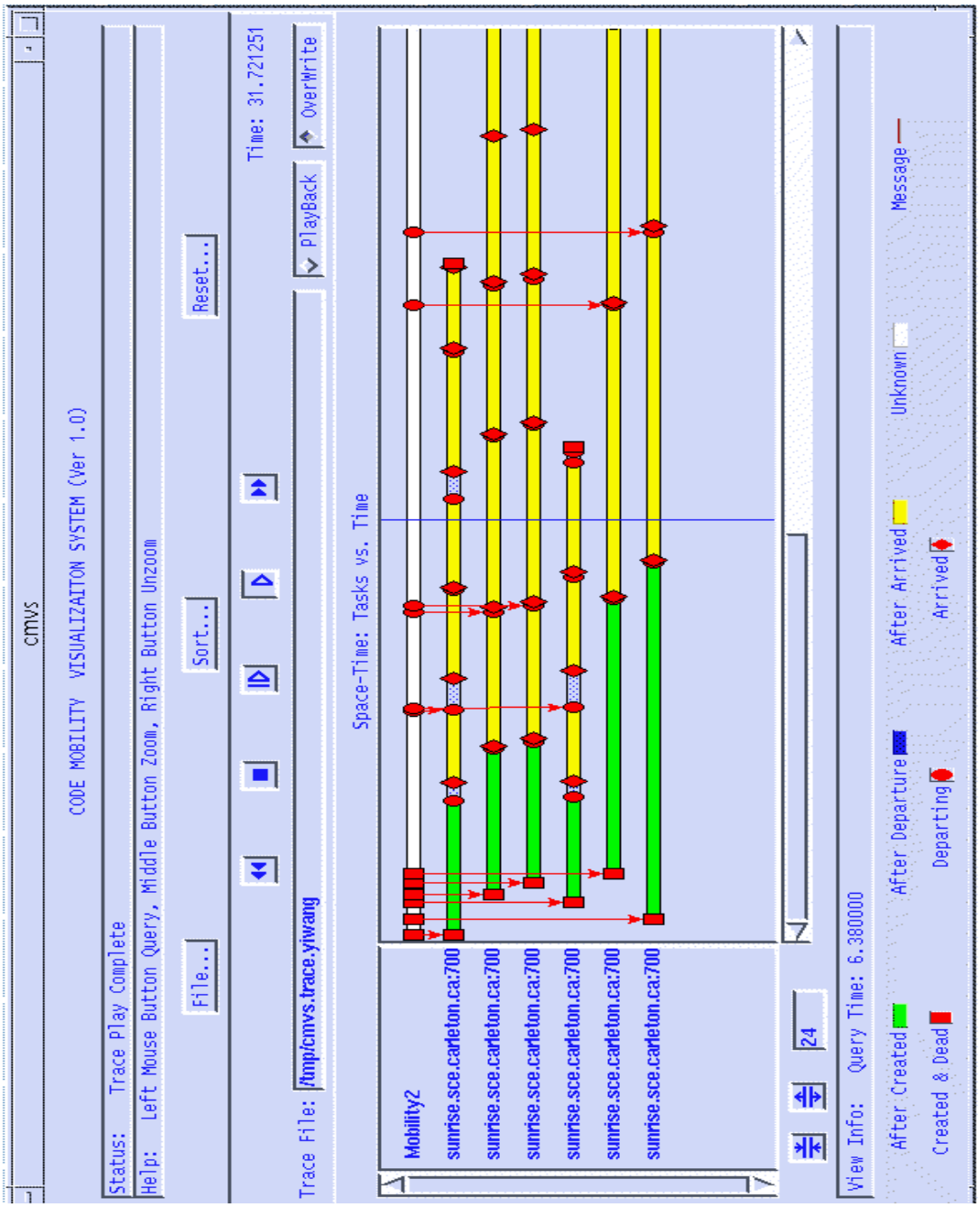


Fig. 6.3: Snapshot of CMVS graphical view and user interface

In line with the visualization objectives presented in the previous chapter, as shown in Fig. 6.3, our visualization system provides the following facilities:

- Trace play control

  In addition to "Play", "Fast Forward" and "Stop" buttons, the control panel also provides "Single Step", which allows a single trace event to be processed, and "Rewind", which resets the visualization system to the beginning of the current trace file.

- Process-time view queries

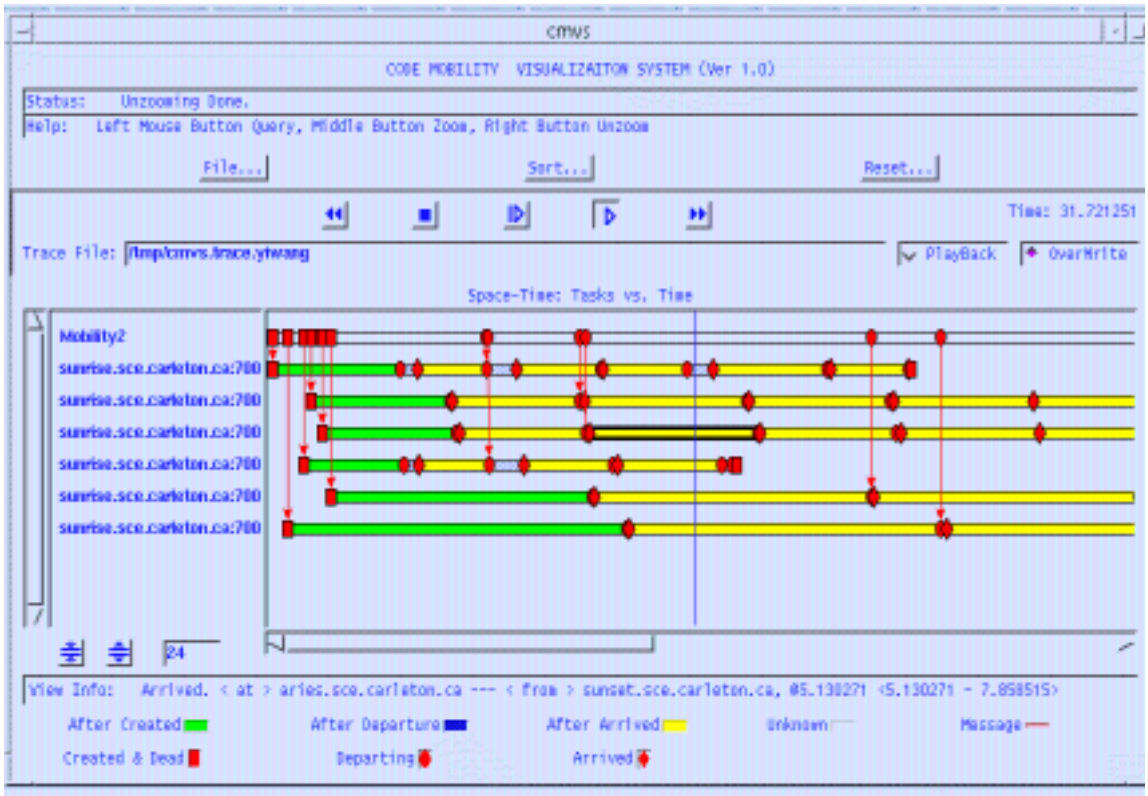


Fig. 6.4: Snapshot of the process-time view query

As shown in Fig.6.4, more detailed information regarding specific object states or invocation can be extracted by clicking and holding on the view area with the mouse button.

- Process-time view zooming

Fig. 6.5: Snapshot of the process-time view zooming

As shown in Fig. 6.5, the process-time view can be zoomed in or out to display different perspectives on the same trace data. This facility is provided to control the level of time detail. If a particular area of the view is zoomed in, it will be enlarged horizontally to fill the entire process-time canvas area to allow the user to take a closer look. Scrolling along the time direction is also supported.

6.4 Summary

In this section, we first summarize the work presented in this chapter and Chapter 5. Then we give a rough evaluation of CMVS.

6.4.1 An Overview of CMVS

Code mobility visualization system (CMVS) is designed to help understand the object-oriented code mobility applications. It collects event data during program execution, then provides on-line or postmortem processing and displays them on a process-time diagram. CMVS has two functional components: the event collection subsystem and the event processing and graphical display subsystem.

The event collection subsystem employs a tracing mechanism to collect event data and allows a program to be annotated with few source code modifications. A set of event records are constructed to meet our current visualization needs. This subsystem adopts the centralized event collection scheme to support both on-line and postmortem visualization. It has two components: the central event collector, which is located on a host that provides visualization functionality, and the local event collector, which resides on each host to which mobile objects migrate. They operate in a typical client/server mode, communicating via TCP/IP, and facilitate event generation and transfer. A mobile object can be uniquely identified by the combination of a URL, the name of its server (the application in which a mobile object is originally created), its class name and an

integer number. A timestamp adjustment strategy is used to help preserve the events causality.

The event processing and graphical display subsystem is responsible for data processing and display. It consists of three functional modules: the event processor, the visualizer and the coordinatior. The first two perform event management and event display respectively. The last one coordinates their activities. The visualizer has three components, the visualization manager, the graphical display and the user interface. The visualization manager is the core of the visualizer. It maintains two kinds of information, the program state information and the visualization information, to translate the generated program events into a meaningful visualization of the executing program. This subsystem provides several facilities that are very helpful in understanding the visually presented program information, such as view zooming and query.

Since no existing visualization tools provide tracing facilities for code mobility, the event collection subsystem, as well as the event processor of the event processing and graphical display subsystem (EPGS) are designed and implemented from scratch.

The visualizer of the EPGS is adapted from XPVM. As described earlier, XPVM supports on-line monitoring and postmortem visualization of PVM program execution. It provides the space-time view, which is similar to the process-time diagram, and some useful facilities, such as view zooming, information query and playback control of trace

files. Nevertheless, it takes more than trivial effort to adapt it to CMVS for the following reasons:

- XPVM is a complex and huge piece of software with 35,000 lines of source code. It is by no means an easy job to understand the program and get a whole picture about it, especially when we lack the help of debugging tools and references. This makes the adaptation very difficult due to the fact that software modification or maintenance greatly relies on program understanding.
- XPVM is a graphical console and monitor for PVM. It provides facilities for the PVM console commands and information. Moreover, it supports six other views except the space-time view to monitor the execution of PVM programs. However, all the facilities for PVM and the views other than the space-time view are irrelevant to our visualization and will not be applied in CMVS. This makes our situation more complicated due to the fact that the pieces of C code and Tcl/Tk scripts related to these facilities and views disperse in the program. It is quite often the case that even a little modification involves many parts of the code.
- XPVM does not meet all of our visualization needs. For example, it does not support the visualization of the events collected from the object-oriented code mobility programs, nor does it facilitate symbols in its space-time view or provide online event reordering. This means that some facilities have to be added in order to support them.

As a result, most parts of the XPVM source code are modified and new facilities are added to support our visualization. XPVM's mechanism used to drive the views is applied in CMVS and we reuse the layout of its graphical interface and some utility procedures, such as those for menu adding, frame resizing, scrolling and playback control of trace file.

CMVS provides similar facilities to some popular visualization tools, such as XPVM, Pvanim and ParaGraph. These tools are widely used and prove themselves useful for application development. Therefore, there is no reason to doubt about the usability of CMVS and we intend to let more application developers use it in the future.

6.4.2 System Evaluation

This section gives a rough evaluation of CMVS in terms of how much work is needed to make a program "visualizable", what is the perturbation effect and how much effort is required to actually visualize a given trace file. This evaluation is by no means comprehensive and the numbers given here are only an indication of the order of magnitude based on tracing one example. A more in-depth exploration of these issues will be left to future work.

Work Needed to Enable Visualization

As described in Chapter 5, we reduce the work needed for program instrumentation by applying class inheritance or aggregation, depending on whether a mobile object class has its own superclass or not.

```
class Drone extends Visualizable  - - -  (1)
{   .....
    Drone ()
     { .....
       creation (...); - - - - - - - - -  (2)
      }
}
class MyApplication
{
     public static void main ()
     {
       .....
       Drone myDrone = new Drone(...);
       .....
        myDrone.destroy();    - - - - - (3)
        .....
}
```

Fig. 6.6: A skeleton application modified using class inheritance to enable visualization

As shown in Fig. 5.2 and Fig. 6.6, if class inheritance is applicable, using CMVS requires three minor modifications to the source code:

- The mobile object class Drone has to inherit from Class Visualizable.

- Creation () should be placed into the constructor of class Drone.
- Destroy () is invoked on a mobile object when will not move any more.

The total number of lines of code needed to be modified/added (NL) is equal to:

$$NL = NC * 2 + NO$$

NC: The number of classes whose instances will be visualized

NO: The number of mobile objects to be visualized

If the mobile object class, Drone, has its own superclass, SuperDrone, due to the limitation of single inheritance from Java, the wrapper pattern using class aggregation is used to insert the instrumentation. As shown in Fig. 5.3 and Fig. 6.7, the following modifications will be required:

- The mobile object class Drone has to implement interface IVisual.
- It has to encompass a instance of class VisualAaptee, adaptee.
- It has to define the following methods: preDeparture(), postArrival() and destroy() by invoking the corresponding methods of adaptee.
- ad_creation () should be placed into the constructor of class Drone.
- ad_destroy () is invoked on a mobile object when it will not move any more.

The total number of lines of code needed to be modified/added (NL) is equal to:

$$NL = NC * 6 + NO$$

NC: The number of classes whose instances will be visualized

NO: The number of mobile objects to be visualized

```
class Drone implements IVisual {                          (1)
    VisualAdaptee adaptee = new VisualAdaptee (...)       (2)
    Drone () {
        adaptee.ad_creation (...);                        (3)
            ......
    }
    preDeparture(...) {
      adaptee.ad_preDeparture (...);                      (4)
    }
    postArrival(...) {
     adaptee.ad_postArrival (...);                        (5)
    }
    destroy() {
      adaptee.ad_destroy (...);                           (6)
    }
          ......
}

class MyApplication {
    public static void main () {
        Drone myDrone = new Drone (...);
         ......
        myDrone.destroy (...);                            (7)
        ......
    }
}
```

Fig. 6.7: A skeleton application modified using wrapper pattern to enable visualization

The Perturbation Effect

As with many other instrumentation approaches, the instrumentation scheme used in CMVS causes perturbations in the instrumented programs, which means a program slows

down by having the event instrumentation active. Generally, to quantify the perturbation effect is difficult and requires a complex perturbation model. Here we give circumstantial evidence by examining only one application, in which objects move between 3 Sun Ultra workstations and generate 1000 events in total. Table 6.1 shows that the instrumented version of the original program requires approximately 7.4% more time to execute. It is acceptable compared with other tracing systems with a perturbation ranging from 5% to 10%.

Table 6.1: Average execution time and overhead

Un-Instrumented	Instrumented	% change
1,280,912 milliseconds	1,287,026 milliseconds	7.4

As discussed earlier, there are some trade-offs between performance and other benefits in the design of CMVS. Future work to address this issue can be attempted in the following directions:

- Reduce the size of the event data structure
- Reduce the cost of event transmission and buffering
- Reduce the side-effect of timestamp adjustment mechanism

The Effort Required to Actually Visualize a Given Trace File

Like any other applications, CMVS consumes CPU time and space. Here we are interested in how much CPU time that CMVS will consume when it visualizes a trace

file. We only give a rough estimate, since processing different event types involves a different amount of overhead.

Table 6.2: Average time required to visualize a trace file

Events in the Trace File	Visualization Time
1000	5698 milliseconds

As shown in Table 6.2, CMVS will take 5698 milliseconds to visualize a trace file with 1000 events. In other words, the processing time per event is 5.698 milliseconds. This means that CMVS can not handle more than 177 events a second. Moreover, this inference is drawn under the assumption that the machine is dedicated to the visualization and no other processing has to be done, which is unrealistic in some cases, such as on-line visualization, when the events have to be received and processed as well. On the other hand, 177 events/second means approximate 88 times of object migration per second (an object moving once generates two events: departure and arrival). This indicates that if the number of migrating objects in a rather big application exceeds a given value, the visualization will become a bottleneck and can not be rendered at speeds close to the actual program execution.

One possible solution to improve the capacity of the visualization is that we have event reading and event display handled by separate threads. In this way, events can be read from the trace file, partially processed and buffered into a queue while the visualization manager is visualizing the previous events.

Chapter 7 Summary and Future Work

7.1 Summary

It is evident that program development and maintenance are two of the most vital activities that occur in computing. Each relies on, and can benefit from, an increased level of program understanding. Distributed applications with code mobility involve all the complexity of distributed programs, plus problems specific to object migration. They are intrinsically complex and hard to understand. With the increasing interest in the application of code mobility, it is important to provide software developers with a tool for understanding, debugging, testing and maintaining such applications. It is equally vital to provide network administrators a way to monitor and manage the mobile objects if they allow them to float in their networks. We believe that event-driven visualization can be and will be an invaluable tool for understanding and monitoring the execution of object-oriented applications with code mobility.

In this thesis, the interesting events that help us understand the execution of object-oriented code mobility applications are identified by carrying out a study of the mobile code technology. Several approaches to the graphical representation of those events are analyzed. An innovative way to visually describe and illustrate code mobility is present. This approach explicitly indicates the location change of mobile objects without increasing

complexity and display space. We surveyed several popular existing visualization systems for distributed/parallel applications. This survey summarizes the contributions that a specific system has made and compares their capabilities in terms of functionality, usability and extensibility. It gives an overview of the current research in the area of program visualization and can help visualization tool developers in designing their own systems. Challenging issues related to program visualization, including preservation of causality, scalability, and quick focus on particular concerns were addressed and solutions provided. An information tracing and visualization infrastructure (CMVS) for understanding the execution of object-oriented code mobility applications was developed. CMVS supports both on-line monitoring and postmortem visualization. It minimizes the need for program annotation by class inheritance or aggregation and provides a means for visualization with little programmer intervention. It facilitates mechanisms that allow users to take a closer look of a particular area of the view, and to get more detailed information regarding a specific event.

Although this infrastructure was developed to visualize programs written in Voyager, its principles and structure can be applied to other event-driven visualization for object-oriented code mobility applications. Only two aspects of its implementation that rely on Voyager have to be modified in order to port CMVS to other platforms:

1) How to locate the reporter

2) How to trace arrival and dispatch events

However, since most frameworks that support object migration, such as aglet, provide some kind of object arrival and dispatch notification, as well as facilities that allow mobile objects to use some resources of the remote nodes, it will not be difficult to adapt CMVS to those platforms.

7.2 Future Work

The work presented in this thesis is intended to improve the understanding of object-oriented code mobility applications. It lays a valuable ground for future work that can take two directions: enhancing CMVS and extending its capabilities. They can be attempted in the following ways:

- Support other interesting events such as method invocation and thread synchronization

  In distributed object-oriented applications, interactions between objects are through method invocations. The objects involved may be located in the same address space or not. Method invocations that involve mobile objects could be helpful for understanding their dynamic behavior. Multi-threading is a technique that is used more and more frequently in distributed applications for its potential performance improvement. It would be desirable to record events related to thread state and synchronization.

- Deal with the vast amount of information involved in understanding the execution of complex applications

If more event types, such as method invocation or thread synchronization, are intended to be visualized, efforts should be put on how to deal with a huge quantity of trace data. Some means of event abstraction may be needed at different abstraction levels.

- Support multiple views

  The visualization subsystem of CMVS currently only supports a single view. It would benefit greatly from extensions allowing it to support multiple views.

- Keep exploring alternative solutions to some challenging issues addressed in this thesis

  As mentioned in the previous chapters, our approaches to some challenging issues, such as on-line event reordering or how to trace the dispatch event when lacking support from the run-time system or language, are by no means the only solutions. There are possible alternatives that can be considered to enhance them.

- Apply the result developed in this research to network and distributed systems management, or performance tuning.
- Use CMVS for more complicated and real-world applications so that feedback from users can be used to add new functionality and capabilities to the system.
- Conduct a more in-depth evaluation of CMVS to understand the additional visualization costs, to identify the factors that affect the performance, and to provide solutions that compensate the perturbation effects.

References

[1] George Coulouris, "Distributed Systems: Concepts & Design", Addison-Wesley Publishing Company, 1994, ISBN: 0201624338.

[2] P. Bates, "Debugging Heterogeneous Distributed Systems Using Event-Based Models of Behavior", SIGPLAN Notices 24, pp. 11-22, Jan. 1989.

[3] M. Friedell, M. LaPolla, et al., "Visualizing the Behavior of Massively Parallel Programs", In Proceedings of Supercomputing ’91, pp. 472-480, Nov. 1991.

[4] T. Lehr, et al., "Visualizing System Behavior", In Proceedings of the 1991 International Conference on Parallel Processing, pp. 117-123, Aug. 1991.

[5] Dror Zernik, Marc Snir, and Dalia Malki, "Using Visualization Tools to Understand Concurrency", IEEE Software, 9(3), pp. 87-92, May 1992.

[6] Cherri M. Pancake, "Visualizing the Behavior of Parallel Programs", Supercomputer, pp. 31-37, Sep. 1990.

[7] Christoph W. Ueberhuber, and Gerald Thomas, "Visualization of Scientific Parallel Programs", Springer-Verlag Publishing Company, 1994, ISBN: 0387577386.

[8] Wentong Cai, Wendy J. Milne and Stephen J. Turner, "Graphical Views of the Behavior of Parallel Programs", The Journal of Parallel and Distributed Computing 18, pp.223-230, 1993.

[9] Object Management Group, "The Common Object Request Broker: Architecture and Specification", CORBA/IIOP2.2, February 1998, available at: http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html

[10] Sun Microsystems Inc., "Java Remote Method Invocation Specification", available at:

http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

[11] ObjectSpace Company, "Voyager Core Technology 3.2 User Guide", available at: http://www.objectspace.com/products/vgrorb.asp

[12] A. Fuggetta, G. Picco, and G. Vigna, "Understanding Code Mobility", IEEE Transactions on Software Engineering, Vol. 24, No. 5, pp. 342-361, 1998.

[13] A. Carzaniga, G. Picco and G. Vigna, "Designing Distributed Applications with Mobile Code Paradigms" In Proceedings of the 19th International Conference on Software Engineering (ICSE'97), pp. 22-32, May 1997.

[14] Sun Microsystems Inc., "Java Language Specification", white paper available at: http://java.sun.com/docs/white/index.html

[15] IBM Corporation, "IBM Aglets Software Development Kit", web page at: http://www.trl.ibm.co.jp/aglets

[16] L. Cardelli, "Obliq: A Language with Distributed Scope", Technical Report, Digital Equipment Corporation, Systems Research Center, May 1995

[17] General Magic Inc., "Telescript Language Reference", web page at: http://www.genmagic.com/technolog/techwhitepaper.html

[18] General Magic Inc., "Odyssey Information", web page at: http://www.genmagic.com/technology/odyssey.html

[19] R. S. Gray, "Agent Tcl: A Transportable Agent System", In Proceedings of the CIKM'95 Workshop on Intelligent Information Agents, 1995.

[20] David Wong, Noemi Paciorek and Tom Walsh, "Concordia: An Infrastructure for Collaborating Mobile Agents", In Proceedings of First International Workshop on Mobile Agents 97 (MA'97), April 1997.

[21] Hector Garcia-Molina and Walter H. Kohler, "Debugging a Distributed Computing System", IEEE Transactions on Software Engineering, 10(3):210-219, March 1984.

[22] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger, "Monitoring Distributed Systems", ACM Transactions on Computer Systems, 5(2):121-150, May 1987.

[23] A. D. Malony, D. H. Hammerslag, and D. J. Jablonowski, "Traceview: A Trace Visualization Tool ", IEEE Software, 8(5):19-28, September 1991.

[24] D. Kranzlmuller, S. Grabner, and J. Volkert, "Event Graph Visualization for Debugging Large Applications", In Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 108-117, May 1996.

[25] E. Kraemer and J. T. Stasko, "The Visualization of Parallel Systems: An Overview", The Journal of Parallel and Distributed Computing, Vol. 18, No.6, pp. 105-117, June 1993.

[26] R. Hood, "The p2d2 Project: Building a Portable Distributed Debugger", In Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 127-136, Philadelphia, Pennsylvania, May 1996.

[27] D. Socha, M. L. Bailey and D. Notkin, "Voyeur: Graphical Views of Parallel Programs", SIGPLAN Notices 24, 1 (Jan. 1989), pp. 206-215.

[28] J. Yan, S. Sarukhai and P. Mehra, "Performance Measurement, Visualization and Modeling Parallel and Distributed Programs Using the AIMS toolkit", Software-Practice and Experience 25:4 (April 1995), pp. 429-461.

[29] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, "MPI: The Complete Reference", MIT press, 1996.

[30] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth and K. A. Shields, "Scalable Performance Analysis: The Pablo Performance Analysis Environment", In Proceedings of the Scalable Parallel Libraries Conference, IEEE Computer Society, 1993.

[31] G. A. Geist, M. T. Heath, B. W. Peyton. and P. H. Worley, "A Users Guide to PICL, a Portable Instrumented Communication Library", Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, TN, October 1990.

[32] Ruth A. Aydt, "The Pablo Self-Defining Data Format", Available at URL: http://www-pablo.cs.uiuc.edu

[33] J. A. Kohl, G. A. Geist, et al., "XPVM 1.0 User's Guide", Technical Report ORNL/TM-12981, Computer Since and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN, April 1995.

[34]] G. A. Geist, A. Beguelin, J. Dongarra, W. Jiang and R. Manchek, "PVM: Parallel Virtual Machine-A User's Guide and Tutorial for Networked Parallel Computing ", The MIT Press, 1994.

[35] J. K. Ousterhout, "TCL: An Embeddable Command Language", 1990 Winter USENIX Conference.

[36] M. T. Heath and J. A. Etheridge, "Visualization the Performance of Parallel Programs", Technical Report ORNL/TM-11813, Oak Ridge National Laboratory, Oak Ridge, TN, May 1991.

[37] B. Topol, J. T. Stasko and V. Sunderam, "Pvanim: A Tool for Visualization in Network Computing Environments", Concurrency: Practice and Experience, Vol. 10 (14) , 1197-1222 (1998).

[38] John T. Stasko and Charles Patterson, "Understanding and Characterizing Software Visualization Systems", In Proceedings of the 1992 IEEE Workshop on Visual languages, pp. 3-10, September 1992.

[39] J. T. Stasko and E. Kraemer, "A Methodology for Building Application-Specific Visualizations of Parallel Programs", The Journal of Parallel and Distributed Computing, Vol. 18, pp.258-264, 1993.

[40]] J. K. Boggs, "IBM Remote Job Entry Facility: Generalize Subsystem Remote Job Entry Facility", IBM Technical Disclosure Bulletin 752, IBM, Aug. 1973.

[41] M. Nuttall, "Survey of Systems Providing Process Migration", Tech. Rep. Doc 94/10, Dept. of Computing, Imperial College, May 1994.

[42] A. Carzaniga, G. Picco, and A. Kershenbaum, "Designing Distributed Applications with Mobile Code Paradigms", In Proceedings of the 19th International Conference on Software Engineering, pp. 22-32, May 1997.

[43] Gang Ao, "Software Hot-swapping Techniques for Upgrading Mission Critical Applications on the Fly", Master Thesis, Department of Systems and Computer Engineering, Carleton University, May 2000.

[44] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides, "Visualizing the Behavior of Object-Oriented Systems", In Proceedings of the ACM OOPSLA'93 Conference, pp. 326-337, Washington, D. C., October 1993.

[45] Michael F. Kleyn and Paul C. Gingrich, "GraphTrace-Understanding Object-Oriented Systems Using Concurrently Animated Views", In Proceedings of the ACM OOPSLA'88 Conference, pp191-205, San Diego, CA, Sept. 1988.

[46] B. Venners, "Inside the Java Virtual Machine", The McGraw-Hill Companies, Inc., 1998, ISBN: 0079132480.

[47] D. J. Taylor, "A Prototype Debugger for Hermes", In Proceedings of the 1992 CAS Conference, Vol.1, pp.29-42, Nov. 1992.

[48] T. Kunz and J. P. Black, "Understanding the Behavior of Distributed Application Through Reverse Engineering", The Journal of Distrib. Syst. Engineering 1 (1994), pp.345-353.

[49] T. Kunz, J. P. Black, D. J. Taylor and T. Basten, "Poet: Target-System Independent Visualizations of Complex Distributed-Application Executions", The Computer Journal, Vol.40, No.8, pp.499-512, 1997.

[50] C. E. McDowell and D. P. Helmbold, "Debugging Concurrent Programs", ACM Comput. Surv. 21, 4 (Dec. 1989), pp. 593-622.

[51] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", Comm. ACM21, 7 (July 1978), pp. 558-565.

[52] B. P. Miller, "What to Draw? When to Draw? An Essay on Parallel Program Visualization", The Journal of Parallel and Distributed Computing, Vol.18, No.2, pp.265-269, June 1993.

[53] Dvid M. Ogle, Karsten Schwan, and Richard Snodgrass, "The Dynamic Monitoring of Distributed and Parallel Systems", IEEE Transactions on Parallel and Distributed Systems, 4(7):762-778, July 1993.

[54] Brad Topol, Vaidy Sunderam, and Anders Alund, "PGPVM Performance Visualization Support for PVM", Technical Report CSTR-940801, Emory University, Atlanta, GA, August 1994.

[55] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Publishing Company, 1996, 0201633612.

[56] Adam Beguelin, Jack Dongarra, Al Geist, and Vaidy Sunderam, "Visualization and Debugging in a Heterogeneous Environment", Computer, 26(6):88-95, June 1993.

[57] C. J. Fidge, " Logical time in Distributed Computing Systems", IEEE Computer, 14(8):258-33, August 1991.