# Abstract

Grids are managed Internet scale distributed services supporting collaboration and virtual organization. Collaboration over networks is becoming more and more popular and important in such areas as online conferencing, distance education, e-Science, e-Business, and e-Learning. We research the topic of shared event collaboration on Grid; we build three prototypes to investigate the issues from different software models and languages; we demonstrate this by making clients (collaboration entities) collaborate on events, taking advantages of the Grid for delivery of event messages and the computing powers of the clients' host computers. Grids are built from Web Services exchanging messages.

Peer-to-peer Grids exploit the observation that the entities in both P2P systems and Grids are autonomous agents (peers, services) whose state is determined by exchanging messages. Our work addresses Peer-to-peer Grids and leverages the advantages of both so that they complement each other. We have developed collaborative prototypes in PowerPoint, Impress, and ReviewPlus applications to demonstrate this, using Message Oriented Middleware (NaradaBrokering Message Service) to provide a Grid messaging

substrate that can support collaborative Grids. The work with ReviewPlus is in Interactive Data language (IDL), where the complete and proper definitions of event structures are supplied and elegant mechanisms of widget programming are guaranteed; the events are systematically and completely defined with respect to structures, levels, configurations, and associated types of event handler routines. The prototypes are instantiations of the Shared Event Model and demonstrations of the collaborative Peer-to-peer Grids.

In order to promote the collaborative applications' usefulness and universal accessibility, we investigate a "Thin Client Collaboration Web Services" client architecture. We further develop a theoretical framework in terms of deterministic finite automata to describe our collaborative systems. We use the prototypes to define a general approach to shared event collaboration on the Grid.

# GRID-BASED COLLABORATION

By

MINJUN WANG

B.S. Jilin University, 1989

M.S. Syracuse University, 1998

# DISSERTATION

Submitted in partial fulfillment of the requirements for the

degree of Doctor of Philosophy in Computer Engineering

in the Graduate School of Syracuse University

August 2006

Approved_____

Professor Geoffrey C. Fox

Date_____

# Table of Contents

# List of Figures

# List of Tables

# Acknowledgements

First of all, I would like to thank my advisor, Dr. Geoffrey C. Fox, for his guidance and instructions during my years of research. They are invaluable for my achievements.

I would like to thank Dr. Marlon Pierce, Dr. Brian Carpenter, and Dr. Shrideep Pallickara for their kind help in the Community Grid Lab at Indiana University.

I would like to show my gratitude for the people at the General Atomics and Affiliated Companies, San Diego, California, such as Mr. David Schissel, Mr. Justin Burruss, and Ms. Kristi Keith. They kindly helped me and generously provided me with their computing environments, tools, and software in the research.

For my Ph.D. proposal research exam committee members, Dr. Jae Oh, Dr. Roman Markowski, Dr. Vojtek Furmanski, and Dr. Geoffrey Fox, I would like to say thank you very much for your kind help and invaluable guidance for my further dissertation research.

For my Ph.D. final defense exam committee members, Dr. Jae Oh, Dr. Roman Markowski, Dr. Amrit Goel, Dr. Simon Catterall, Dr. Marlon Pierce, and Dr. Geoffrey Fox, I would like to say thank you very much for your kind help and invaluable advices for my future research.

Dr. Carlos R.P. Hartmann and Dr. Kishan Mehrotra of Syracuse University helped me a lot with regard to my minor field in Computer Science. It is really beneficial

in my further study, research, and more to come. I appreciate your patience and help very much.

Our department staff, such as Ms. Barbara Hazard and Ms. Maureen Marano, has been very helpful during all the years of my academic life in the Electrical Engineering and Computer Science department of Syracuse University. I appreciate your great service.

I owe my family and relatives countlessly; they have sacrificed so much for me. My parents (especially my mother) and my sisters have missed me so much during the long years of separation. Yet, everyone has been showing me great love and support; everyone has been encouraging me all the time, on the phone, in the mail, in the package … Without this, I wouldn't have gone so far; without this, I wouldn't have achieved this much.

Last but not least, I would like to show my gratitude to my friends, such as Gang Lu, Bin Li, and Xi Rao, for their help and friendship all these years, through the sunny days and raining days, sharing the happiness and burdens. Thank you, Jennifer Carter, for your prayers; your good words and encouragements really have enlightened my soul and lifted me high across the hot waters.

While a simple "Thank you" would never express my sincere gratitude enough to everyone, I would love to say it again – Thank you, thank you very much.

To my family

# Chapter 1

# Introduction

This dissertation is about Grid-based collaboration. In this chapter, we summarize key issues that we determined in the dissertation research about possible architectures and implementations for collaboration on a Grid. They span both theoretical principles and frameworks and practical engineering issues from our three major prototypes – Collaborative PowerPoint, Impress, and ReviewPlus applications. The work builds on earlier principles and prototypes built in Dr. Fox's research groups at Syracuse University and Indiana University as well as broadly understood issues in collaboration and Grid systems.

## 1.1 Research in the Context of Previous State of the Art

Grids can be considered as managed Internet scale distributed services and are often thought of as supporting collaboration [Fox+CTS, Fox+P2PGrid, Snelling+GGF16]; they are substrate supporting the virtual organization that corresponds to the international team, as in modern scientific research. However, most work on Grids focuses on the hard and important problem of secure efficient sharing of data and compute resources. The virtual observatory in Astronomy (International Virtual Observatory Alliance) [IVOA] and the TeraGrid (Teragrid.org) [TeraGrid] are good examples in data and compute areas, respectively. These Grid applications are asynchronous collaboration problems that differ significantly in terms of key issues from the synchronous collaboration considered in this dissertation. Nevertheless, there are common points and often one wants to support both synchronous and asynchronous applications. Peer-to-peer systems and many recent Web 2.0 community (social networking) tools are further important asynchronous Internet collaboration systems [Fox+CTS'06]. Grids are built today totally in web service architecture and are natural for supporting collaboration as all interactions between components in terms of explicit SOAP messages.

Grids are built from Web Services exchanging messages; the Community Grids Laboratory (CGL) at Indiana University has stressed the value of using Message Oriented Middleware (closely related to Enterprise Service Buses like MQSeries used in Industry [MQSeries]) to provide a Grid messaging substrate that can support collaborative Grids. This work provides a secure high performance messaging layer that allows for the needed message multi-cast for collaboration. Earlier work by Dr. Uyar [Uyar+Dissertation] and Qiu established that the simple and general publish-subscribe mechanism was able to

support collaboration with flexibility intrinsic from its architecture, and that a 1-2 millisecond overhead that was negligible. Two upcoming theses (by Mr. Oh and Aydin of CGL of Indiana University) discuss how high bandwidth systems can be designed using the SOAP Infoset. Our thesis tackles complex collaborative applications that refine the requirements for the NaradaBrokering messaging system built in CGL based on the original Dr. Pallickara's dissertation [Pallickara+Dissertation] of Syracuse University, which was partly motivated by the TangoInteractive collaboration system [CollabWorx] built at Northeast Parallel Architectures Center (NPAC) of Syracuse University.

The generality of the shared display approach is obtained at large cost in network traffic. Most approaches to collaboration use what CGL calls major and minor events. Major events contain a complete state description of the finite automata and are used to cope with transmission errors (inevitable in UDP based audio-video conferencing) and new clients that arrive after the session has started. Minor events contain updates and must be applied in order after the preceding major event. Major events require more processing and are only sent occasionally so they impact the overall performance by much.

Peer-to-peer Grids [Hwang, Fox+ACM] exploit the observation that the entities in both P2P systems and Grids are autonomous agents (peers, services) whose state is determined by exchanging messages. Grids typically have identified "central" servers and "thin" peer clients while P2P systems combine client and server (service) in the same machine. Our work addresses both of these cases.

Collaboration environments support Virtual Organizations (VO) or equivalently sessions (typical collaboration term) that at their heart have a list of clients and

"applications" (such as PowerPoint or an audio-video stream) together with a set of authorization attributes. The session service specifies which clients participate in which applications and their roles that include administrative privileges, and possession of the master token (floor control). XML General Session Protocol (XGSP) has been developed in CGL to support this, but it is incomplete and not integrated with approaches like Virtual Organization Management System (VOMS) [VOMS] from the Grid. Our thesis research showed how complex applications can use this VO model.

## 1.2 Introduction of Our Work

The Internet is a network of networks. It brings intelligence, knowledge, computing power, data, and people together from virtually every corner of the world. The advantages of it are enormous. Cooperation across the boundaries of companies, organizations, and institutions is becoming more regular; collaboration within or between groups of people is becoming more important. Grids offer consistent computational and informational environments that enable applications to make use of resources managed by diverse organizations worldwide. Grid-based collaboration has many strengths in respect to such areas as management performance, fault tolerance, and security.

There has been a lot of software developed for collaboration over the Internet. Application areas include online conferencing, distance education, e-Science, e-Business, e-Learning, and web services. Applications already in industry, such as WebEx [WebEx], CollabWorx (Tango) [CollabWorx], Interwise Glance [Interwise], Groove Networks [Groove], and Placeware [Placeware], have proved to be useful, efficient, and beneficial.

This means – among other things – that distance or location is no longer a barrier or restriction, and time, resources, and money are saved. Communication systems and brokers are used to communicate the information among the collaborators.

One way of collaboration is to actually transfer huge amounts of data over the Internet to the collaborators for rendering. In Shared Display such as Virtual Network Computing (VNC) [VNC], it is based on transferring image data like bitmaps. In Audio/Video conferencing systems, it is based on transferring blocks of audio and video data, as shown in Dr. Ahmet Uyar's dissertation [Uyar+Dissertation]. These cases are suitable in dynamic situations such as impromptu conferencing, and they need heavy network traffic.

Another way of collaboration is to collaborate between the collaborators by transferring small text event messages and rendering the same results on the event messages. It is suitable in less dynamic situations such as using presentation slides, files, and interactive interfaces in the collaboration. Before the session of the collaboration, the necessary prepared data (slides, files, etc.) for the presentation are deployed or downloaded to the host computers of the collaborators in consistent directories, and the relative supporting software (e.g., PowerPoint) is installed on all the host computers.

In this dissertation we explore the concept of Grid-based collaboration using event. We demonstrate the concept by making three interface applications collaborative between computers over networks, using a common message broker as the underlying communication system. To achieve the global collaboration, we have brought together in the research a Shared Event model, different implementing structures, methodologies,

and technologies. We represent the applications' event structures in messages, which are coordinating the Grid-based collaboration.

We have explored the Grid collaboration architecture with the three collaborative applications, which are Grid-based collaboration tools using the Shared Event Model. These applications are the Collaborative PowerPoint applications [Wang+JDIM, Wang+CATE'04], the Collaborative Impress applications [Wang+KSCE], and the Collaborative ReviewPlus IDL applications [Wang+ITCC'05]. They are instantiations of the Shared Event Model [Wang+SCI, Wang+KSCE] in Grid-base Collaboration and can be used in e-Learning, distance education, online conference, and e-Science.

We have designed the overall structure of each type of the three collaborative applications to consist of a Master (or Master Client) and a Participant (or Participating Client) using small text event messages for the communication between them. During a session, the Master captures events in its process, deals with them, generates delimited event messages, and sends the event messages to the Participant for rendering the displays in the Participant's process, so that both of them can share the screen displays simultaneously. There can be multiple Participants working with the Master concurrently and independently.

We have used a common message broker – the NaradaBrokering Message Service [Fox+JGI, Pallickara+JDIM, Pallickara+Dissertation] – as the media for message communication implied by the Shared Event Model. The supporting software – Microsoft Office, OpenOffice/Star Office, or RSI IDL – is installed on both the host computers of the Master and the Participant. If files are needed in a session, they are deployed beforehand on the same directories of the hosts. This deployment guarantees that the

files' access is correct on the hosts under the control of the event messages. All clients are required to be in a session and keep in that session for the whole collaboration session, because an event message coordinates each client to change its current status, and the correct transition to a subsequent status depends on the previous one.

Although the three types of collaborative applications are developed in different languages, technologies, structures, and strategies suitable for their native systems, they share a common nature – collaboration on event messages, which decreases the traffic on the Networks and increases the usage of the computing powers of the clients' host computers. This nature therefore highlights the main tenet of this dissertation: Grid-based collaboration on event messages.

We have realized the shared event idea in the collaboration. We use event messages in the coordinating and controlling of a presentation process between the clients. Compared to Shared Display and Audio/Video conferencing, the short event messages save great bandwidth over the Internet and better leverage the computing powers of the hosts. This realization is fast and efficient.

The clients of each of the three collaborative applications work on Peer-to-Peer Grid computing. There are two categories of systems in this: the Grid system and the Peer-to-Peer systems. High-performance and stable services are in the Grids, such as the Web Services, the Grid Services, and the Common Message Brokers, while the services of the Peers are more convenient and accessible, such as the user developed applications and the commodity applications.

The infrastructure of the Internet ties up and correlates the two categories. It enables the Peer-to-Peer Grid computing to be useful, which harnesses the advantages of the two

categories so that they complement each other. The Peer-to-Peer Grid computing is the basis of the collaboration. The Grid system largely comprises stable, formal, efficient, and high-functionality services, such as the Web Services, the Grid Services, and the Common Message Brokers, which are deployed in the Grid on structured, well-organized, and powerful supercomputers. The Peer-to-Peer system offers user-friendly, convenient, intuitive, and easy accessible applications and services, such as the popular commodity software used daily and everywhere. They are installed on a variety of personal devices, such as desktops, laptops, PDA's, and smart phones. The Grid system offers robust, structured, and security services that scale well in pre-existing hierarchically arranged enterprises or organizations; it is largely asynchronous and allows seamless access to supercomputers and their datasets. The Peer-to-Peer system is more convenient and efficient for the low-end clients to advertise and access the files on the communal computers; it is more intuitive, unstructured, and largely synchronous.

We have designed and developed the collaborative applications to demonstrate the Peer-to-Peer Grid computing idea. We deploy the NaradaBrokering Message Service as the broker in the Grids and use it for event message communications between the Master and Participants of the collaborative applications; we deploy the Master and Participants as the Peers at the edge so that they collaborate on event messages through the Grids in the core.

Scientifically, we can generalize the collaborative applications on PowerPoint, Impress, and ReviewPlus to "Deterministic Finite Automaton-based Collaboration Entities." We can think of the elements of these collaborative applications (the Master clients and the Participant clients) as Collaboration Agents, or preferably, Collaboration

Entities in Peer-to-Peer Grid [Oram, Foster+Kesselman, Berman+Fox+Hey]. The elements in all the collaborative applications are just different types of entities. These entities are finite automaton-based in a collaboration session; in essence, they are just finite automata, or deterministic finite automata in the session. Intuitively, the entities in collaboration (the Master and Participants) collaborate on events to keep showing the same output displays at each step, with the Master in controlling by capturing events, generating, and sending out event messages to Participants through a message broker, and with the Participants in responding by rendering the received event messages. Specifically, they collaborate to share a common finite automaton in their respective instantiations (the finite automata in them are the same) and reach a common state of the finite automaton at any collaboration step. Collaboration of the entities is therefore defined by being in a same state of the finite automaton at each event.

We demonstrate that the collaboration entities of a type converge on a common state of the deterministic finite automata at a collaboration step, even though they diverge in many other respects. The Master and Participant are designed for different purposes, in different architectures, implementing mechanisms, and shapes of codes; they are divergent. At the same time, they have the same logic as to the state transitions on events and get to the same state at the end of the process of each event; they are convergent.

The Web Service Architecture is designed to promote software's usefulness, interoperability, availability, extensibility, accessibility, etc. If the collaboration applications are made to be Web Services (WS), totally or partially, they will be more powerful and beneficial to people – different research groups, institutions, organizations, and even ordinary users. In this dissertation, we describe some possible ways of making

such software to be Web Service, propose and focus on discussing the idea of Thin Client Collaboration Web Services, and explore some potential scenarios of this idea in which it shows its merit and the freedom resulted in collaboration.

The idea of the Thin Client Collaboration Web Services is as follows: instead of packaging the whole package of a collaboration software application as Web Service, it separates the native interfaces from the rest of the software; it correlates the native interfaces (in whatever format and language) to web service friendly user interfaces (such as in SVG format and Java language), including their screen displays and events originated from the triggering of the widgets inside the displays; the users access the resulting user interfaces as if they were the native interfaces, relying on the fact that the Web Service has made them one-to-one correspondence functionally, both on the corresponding parts of the displays and on the corresponding events of the widgets.

Such a Web Service has two sets of ports: User-facing Input/Output ports and Resource-facing Input/Output ports. The user-facing I/O contacts a Web Service viewer, and the resource-facing I/O contacts a collaboration application. Hence, the role of the Web Service is to transcode in both directions between the two sets of ports with respect to displays and events, so that the user accesses the Web Service viewer as if he were accessing the collaboration application itself. We use the three types of collaborative applications as resources and projects in SVG as the demonstration of our initial effort toward the implementation of a General Thin Client Collaboration Web Service.

We have used many different methods, structures, and technologies in the research of Grid-based collaboration on events. We are always open-minded to explore new strategies and approaches.

## 1.3 Related Work

Our research is related to the technologies and researches in areas including online conferencing, distance education, e-Learning, e-Science, and e-Business. Some products in the areas are already prospering in industry, and the others are in earnest research and of great interest to many people. We outline some of the areas as follows.

WebEx [WebEx] is a commercial package of services for web conferencing and online meetings. It offers video conferencing service via web browser and audio conferencing service via phone. It uses a private, global network called WebEx MediaTone Network for the on-demand delivery of WebEx applications, and it is secure.

Microsoft Office Live Meeting [Placeware] is an online, hosted service for conferencing. It allows people to collaborate online either in individuals or with large groups. It offers media-rich set of tools such as Whiteboard, Text Slide, Web Slide, Annotations, and Chat. The feel and look of Live Meeting is as Microsoft Office. Live Meeting enables users to initiate meetings from Microsoft Office applications such as Word, Excel, PowerPoint, Outlook, Project, and Visio, or from instant messaging applications such as MSN Messenger and Microsoft Windows Messenger, so that Windows/Microsoft Office users feel at home in using the service.

Virtual Network Computing (VNC) [VNC] is a cross-platform remote control solution. It works on the Shared Display model, and it is open-source, small, and simple. It enables you to access another computer from one computer, as if you are controlling that computer directly, no matter whether the computers are of the same platform or not.

With it, you can even access any computer in any platform from a Java Viewer or Java-capable browser. Therefore, it creates a hot-desking and road-warrior environment. People can access their desktops and severs in office from their PCs at home or from wireless devices on their trip. VNC can also allow simultaneously access one computer from multiple computers. This can be used in distance education. Students from different locations can access their instructor's computer at the same time and share the same contents as in a virtual classroom. Likewise, the instructor can access the students' computers and provide assistance.

Blackboard Learning System [Blackboard] and WebCT Learning System [WebCT] are commercial software that innovate education and learning throughout educational institutions worldwide. They make use of the full power of the Internet, bring students together who are distributed around the world, and promote e-Learning. In other words, they help breakdown all kinds of barriers and increase learning opportunities. Through the Internet, they enable collaboration between the instructor and students in the form of a virtual classroom, or among students in the forms of study groups. They facilitate education, learning, and management.

Access Grid (AG) [AG] has strength in collaboration for large groups of participants and sites instead of small size of desktop-to-desktop individuals. It is an ensemble of resources (projectors, cameras, microphones) connected by networked computers for audiovisual conferencing. The global conferencing service takes advantages of the Grid infrastructure for its high performance. AG also provides interfaces to Grid middleware and visualization environments for further development of new collaborative tools and

applications to enhance the quality and performance of its global audiovisual conferencing.

JXTA peer-to-peer technology [JXTA] is a set of open protocols that allow diverse connected devices (ranging from cell phones, personal digital assistants (PDA), personal computers, to servers) to communicate and collaborate in a peer-to-peer manner. It is platform independent, embracing diverse languages, platforms, and networks. It allows different peer-to-peer systems and communities to interoperate. Its interesting usage includes 1) finding peers and resources on the network, even those behind firewalls and NATs; 2) sharing files with peers online; 3) dynamically creating groups of peers across different networks; 4) communicating with peers securely via the open networks.

CollabWorx [CollabWorx] is a package of Integrated Web Collaboration Solutions. It is based on web browsers with collaboration tools added to the browsers, so that collaboration is achieved in the cyberspace through the users' familiar browser environments. The collaboration tools serve in areas of audio, video, instant messaging, browsing, and file and application sharing. The solutions include communication and collaboration web services (e.g., secure audio and video conferencing, secure instant messaging, and secure virtual meeting), knowledge management web services (e.g., virtual classroom, courseware management system), and on-line customer support. CollabWorx provides secure and high-performance end-to-end solutions for collaboration, communication, and distance learning on multi-platforms. All the collaboration tools have developed on the CollabWorx platform, which leverages backend databases, clusters of HTTP/HTTPS servers, real-time messaging servers (meeting engines), middleware layer plug-ins, and the browsers. The collaboration tools are in the forms of

Java applets, ActiveX controls, and scripted web pages, which are added to the standalone browsers to make them collaborate. The CollabWorx platform uses event sharing technology, which synchronizes the states of a session on events. An event occurs when a URL is loaded in a browser, data are input in a web page's form, or a page is scrolling. The collaboration tools for the instances of the browsers cooperate with each other on the events (through the middleware plug-ins and the messaging servers), so that browsers are now collaborating. For audio and video collaboration, the CollabWorx platform differentiates the events and the media data and processes them differently; after the events, the media data are transmitted on Real Time Streaming Protocol (RTSP) or Real Time Protocol (RTP) via the private channels of native applications. Compared to the media data, the size of event data is much smaller. Document Object Model (DOM) is used for communication between system objects for the collaborative browsers.

Other related areas in industry include Polycom Conferencing Environment [Polycom], Groove Desktop Collaboration Software [Groove], Centra Collaboration Environment [Centra], Interwise Enterprise Communications Platform [Interwise], Jetspeed Enterprise Portal from Apache [Jetspeed], and Jabber Instant Messenger [Jabber].

In our research we need an efficient, high-performance, and secure message service (such as Java Message Service [JMS, Monson-Haefel]) as the underlying communication system for our collaborative applications. We have chosen the NaradaBrokering Message Service for the purpose, which was first developed by Dr. Pallickara and later has been improved and developed by a team led by him in the Community Grid Lab at Indiana University, USA. The NaradaBrokering Message Service is a system that supports

messaging in Peer-to-Peer Grid environment; it uses a generalized publish/subscribe mechanism; it handles dynamic protocol choice and tunneling through firewalls; it supports TCP, UDP, multicast, SSL and RTP; it can run in client-server mode as Java Message Service (JMS) or in distributed Peer-to-Peer mode like JXTA; it can be used in real-time synchronous collaboration.

## 1.4 Motivation

Collaboration over the Internet is becoming more and more popular and important in learning, education, engineering, science, and business. It presents itself mainly as a humble prefix e- to the categories (e.g., e-Learning), yet it transforms them magically and brings new life and challenges to them. “e” is for electronic. To collaborate electronically needs huge amounts of information communicated, as in the cases of Virtual Network Computing (VNC) and Audio/Video conferencing. This causes great traffic on Network, and the bandwidth of Network becomes the bottleneck for the performance of the collaboration. While people can always improve the bandwidth and ability of Networks, it is a good effort to try other ways of collaboration that will reduce the traffic on Networks and make full use of the computing powers of host computers. This motivates us to do the research: Grid-based collaboration on events.

The influencing collaboration tools in research and industry, such as Virtual Network Computing (VNC), Microsoft Windows NetMeeting [NetMeeting], Microsoft Office Live Meeting, WebEx, Blackboard Learning System, WebCT Learning System, Access Grid, Polycom Conferencing Environment, Groove Desktop Collaboration Software,

Centra Collaboration Environment, and Interwise Enterprise Communications Platform, are basically working on the Shared Display model. However, few collaboration tools are taking the advantages of event sharing technology. A very successful company new in industry is CollabWorx, Inc., which uses its patented event sharing technology in developing collaborative web browsers. An event occurs when a URL is loaded in a browser, data are input in a web page's form, or a page is scrolling; the collaborative web browsers collaborate on the event. The whole software package is called CollabWorx. CollabWorx has integrated web collaboration solutions in areas of audio, video, instant messaging, browsing, and file and application sharing. It offers distinct benefits over other products, such as its extensive set of collaboration tools, adaptability, extensibility, scalability, open platform, low total cost of ownership for end users, authoring system independence, and easy data sharing [CollabWorx]. Take its scalability for an example, CollabWorx can handle even thousands of concurrent users (depending on the network conditions and the event interactivity level), while the other Shared Display based products break down when the number of users exceeds a handful. The achievements and potentials of CollabWorx (former Tango) further motivate us and encourage us to do the research: Grid-based collaboration on events.

In this research, we will make the collaborators/clients collaborate on events, with one client (the Master) in controlling of the process by capturing the events, getting and sending out the event messages to the other clients (Participants) for rendering the same output results. The event messages are small text strings that contain information to coordinate/control the process of the collaboration among the clients. This uses the Shared Event Model.

The collaboration also takes advantages of the high performance of the Grid infrastructure and services for the deliverance of the event messages, and the computing powers of the host computers of the clients. The clients work on a Grid-based Collaboration Paradigm, in which the Shared Event Model acts as the messenger and the Peer-to-Peer Grid computing acts as the basis. There are two categories of systems in this paradigm: the Grid system and the Peer-to-Peer systems. The Grid is in the core and the Peers are at the edge of the paradigm. High-performance and stable services are in the Grid, such as Web Services, Grid Services, and Common Message Brokers; the services of the Peers are more convenient and accessible, such as user developed applications and commodity applications. The Peer-to-Peer Grid computing is the basis of the collaboration. It is a new trend in scientific computing and collaboration. It is based on the Peer-to-Peer and Grid technologies and leverages the advantages of both.

We will demonstrate the idea by making three interface applications collaborative between computers over networks, using a common message broker as the underlying communication system. The three types of collaborative applications will be the Collaborative PowerPoint applications, the Collaborative Impress applications, and the Collaborative ReviewPlus IDL applications. They will be instantiations of the Shared Event Model in Grid-base collaboration and collaboration tools that can be used in e-Learning, distance education, online conference, and e-Science.

We can deploy the NaradaBrokering Message Service as the broker in the Grid and use it for event message communications between the Master and the Participants of a type of collaborative applications; we can deploy the Master and the Participants as Peers

at the edge so that they collaborate on event messages through the Grid in the core. This process will be demonstrations of Peer-to-Peer Grid computing.

The success of our prototypes and demonstrations will imply that the idea can be applied to the Microsoft office suite, OpenOffice suite, and other IDL applications, because PowerPoint is one application of Microsoft office suite, Impress is one application of OpenOffice suite, and ReviewPlus is one application in IDL. It will also imply that it can be applied to other categories, such as Matlab, Mathematica, and Maple.

In engineering, we will make the clients have the same output displays at each event (or collaboration step) so that collaboration is achieved. In science, we will generalize the efforts behind all the types of the collaborative applications; we will model and analyze the collaboration process. Theories like finite automaton and Petri net will be good candidates for the purpose regarding the idea of Grid-based collaboration on events. We will also research the timing issues in collaboration with respect to the clients, message brokers, and the networks.

In order to promote the collaborative applications in areas such as usefulness, availability, and universal accessibility, we will explore and resort to solutions in web services; we will try to find some possible ways of making such software to be web services or making use of the software in some web services. Scalable Vector Graphics (SVG) is W3C's vector representation of text and graphics. SVG data can be rendered in popular web browsers and various SVG viewers, on workstations, desktops, and wireless devices like PDA's. If the web services can translate the native interfaces of the collaborative applications to the SVG equivalences, and correlate the corresponding output displays and events in both directions of the two parts, so that users access the

browsers and SVG viewers as if they were accessing the native interfaces of the collaborative applications, that will fulfill the goal of promotion. The collaborative applications can be deployed globally on powerful servers, and the web services can use the collaborative applications on the servers that are closest to them. The collaborative applications can be used by themselves alone; the web services will make them more useful, available, and universally accessible.

We will always like to find new approaches for collaboration over the networks. We will try to find some general solutions or ideas for the literature, at least in a category (such as IDL).

## 1.5 Research Questions

Driven by the motivation, we have the following research questions.

1. Is it possible and how is it possible to achieve Grid-based collaboration on events or event messages? Please demonstrate the idea using implementations or prototypes that are developed in different technologies, structures, programming languages, and platforms.
2. Given the diverse collaboration prototypes, what is the common model in them that coordinates the collaborations on events?
3. What are the strategies and therefore the implementing structures for the collaborative applications as to capturing of events and generating of events?
4. How does Peer-to-peer Grid computing play important roles in Grid-based collaboration on events? How do the prototypes along with common message

brokers serve as the demonstrations of the idea of Peer-to-peer Grid computing?

5. What are the scientific bases that serve to model and analyze the Grid-based collaboration on events?
6. How to promote the collaborative applications' usefulness and universal accessibility with the aids of dedicated web services?

With the research questions as guidelines, we unfold the rest of the dissertation with descriptions, illustrations, demonstrations and conclusions.

## 1.6 Organization of the Dissertation

We organize the rest of the dissertation as follows. In Chapter 2, we describe the theory of finite automaton and the characteristics of our usage with the Deterministic Finite Automaton (DFA) in the modeling and analysis for the prototypes. We point out that the Master and Participant clients of a prototype diverge in many other respects, but in logic they converge at a same state of the DFA at each event or collaboration step. The chapters following it are basically demonstrations, proofs, or highlights for this idea. This chapter is the theory basis for the other chapters.

Chapter 3 describes the implementation of the collaborative PowerPoint applications and the logic in collaboration with respect to the DFA. Chapter 4 describes the implementation of the collaborative Impress applications and the logic in collaboration

with respect to the DFA. Chapter 5 describes the implementation of the collaborative ReviewPlus IDL applications and the logic in collaboration with respect to the DFA.

Chapter 6 compares all the three prototypes in respects of implementing structures and event structures. It further implies that in engineering the prototypes are quite different from each other; in science the modeling with DFA generalizes them and the like.

In order to promote the collaborative applications in areas such as usefulness, availability, and universal accessibility, we have proposed and focused on discussing the idea of Thin Client Collaboration Web Services in Chapter 7, and explored some potential scenarios of this idea in which it shows its merit and the freedom resulted in collaboration. Finally, in Chapter 8 we conclude the dissertation and point out the future work.

# Chapter 2

# Collaboration Entities on Deterministic Finite Automata

In this chapter we introduce several types of collaborative applications we have developed in the dissertation research. They are realizations of the Shared Event Model in Grid-base Collaboration and examples of Peer-to-Peer Grid computing. Each application type consists of collaboration entities, and they play different roles in collaboration. These entities are finite automaton-based in a collaboration session; in essence, they are just deterministic finite automata in the session. Intuitively, the entities in collaboration collaborate on events to keep showing the same output displays at each step, with one entity in controlling by capturing events, generating and sending out event messages to the others through a message broker, and the others in responding by rendering the received event messages. Specifically, they collaborate to share a common finite automaton in their respective instantiations, and reach a common state of the finite

automaton at any collaboration step. Collaboration of the entities is therefore all about being in a same state of the finite automaton at each event.

## 2.1 Introduction

We have developed Collaborative PowerPoint applications [Wang+JDIM, Wang+CATE'04], Collaborative Impress applications [Wang+KSCE], and Collaborative ReviewPlus IDL applications [Wang+ITCC'05]. They are instantiations of Shared Event Model [Wang+SCI, Wang+KSCE] in Grid-base Collaboration, and can be used in e-Learning, distance education, online conference, and e-Science. They work on a Grid-based Collaboration Paradigm [Wang+SCI], in which Shared Event Model as messenger and Peer-to-Peer Grid computing [Fox+CTS, Berman+Fox+Hey, Wang+JDIM] as basis.

They are desktop windows collaborative applications. They are based on the stand-alone PowerPoint, Impress, and ReviewPlus applications; they are developed as collaborative applications that enable the otherwise stand-alone ones of each type to collaborate within themselves over the networks. Impress of OpenOffice is a presentation application similar to Microsoft PowerPoint and has similar functionality. ReviewPlus is a general-purpose data visualization tool developed in Interactive Data Language (IDL) by General Atomics of USA. It is used in physics and engineering for displaying 2D and 3D graphs and signals.

We design the overall structure of each of the three collaborative applications to consist of a type of Master (or Master Client) and a type of Participant (or Participating Client) using small text event messages for the communication between them. During a

session, the Master captures events in its process, deals with them, generates delimited event messages, and sends the event messages to the participant for rendering the displays in the participant's space, so that both of them can share the screen displays simultaneously. The Master is in active mode and controls the process of a session. The Participant is in passive mode and is not in control of the process; it just receives the event messages and renders the displays. There can be multiple Participants working with the Master concurrently and independently.

The collaboration is on the Shared Event Model; small size text event messages are communicated between the Master and Participants to synchronize their displays. Compared to other approaches of achieving synchronized views such as Shared Display (that communicates image data like bitmap) in Virtual Network Computing, this method uses small network traffic.

We use a common message broker – NaradaBrokering (NB) Message Service – as the underlying message communication system between the Master and Participant clients. NB is deployed in Grid as a Grid service; the clients are deployed on user computers and are running as Peers using the service for message communication; together they perform Peer-to-Peer Grid computing.

The base software – Microsoft Office, OpenOffice/Star Office, or RSI IDL – is required to install on both the hosts of the Master and the Participant; if files are needed in a session, they are deployed beforehand on the same directories on the hosts. This deployment guarantees the accesses of the files are correct on the hosts under the control of event messages. All clients are required to be in a session and keep in that session for the whole collaboration, because an event message coordinates each client to change its

current status, and the correct transition to a subsequent status depends on the previous one.

In a collaboration session, we can generalize the collaborative applications on PowerPoint, Impress, and ReviewPlus to "*Deterministic Finite Automaton-based Collaboration Entities*." In the session, we can think of the elements of these collaborative applications (the Master and Participant clients) as *Collaboration Agents* in Peer-to-Peer Grid [Oram, Foster+Kesselman, Berman+Fox+Hey], or preferably, *Collaboration Entities*. The elements in all the collaborative applications are just different types of entities. We can model the entities of a type in a collaboration session using finite automata; these entities in the session are finite automaton-based; in essence, they are just finite automata, or deterministic finite automata.

It is viable of this modeling. First, a collaboration session is finite, because human life is finite, and we are only interested in modeling those adequate and meaningful sessions that are finite in time and started and ended normally. Therefore, the events invoked (by a user's interaction with the interfaces on the Master client) and the event messages communicated (between the Master and Participants) are finite.

Second, the interfaces and widgets of an interactive windows application are finite. The event messages from some of them are the same from invocation to invocation, such as a button widget titled "Next," while others are dynamic depending on the interactive inputs, such as a text field widget. Even though in this case the event messages are different in all the widget's invocations, the invocations in a collaboration session are finite and so the associated event messages.

Third, since all our collaborative applications are designed to collaborate on events, we are only interested in the events that actually happened in a collaboration session and model the process of the session on those events. Those events are finite. The occurrence and sequence of events in a session may be different from that in another session, and so the associated modeling finite automata.

The meaning of this modeling is that, we can get a simple, clear, and consistent picture with regard to the collaboration between the entities in a session; we can see through the differences between the entities and logically abstract them to share a common finite automaton in their instantiations in collaboration; we can see the important roles of events and the shared event model in collaboration. The Master and Participant collaboration entities are designed for different purposes, in different architectures, implementing mechanisms, and shapes of codes; they are divergent. At the same time, they have the same logic as to the state transitions on events and get to the same state at the end of the process of each event; they are convergent. Intuitively, the entities in collaboration – the Master and Participants – collaborate on events to keep showing the same output displays at each step, with the Master in controlling by capturing events, generating and sending out event messages to Participants through a message broker, and the Participants in responding by rendering the received event messages. Specifically, they collaborate to share a common finite automaton in their respective instantiations (the finite automata in them are the same) and reach a common specific state of the finite automaton at any collaboration step. Collaboration of the entities is therefore all about being in a same state of the finite automaton at each event.

## 2.2 The Finite Automaton

There are two types of Finite Automaton: the Deterministic Finite Automaton (DFA) and the Nondeterministic Finite Automaton (NFA) [Hopcroft]. Both of them can be represented as a five-tuple notation as follows.

$A = (Q, \Sigma, \delta, q_0, F)$, where

A is the name of the automaton;

Q is the finite set of states of the automaton;

$\Sigma$ is the finite set of input symbols;

$q_0$ is the start state;

F is the finite set of final or accepting states, which is a subset of Q;

$\delta$ is the transition function, which takes as parameters a state from Q and a symbol from $\Sigma$ and returns a state in Q for DFA or a set of states from Q for NFA.

The difference between a DFA and an NFA is that, for a DFA, it is in a single state at any time, while for an NFA, it has the power to be in multiple states at a time. NFA is more succinct and easier to design, while DFA is more feasible and safe in implementation and programming. Any NFA can be converted to a DFA using subset construction (the power set of the set of states of the NFA), and the two are mathematically equal. Even though in the worst case the number of states of the DFA constructed from an NFA is exponentially larger than that of the NFA ($2^n$ vs. n), in most cases and practically, the numbers are almost equal, because most of the states of the constructed DFA are inaccessible or unreachable from the start state, and therefore can be

eliminated. A useful technique in doing so is the one called lazy evaluation, which is effective in keeping all those accessible or reachable states in the power set. Deterministic Finite Automata are suitable for each case of our projects in collaborative PowerPoint, OpenOffice, and IDL applications; the collaboration entities in them are in essence Deterministic Finite Automata. We show the characteristics and demonstrations of the cases in the following sections.

## 2.3 Characteristics of the DFA for Collaboration Entities

There are characteristics in the Deterministic Finite Automata of the collaboration entities in our projects. They are the specialties in the set of input symbols Σ and hence the transition function δ. Traditionally, the symbols in Σ are alphabets, digits, or any printable ASCII characters; the transition function δ takes as parameters a state $q_i$ in Q and a single symbol $s_i$ in Σ (a, b, c, 1, 2, 3, %, $, &, etc.) and returns a state in Q. Specifically, we define in our cases the symbols or units in Σ to be event messages, which are independent text strings. For example, we use the strings "OpenFile;*Dir/filename*", "Goto;*CertainSlide#*", "Previous", and "Next" for collaborative PowerPoint and Impress applications and the string "{Widget_Base;ID:10;TOP:8;HANDLER:10;X:123;Y:456}" (representing the base widget event structure) for collaborative IDL applications. Each such message is defined as a "symbol" in Σ. The transition function δ takes a state in Q and such a symbol in Σ as input and transits to the next state in Q, usually a different one.

By doing so, we encapsulate the low level chores – such as capturing events and getting the event messages, serializing them in text strings for transmitting (on the Master side), de-serializing and parsing the message strings, and building up the event structures (on the Participant side) – into the collaboration entities, thus simplify the modeling, and make it clear that the Deterministic Finite Automata in them are all about collaborations on event messages. This is to use semantically complete event messages as the basic units in the set of input symbols Σ and make the automata concentrate on describing the message-based collaborations.

## 2.4 Unification of the Collaboration Entities

If we observe the collaboration entities in a project – the Master and Participant clients – on lower levels (e.g., design and implementation), they are different things, with respect to strategies, architectures, languages and technologies used, and roles supposed. However, if we consider the entities on higher levels as to state transitions of DFA in question, they share the same state of logic at any step in a collaboration session; therefore in essence, they have the same DFA in their respective instantiations and collaborate on it using event as the messenger. In practice, the entities of the Master and Participant are created for different purposes; they are binary. In theory, they follow the same logic of collaboration and manage to share the same state of a common DFA at a step in a session; they are unity. They are binary so that they serve and satisfy the special requirements as to the capturing of events in the Master and the generating of events in

the Participant. They are unity so that they have the same logic state at any collaboration step in the form of the same output screen.

In more detail, on the entity of the Master client, the user controls the process of a session by physically controlling the interface of the entity using mouse clicks, keystrokes, etc., which we can call physical events. The entity responds to these physical events and navigates through each of the corresponding states; at the same time for each of these events, it builds up an event message regarding information about the event. The event message is a delimited text string and the intermediate representation of the event for transmission / broadcasting via the message broker.

On the entity of the Participant client, it parses the delimited text string [Aho] after receiving it; based on the information, it arranges which function to call, converts all the types of data represented in string to its system's interior representation, and builds up the native event structure. It then automates through each of the states as in the Master by calling a function, mostly with a property or an event structure as the parameter. It is controlled during the process of a session programmatically, which is called automation.

Considered logically, both the Master and Participant entities can be modeled as a DFA in a session. They maintain the same set of states and collaborate on event to be in a same state at any step. The transition function $\delta$ takes the current state q and an event message (for convenience, we use event message to refer to even the native event representation of a system) as parameters and transits to the next state p of the DFA.

In Object-oriented Programming languages like C++, polymorphism is used to refer to objects of classes in different shapes, builds, and configurations yet performing the same logical functions using the same interfaces, as the "print" objects for different

devices of printing hardware as well as the monitor screen. In Peer-to-Peer Grid computing, we can use polymorphism in higher level to reference the instantiations of the collaboration entities (the Master and Participant clients), which are different in shapes, builds, and configurations, but are same in logic of the unity of the Deterministic Finite Automata. This is the Unification of the Collaboration Entities.

## 2.5 A DFA Example in Collaborative PowerPoint and Impress Applications

Let us use a DFA example suitable in collaborative PowerPoint and Impress of OpenOffice to demonstrate the idea of finite automaton-based collaboration entities. Suppose there is a presentation file either in the format of .ppt for PowerPoint or .sxi for Impress. There are three slides in this file: slide 1, 2, and 3. Accordingly, the finite set of states is as follows.

$Q = \{q_0, q_1, q_2, q_3, q_4\}$, where

$q_0$ is the state when the application is started;

$q_1$ is the state for slide 1;

$q_2$ is the state for slide 2;

$q_3$ is the state for slide 3;

$q_4$ is the state when the application is ended.

The finite set of input symbols is as follows.

$\Sigma = \{a_0, a_1, a_2, a_3, a_4, a_5, a_6\}$

Each $a_i$ is an event message, with

$a_0$ = "Openfile;C:/file1.ppt" (or file1.sxi, meaning open this file),

$a_1$ = "Goto;1" (meaning go to slide 1),

$a_2$ = "Goto;2" (meaning go to slide 2),

$a_3$ = "Goto;3" (meaning go to slide 3),

$a_4$ = "Exit" (meaning the application exits),

$a_5$ = "Previous" (meaning go to the previous slide),

$a_6$ = "Next" (meaning go to the next slide).

The start state is $q_0$.

The finite set of accepting states is

$F = \{q_4\}$

We define $q_4$ – the state when the application is exited – to be the accepting state of the automaton. That means the presentation session is normally and adequately finished. As for the transition function $\delta$ is concerned, instead of using many equations $\delta(q_i, a_i) = p_i$ to represent the transition from state $q_i$ to $p_i$ on input symbol (event message) $a_i$, we

make use of two more convenient and clearer means to do the job. They are the transition diagram and the transition table. We will demonstrate them with the example next.

So, the five-tuple notation $A = (Q, \Sigma, \delta, q_0, F)$ for the DFA in this example becomes

$$A = (\{q_0, q_1, q_2, q_3, q_4\}, \{a_0, a_1, a_2, a_3, a_4, a_5, a_6\}, \delta, q_0, \{q_4\})$$

## Transition Diagram

Next, we give the transition diagram for the transition function δ and explain how the transitions go through the states of Q in this example. The transition diagram is in Figure 2.1.

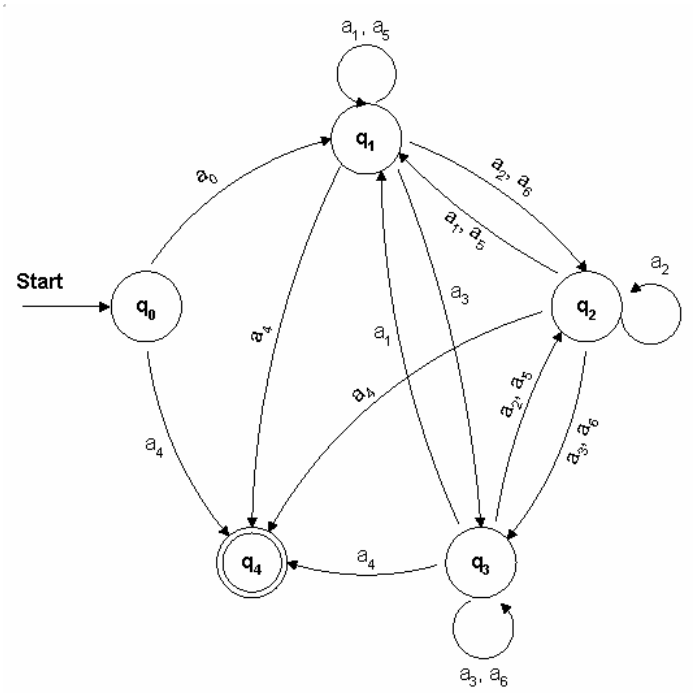



**Figure 2.1 The transition diagram for the DFA of the collaboration entities working on a presentation file in PowerPoint or Impress of OpenOffice.**

## Explanation of the Example

After a collaboration entity is instantiated, it is in state $q_0$, which is the start state and denoted by an arrow. From here, it can exit immediately without doing anything by going to state $q_4$ (on message $a_4$ = "Exit"), or it can go to state $q_1$ (on message $a_0$ = "Openfile;C:/file1.ppt"), at which the presentation file is opened and by default is on slide 1.

From state $q_1$ it can go to state $q_2$ (on message $a_2$ = "Goto;2" or $a_6$ = "Next"), which is the state for slide 2, or it can go to state $q_3$ (on message $a_3$ = "Goto;3"), which is the state for slide 3, or it can go to state $q_1$ itself (on message $a_1$ = "Goto;1" or $a_5$ = "Previous"), or it can go to state $q_4$ (on message $a_4$ = "Exit"), which is the state when the collaboration entity is killed. For slide 1, there is no previous slide for it; so it stays on message $a_5$.

From state $q_2$ it can go to state $q_1$ (on message $a_1$ = "Goto;1" or $a_5$ = "Previous"), which is the state for slide 1, or it can go to state $q_3$ (on message $a_3$ = "Goto;3" or $a_6$ = "Next"), or it can go to state $q_2$ itself (on message $a_2$ = "Goto;2"), or it can go to state $q_4$ (on message $a_4$ = "Exit").

From state $q_3$ it can go to state $q_1$ (on message $a_1$ = "Goto;1"), or it can go to state $q_2$ (on message $a_2$ = "Goto;2" or $a_5$ = "Previous"), or it can go to state $q_3$ itself (on message $a_3$ = "Goto;3" or $a_6$ = "Next"), or it can go to state $q_4$ (on message $a_4$ = "Exit"). For slide 3, there is no next slide for it, so it stays on message $a_6$.

State $q_4$ is the state when the collaboration entity is ended and is the accepting state. Nothing will happen from here; therefore there is no label leading out from it. It is denoted by a double circle.

In this example, we have discussed all flow possibilities for a 3-slide presentation. It implies many possible collaboration sessions. For example, in one session slides 1, 2, and 3 are presented in that order and then the session is ended; in another session the slides may be presented randomly and in any number. In the modeling of one actual collaboration session, the corresponding deterministic finite automaton is a sub-graph of the transition diagram of Figure 2.1.

## Transition Table

The transition table is a two dimensional array representation of the DFA; it is functionally equivalent to the transition diagram. The transition table and the transition diagram can be converted to each other. The transition table for the example is as follows in Table 1. It describes the same results as we mentioned above.

**Table 2.1 The transition table for the DFA of the collaboration entities working on a presentation file in PowerPoint or Impress**

| | $\mathbf{a_0}$ | $\mathbf{a_1}$ | $\mathbf{a_2}$ | $\mathbf{a_3}$ | $\mathbf{a_4}$ | $\mathbf{a_5}$ | $\mathbf{a_6}$ |
|---|---|---|---|---|---|---|---|
| $\rightarrow\mathbf{q_0}$ | $q_1$ | ø | ø | ø | $q_4$ | ø | ø |
| $\mathbf{q_1}$ | ø | $q_1$ | $q_2$ | $q_3$ | $q_4$ | $q_1$ | $q_2$ |
| $\mathbf{q_2}$ | ø | $q_1$ | $q_2$ | $q_3$ | $q_4$ | $q_1$ | $q_3$ |
| $\mathbf{q_3}$ | ø | $q_1$ | $q_2$ | $q_3$ | $q_4$ | $q_2$ | $q_3$ |
| $*\mathbf{q_4}$ | ø | ø | ø | ø | ø | ø | ø |

The first column lists all the states, the first row lists all the input symbols, and each cell at the cross of a row and a column lists the returned value of the transition function δ. For instance, $q_1$ is in the cell at the cross of the row $\mathbf{q_0}$ and the column $\mathbf{a_0}$. With $\delta(q_0, a_0) =$

$q_1$, $q_1$ is the next state from $q_0$ on input symbol $a_0$. The start state $\mathbf{q_0}$ is preceded by an "→" sign, and the final state $\mathbf{q_4}$ is preceded by an "*" sign. The ø in the cells of the table represents "not applicable", "not defined", "ignored", or "dead state". Take the last row of state $\mathbf{q_4}$ as an example. Because $\mathbf{q_4}$ is the final state, any attempt to get to any other state on any input symbol is just meaningless or will lead to a dead state.

## 2.6 Issues about DFA with Collaborative ReviewPlus Applications

In a session, the DFA is determined by the instantiation of the collaboration entity and its inputs later on. In some DFA, all their states are built at early stage; in the others, some of their states are built up on the fly because of interactive inputs. For example, in the example above, all the states of the DFA are decided as early as when the collaboration client is instantiated (state $q_0$) and loads the presentation file (state $q_1$), as $\delta(q_0, a_0) = q_1$. At this stage, all the states are known and also the relationships between them (decided by the presentation file itself), as shown by the labeled arcs between states in the transition diagram in Figure 2.1.

As for the DFA with Collaborative ReviewPlus, some of its states are decided at early stage (e.g. the states related to the original menus and sub-menus of the interfaces when the application is first started), and the others, especially those related to interactive input requirements, are built up dynamically. In such cases, we might as well think the DFA is extended stepwise with the collaboration steps.

Collaboration here is all about the synchronization of the interfaces between the Master and Participant clients at each step. In IDL, the interface consists of all kinds of widgets such as buttons, lists, sliders, tabs, and text fields. The constitution, configuration, and layout of the widgets in the interfaces of an application are coded in its widget programs. The states of the DFA in question are based on the widgets. The relationships between the widgets and the states may be one-to-one correspondence, such as simple button widget (one button click causes the DFA to transit to the next state), or it may be one-to-many, in which one widget corresponds to many states, such as the text field widget. For example, if the text field widget is configured using keyword "/all_events" that means an event is fired whenever the contents of the text field have changed, then it is associated with all the states. Each of the states is the result of an event.

Hence, when the collaboration entity is instantiated (the application is invoked), the initial states associated with the widgets in the interfaces of menus and sub-menus are built up with the instantiation; these are the original states which are determined by the widget programs with their initial values. More states are built up on the fly with new context, contents, and inputs.

We list some parts of the interfaces in ReviewPlus below in Figure 2.2 and 2.3. The widgets in these parts are of the type of one-to-one correspondence; each widget corresponds to the transition to one state when triggered.

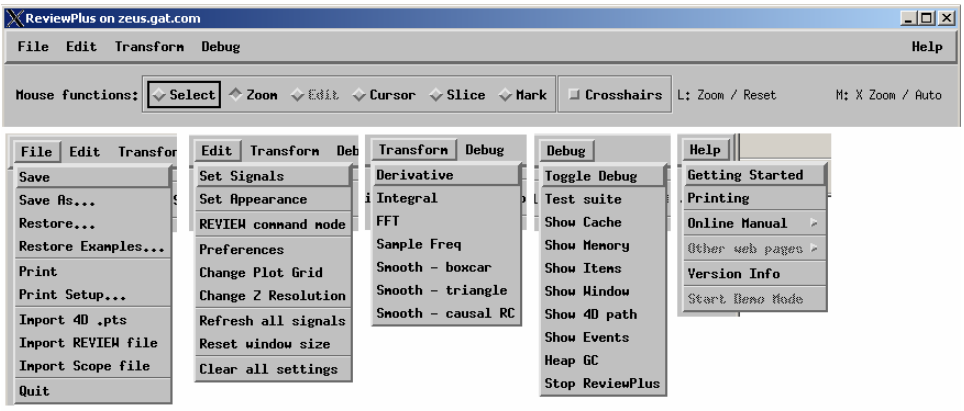



**Figure 2.2 Widgets captured from the interfaces of ReviewPlus that each corresponds to the transition to one state of the DFA when it is triggered at a step of a session.**

**Example 1:** If we click on the "Set Signals" button from the sub-menu of "Edit" in the main menu, the DFA goes from the current state to the next state, which brings up an input table.

**Example 2:** Suppose we are in the context of a selected signal of the display of ReviewPlus. If we click on the "Derivative" button from the sub-menu of "Transform" in the main menu, the DFA goes from the current state to the next state, which shows the derivative of the signal.

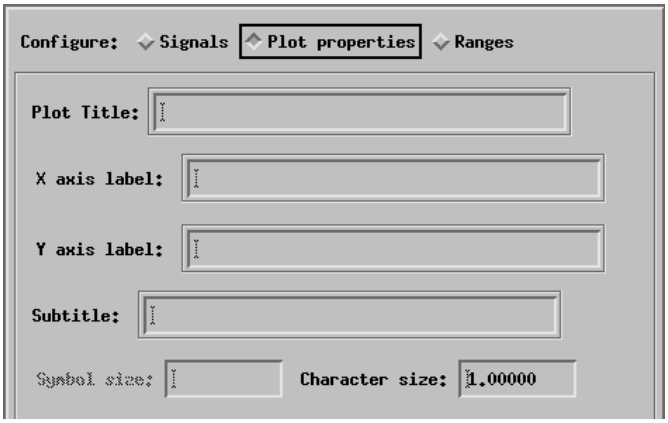



**Figure 2.3 Widgets captured from an interface of ReviewPlus that each corresponds to the transition to one state of the DFA when it is triggered by the event of the carriage return.**

**Example 3:** All the "cw_field" widgets in Figure 2.3 above are configured with the keyword "return_events" in the widget program, which means that an event is fired when the carriage return key is pressed in the text field. The fields that are titled "Plot Title:," "X axis label:," "Y axis label:," and "Subtitle:" are of string type. Any string content typed in the field is reflected in the event structure as a single string value after the pressing of the return key, and the DFA transits to the next state with this value. The same is true for the last two fields titled "Symbol size:" and "Character size:" except that they are of floating-point type.

Next we list in Figure 2.4 some parts of the interfaces in ReviewPlus where the widgets are of the type of one-to-many correspondence; each widget corresponds to the transition to one or more states when the input is finished.

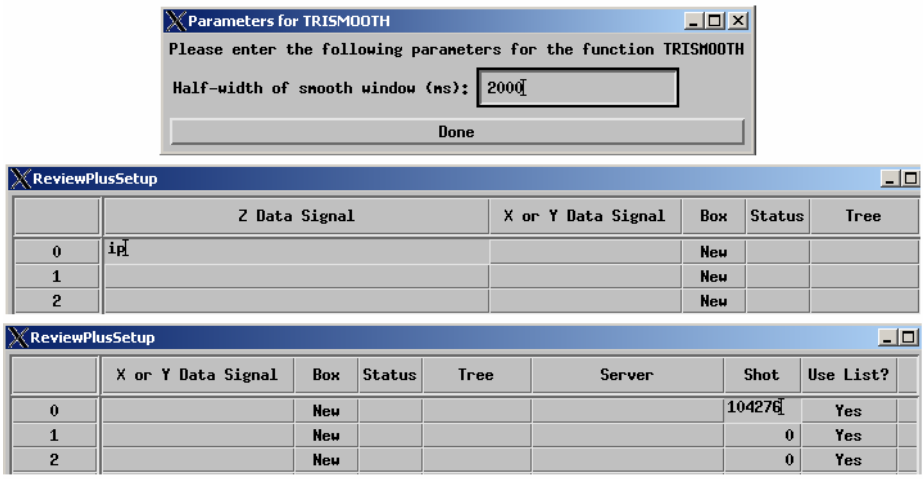



**Figure 2.4 Widgets captured from the interfaces of ReviewPlus that each corresponds to the transition to one or more states of the DFA when the input is finished.**

**Example 4:** All the input cells of the widgets above are configured in the widget programs with the keyword "all_events", which means an event is fired whenever the contents of the text field have changed. The field that is titled "Half-width of smooth window (ms):" is of long integer type, and the final value is 2000 before clicking the "Done" button below it. Each input character triggers an event, including the ending hexadecimals "0a" and "0d" for line feed and carriage return, if there are some. If we look at the values only, the DFA transits through the states with the values 2, 20, 200, and 2000, one by one before reaching the final value 2000. The same is true for the fields with the string value "ip" and the long integer value "104276" under the titles "Z Data Signal" and "Shot", respectively. They are used to define a signal display in ReviewPlus. This feature of "all_events" for a widget makes it possible to show more detailed process in collaboration, as opposed to the case with the feature "return_events" in Example 3,

where only the final values in the fields are collaborated, without showing the details of the inputs in the participant clients.

## 2.7 Extended Transition Function with Collaboration Entities

The type of collaboration entities we have described so far is synchronous; the Master and Participants are cooperating in the same session and sharing the same output displays in real time. The other type of collaboration entities we need to describe is asynchronous. The data about the collaboration during a synchronous session are recorded and saved. The asynchronous entities can access the data at any time thereafter, in any possible way, taking advantage of the *Extended Transition Function* $\Delta$ [Hopcroft]. More specifically, we can make the synchronous entities save the event messages $a_i$ in a session in the order they happened and connect them in a string $\omega$, as in

$$\omega = a_0 a_1 \ldots a_n$$

Later on in the asynchronous access, the Extended Transition Function $\Delta$ makes use of any prefix of the string $\omega$ (e.g., $a_0 a_1 \ldots a_i$ with $0 \leq i \leq n$) and transits to state $q_{i+1}$. This means that the users with the asynchronous entities can review the content happened in a session in a way that is sequential access, random access, or even "keyword search" based access to any history display of the contents. The "keyword search" based access forms the concept of *Reverse Indexing on Event Messages*.

## 2.7.1 Extended Transition Function

The Extended Transition Function Δ is a function that takes a state q and a string ω and returns a state p, as

$$\Delta (q, \omega) = p$$

The automaton starts at state q, processes the sequence of string ω, and finally reaches state p. It is defined by induction on the length of the string ω, as follows.

**BASIS**: $\Delta (q, \varepsilon) = q$. That is, if the automaton is in state q and reads no input or a null string, it is still in state q.

**INDUCTION**: Suppose $\omega = a_0a_1 \ldots a_n$, $x = a_0a_1 \ldots a_{n-1}$, $a = a_n$, we can write $\omega = xa$, in which "a" is the last symbol of the string ω and "x" is the rest of it. Then,

$$\Delta (q, \omega) = \delta (\Delta (q, x), a)$$

The Extended Transition Function Δ is based on the Transition Function δ. Let

$$\Delta (q, x) = p$$

Then,

$$\Delta (q, \omega) = \delta (\Delta (q, x), a) = \delta (p, a)$$

That is, for any length of string x, if the final state is p due to the transitions on the sequence of x, then the next state on one more input symbol a is decided by the transition function δ, as in $\delta(p, a) = r$.

## 2.7.2 The Language of a DFA

We have defined the symbols of Σ to be event messages. In a collaboration session between the entities, the actual event messages are finite, which is mainly decided by the finiteness of the session. Let

$$\Sigma = \{a_0, a_1, a_2, \ldots, a_n\}$$

All the event messages $a_i$ in Σ could have formed random strings in any length, any combinations of the $a_i$'s, and in any order. Examples are ε, $a_0$, $a_1$, $a_2$, $a_0a_0$, $a_1a_1a_1$, $a_0a_1a_2$, $a_0a_1a_2\ldots a_n$, $a_3a_na_2\ldots a_0$, and the like. We denote the set of all strings constructed from the symbols in Σ to be $\Sigma^*$.

Not all the strings in $\Sigma^*$ are possible or meaningful for a collaboration session. We are only interested in those strings that cause the DFA to go through transitions from the start state $q_0$ to an accepting state in F, such as $\omega = a_0a_1 \ldots a_n$. When we refer to a collaboration session, we mean such a successful one that leads to an accepting state.

One collaboration session is associated with one event message string $\omega = a_0a_1 \ldots a_n$. All such strings form a *Language* for a type of collaboration entities. The type can be Collaborative PowerPoint, Collaborative Impress, Collaborative ReviewPlus, or others.

If the DFA for the type of collaboration entities is $A = (Q, \Sigma, \delta, q_0, F)$, then the language L (A) is defined as

$$L(A) = \{\omega \mid \Delta(q_0, \omega) \in F\}$$

That is, the set of strings which cause the DFA to go through transitions from the start state $q_0$ to an accepting state in F.

## 2.7.3 Random and Sequential Access

In random access of an asynchronous session, the user directs the entity to randomly go to an event message $a_i$ $(0 \leq i \leq n)$ in string $\omega = a_0 a_1 \ldots a_n$, to generate the corresponding state p, and to render the output display. The entity does the job by taking advantage of the Extended Transition Function $\Delta$, as

$$\Delta(q_0, x) = p, \text{ where } x = a_0 a_1 \ldots a_i.$$

The entity basically begins with the start state $q_0$, goes through all the transitions in response to each $a_j$ $(0 \leq j \leq i)$ in x, and finally gets to state p on input symbol $a_i$.

In sequential access, the Extended Transition Function $\Delta$ can be used in the same way as in random access, but since in sequential access the symbols in string $\omega = a_0 a_1 \ldots a_n$ are accessed sequentially one by one from $a_0$ going forward to $a_n$, the entity can just take advantage of the current state p in memory, get the next symbol $a_j$ $(0 \leq j \leq n)$ in the

remaining string $a_j a_{j+1} \ldots a_n$ of $\omega$, and go to the next state r by the transition of the Transition Function $\delta$, as $\delta(p, a_j) = r$. The basis of $\Delta$ is $\delta$.

## 2.7.4 Reverse Indexing on Event Messages

The Web Browsers nowadays have keyword search mechanisms to find relative web sites based on the input keywords and list them for the user to click on. One of the most popular search engines is Google. The technique they use is the one that is called "Reverse Index" – keywords associate with web sites. We can apply this technique to the event messages $a_i$ of string $\omega$ in an asynchronous session.

An event message $a_i$ in this case corresponds to a state q; the state q in turn corresponds to an output display; the output display corresponds to some contents; from the contents keywords can be generated; therefore we can associate the keywords with the event message $a_i$, and hence this becomes *Reverse Indexing on Event Messages*. Later on, the user with the asynchronous collaboration entity can use keywords to get event messages $a_i$ and then $x = a_0 a_1 \ldots a_i$; then he use $\Delta(q_0, x) = p$ to get to the states and find the contents. Further more, this can bring different languages into collaboration. Let us use $\omega_1 = a_0 a_1 \ldots a_k$ to denote the strings in the language of the entities of the Collaborative PowerPoint, $\omega_2 = b_0 b_1 \ldots b_m$ to denote those of the Collaborative Impress, and $\omega_3 = c_0 c_1 \ldots c_n$ to denote those of the Collaborative ReviewPlus. They are different languages for different purposes. PowerPoint and Impress are designed mainly for the presentation of text, while Reviewplus for graphics and images, 2D or 3D.

Suppose a lecture was presented using all the three types of the above collaboration entities, the event messages were saved in all the three languages, and the keywords

(associated with the event messages) for the related contents are consistent. Then the user in an asynchronous session can use keyword search to find the text representations of the contents (in both the asynchronous entities of the Collaborative PowerPoint and the Collaborative Impress) and the graphic/image representations (in the asynchronous entity of the Collaborative ReviewPlus).

## 2.8 Logical Consensus

In this section, we use the Collaborative ReviewPlus as an example to describe some issues of the collaborative applications, mainly focusing on event and logic with the applications. We shall see that the Master and Participant clients share a common Deterministic Finite Automaton (DFA) in a session, have the same logic with regard to the state transitions, and converge on the same state on each event. The same holds for Collaborative PowerPoint, Collaborative Impress, and other such collaborative applications.

### 2.8.1 Units and Unity

We have developed the Collaborative ReviewPlus applications – the Master and Participant collaboration entities – from the original ReviewPlus application, without changing the overall logic related to state transitions. So they have the same logic with regard to the state transitions on events. The logic corresponds to the transition function $\delta$ of the DFA, or the extended transition function $\Delta$. The logic is composed of many IDL routines – procedures and functions with unique names. We can think of the routines as

the building blocks or *units* of the logic and the logic as the *unity* of the routines. So, routines are the units, and δ or Δ is the unity of the units. On an event, only one or some routines are executing to do the transition; in other words, only one part or some parts of the unity are actually functioning. But we can indistinguishably say that δ or Δ is reacting on the event and transiting to the next state.

## 2.8.2 Divergence and Convergence

The Master and Participant collaboration entities are designed for different purposes, in different architectures, implementing mechanisms, and shapes of codes; they are divergent. At the same time, they have the same logic as to the state transitions on events and get to the same state at the end of the process of each event; they are convergent.

Let us describe it in more detail using the implementation of the Collaborative ReviewPlus applications as the example. It is similar for the other collaborative applications. On the Master client, each widget that fires event is associated with an event handler – either a procedure or a function – in the widget construction programs, which are registered at the end of the constructions with the IDL system routine ("xmanager.pro"), which in turn is managing the life-cycle of the widgets and listening for events from them. Whenever a widget is triggered by the user through the interface, the system automatically gathers the information for the event, fits in the event structure, and invokes the event handler with the event structure as the only parameter.

We add the code for collaboration at the beginning of each event handler to capture the event, get the information of it for every field of the event structure, convert them into flat strings, serialize them into a delimited single string along with names of the event

structure and the event handler, and send this result string to NB message broker for broadcasting to participant clients. NB broadcasts the string to the participant and saves it in a public variable, which is one element of a synchronized linked list added in one of NB's interface class; NB also updates the event flag variable, which reflects the number of strings saved.

The Participant client is developed using a Polling Structure. It is a main loop that is constantly polling the public variables – testing the event flag to see if it is non-zero; if it is, then removing a string from the head of the linked list to do further process. In the process, it parses the string on the delimiter [Aho] to get all the field pieces, the event structure name (or widget name), and the event handler name, converts the field pieces to native type values of the event structure, constructs the event structure using these values (according to the event structure name), and finally renders the display by calling the event handler routine with the event structure as parameter (according to the event handler name). As to the interactive input value in an input field such as text field, on the Master side, the user input them physically; on the Participant side, after it gets the value, it sets it in the field programmatically.

From the description above, we can see that the entities of the Master and Participant clients diverge in the shapes of codes, architectures, implementing mechanisms, and purposes. They are in diversity under the goal of collaboration. However, on each event, we have made them have the same input value in the input field, call the same routine of event handler with the same event structure as parameter, and, at the end of the processes of the event, have the same output display; in other words, they converge on the same

state of the DFA on each event, from the start state to the final accepting state, which is a well-defined session.

### 2.8.3 Collaboration on Event and Transition Function

We now describe the collaboration between the entities of the Master and Participant clients using pieces of code from the Collaborative ReviewPlus applications, mainly focusing on event and the transition function. We just give one collaboration step here that illustrates the idea of collaboration in terms of convergence on the same state of the DFA at the end of the process of the event, with the transition function doing the real job of state transition. In *Appendix B, A Description of the Implementation of Collaborative ReviewPlus* [Wang+TR0705], we give more such typical and interesting ones that would sufficiently help to get the idea.

We can see from the one step demonstration that, the Master and Participant collaboration entities are designed for different purposes, in different implementations, and shapes of codes; they are divergent. At the same time, they have the same logic as to the state transitions on the event of the step and get to the same state at the end of the process of the event; they are convergent. Since the output displays of both the Master and Participant clients at the event are the same, we just show a single set of image captures in the demonstration of the step. We begin with the invocation of the collaboration entities, as shown in Figure 2.5. This corresponds to the start state $q_0$ of the DFA.

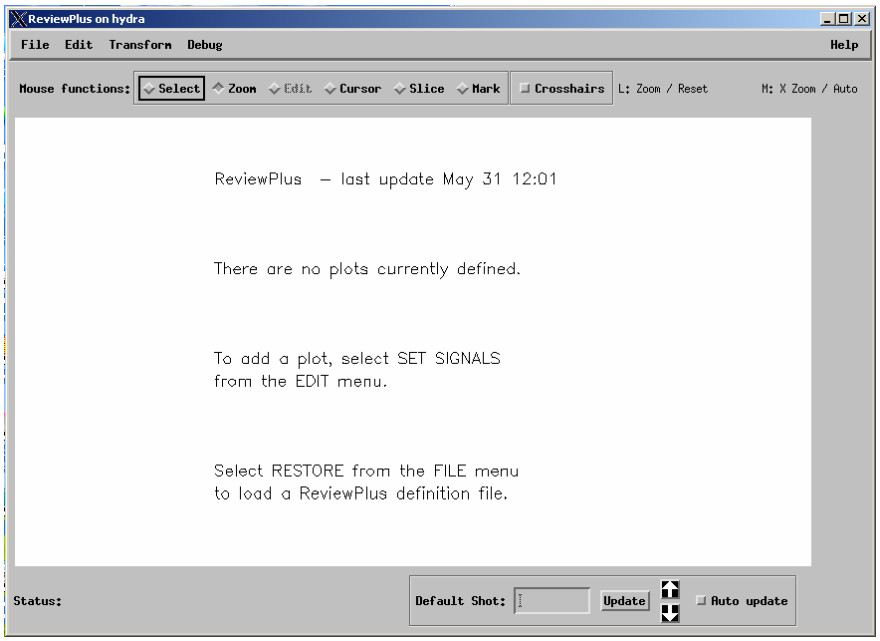



**Figure 2.5 The initial interface and display of ReviewPlus.**

From this interface on the Master client, if we click on the "Edit" item from the main menu, a sub-menu will appear, as shown in Fig. 2.6.

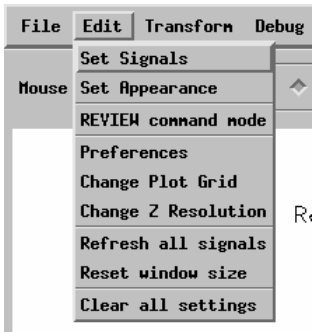



**Figure 2.6 A sub-menu from the main menu of ReviewPlus.**

If we then click on the "Set Signals" item from the sub-menu, an event is fired. This is a button widget, and an event handler routine is defined for the event. We describe the

pieces of code for both the Master and Participant clients in response to this event as follows.

**The Master Client Side**

- Widget creation

```
x = widget_button(mEdit, value='Set Signals', $
                  event_pro='ReviewPlus_SignalDialog_event')
```

From the code above we know that this button widget has `'Set Signals'` as its value shown on the widget's appearance and is associated with an event procedure named `'ReviewPlus_SignalDialog_event'`. When the button is clicked, the procedure is called by the IDL system.

- Definition of event structure for widget

Here is the definition of the event structure for widget button:

{WIDGET_BUTTON, ID:0L, TOP:0L, HANDLER:0L, SELECT:0L}

It has a name WIDGET_BUTTON and 4 fields: ID:0L, TOP:0L, HANDLER:0L, and SELECT:0L, each with a field name, a colon, and a type value. In this case all the values of the fields are of `long` type indicated by the suffix letter L. SELECT: If the button is pressed, the value is 1; if it is released, the value is 0.

- Event handler

```
pro ReviewPlus_signaldialog_event,event
;;;;;;;; collaboration code added ;;;;;;;;
eventMessage = "ReviewPlus_signaldialog_event;"+"WIDGET_BUTTON;"+"ID;"$
  +string(event.ID)+";TOP;"+string(event.TOP)+";HANDLER;"$
  +string(event.HANDLER)+";SELECT;"+string(event.SELECT)
  COMMON BROKER, joChat2
  joChat2 -> writeMessage, eventMessage
;;;;;;;; end of collaboration code ;;;;;;;;
  widget_control,event.top,get_uvalue=info
  info.oReview->SignalDialog
end
```

From the code above we can see that the collaboration code captures the event and gets its field information from `event.ID, event.TOP, event.HANDLER,` etc., converts them into strings, and serializes the strings into a semicolon delimited string, along with the event structure name `"WIDGET_BUTTON"` and the event handler name `"ReviewPlus_signaldialog_event"`. This result string is the event message, which is sent to the NB broker for broadcasting to the Participants.

**The Participant Client Side**

- Parsing of event message

```
          result = STRSPLIT(uval, ';', COUNT=count, /EXTRACT,
/PRESERVE_NULL)
```

```
which_event = result[0]
which_widget = result[1]
```

The next event message string for the Participant client to process is saved in variable `uval`. The IDL system function `STRSPLIT` is called to parse it with `';'` as the delimiter. All the pieces of information around the delimiter are extracted and saved in the array `result` with null string preserved as a piece; the total number of the pieces is saved in variable `count`. The event handler name is in `result[0]` or `which_event`, and the event structure name (or widget name) is in `result[1]` or `which_widget`. The rest of the pieces are all for the fields of the event structure and are saved in the rest elements of the array starting with `result[2]`.

- Conversion to IDL native types

```
FOR i=2, count-1, 2 DO BEGIN
   IF (result[i] EQ 'ID') THEN BEGIN
      id_name = 'ID'
      id_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
      top_name = 'TOP'
      top_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
      handler_name = 'HANDLER'
      handler_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'SELECT') THEN BEGIN
      select_name = 'SELECT'
```

```
            select_value = long(result[i+1])
                                :
        ENDIF
    ENDFOR
```

The code above converts the information (in string) of the fields of the button event structure to its IDL native types; each pair of the strings, i.e., those stored in `result[i]` and `result[i+1]`, decide the field's value and the type of the value, with the former indicating the name and type of the value (due to the unique association of a name with a type, the name alone can also indicate a type, e.g., `ID` is a `long` type) and the latter the value in string. In this case, all the values of the fields are of `long` type; therefore the strings are converted to IDL type `long`.

- Construction of event structure

```
IF (which_widget EQ 'WIDGET_BUTTON') THEN $
   event_structure = {WIDGET_BUTTON,id:id_value,$
   top:top_value,handler:handler_value,select:select_value}$
ELSE IF ...
```

The code above constructs the widget button event structure using the converted native values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`.

- Invocation of the routine of event handler

```
...
ELSE IF (which_event EQ 'ReviewPlus_signaldialog_event') THEN BEGIN
            ReviewPlus_signaldialog_event, event_structure
     ENDIF ELSE IF ...
```

The code above calls the routine of the event handler `ReviewPlus_signaldialog_event` with the constructed event structure `event_structure` as the only parameter.

**Step Summary**

In the process on the event, both the Master and Participant clients call the same routine (the event handler `ReviewPlus_signaldialog_event`, which is a unit of the transition function δ) with the event structure as the only parameter. The event message acts as the messenger, the information source, and the coordinator. With $\delta (q_0, a_0) = q_1$, the Master and Participant clients converge on the same state $q_1$ of the DFA on event message $a_0$ at the end of the process of the event; therefore they have the same output display, as in Figures 2.7 and 2.8, which are two parts of a big interface. Note that, inside an event handler, other routines can be called in any sequence and order, which we don't have to worry about but just think of the whole as the encapsulation and abstraction of the event handler.

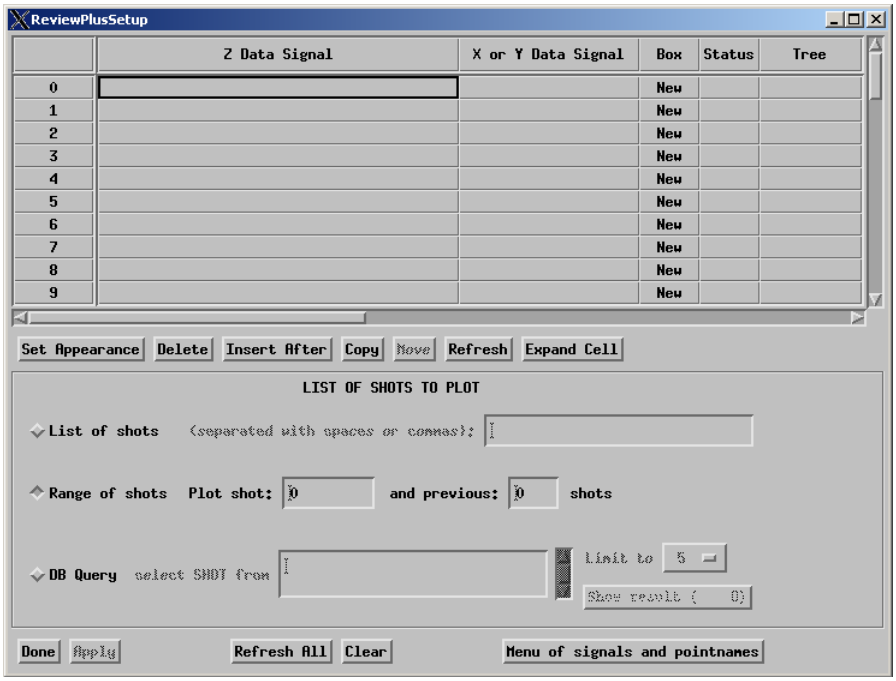



**Figure 2.7 A part of a big interface in ReviewPlus for setting up and managing of signals.**

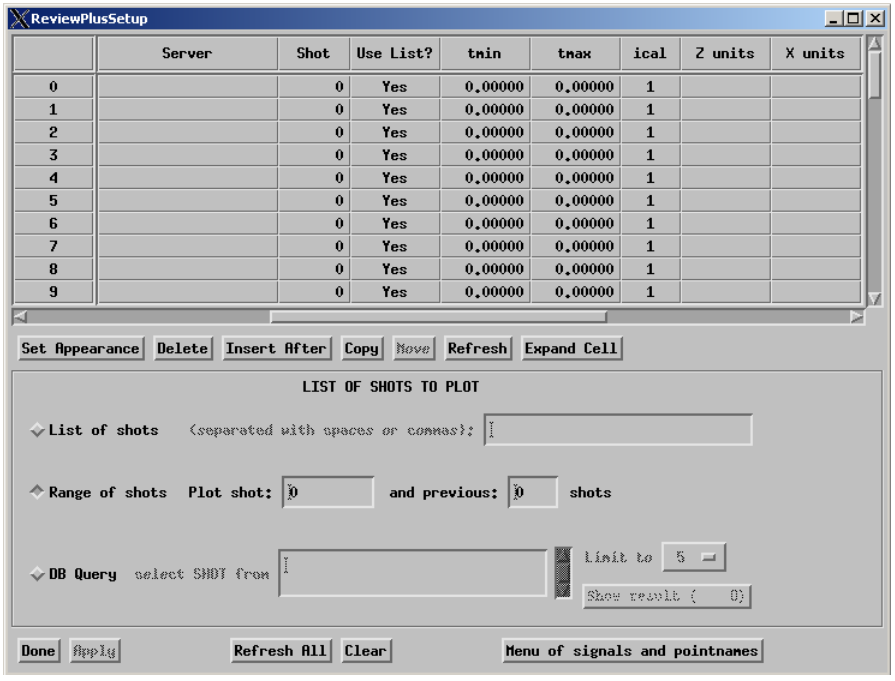


**Figure 2.8 Another part of the big interface in ReviewPlus for setting up and managing of signals.**

# Chapter 3

# Collaborative PowerPoint Applications

In this chapter, we will describe the Collaborative PowerPoint Applications. We will elaborate the components of the applications (the Master client and Participating clients) on their purposes, architectures, and implementing mechanisms. The clients collaborate with each other on events to keep showing the same output displays at each step, using the NaradaBrokering Message Service as the underlying communication system for delivering of event messages.

We further show that, as we have mentioned in chapter 2, the clients actually collaborate to share a common Deterministic Finite Automaton (DFA) in their respective instantiations (i.e., the finite automata in them are the same) and reach a common state of the DFA at any collaboration step. Collaboration between the clients is therefore all about being in a same state of the DFA at each event. The Master and the Participating collaboration clients are designed for different purposes, in different architectures, and implementing mechanisms; they are divergent. At the same time, they have the same

logic as to the state transitions on events and get to the same state of the DFA at the end of the process of each event; they are convergent.

## 3.1 The Big Picture

We have developed the collaborative PowerPoint applications that consist of a Master client type and a Participating client type. The Master client is in control in a collaboration session and broadcasts its event messages to all Participating clients. On the host computers of both the Master client and the Participating clients, the Microsoft PowerPoint application is installed, and the PowerPoint presentation files for the collaboration session are deployed before the session starts. These presentation files are deployed in consistent directories on the computers. The files' consistent deployment makes it possible to collaborate between the clients by communicating text event messages. The Master client captures events such as "file opened," "slide changed," and "window selection changed" during the session, translates them into event messages, and sends the messages to the Participating clients. The Participating clients then generate the same displays according to the directions of the received messages. This way, they work synchronously in Collaboration. By using text event messages we have improved the speed and efficiency of the applications' performance, because it lowers the Internet traffic greatly as compared to Shared Display, which is based on transferring image data like bitmap. We use NaradaBrokering messaging service [Pallickara+Dissertation, Fox+Pallickara+PDPTA'02, Fox+Pallickara+IC'02] as the message environment to communicate event messages during the session.

We have also developed and deployed Instant-Messaging Web Services to further improve the message communications and modularity. The messages are processed to become Metadata by being attached with eXtensible Markup Language (XML) tags. We make the collaborative PowerPoint applications use the Instant Message as a Web Service. We save the Metadata of the PowerPoint presentations and make it a Web Service, so that the presentations can be rendered asynchronously as well as synchronously. Conference sessions may be registered with a session server using XML General Session Protocol (XGSP), which is an XML-based protocol to describe registration, session parameters, session membership, negotiation, etc.; the session server defines session information for both general and the Audio/Video subsystems [Fox+IC'02, Bulut+CIIT]. Therefore, sessions can be set up with the session server so that the collaborative PowerPoint applications can be invoked and connected with the NaradaBrokering Message Service at the scheduled times for collaboration. Along with some Audio services, the collaborative PowerPoint applications can be used in situations such as Distance Education [Fox+Education, Fox+Distance], On-demand Education, and Web Conferencing [WebEx].

## 3.2 The Master Client

The Master client is the one that captures the event and sends the messages to the Participating clients during the presentations of a session. It uses Automation, Connection Point object, and Event sinks technologies [Eddon, Microsoft+KB] in doing this. Automation is a technology that enables the otherwise end-user applications to expose

their functionality through interfaces, and the other applications can reuse the posted functions in their programs by using the methods in the wrapper classes. The Master client controls the functionality of the PowerPoint application server through automation; the Master client calls the functions of the wrapper classes of the server as if the functionality were its own. Under the hood, the wrapper class functions were mapped to the actual functions of the application server through Dispatch map or Dispatch identifiers (DISPIDs). When the function returns from the server, it maps the returned value to its caller in the client's code.

Microsoft has designed the Connectable Object technology that enable client and server object to communicate with each other in both directions. During the collaboration, when something interesting happened in the server object, it informs the client immediately. That is what we call an event. The Connection Point objects are managed by the Connectable Object. This is where the outgoing interfaces are defined, but their implementations are in the client event sinks. Each Connection Point is associated with only one outgoing interface. This is where the events occur and is therefore called the source interface for the client sink interface. The sink interface is where the event handlers are implemented; the events are handled and processed in the event handlers. This is illustrated in Figure 3.1.

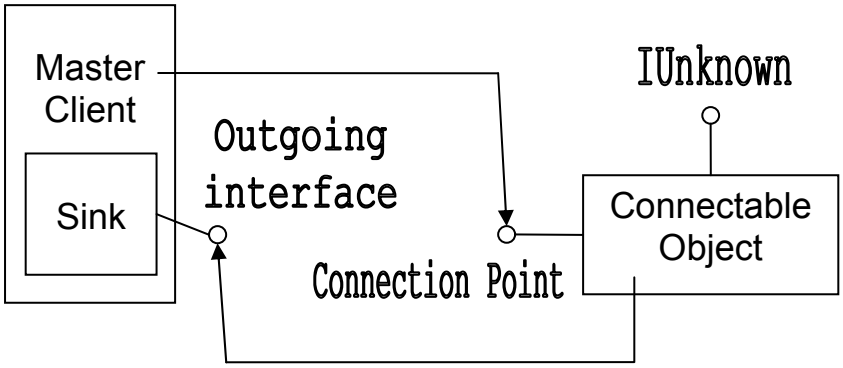



**Figure 3.1 The Connectable object calls the outgoing interface implemented in the sink; the Master client handles events fired from the connectable object through the sink.**

During the presentation, the events are sent from the Connection Point object to the sink object, where the events are identified and processed and the event messages are sent out to the NaradaBrokering message broker for distributing to the Participating clients. This way, the PowerPoint events are captured and dispatched.

## 3.3 The Participating Clients

When the NaradaBrokering messaging broker receives event messages from the Master client, it notifies the Participating clients and broadcasts the messages to them. Each Participating client has its own copy of Microsoft PowerPoint application and the presentation files about the topic to which they have subscribed; the presentation files have been deployed to the same directories as the Master client. Each Participating client processes the event messages independently.

When the Participating client receives the message, it parses it and gets the different parts of information such as event type and its properties. It then dispatches the event type to the appropriate event handler to process. The event type is the key to call different processing functions. The associated properties are used in the functions to generate the correct presentation results. The Participating client uses automation technology [Eddon, Microsoft+KB] in the process. It calls the functions in a PowerPoint wrapper class under the instructions of the event message. The functions are actually mapped to the functions in the Microsoft PowerPoint application. PowerPoint functions get called, do the tasks such as navigating through presentations and slides, and return the result values eventually to the caller functions in the wrapper class. This is illustrated in Figure 3.2. Thus, the Participating clients are generating the same output displays as the Master client independently and simultaneously.

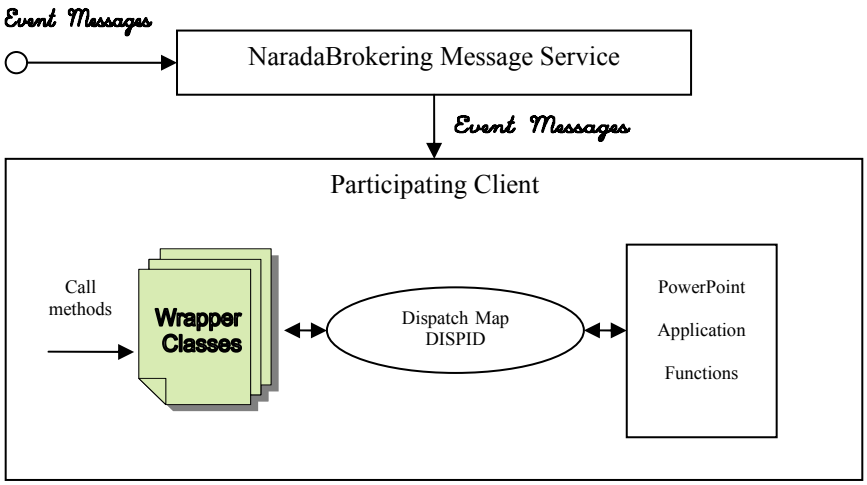



**Figure 3.2 The event messages invoke methods of the wrapper class; the methods then are mapped to functions of PowerPoint application through Dispatch Map/DISPID; the functions are executed and result/status codes are returned.**

## 3.4 The Event Models

We abstract the event models of the collaborative PowerPoint applications to be of three levels of events: physical event, semantic event, and rendering event, from low to high, in that order. The physical event is the event when a cursor is on a specific area of the screen, a mouse clicking, or a keyboard stroking. When the Master client is on a presentation session, the lecturer might use all combinations of the physical events to control the process. The PowerPoint application converts these physical events to meaningful instructions to the applications, such as "change slides," and "change windows." These meaningful instructions are the semantic events.

In our programs we make use of the Dispatch event interfaces of the PowerPoint application, connectable object, connection point technology, and event sink to catch these semantic events. They are listed in Table 3.1. For some reasons, in Microsoft PowerPoint, one can only get the hexadecimal codes of these events instead of meaningful string name descriptions as in the other applications of the Microsoft office suites. With codes like this, one can not know their meanings and can not figure out which is which. We have done logical analysis according to the input/output of the presentation processes and mapped each of the code to its corresponding meaningful string name in the event interface (EApplication) of the PowerPoint application. We call this process a translation.

**Table 3.1 The events that are posted in EApplication interface of PowerPoint and that can be captured and processed**

| |
|---|
| WindowSelectionChange |
| WindowBeforeRightClick |
| WindowBeforeDoubleClick |
| PresentationClose |
| PresentationSave |
| PresentationOpen |
| NewPresentation |
| PresentationNewSlide |
| WindowActivate |
| WindowDeactivate |
| SlideShowBegin |
| SlideShowNextBuild |
| SlideShowNextSlide |
| SlideShowEnd |
| PresentationPrint |
| SlideSelectionChanged |
| ColorSchemeChanged |
| PresentationBeforeSave |
| SlideShowNextClick |

After getting them, the Master client sends the semantic events through NaradaBrokering message service to the Participating clients. The Participating clients then call the functions of the PowerPoint through automation according to each command of the semantic event and render the process of the presentation. We call this kind of event rendering event.

We've realized some multimedia effect in our project, making sounds, animations, and transitions collaborative, that is, they can be played synchronously between the Master client and Participating clients. This effect makes learning and education more enjoyable and more impressive; it vivifies the lectures and classrooms and brings more lights to the soul. This is done by triggering them when a specific slide is navigated to, or when a specific animation item is invoked, but, it is very limited. For example, if a sound

file, an animation file, or a movie file is embedded in a slide, and the Master client invokes the file by moving the mouse over the item or clicking on it and wants to inform the Participating clients to generate the same effect, there is no way to do that, because there is no such event exposed for us to capture and use. The event types that are potentially useful but absent are listed in Table 3.2. We could have developed much more interesting and enlivening stuffs in our applications had the interfaces of those event types been posted over there.

**Table 3.2 The types of events that are absent in PowerPoint but potentially useful in further application developing**

| | |
|---|---|
| Physical events | MouseOver, MouseClicked, MouseDoubleClicked, KeyDown, KeyUp, KeyStroke, etc. |
| Events about sounds | SoundClipPlayed, SoundFilePlayed, etc. |
| Events about animations and transitions | AnimationClipPlayed, MoviesPlayed, AnimationFilePlayed, etc. |

## 3.5 NaradaBrokering Message Service

We integrate the NaradaBrokering Message Service with the collaborative PowerPoint applications to transmit event messages between clients. The NaradaBrokering is a system that supports messaging in Peer-to-Peer Grid [Fox+CTS, Fox+P2PGrid, Wang+ITCC'04] environment; it is a generalized publish-subscribe mechanism; it handles dynamic protocol choice, tunneling through firewalls; it supports TCP, UDP, multicast, SSL, and RTP; it can run in client-server mode as Java Message Service (JMS) [Fox+Pallickara+IC'02, JMS, Monson-Haefel] or in distributed Peer-to-

Peer mode like JXTA [Fox+JGI, JXTA]; it can be used in real-time synchronous collaboration.

The NaradaBrokering system was written in Java language, and our collaborative PowerPoint applications have been developed in C++. In order to communicate information between the two environments, we use Java Native Interface (JNI) [Liang] as a tool to fulfill this task. The communication is a two-way conduit, both from C++ sending event messages to Java and from Java to C++.

The Master client captures the event in PowerPoint and sends messages to the NaradaBrokering message service system using the functions in the JNI interface. In doing so, it creates and embeds a Java Virtual Machine inside the C++ environment, maps data types between them, and calls the JNI functions through the virtual machine.

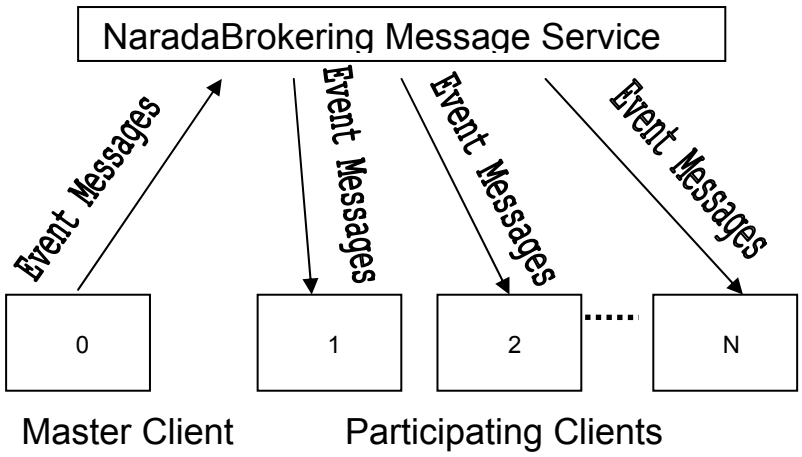



**Figure 3.3 The collaboration structure for the Master client and the Participating clients with the NaradaBrokering Message Service as the underlying communication system.**

As soon as the NaradaBrokering system receives a message, it broadcasts the message to every Participating client, using the notification mechanism as shown in Figure 3.3. Here, the transformation of the message is from Java to C++ environment. The notifying method, i.e., onMessage(), is overridden to include native function calls to C++, so that the message type commands the appropriate C++ functions in the Participating client application to perform the rendering process of the presentation. The functionality of the Participating client is divided into C++ methods, and contained in a dynamic link library component (e.g. collabPPT.dll), which is loaded in the Java environment so that the Java native functions can make use of it. The JNI interface plays an important role in this communication direction. Thus, the Master client and the Participating clients of the collaborative PowerPoint applications communicate and cooperate with each other through the NaradaBrokering Message Service system.

## 3.6 Collaboration on Deterministic Finite Automaton

As we have noticed so far, the Master and Participating collaboration clients are designed for different purposes, in different architectures and implementing mechanisms; they are divergent. At the same time, they collaborate on messages to present the same output displays at each event; in other words, they have the same logic as to the state transitions on events and get to the same state of the Deterministic Finite Automaton (DFA) at the end of the process of each event; they are convergent. They actually collaborate to share a common DFA in their respective instantiations (the finite automata in them are the same) and reach a common state of the DFA at any collaboration step.

Collaboration between the clients is therefore all about being in a same state of the DFA at each event.

As we have shown in the example of chapter 2, the Master and Participating collaboration clients are at the start state $q_0$ of some DFA when instantiated. At this stage, the automata in their respective instantiations are not completely determined with respect to some presentation files. The user on the Master client opens a presentation file by physically controlling the interface of the PowerPoint application using mouse clicks, and keystrokes (e.g., File > Open > File name …). The Master client responds to these physical events. As a result, the event "PresentationOpen" is fired in the Connection Point object of the PowerPoint server object "EApplicaton", and the Connection Point object informs the client sink object immediately with the hexadecimal identification of the event. The sink object is where the event handlers are implemented. It builds up the corresponding event message containing information about the function to call (which performs presentation file opening) and the property (the path and name of the presentation file). The event message is a delimited text string (e.g., "PresentationOpen;C:/file1.ppt") and the intermediate representation of the event, and it is sent out to the NaradaBrokering message broker, where the messages are distributed to the Participating clients.

Each Participating client parses the delimited text string after receiving it, converts the property data to its system's native representation, and, based on the information, arranges the function to call (e.g. Open(LPCTSTR(C:/file1.ppt), …)). This way, it automates to open the same presentation file as the Master client programmatically.

At this stage, the automata in the instantiations are determined with respect to the opened presentation file, as to the total states and the relationships between the states. We can call this event the ***major event*** because it defines the states of the automata. From this state (state $q_1$ in the example), we can navigate through the slides of the presentation file on events such as "Goto;*CertainSlide#*," "Previous," and "Next." We call these events the ***minor events*** because they just change the states of the defined automata. The collaboration on minor events is similar to the major event we have just described, so we omit the description.

## 3.7 Instant-Messaging as a Web Service

The event messages can be marked up using XML tags, so that an XML document can be generated corresponding to the Document Object Model (DOM) format [Deitel+XML, DOM]. This DOM-based XML message can then be used as the unit of message communications between the clients of the collaborative PowerPoint applications; it is transferred through the Internet using Simple Object Access Protocol (SOAP); it is the basis of Instant Message communication. The XML message includes session information such as session identifier, topic title, source, and destination, as in

<message sessionID = "aSessionNumber" topic = "aTitile" to = "receiver" from = "sender"> an event message </message>

As such, each group of people in a session can send and receive messages correctly in a concurrent sessions support environment such as NaradaBrokering Message Service. We have developed and deployed Instant-Messaging Web Services [OASIS] for the communication in this project. The main services include functions that markup event messages as DOM-based XML document and functions that get the event messages out of the XML document. The information of the web services such as its Universal Resource Identifier (URI) endpoint and its exposed methods are described in the Web Service Description Language (WSDL) file, and web services are deployed using this file. The users can find the web services using Universal Discovery, Deployment, and Integration (UDDI). The users then bind the web services to their applications and use web services via the internet [Deitel+WS].

In the collaborative PowerPoint applications, the Master client send its event messages to the NaradaBrokering message service, which has discovered and bound to the Instant-messaging web services beforehand. The NaradaBrokering message service makes use of the exposed methods of the web services to convert the received plain messages to DOM XML document; then it transfers and distributes the document via the internet to the Participating clients for dealing and rendering. The Participating clients leverage the functionality of the Instant-messaging web services through the NaradaBrokering message service to get the plain event messages out of the XML document and operate on the instructions of the messages syntactically, rendering the exact process which the Master client is going through. This is shown in Figure 3.4.

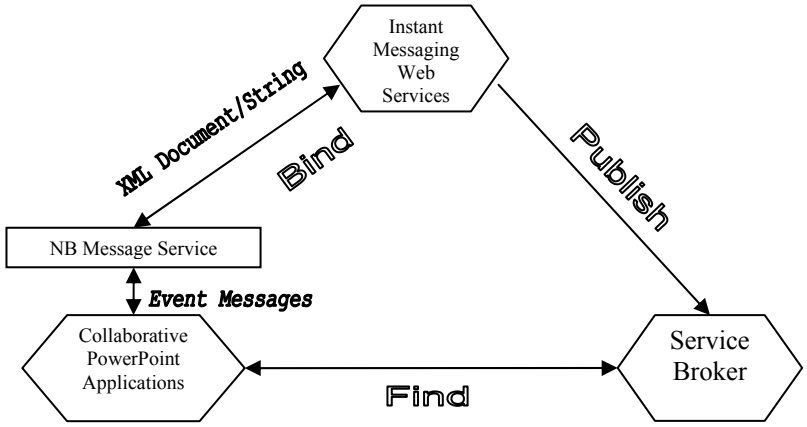



**Figure 3.4 Instant-messaging web services with the collaborative PowerPoint applications.**

## 3.8 Metadata and On-demand Education

The Metadata generated by the Instant Message Web Service can be saved as files or in a database and can be made as a Web Service. The Metadata Web Service can be registered with a session server and can be reflected in session server's information page. One of such session servers is the one used in our lab. The session server uses an XML General Session Protocol (XGSP), which is an XML-based protocol to describe registration, session parameters, session membership, negotiation, etc. It defines session information for both general and the Audio/Video subsystems. Sessions can be set up with the session server, and the session server manages the sessions intelligently with the Metadata.

Metadata plays an important role in today's computerized world full of data scattered anywhere and everywhere. It makes intelligence possible in this highly connected networks and internet space. More likely, this world has enough data almost about anything, but most likely, people do not have enough useful information when they need it. How to get the useful information out of the flat data is an interesting and meaningful task. Metadata will give a big help in doing this.Metadata is a gateway to information [Tannenbaum]. Along with other methodologies, we can use Metadata to generate information on demand.

In on-line education, e-learning, and virtual classroom, to get the intended information on demand is attractive. Apart from regular lecture sessions, people can go through the lectured slides of a presentation at their own pace, or skip to specific slides within or even between lecture presentations. They can benefit from this and achieve the best education effects. We have developed a special application in our collaborative PowerPoint applications that can do things like that. It inherits functions from both the Master client (such as message broadcasting) and the Participating client (such as generating displays) applications. It binds to and makes use of the Metadata Web Services. It works like this. From a session server, the user selects a Metadata Web Service (that corresponds to a lecture stored) and binds the application to the Web Service. Next, the user clicks buttons on the application such as "Next slide," and "Go to slide # x" to control the navigation. Under this control, the application contacts with the Metadata Web Service, gets messages out of the Metadata, and renders the displays according to the instructions of the messages, just as the Participating clients do.

Combined with the Participating client application, this on-demand application can be used in forming dynamic virtual study groups (after class). The on-demand application sends the message out to its group members for rendering. Each member in the group can have both of the applications, and each one can have a chance to control the process. This way, they can discuss the contents on some slides of some presentations in some lectures.

## 3.9 The Extended Transition Function and Access to Metadata

The Metadata Web Service is the one that stores and manages the metadata of the event messages in collaboration. We can think of the metadata as a concatenated string $\omega = a_0a_1 \ldots a_n$, with each $a_i$ $(0 \leq i \leq n)$ being an XML tagged event message.

The special application (designed for asynchronous access) binds and makes use of the web service. The user controls the widgets such as "Next slide," "Go to slide # x" on the interface of the application to navigate to certain slide. If the user wants to do some sequential access, he can click the button "Next slide" continuously from the beginning. The Extended Transition Function $\Delta$ of the DFA takes the symbols in string $\omega = a_0a_1 \ldots a_n$ sequentially, one by one from $a_0$ going forward to $a_n$. It takes advantage of the current state p in memory, gets the next symbol $a_j$ $(0 \leq j \leq n)$ in the remaining string $a_ja_{j+1} \ldots a_n$ of $\omega$, and goes to the next state r on the transition of the Transition Function $\delta$, as in $\delta (p, a_j) = r$.

If the user wants to do some random access, he can control the input and the button "Go to slide # x" to get to a specific slide. The Extended Transition Function $\Delta$ takes as

parameters the start state $q_0$ and the prefix (which corresponds to the desired slide) of the string $\omega$ and transits to the corresponding desired state p, as in $\Delta (q_0, x) = p$, where $x = a_0a_1 \ldots a_i$. The Extended Transition Function $\Delta$ basically begins with the start state $q_0$, goes through all the transitions in response to each $a_j$ $(0 \leq j \leq i)$ in x, and finally gets to the state p on input symbol $a_i$.

# Chapter 4

# Collaborative OpenOffice Applications

In this chapter, we will describe our effort in developing of the Collaborative OpenOffice Applications. Impress is a presentation application in OpenOffice/Star Office; it has similar functionality as Microsoft PowerPoint. Making Impress collaborative across computers is useful in distance education, e-Learning, and online conference. We have developed the collaborative Impress applications which make use of the functions of the Impress. We will elaborate the components of the applications – the Master client type and the Participating client type – on their purposes, architectures, and implementing mechanisms. The clients collaborate with each other on events to keep showing the same output displays at each step, using the NaradaBrokering Message Service as the underlying communication system for delivering of event messages.

We will further show that, as we have mentioned in chapter 2, the clients actually collaborate to share a common Deterministic Finite Automaton (DFA) in their respective instantiations (i.e., the finite automata in them are the same) and reach a common state of

the DFA at any collaboration step. Collaboration between the clients is therefore all about being in a same state of the DFA at each event. The Master and the Participating collaboration clients are designed for different purposes, in different architectures and implementing mechanisms; they are divergent. At the same time, they have the same logic as to the state transitions on events and get to the same state of the DFA at the end of the process of each event; they are convergent.

## 4.1 The Big Picture

OpenOffice [OpenOffice] is an open source office suite; it has similar functionality as Microsoft Office suite. Because it is free downloadable, and every one has an opportunity to develop and contribute to the resource using a broad range of programming languages and protocols, it has the features such as availability, usability, extensibility, popularity, and versatility. It is people's common property and wisdom. The Impress is a presentation application in the OpenOffice/Star Office; it has similar functionality as Microsoft PowerPoint.

Developing the Impress to be collaborative across computers is useful in e-Learning, web conference, and distance education. We have developed collaborative Impress applications, which make use of the functions of the OpenOffice/Star Office. The Master client and the Participating client collaborate with each other to share the same presentation slide displays. We have used a common message broker, the NaradaBrokering Message Service [Fox+Pallickara+PDPTA'02, Pallickara+Dissertation],

as the underlying messaging environment [Fox+JGI, JMS] to communicate messages between the clients in a session.

We have realized the shared event model in collaboration [WebEx, Placeware]. We use event messages in the controlling of a presentation process. Compared to shared display, the short text messages save great bandwidth over the Internet. This realization is fast and efficient. The shared event model plays an important role in collaboration. This realization of the collaborative Impress applications is a complementary effort to the OpenOffice/Star Office.

## 4.2 Shared Event Model

A commonly used model in collaboration is Shared Display. In this model, some amount of the screen image in a format (e.g. bitmap) is sent over the networks between the collaborating computers each time when the image of the screen is changed, either partially or totally. The *Remote Desktop Connection* of Microsoft Windows XP and the *Virtual Network Computing (VNC)* [VNC] are using this model. It is appropriate in situations where the screen output is random, such as online meeting, discussion, sharing data (text, graph, image), or impromptu presentations using some software. The disadvantages of Shared Display include the following.

- It consumes big bandwidth of the networks because of the image transferring. Thus it is relatively slow and the latency is big.

- The feeling of using it is not smooth. The waiting time depends on the amount of the screen image that needs to update. The worst case is when the image changes abruptly, say, the whole screen.

Let us examine a case of using presentation files in collaboration, such as Microsoft PowerPoint (.ppt) or OpenOffice/Star Office Impress (.sxi) files. Each slide in a presentation file is different, and the contents of a whole screen need to update. In this case, the disadvantages of the Shared Display model are the worst. To solve this problem, a competent collaboration model is meant to take place and play an important role. We believe a "lightweight" model, the *Shared Event Model*, will work efficiently and gracefully. The idea is to catch events and generate event messages in the driving side (let's call it the *Master Client*) of the collaboration during a presentation session, send the event messages through the common message brokers to the accepting side (*Participating Clients* or *Participants*), and render the slide displays over there. The event messages are short text strings, as "Presentation Open," "Slide Change," etc.

The presentation files are deployed or downloaded to the same directories on the host computers of both the Master client and the Participating clients before a session begins. This way, the collaborative applications can locate, open, and navigate through them correctly. The shared event model overcomes the disadvantages of the Shared Display model; therefore the shared event model has the following advantages.

- It is fast and efficient, because the small text string messages greatly reduce the network traffic, and it makes full use of the computing powers of both sides.
- It gives consistent and smooth feelings of presentations. The size of the messages are small and approximately equal, so the time for transferring them

will remain in a relatively constant range, just as mentioned in the paper "The Rule of the Millisecond" [Fox+CISE+March2004].

At the same time, because this model makes use of modern common message brokers, it shares the advantages of the brokers such as tolerance and quality of service. It also contributes to Peer-to-Peer Grid computing [Fox+CTS, Berman+Fox+Hey, JXTA]. Collaboration such as web conference, distance education, or e-learning is a trend in today's information revolution. The usage of presentation files accounts for a considerable proportion in all the visual aids. So, the shared event model will play an important role and show its power.

## 4.3 Collaboration Structure

In the collaborative Impress applications, one type is the Master client, and the other is the Participating client, or Participant. Both of the client types cooperate with a common message broker, the NaradaBrokering Message Service, as the underlying communication environment to communicate messages between the clients in a session. The master client lectures and broadcasts its event messages to all participating clients. The applications make use of the functions of the OpenOffice/Star Office. The Master client and the Participant clients collaborate between them and share the same slide displays.

On the host computers of both the Master client and the Participating clients, the OpenOffice/Star Office suite (or just the Impress application) is installed and the presentation files (in the scheduled lecture) are deployed or downloaded in consistent

directories before the session begins. This makes it possible for the clients to collaborate by communicating text event messages. The master client captures events such as "file opened," and "slide changed" during the session, translates them into text messages, and sends the messages to the Participating clients through the NaradaBrokering Message Service. The Participating clients then generate the show of the lecture according to the directions of the received messages. This way, the Master client and the Participating clients work synchronously in collaboration. This is illustrated in Figure 4.1.

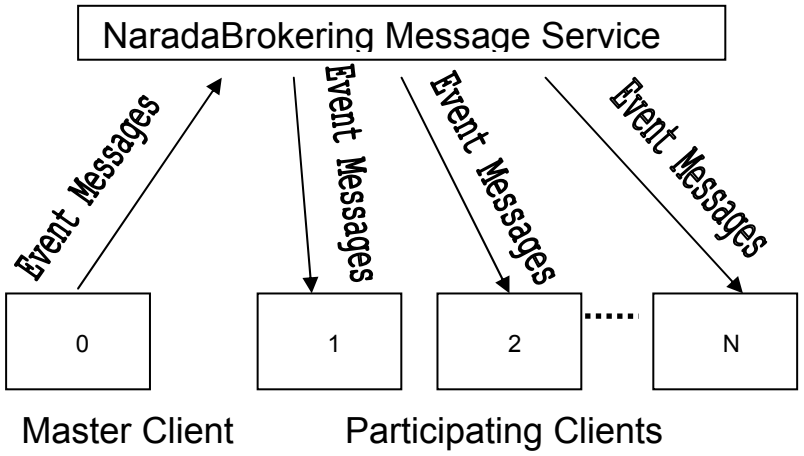



**Figure 4.1 The collaboration between the Master client and the Participating clients via the NaradaBrokering Message Service.**

## 4.4 New Concept for Collaboration

Some features in the OpenOffice/Star Office are elegant for universal programming and the Web. These features include Universal Network Object (UNO) technology, diverse programming environment, fine-grained Application Programming Interfaces

(API), and Frame-Controller-Model Paradigm. They form a new concept for modern programming and global Collaboration [Fox+WS].

### 4.4.1 Universal Network Object

Universal Network Object (UNO) is a component technology that is designed for universal programming and application. Components in the UNO can interact with each other across programming languages, component technologies, computer platforms, and networks. The UNO works with programming languages such as C++, Java, Java Script, Visual Basic, VBScript, and Delphi. The UNO is the base component technology for the OpenOffice/Star Office; it can also cooperate with other component technologies such as Java Beans and Microsoft COM/DCOM [Eddon]. The UNO is available on UNIX, Linux, and Windows platforms; thus it has the features of availability and popularity. The UNO makes it possible that components are able to collaborate through networks. In a word, just as its name implies, the UNO enables objects to function well across networks and makes universal programming and application of objects a reality.

Through the UNO technology, application programs connect to local or remote instances of the OpenOffice/Star Office from C++, Java, or COM/DCOM. The programs then access the functionality of the instances using their APIs, control and automate the process, either sequentially or interactively. The purpose of the UNO is to treat applications/components as reusable objects, which are accessible universally through the underlying infrastructure networks, as long as those objects are cooperative by providing programming interfaces, type libraries, etc. The APIs of the OpenOffice provide comprehensive specification of its programming features. In our collaborative Impress

applications, both the Master client and the Participating clients connect to an instance of the Impress application via the UNO and access the functions in the application's APIs.

### 4.4.2 Diverse Programming Environment

The OpenOffice offers Diverse Programming Environment. It enables people to develop codes in languages such as C++, Java, Java Script, Visual Basic, VBScript, and Delphi, and on platforms such as UNIX, Linux and Windows. It has the features of diversity, versatility, and popularity. Therefore, people can program in their most familiar languages and on their most convenient platforms. This is a factor for productivity and quality of software.

People can add new functions to the OpenOffice. For example, people can program new file filters, database drivers, linguistic extensions, or even complete groupware applications. People can even integrate the OpenOffice environment with the Java Integrated Development Environment (IDE) through the UNO components and work with Office documents in Java Frames.

We can connect to a local or remote instance of the OpenOffice from C++, Java, or COM/DCOM. The extended Java API of the OpenOffice is neat, efficient, and secure. For instance, it has similar methods as COM/DCOM for connecting objects, as follows.

```
queryInterface(),
addRef(),
deleteRef(),
…
```

People who are familiar with COM/DCOM technology will feel comfortable in developing and using the OpenOffice objects. At the same time they can take all the advantages of the Java language, such as its features for the Web and Internet.

### 4.4.3 Fine-grained API

The OpenOffice defines a comprehensive specification describing its programmable features. It is called the Application Programming Interfaces (API). These Interfaces are fine-grained: each method (function) is for a sole and clear purpose; relative methods are grouped in a class; relative classes form a package that is called a module; relative modules form a parent module, and parent modules can have their own parent along the tree structure, until a root module is reached, as in

```
com.sun.star.frame.XDesktop
```

From the software engineering point of view, this design has the following strengths.

- Because of its fine-grained API, it is of high level of reuse; therefore it will survive through time.
- Program can just integrate with small and necessary parts of the interfaces to fulfill its functionality, instead of having to include conglomerate blocks which contain lots of unnecessary functions. Thus the fine-grained API makes program more effective and efficient.
- It makes the software highly extensible. Extensibility is of vital importance in modern software industry.

- It makes the software very easy to manage and maintain. The cost of software management and maintenance is always a big part in the life cycle of software engineering.

In both the Master client and the Participating clients of the collaborative Impress applications, the programs make use of the functions in the API. The clients collaborate with each other to share the same screen displays simultaneously. As an example, we list the event listener interfaces and their corresponding event types, which we have tried in our programs, in table 4.1.

**Table 4.1 Event listener interfaces and their corresponding event types**

| | |
|---|---|
| XPropertyChangeListener | PropertyChangeEvent |
| XSelectionChangeListener | EventObject |
| XFrameActionListener | FrameActionEvent |
| XKeyListener | KeyEvent |
| XMouseListener | MouseEvent |
| XMenuListener | MenuEvent |
| XWindowListener | WindowEvent |
| XContentEventListener | ContentEvent |
| XFocusListener | FocusEvent |
| XFormControllerListener | EventObject |
| XModeChangeListener | ModeChangeEvent |
| XChangeListener | EventObject |
| XContainerListener | ContainerEvent |
| XEventListener | EventObject |
| XTerminateListener | EventObject |

## 4.4.4 Frame-Controller-Model Paradigm

The Model-View-Controller (MVC) design pattern [Gamma] is popular and widely applied to interactive software development. Based on this pattern, the OpenOffice adopts a new paradigm in its developing. It is called the Frame-Controller-Model (FCM)

Paradigm. We discuss them and point out the advantages of FCM paradigm in the OpenOffice programming next.

In MVC, the Model is the application object, the View is its screen presentation, and the Controller is the encapsulation of a response strategy that defines the way a user interface responds to user input. MVC changes the previous monolithic programming style in which the model, view, and controller are undistinguishable and mingled together to be one unit or object; MVC decouples the model, view, and controller from the megalithic lump to make software more flexible and reusable.

A model can have multiple views, and new views can be added. The appearances of the views reflect the state of the model. Between the view and model, there is a Subscribe/Notify mechanism. Each view subscribes and listens to the model. Whenever the values of the model have changed, it notifies the views about this. The views then access the model and update their screen appearances.

The controller is a response mechanism; it defines the way the user interface (the view) responds to user input. The controller object encapsulates the response strategy which represents an algorithm. The view associates with a controller instance at a time. The view can have multiple controllers in store and can add new controllers. A view can change a controller instance at run time; thus it can change the way it responds to user input dynamically. The controllers associated with the view may have different strategies or variant algorithms about the responses of user interfaces to user inputs. The view switches to a different controller object either statically or dynamically, without changing its screen appearance.

The Frame-Controller-Model (FCM) paradigm of the OpenOffice has some common properties and works similarly as MVC, but it has its own specialties and is more suitable for the Universal Network Object (UNO) programming. In the FCM, the Model is the document object; it has document data and also methods that access the data. The methods can change the data directly without having to use a controller object. The controller is the screen interaction with the model; it observes the changes made to the model, and manages the presentation of the document. The frame is the controller-window linkage; it contains the controller, and has knowledge about the window, but not the functionality of the window. That functionality is encapsulated in the underlying windows system – whatever platform it is. This decouples specific windows implementation from the frame, thus makes it possible to use a single frame implementation for different windows in Open Office. The specific windows work with the frame to make the screen presentation.

People can develop new models for new document types in the OpenOffice without having to worry about the frame and the management of the underlying windows system. Each model can have multiple controllers associated with it. The controller depends on the model and controls the manner of the model's presentation. A controller can be replaced by another one without changing the model or frame. New controllers can be created for the model.

In programming, from a model object, we can use the method `getCurrentController()` of the API to get the controller object associated with this model; from this controller, we can use the method `getModel()` to get the model object. Likewise, we can use the method `getFrame()` to get the frame object from the

controller object; we can use the method `getController()` to get the controller object from the frame object. From the frame object, we can even get the Container Window of the frame and Component Window of the Component using the methods `getContainerWindow()` and `getComponentWindow()`, respectively. Through those window objects, we can do jobs related to the management and the control of the window system. This is convenient and powerful. The FCM paradigm is just the right thing for the Universal Network Object (UNO) programming.

## 4.5 The Client/Server Communication Bridge

In a session, the Master client connects to the OpenOffice/Star Office that serves as a server, listens to the events fired there, and sends the event messages to the message broker for broadcasting to the Participating clients for generating the screen displays as those of the Master client. So, the Master and Participants are working synchronously in the session. The client (either the Master or the Participant) communicates information with the office server through TCP/IP socket. The office server listens to clients' TCP/IP connections using a connection URL as the parameter, which includes the hostname/IP address, the port number, and the protocol, as in Figure 4.2.

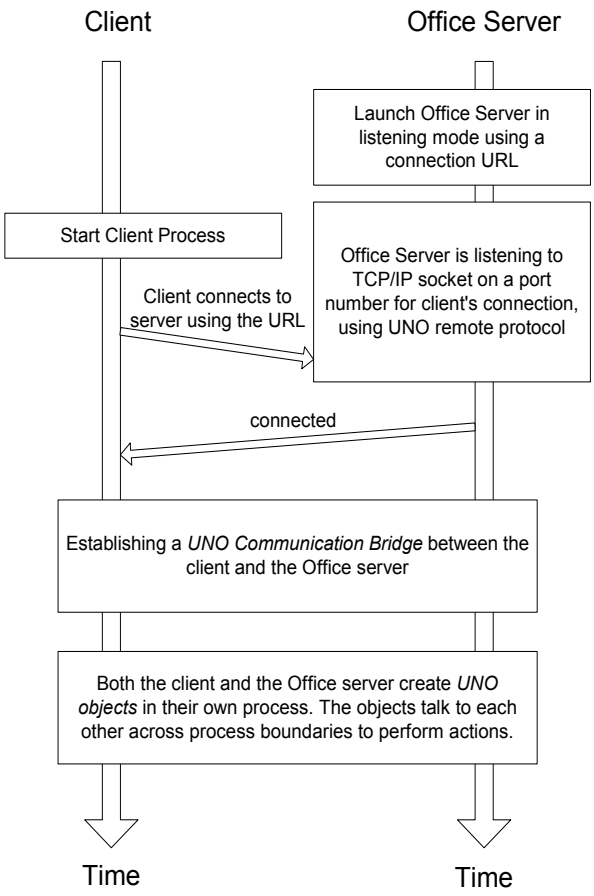



**Figure 4.2 The process of the launch, connection, and interaction between the Client and the Office Server.**

We launch the office server in listening mode by issuing the command line:

```
soffice -accept=socket, host=localhost, port=8100; urp;
StarOffice.ServiceManager
```

Here, the office server is running on the local host; it is listening to the socket on port number 8100 for connection; it is using UNO remote protocol for communication. We

make the client and server running on the same host for convenience, though they can run on different hosts in UNO programming environment.

As in other object oriented languages, objects are used in UNO programming to perform specific tasks. They are referred to as services in the UNO context. A service manager is a factory of services, which creates services and other data used by the services. A component context consists of the service manager. Both the client and the office server have their own component context and service manager. The client creates services, or UNO objects, through the service manager in the client process, and the server creates UNO objects in the server process. Only the UNO objects created by the service managers can talk to each other across process boundaries.

On the Master client side, the user controls the office files on the Office server, opening, loading, or accessing the data. In our case, the user controls the Impress presentation files on the Impress Office server, and the Master client catches the events fired over there. The Participating client processes the received event messages and calls the functions of the Office Server (under the instructions of the messages) for generating the same displays as the Master client. The automation technology is used in this process. In order to do their jobs and to work with the data located on the Office servers, both the Master client and the Participating clients need to establish a communication bridge with their respective Office Servers and get their servers' service managers. The process of building such a bridge is described in Figure 4.3.

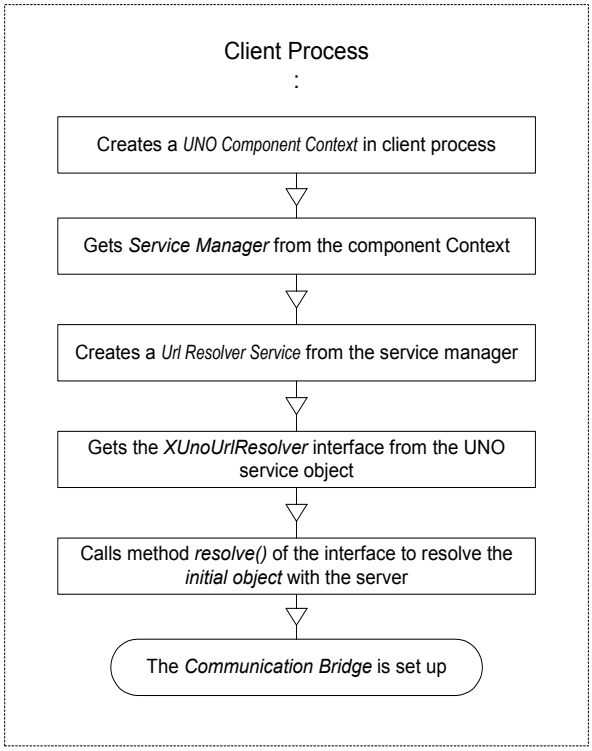



**Figure 4.3 Establishing of a Communication Bridge between the Client and the Office Server.**

We will describe the steps next, using explanations and snippets of codes in our programs. First, the client creates a local UNO component context as follows:

```
   XComponentContext xLocalContext =
com.sun.star.comp.helper.Bootstrap.createInitiComponentCont
ext(null);
```

The client's local component context contains a service manager which creates services necessary to talk to the server's remote component context. We can get this local service manager from the local component context as follows:

```
   XMultiComponentFactory xLocalServiceManager =
xLocalContext.getServiceManager();
```

The local service manager then creates a service called `com.sun.star.bridge.UnoUrlResolver`, which is an object to be used in the connection, as follows:

```
   Object urlResolver  =
xLocalServiceManager.createInstanceWithContext("com.sun.sta
r.bridge.UnoUrlResolver", xLocalContext );
```

From this object, we retrieve the `XUnoUrlResolver` interface, which supplies methods to resolve the initial object with the server, as follows:

```
   XUnoUrlResolver xUnoUrlResolver = (XUnoUrlResolver)
UnoRuntime.queryInterface(
   XUnoUrlResolver.class, urlResolver);
```

The method `resolve()` of the interface is called to resolve the initial object with the server using the same connection URL when the server was launched, as follows:

```
   Object initialObject =
xUnoUrlResolver.resolve( "uno:socket, host=localhost,
port=8100; urp; StarOffice.ServiceManager" );
```

Now we have set up a bridge between the client and the server. The client makes use of this initial object (associated with the server) to access the server's data and to control the server's functions, as if they were its own. The client needs to use the server's default context; so it first obtains the `XPropertySet` interface and then gets the default context as follows:

```
   XPropertySet xPropertySet =
(XPropertySet)UnoRuntime.queryInterface(XPropertSet.class,
initialObject);
   Object context =
xPropertySet.getPropertyValue("DefaultContext");
```

Next, the client gets the server's component context as follows:

```
   XComponentContext xRemoteContext =
(XComponentContext)UnoRuntime.queryInterface(XComponentCont
ext.class, context);
```

Finally, the client gets the server's service manager as follows:

```
   XMultiComponentFactory xRemoteServiceManager =
xRemoteContext.getServiceManager();
```

Now, the client has a reference to the server's service manager. Thereafter, the client can use the reference to get the server's "Desktop" (`com.sun.star.frame.Desktop`) object and its interface, which is used to load and access documents (such as presentation files) and to get the current one.

```
   Object desktop =
xRemoteServiceManager.createInstanceWithContext(
   "com.sun.star.frame.Desktop", xRemoteContext);
   XDesktop xDesktop =
(XDesktop)UnoRuntime.queryInterface(XDesktop.class,
desktop);
```

With the `XDesktop` interface, the client can call the interface's methods (such as `getCurrentFrame()`) to get the server's Frame, Controller, and Model, either directly or indirectly. With the FCM paradigm, as we have discussed previously, the client can take control of the server's process. This is described in Figure 4.4.

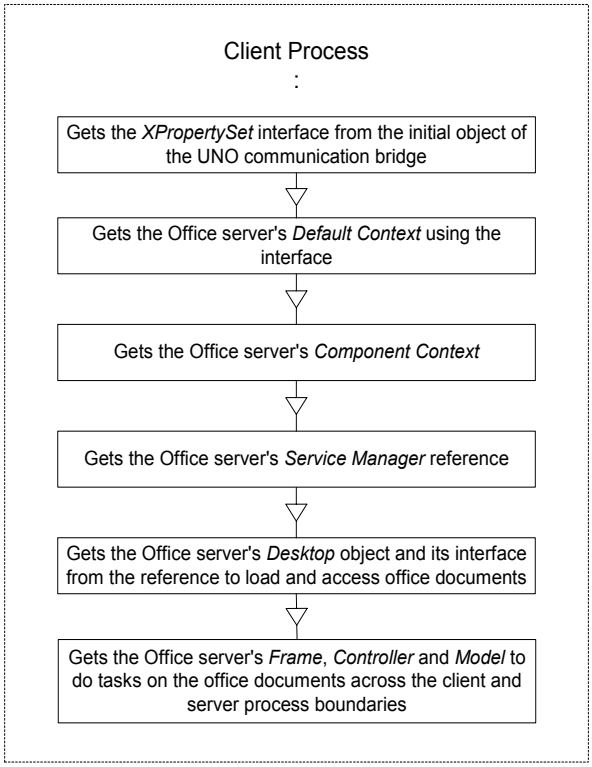



**Figure 4.4 The client's accessing of the Office Server's functionality through the established UNO communication bridge.**

# 4.6 The Master Client

After the procedures described in the previous section, the Master client has set up the remote bridge and is taking control of the programming features via the FCM paradigm. The Master client gets the current frame, which in Impress corresponds to the current opened presentation file. The client keeps testing for the current presentation file. If a change is detected, it means that either a new presentation file is opened or an opened one is switched to. The Master client then gets the URL of this presentation file through a method called `getURL()` in the interface of the Model. The Master client also registers listeners at the remote bridge to listen to events fired at the Office server, as in Figure 4.5.

One of the registered listeners is the “Property Change Listener,” which listens to property change events of an object. The client makes the listener listen to changes of the “Current Page” of the presentation file object.

```
   PropertyChangeListener propertyChangeListener = new
PropertyChangeListener();
   xPropertySet.addPropertyChangeListener("CurrentPage",
propertyChangeListener);
```

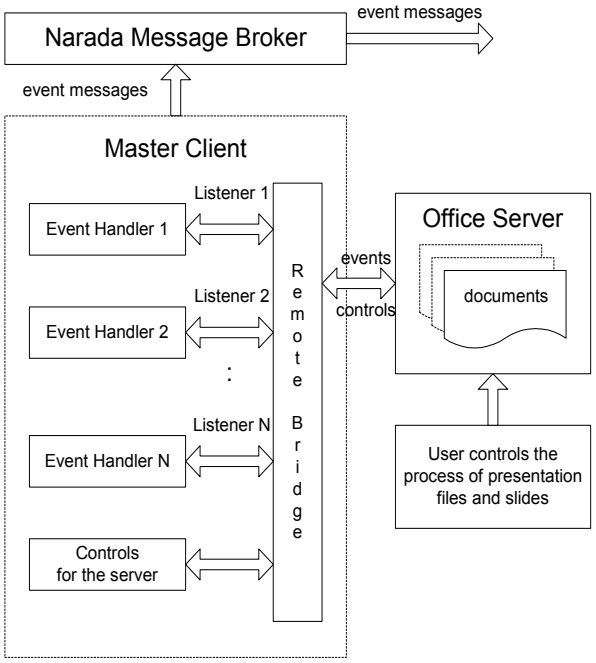



**Figure 4.5 The function structure in the side of the Master client applications.**

Whenever a presentation slide changes in the Impress server, the listener catches the event and notifies the event handler to do further processing. The event handler first gets the slide number using method `getPropertyValue(“Number”)` of the

`XPropertySet` interface. Then, it deals with the current slide number by adding appropriate eXtensible Markup Language (XML) [XML] tags and properties to address session information such as session identifier, topic title, source, and destination, as in

```
<event sessionID = "aSessionNumber" topic = "aTitile" to
= "receiver" from = "sender"> a slide number </event>
```

This way, each group of people in a session can send and receive messages correctly in a concurrent public message broker environment such as the NaradaBrokering Message Service. The Master client deals with the URL of the current presentation file in the same way, as in

```
<presentation sessionID = "aSessionNumber" topic =
"aTitile" to = "receiver" from = "sender"> a URL of a
presentation file </presentation>
```

As soon as such an XML message is generated, the Master client sends it to the NaradaBrokering Message Service for broadcasting to all the subscribed Participating clients for generating the same displays concurrently.

# 4.7 The Participating Clients

When the NaradaBrokering Message Service receives event messages from the Master client, it notifies the Participating clients and broadcasts the messages to them, as in Figure 4.6. Each Participating client connects to, controls, and makes use of the Office server. The OpenOffice application is installed on the host computer of the Participating client, and the presentation files have been downloaded or deployed beforehand to the same directories as the Master client. Each client processes the received event messages and generates the same displays as the Master client simultaneously.

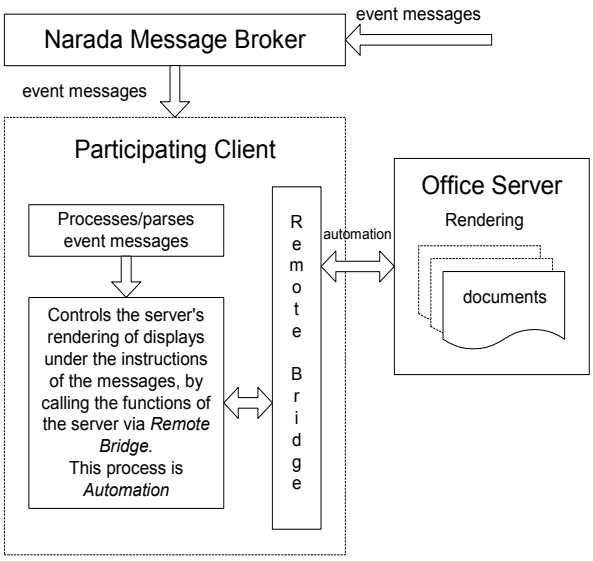



**Figure 4.6 The function structure in the side of the Participating client applications.**

To connect to the Office server, the Participating client goes through the same procedures described previously as the Master client, such as creating the remote bridge and getting the server's component context and service manager. The client then gets control of the server's Frame, Controller, and Model and makes use of the FCM

paradigm to call the server's functions. For example, when the client receives a message from the NaradaBrokering Message Service, it parses the message and gets the different information parts, such as the event type, the associated properties, and the URL of a presentation file; then it may call the functions of the server, such as `loadComponentFromURL()`, to open or switch to a presentation; then it may call the method `getDrawPages()` of the `XDrawPagesSupplier` interface, the method `getByIndex(index)` of the `XDrawPages` interface, and the method `select( xDrawPage )` of the `XSelectionSupplier` interface to navigate to a specific slide of an opened presentation file. The event type is the key to call different processing functions, and its associated properties are used in the functions to generate the correct presentation results. This process is automation; the functions of the Office server are called programmatically under the instructions of the event messages. Thus, the Participating clients are generating the same displays as the Master client, independently and simultaneously.

## 4.8 Collaboration on Deterministic Finite Automaton

As we have noticed so far, the Master and the Participating collaboration clients are designed for different purposes, in different architectures and implementing mechanisms; they are divergent. At the same time, they collaborate on messages to present the same output displays at each event; in other words, they have the same logic as to the state transitions on events and get to the same state of the Deterministic Finite Automaton (DFA) at the end of the process of each event; they are convergent. They actually

collaborate to share a common DFA in their respective instantiations (i.e., the finite automata in them are the same) and reach a common state of the DFA at any collaboration step. Collaboration between the clients is therefore all about being in a same state of the DFA at each event.

As we have shown in the example of chapter 2, the Master and Participating collaboration clients are at the start state $q_0$ of the DFA when instantiated. At this stage, the automata in their respective instantiations are not completely determined. The user on the Master client opens a presentation file by physically controlling the interface of the Impress application using mouse clicks and keystrokes (e.g., File > Open > File name). The Impress server responds to these physical events. The Master client gets the current frame, which in Impress corresponds to the current opened presentation file. The Master client keeps testing for the current presentation file. If a change is detected, it means that either a new presentation file is opened or an opened one is switched to. The Master client then gets the URL of this presentation file through a method called `getURL()` in the interface of the Model. Then, it builds up the corresponding event message containing information about the function to call (which performs presentation file opening) and the property (the path and name of the presentation file). The event message is a delimited text string (e.g., "`document: C:/file1.sxi`") and the intermediate representation of the event; the message is sent out to the NaradaBrokering Message Service for broadcasting to the Participating clients.

Each Participating client parses the received delimited message string, converts the property data to its system's native representation, and, based on the information, arranges the function to call. After it constructed the native representation of the URL of

a presentation file from the received event message, the Participating client calls the relative function of the server with the native URL as the parameter, such as `loadComponentFromURL(C:/file1.sxi)`, to open or switch to that presentation. This way, it automates to open the same presentation file as the Master client programmatically.

At this stage, the automata in the instantiations are determined with respect to the opened presentation file, as to the total states and the relationships between the states. This event is a *major event* because it defines the states of the automata. From this state (state $q_1$ in the example), we can navigate through the slides of the presentation file on events such as "`slide: CertainSlide#`," "`Previous`," and "`Next`." These events are *minor events* because they just change the states of the defined automata. The collaboration on the minor events is similar to the collaboration on the major event we have just described, so we describe it briefly next.

The Master client also registers listeners at the remote bridge to listen to events fired at the Office server, as in Figure 4.5. One of the registered listeners is the "Property Change Listener," which listens to property change events of an object. The client makes the listener listen to changes of the "Current Page" of the presentation file object. Whenever a presentation slide changes in the Impress server, the listener catches the event and notifies the event handler to do further processing. The event handler first gets the slide number using method `getPropertyValue("Number")` of `XPropertySet` interface; then it builds up the event message and sends it out.

When the Participating client receives a message from the NaradaBrokering Message Service, it parses the message and gets the different information parts, such as the event

type and its properties. The Participating client can call the method `getDrawPages()` of the `XDrawPagesSupplier` interface, the method `getByIndex(index)` of the `XDrawPages` interface, and the method `select( xDrawPage )` of the `XSelectionSupplier` interface, to navigate to the specific slide of the opened presentation. The event type is the key to call different processing functions, and the event's associated properties are used in the functions to generate the correct presentation results. This process is automation; the functions of the Office server are called programmatically under the instructions of the event messages. Thus, the Participating clients generate the same displays as the Master client. We list the environments and the output displays of the Master client and the Participating client at the event, in Figures 4.7 and 4.8, respectively.

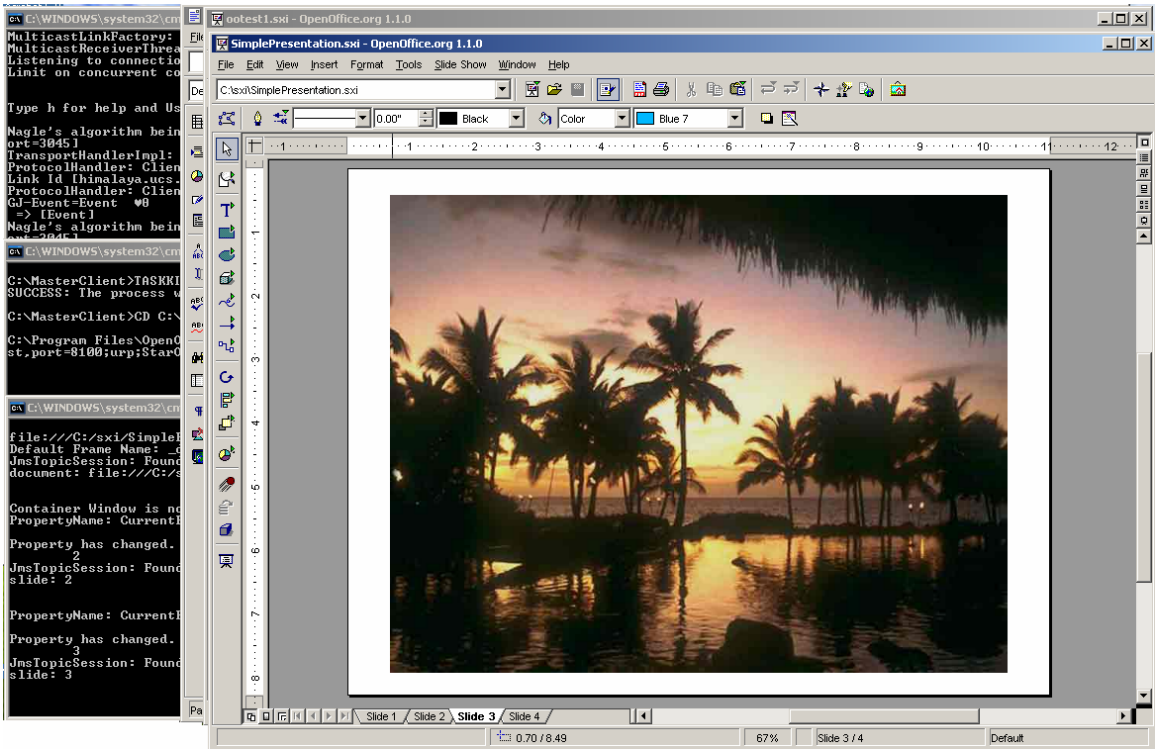


**Figure 4.7 The environments and an output display at an event on the side of the Master client of the collaborative Impress applications.**

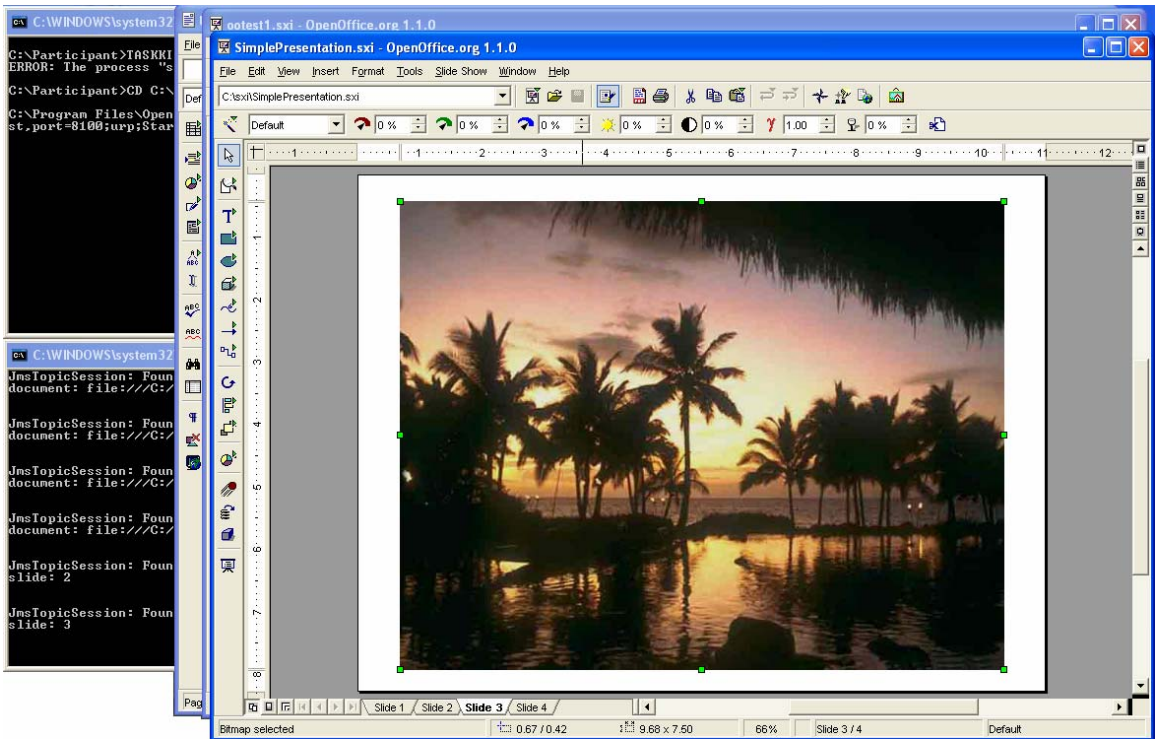



**Figure 4.8 The environments and the corresponding output display at the same event as the Master client on the side of the Participating client of the collaborative Impress applications.**

In short, the Master client and the Participating clients collaborate to share a common DFA and reach a common state of the DFA at any collaboration step. Collaboration between the clients is all about being in a same state of the DFA at each event. The Master client and the Participating client are designed for different purposes, in different architectures, implementing mechanisms, and code shapes; they are divergent. At the same time, they collaborate on event messages to present the same output displays

at each event; they have the same logic as to the state transitions on events and get to the same state of the DFA at the end of the process of each event; they are convergent.

# Chapter 5

# Collaborative Interactive Data Language (IDL) Applications

In this chapter, we describe our effort in developing Collaborative Interactive Data Language (IDL) Applications. Interactive Data Language (IDL) is an array-oriented data analysis and visualization application environment, which is widely used in research, commerce, and education. It is meaningful to make user IDL applications collaborative between computers over networks, using a common message broker as the underlying communication system.

In order to achieve the global collaboration, we have brought together in our research a Grid-based Collaboration paradigm, a Shared Event model, different implementing structures, methodologies, and technologies. We have first succeeded in our prototype codes through one approach (a Notifying structure); later we have worked on a real life IDL application package (ReviewPlus) and succeeded in making it collaborative through

another approach (a Polling structure). At the same time, we are trying to find more and potentially better structures and methods for collaborations in general user IDL applications, as we have deduced a Dynamic structure and an Embedded structure for further implementation.

In each structure, the collaborative IDL applications consist of the components of the Master and Participating clients. We elaborate the components of the collaborative applications on their purposes, architectures, and implementing mechanisms. The clients collaborate with each other on events to keep showing the same output displays at each step, using the NaradaBrokering Message Service as the underlying communication system for delivering of event messages.

We further show that, as we have mentioned in chapter 2, the clients actually collaborate to share a common Deterministic Finite Automaton (DFA) in their respective instantiations (the finite automata in them are the same), and reach a common state of the DFA at any collaboration step. Collaboration between the clients is therefore all about being in a same state of the DFA at each event.

The Master and Participating collaboration clients are designed for different purposes, in different architectures and implementing mechanisms; they are divergent. At the same time, they have the same logic as to the state transitions on events, and get to the same state of the DFA at the end of the process of each event; they are convergent.

## 5.1 The Big Picture

Interactive Data Language (IDL) is an array-oriented data analysis and visualization application environment, which is widely used in research, commerce, and education [Gumley, RSI]. Its application areas include engineering, medical physics, astronomical and space science, and earth science. It offers rapid interactive data analysis and visualization, a programming environment, and end user applications.

IDL is available for Windows, UNIX, Linux, Macintosh, and VMS platforms and Operating Systems. This high availability facilitates data analysis and visualization in multi-platform environment, and ensures high code portability among platforms and systems. People from different categories around the world have developed and used diverse IDL applications in their respective areas. Also, users worldwide are continually contributing to Internet-based IDL libraries [IDL+Lib1, IDL+Lib2, IDL+Lib3, IDL+Link] that are freely available.

It is contributing and meaningful to make IDL applications collaborative between computers of same or different platforms, using a common message broker – such as the NaradaBrokering Message Service – as the underlying communication system. In today's Information revolution society, collaboration is becoming more and more important. It means breakthroughs in new abilities, which otherwise would be hardly possible; it means achievements in the advances of science and technology; it also means great contributions to economy.

We have designed the overall structure of the collaborative IDL applications to consist of a type of Master (or Master Client) and a type of Participant (or Participating Client), using small text event messages for the communication between them. During a

collaboration session, the Master captures events in its process, deals with them, builds up and sends out the event messages to the Participant for rendering the displays in the Participant's process, so that both of them can share the screen displays simultaneously. There can be multiple Participants working with the Master concurrently and independently. We use the NaradaBrokering Message Service [Fox+JGI, Pallickara+JDIM, Pallickara+Dissertation] for the message communication.

The RSI IDL software is installed on both the host computers of the Master and the Participant. If files are needed in a session, they are deployed beforehand on the same directories on the hosts. All clients are required to be in a session and keep in that session for the whole collaboration, because an event message coordinates each client to change its current status, and the correct transition to a subsequent status depends on the previous one.

We have researched and explored this area by using:

- A Grid-base Collaboration paradigm, in which Shared Event Model acts as the messenger, and Peer-to-Peer Grid computing [Berman+Fox+Hey, Fox+CTS, Wang+JDIM] acts as the basis.
- Different Implementation Structures for the collaborative IDL applications – Notifying Structure and Polling Structure.

We illustrate the mechanisms, methodologies, and technologies used in each structure, and analyze their strengths and limitations in the context of applications. We have developed prototypes of collaborative event-driven GUI programs by first using the Notifying structure and then using the Polling structure; we have demonstrated the global collaborations of the prototypes through the Internet. Later, we have worked on a real-life

IDL application package (ReviewPlus [ReviewPlus] that is a general-purpose data visualization tool from General Atomics of USA [GA]), and succeeded in making it collaborative using the Polling structure. We will describe the development and special issues in the implementation later in this chapter. Our research contributes to Grid-based Collaboration in Interactive Data Language Applications.

## 5.2 Grid-based Collaboration Model

We have used a Grid-based Collaboration Model in the design and development of the collaborative Interactive Data Language (IDL) applications, as shown in Figure 5.1.

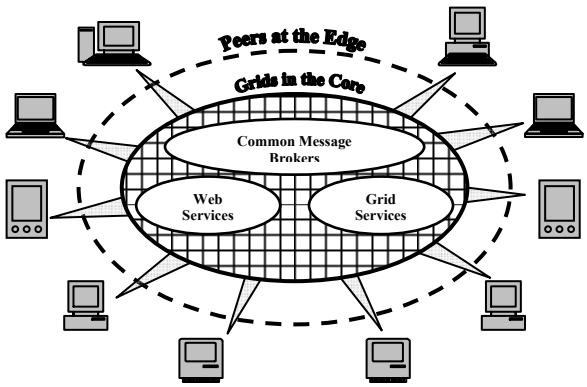



**Figure 5.1 A grid-based collaboration model.**

In this model, there are two categories of systems – the Grid system and the Peer-to-Peer systems. The Grid system [Gannon, Fox+CISE+July2004, Santo, Globus] is the basis; it largely comprises stable, formal, and efficient high-functionality services such as Web Services, Grid Services, and Common Message Brokers, which are deployed in the

Grid on structured, well-organized, and powerful supercomputers. They are in the core of the model. The Peer-to-Peer system is the interface to this world; it offers user-friendly, convenient, intuitive, and easy accessible applications and services such as the popular commodity software used daily and everywhere. They are installed on a variety of personal devices, such as desktops, laptops, PDAs, and smart phones. They are at the edge of the model.

The infrastructure of the Internet ties up and correlates the two categories. It enables the Peer-to-Peer Grid computing to be a trend, which harnesses the advantages of the two categories (so that they complement each other) and brings new opportunities and challenges to computing in all. The Grid system offers robust, structured, and security services that scale well in pre-existing hierarchically arranged enterprises or organizations; it is largely asynchronous and allows seamless access to supercomputers and their datasets. The Peer-to-Peer system is more convenient and efficient for the low-end clients to advertise and access files on the communal computers; it is more intuitive, unstructured, and largely synchronous.

In our design and development of the collaborative IDL applications, we have realized the Peer-to-Peer Grid computing idea. We have deployed the Narada Message Broker in the Grid and used it for message communication between the Master and Participants of the collaborative applications; we have deployed the Master and Participants as Peers at the edge and made them collaborate on events.

## 5.3 Shared Event Model

We use a Shared Event Model in the communication between Peers [Wang+KSCE]. In this model, small text event messages are transmitted via the Grids of common message brokers and are used to coordinate the operations between the peers, so that the peers cooperate concurrently and share the output displays simultaneously. In our design of the collaborative IDL applications, one type of the Peers is the Master client, and the other is the Participant client. During a collaboration session, the Master client captures events in its process, deals with them, and sends the event messages to the Participant client for rendering the displays in the Participant client's process, so that both clients can share the screen displays simultaneously [Fox+CISE+March2004]. There can be multiple Participant clients working with the Master client concurrently and independently. We use the Narada Message Broker in the Grid for the message communication. The RSI IDL software is installed on both the host computers of the Master and the Participant; if files are needed in a session, they are deployed beforehand on the same directories on the hosts. This deployment guarantees that the files' access is correct on the hosts under the controls of the event messages.

There are a variety of primitive widgets in IDL, such as Button, Slider, Text field, and Draw area. The event structure for each widget is different; each one contains state information specific to that widget, e.g., flags and values. However, there are three common items in all the event structures; they are ID, TOP, and HANDLER, which are long integers and the first three items in each structure.

1. ID is the widget ID number of the widget that generates the event.

2. TOP is the widget ID number of the top-level base that contains the widget that generates the event.
3. HANDLER is the widget ID number of the widget that is associated with an event handler.

For instance, the event structure for BASE widget is:

{WIDGET_BASE, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L}

Where X is the width of the base, and Y is the height.

On the Master, the client captures the event, gets the event structure, packages the information from it into a delimited string, as in `{widget_base|id 0|top 0|handler 0|x 0|y 0}`, with possibly other information such as session, source, and destination, and sends the result message string to the Narada message broker for broadcasting to Participants. This is mainly a serialization process. On the participant, the client parses the received message string, gets all the different parts of the delimited string, and rebuilds the IDL event structure by converting the sub-string sections (like "`id 0`") to corresponding IDL types. This is mainly a de-serialization process. The constructed event structure is then used as a parameter for its event handler, which is invoked by the Participant client programs to generate the same event result as that happened on the Master client.

## 5.4 Notifying Structure

Based on the Notifying structure, we have developed and made a simple IDL GUI application collaborative between the Master and Participating clients. The Master client displays a GUI containing a lot of button widgets, which represent JPEG images. When a user clicks a button, the following will happen:

- The corresponding image displays in IDL environment.
- The Master client captures the event and sends the message to the NaradaBrokering to broadcast to the Participating clients for rendering.

The Participant receives the event message broadcasted from the NaradaBrokering, and renders the display as that of the Master in its own IDL environment. There can be multiple instances of Participant clients. The interface and an IDL display on both the Master and the Participant are shown in Figure 5.2.

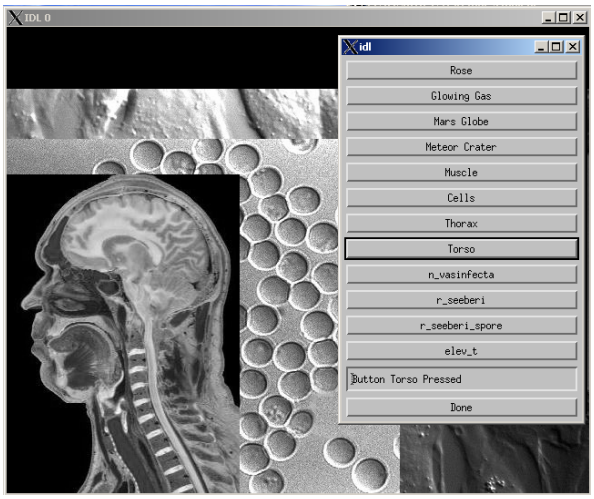



**Figure 5.2 The interface and a display from a collaborative IDL application.**

In this structure, we just naturally followed the nature and technologies in IDL and the NaradaBrokering (written in Java) to program the application to be collaborative. The technologies for this issue include IDL-Java Bridge, Callable IDL, Shared Library, Java Native Interface (JNI) [Liang], and Subscribe/Notify mechanism. Whenever the NaradaBrokering receives an event message from the Master client, it will be *Notifying* the Participating client immediately through its method `onMessage()`, which in turn gets the message and invokes all kinds of IDL routines accordingly to render the display.

More specifically, the Master client is written in IDL programs. It consists of a GUI building and managing part, and an event handling part.

- It makes use of the IDL-Java Bridge, calls methods in a Java program to connect to the NaradaBrokering Message Service.
- It captures the event and gets the event message in an event handler whenever a user clicks a button in the GUI.
- It sends the event message over to the NaradaBrokering Message Service for broadcasting to the Participant clients.

This process is elaborated in Figure 5.3.

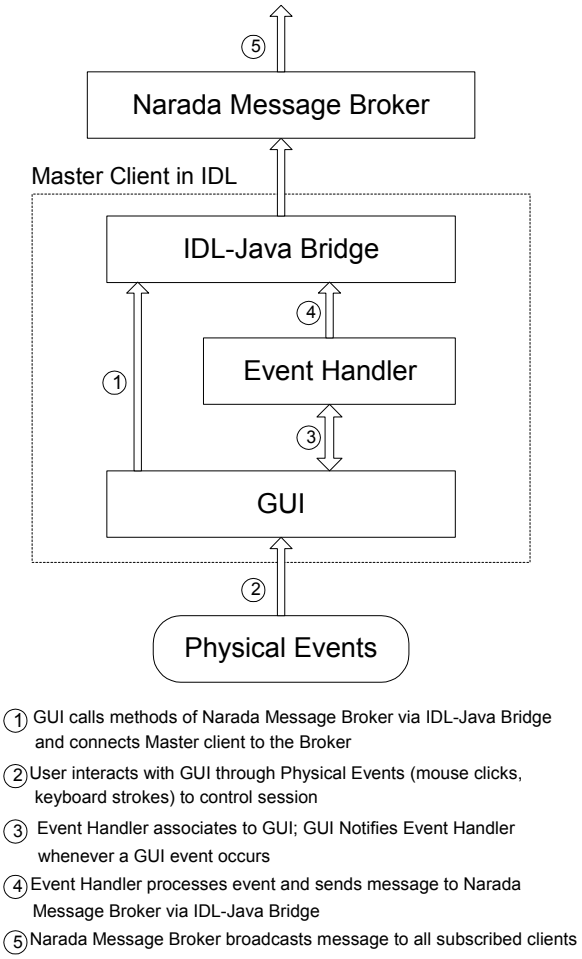



**Figure 5.3 The mechanism of the Master client.**

The Participant is written in a Java program.

- It connects to the NaradaBrokering and receives event messages from it.
- The Java program controls the rendering process according to the event messages it receives.
    - It makes use of the Callable IDL technology and the JNI technology.
    - It calls the IDL routines (procedures or functions) for the rendering.

- In order to do that, it has to call the IDL routines through a C program; in other words, that C program calls IDL routines directly through Callable IDL technology.
- A shared library (`libCallableIDL.so`) is generated from the C program, and the Java program calls the native functions in the shared library through JNI.

This process is elaborated in Figures 5.4 and 5.5. This way, it renders the images simultaneously with the Master client.

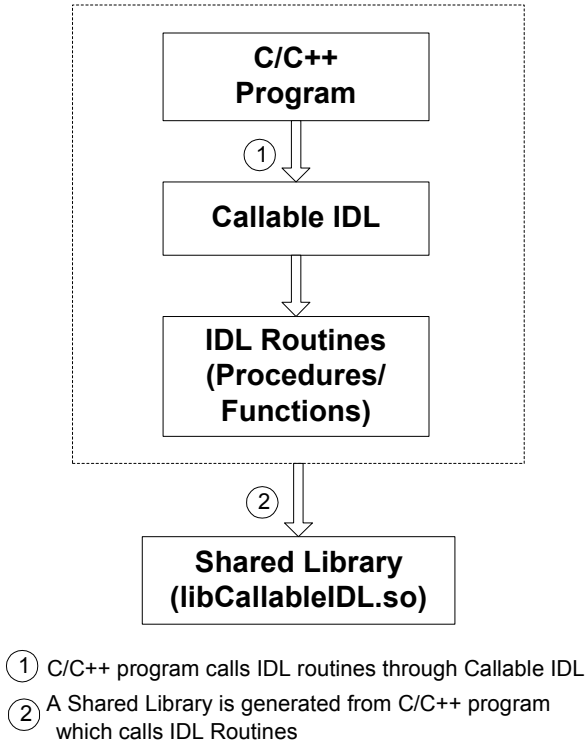



**Figure 5.4 Generating of a shared library.**

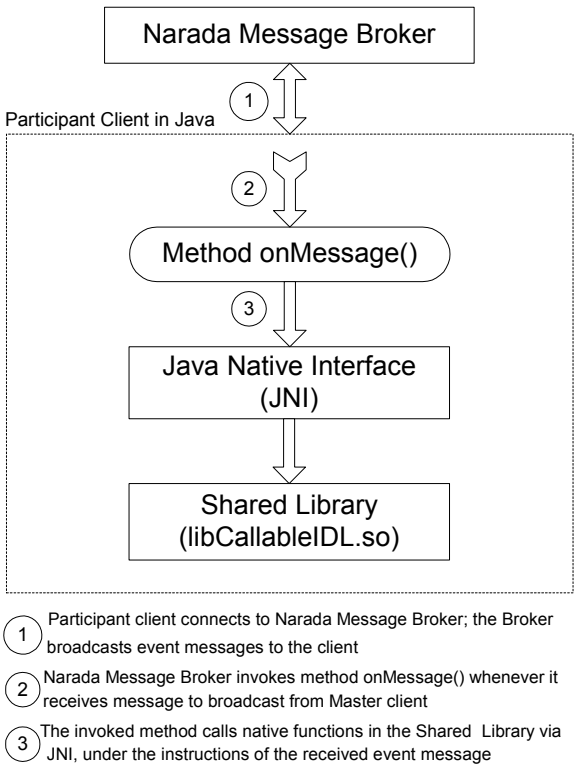



**Figure 5.5 The mechanism of the Participant client.**

This structure works just fine for our small demonstrating collaborative IDL GUI applications. It follows the nature of the IDL and the Narada Message Service systems, and it builds on top of those systems and just makes use of their technologies and functions in achieving collaboration without changing anything in either of them. In other words, all of the design and programming is just within our collaborative applications. This is architectural abstraction thinking and complies with the object-oriented design in systems level.

There is a limitation in this structure: the types of the programs for the Master client and the Participant client are different. The code for the Master client is in IDL (.pro) language; the code for the Participant client is in Java (.java), calling native functions in a

shared library through JNI. The shared library in turn is generated from a C program, which calls IDL routines directly through the Callable IDL technology. In other words, all the functionality of the IDL routines is compiled into binary and put in the shared library (.so). This is not good enough for large application development like ReviewPlus, which is huge itself and also refers to many (if not huge) other IDL routines in other IDL applications such as MDSplus [MDSplus].

This limitation implies:

1. It is complicated and inconsistent in the codes between the Master client and the Participant client; they look totally different things, and there is no similarity.
2. The time and efforts in developing would be at least doubled; two different types need to be developed – one for the Master, one for the Participant.
3. It is error-prone in programming, debugging, and testing, bringing different technologies and environments together.

## 5.5 Polling Structure

Now that we have succeeded in making a simple IDL application collaborative, we begin to think further and ask a question: is it possible to develop the codes for both the Mater and the Participant in pure IDL and make the codes for the Master and the codes for the Participant as same as possible? If we succeed, we can overcome the limitation and shortcomings mentioned in the Notifying structure; we can simplify the collaboration system and make the codes consistent and clear; we can save significant time, effort, and

cost in software development and maintenance. Thereof, we can really make Grid-based collaboration for large IDL applications (such as ReviewPlus) practical and feasible.

For this, we have tried and succeeded in our simple GUI IDL collaborative applications using the Polling Structure. There is a trade-off. In order to achieve this, we have to change some parts of the underlying systems in some cases, thus suffering some design abstraction; in our case, we have changed an interface of the underlying Narada Message Broker.

In the Polling Structure, both the Master client and the Participant client make use of the IDL-Java Bridge to connect to the NaradaBrokering and communicate with it. The methods used in the Bridge belong to IDL. As before, the Master client captures events and sends event messages to the NaradaBrokering, which then broadcasts them to all the Participating clients, as in Figure 5.3. In a Java class, we add public global variables for event change flag and event message, and make a notification related method onMessage() update them whenever the Broker broadcasts event messages to the clients. The update includes increasing the event flag and storing the event message in the variables (to the tail of a linked list). The Java class is an interface of the NaradaBrokering; the Participating clients codes instantiate the class and make use of it.

The Participating client code now has an instance of the Java class; it is constantly testing, or *Polling*, the instance variable – the event flag. If it finds that the flag is positive (indicating there is at least one event message left in the linked list), it decreases the event flag and retrieves an event message from the head of the linked list. It then follows the instructions of the message to execute different parts of the IDL programs to do the rendering. This process is elaborated in Figure 5.6.

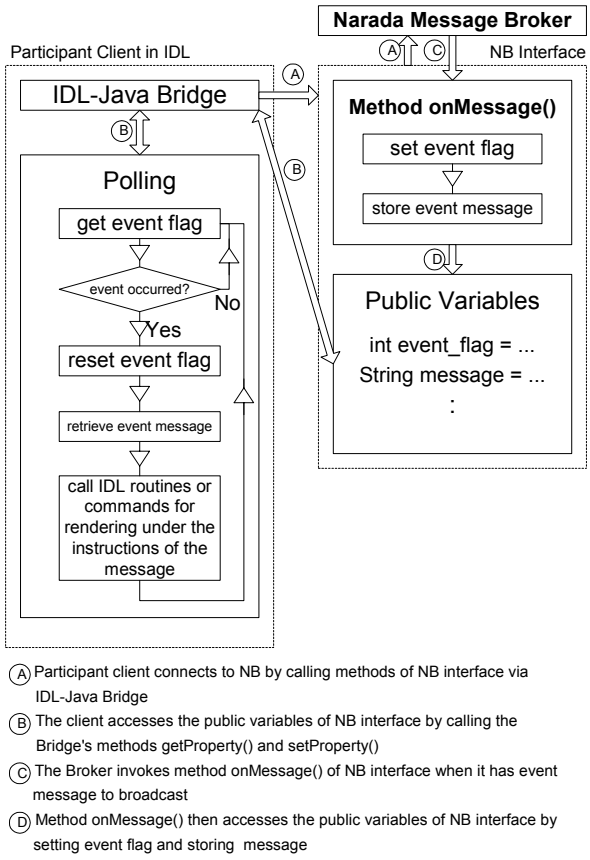



**Figure 5.6 The mechanism of the Participant client in a polling structure.**

This way, the Polling structure makes the collaboration working. It has advantages in working with large IDL applications. We have used it in the design and implementation of the collaborative ReviewPlus applications. One of the interfaces and displays from ReviewPlus is shown in Figure 5.7.

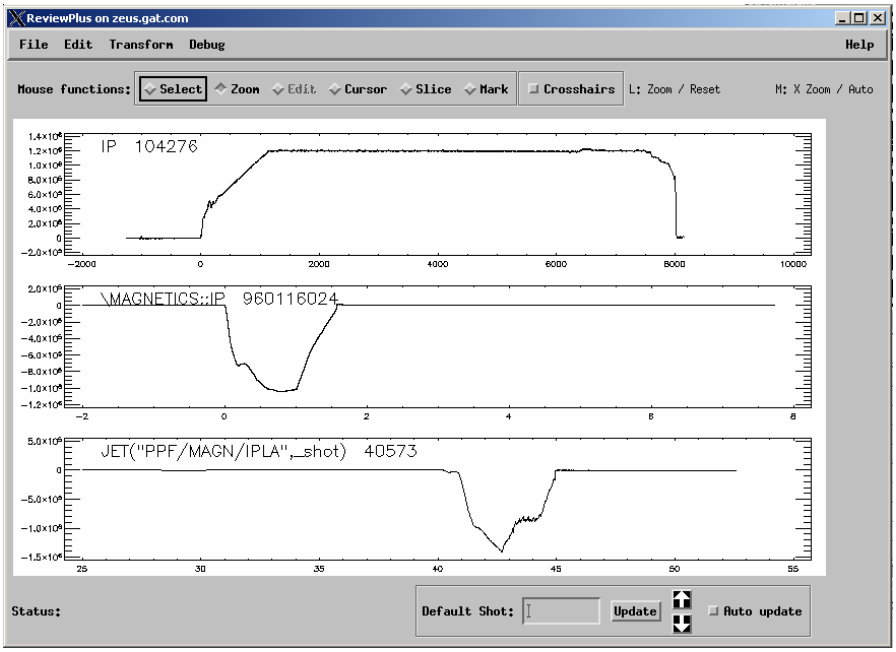



**Figure 5.7 One interface and display from ReviewPlus IDL application.**

As in IDL widget programming, the structure of ReviewPlus includes two parts: one is the widgets definition part, and the other is the event handling routine part. All the required widgets in the application are defined and realized in the former, and the event handlers are contained in the latter. The event handling part is put first in a program unit, and the definition part follows. People specify the event function or procedure to be called when a widget is invoked by using the keywords `EVENT_FUNC` or `EVENT_PRO` in the definition of the widget. This way, when the widget is invoked, the corresponding event handler is called to process the event. For example, in ReviewPlus, there is a piece of code:

```
mEdit = Widget_Button(menubase, Value='Edit')
```

```
     x = widget_button(mEdit, value='Set Signals', $
event_pro='ReviewPlus_SignalDialog_event')
```

This is for the item “Set Signals” on menu “Edit”. When this widget fires an event, the event handler “`ReviewPlus_SignalDialog_event`” is invoked. Both the Master and the Participant are developed on the basis of ReviewPlus and make use of its codes as much as possible. On the Master, when we choose the “Set Signals” item from menu “Edit”, the event handler “`ReviewPlus_SignalDialog_event`” is invoked, and the tasks in the handler are processed. In it, we can put statements to get information of the event structure and put pieces of substrings for the fields of the event structure in a message string, as we have described in Section 5.3, Shared Event Model. We put a substring for locating the desired event handler (e.g., “Edit>SetSignals”) followed by a delimiter in the event message string. The NaradaBrokering then broadcasts the message. On the Participant, it receives the string from the public variables in the polling structure. After it parses and gets the substrings, it locates the event handler and rebuilds the event structure in IDL types. It then calls the event handler with the event structure like this:

```
   ReviewPlus_SignalDialog_event, {WIDGET_BUTTON, ID:15,
TOP:1, HANDLER:15, SELECT:1}
```

## 5.6 Collaboration on Deterministic Finite Automaton

As we have noticed so far, the Master and the Participating collaboration clients are designed for different purposes, in different architectures and implementing mechanisms; they are divergent. At the same time, they collaborate on messages to present the same output displays at each event; in other words, they have the same logic as to the state transitions on events, and get to the same state of the Deterministic Finite Automaton (DFA) at the end of the process of each event; they are convergent. They actually collaborate to share a common DFA in their respective instantiations (i.e., the finite automata in them are the same), and reach a common state of the DFA at any collaboration step. Collaboration between the clients is therefore all about being in a same state of the DFA at each event.

Next, we describe the collaboration between the entities of the Master and Participant clients of the collaborative ReviewPlus applications, using pieces of code from them to demonstrate one collaboration step, mainly focusing on the event and the transition function. This collaboration step illustrates the idea of collaboration in terms of convergence on the same state of the DFA at the end of the process of the event, with the transition function doing the real job of the state transition. As the others, this collaboration step also shows this: even though the code shapes of the Master and Participant clients diverge, the clients converge on the same state of the DFA at the event.

Since the output displays of both the Master client and the Participant client at the event are the same, we just show image captures on one side in the demonstration of the collaboration step. We begin with the invocation of the collaboration entities, as shown in Fig. 5.8. This corresponds to the start state $q_0$ of the DFA.

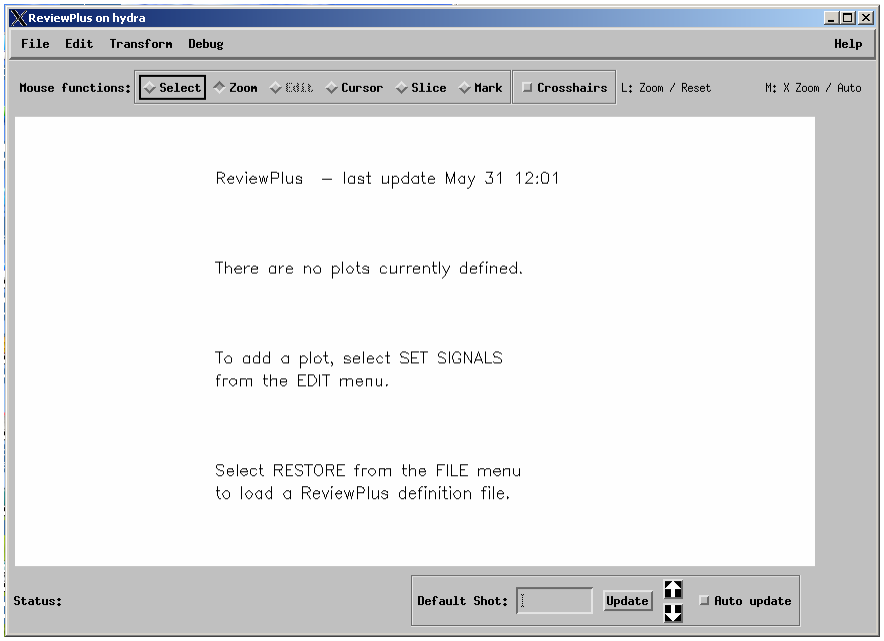



**Figure 5.8 The initial interface and display of ReviewPlus.**

At this stage, the initial interface, the subsequent interfaces that would be produced from it, the widgets of the interfaces, and the relationships between them are determined. Correspondingly, the initial states and the relationships between the states of the automata in the instantiations are determined. This initial determination is due to the initial construction, configuration, logic, and condition of the programs of the clients. This invocation is an event, and it is a *major event*, because it defines the initial states of the automata.

From the initial interface (also as shown in Figure 2.2), we can get to other interfaces, display results, or states by invoking the widgets on the menu or sub-menus, such as “Edit>Set Signals,” “Edit>Change Plot Grid,” “Debug>Show Items,” or “Help>Getting

Started." These events are *minor events* because they just change the states of the current automata.

The automata can be extended dynamically by other *major events* to include more states and relationships with the process of the collaboration. For instance, after some data are input into some cells on the interface of "ReviewPlusSetup" (see Figure 5.10) to define some signals, the widget "Done" or "Apply" on the interface can be invoked to generate signals similar as those shown in the display of Figure 5.7. The events of invoking widgets "Done" and "Apply" are *major events*. In programming, we treat all the events indistinguishably as *flat events*. To distinguish between *major events* and *minor events* is of importance to theory. From the interface of Figure 5.8 on the Master client, if we click on the "Edit" item from the main menu, a sub-menu will appear, as shown in Figure 5.9.

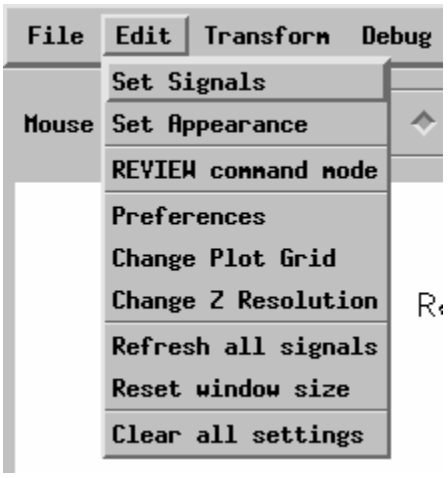



**Figure 5.9 A sub-menu from the main menu of ReviewPlus.**

If we then click on the “Set Signals” item from the sub-menu, an event is fired. This is a button widget; an event handler routine is defined for the widget. We describe the pieces of code for both the Master client and the Participant client in response to this cooperation as follows.

**The Master Client Side**

- Widget creation

```
x = widget_button(mEdit, value='Set Signals', $
                  event_pro='ReviewPlus_SignalDialog_event')
```

From the code above we know that this button widget has `'Set Signals'` as its value shown on its appearance, and it is associated with an event procedure named `'ReviewPlus_SignalDialog_event'`. When the button is clicked, the procedure is called by the IDL system.

- Definition of event structure for widget

Here is the definition of the event structure for widget button:

```
{WIDGET_BUTTON, ID:0L, TOP:0L, HANDLER:0L, SELECT:0L}
```

It has a name WIDGET_BUTTON and 4 fields – ID:0L, TOP:0L, HANDLER:0L, and SELECT:0L, each with a field name, a colon, and a type value. In this case all the values of

the fields are of `long` type indicated by the suffix letter L. SELECT: If the button is pressed, the value is 1; if it is released, the value is 0.

- Event handler

```
pro ReviewPlus_signaldialog_event,event
;;;;;;;; collaboration code added ;;;;;;;;
eventMessage = "ReviewPlus_signaldialog_event;"+"WIDGET_BUTTON;"+"ID;"$
  +string(event.ID)+";TOP;"+string(event.TOP)+";HANDLER;"$
  +string(event.HANDLER)+";SELECT;"+string(event.SELECT)
  COMMON BROKER, joChat2
  joChat2 -> writeMessage, eventMessage
;;;;;;;; end of collaboration code ;;;;;;;;
  widget_control,event.top,get_uvalue=info
  info.oReview->SignalDialog
end
```

From the code above we can see that the collaboration code captures the event and gets the field data from `event.ID, event.TOP, event.HANDLER,` etc., converts them into strings, and serializes the strings into a semicolon delimited string, along with the event structure name `"WIDGET_BUTTON"` and the event handler name `"ReviewPlus_signaldialog_event"`. This resulting string is the event message and is sent to the NB broker for broadcasting to the Participants.

**The Participant Client Side**

- Parsing of event message

```
            result = STRSPLIT(uval, ';', COUNT=count, /EXTRACT,
/PRESERVE_NULL)
            which_event = result[0]
            which_widget = result[1]
```

The next event message string for the Participant client to process is saved in variable `uval`. The IDL system function `STRSPLIT` is called to parse it with `';'` as the delimiter. All the pieces of information around the delimiter are extracted and saved in the array `result` with null string preserved as a piece, and the total number of them is saved in variable `count`. The event handler name is in `result[0]` or `which_event`, and the event structure name (or widget name) is in `result[1]` or `which_widget`. The rest of the pieces are all for the fields of the event structure; they are saved in the rest elements of the array starting with `result[2]`.

- Conversion to IDL native types

```
            FOR i=2, count-1, 2 DO BEGIN
               IF (result[i] EQ 'ID') THEN BEGIN
                  id_name = 'ID'
                  id_value = long(result[i+1])
               ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
                  top_name = 'TOP'
                  top_value = long(result[i+1])
```

```
   ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
      handler_name = 'HANDLER'
      handler_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'SELECT') THEN BEGIN
      select_name = 'SELECT'
      select_value = long(result[i+1])
                        :
   ENDIF
ENDFOR
```

The code above converts the data (in string) of the fields of the button event structure to the IDL native types; each pair of the strings (those stored in `result[i]` and `result[i+1]`) decide the field's value and the type of the value, with the former indicating the name and type of the value (due to the unique association of a name with a type, the name alone can also indicate a type, e.g., `ID` is a `long` type), and the latter the value in string. In this case, all the values of the fields are of `long` type, and therefore the strings are converted to IDL type `long`.

- Construction of event structure

```
IF (which_widget EQ 'WIDGET_BUTTON') THEN $
   event_structure = {WIDGET_BUTTON,id:id_value,$
   top:top_value,handler:handler_value,select:select_value}$
ELSE IF ...
```

The code above constructs the widget button event structure using the converted native IDL values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`.

- Invocation of the routine of event handler

```
...
ELSE IF (which_event EQ 'ReviewPlus_signaldialog_event') THEN BEGIN
           ReviewPlus_signaldialog_event, event_structure
     ENDIF ELSE IF ...
```

The code above calls the routine of the event handler `ReviewPlus_signaldialog_event` with the constructed event structure `event_structure` as the only parameter.

**Step Summary**

In the process of the event, both the Master client and the Participant client call the same routine – the event handler `ReviewPlus_signaldialog_event`, which is a unit of the transition function δ, with the event structure as the only parameter. The event message acts as the messenger, the information source, and the coordinator. With $\delta(q_0, a_0) = q_1$, the Master client and the Participant client converge on the same state $q_1$ of the DFA on event message $a_0$ at the end of the process of the event; therefore they have the same output displays, as in Figure 5.10, which is a part of a big interface. Note that, inside an event handler other routines can be called in any sequence and order, which we don't

have to worry about but just think of the whole as the encapsulation and abstraction of the event handler.

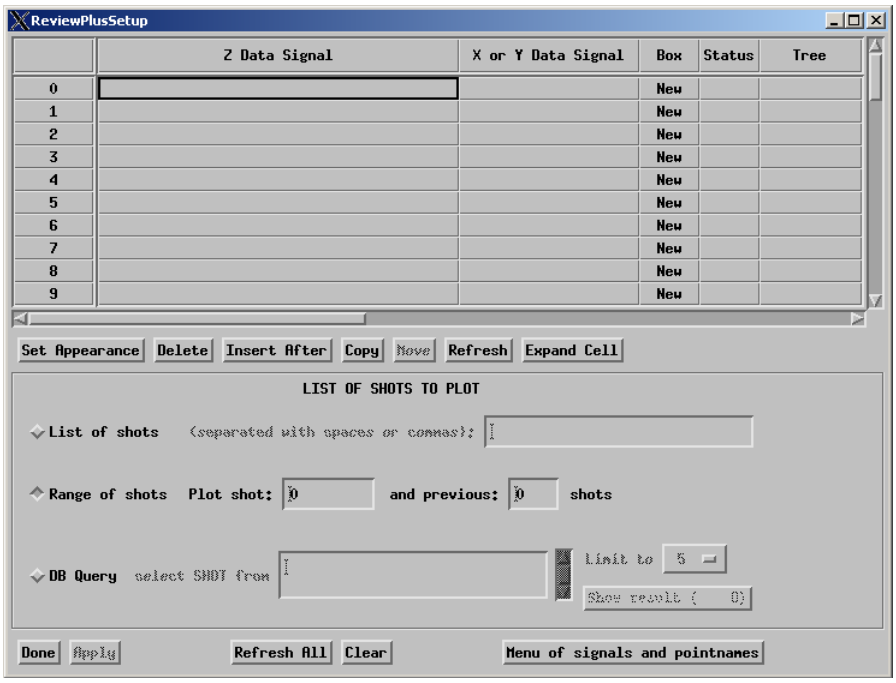



**Figure 5.10 A part of a big interface in ReviewPlus for setting up and managing of signals.**

By now, we have demonstrated one collaboration step on an event between the clients. We have given more demonstrations of collaboration steps on events in *Appendix B*. There, instead of using exhausting enumeration of each and every widget cases in the interfaces, we give some typical and interesting ones that sufficiently illustrate the idea of collaboration in terms of convergence on the same state of the DFA at the end of the process of each event, with the transition function doing the real job of state transitions.

We run the Master client and the Participant client on a desktop computer and captured some screen shots of their interfaces for demonstration. We list the

environments and the output displays of the Master client and the Participant client in Figures 5.11 and 5.12, with the Master client in the front and the Participant client in the back.

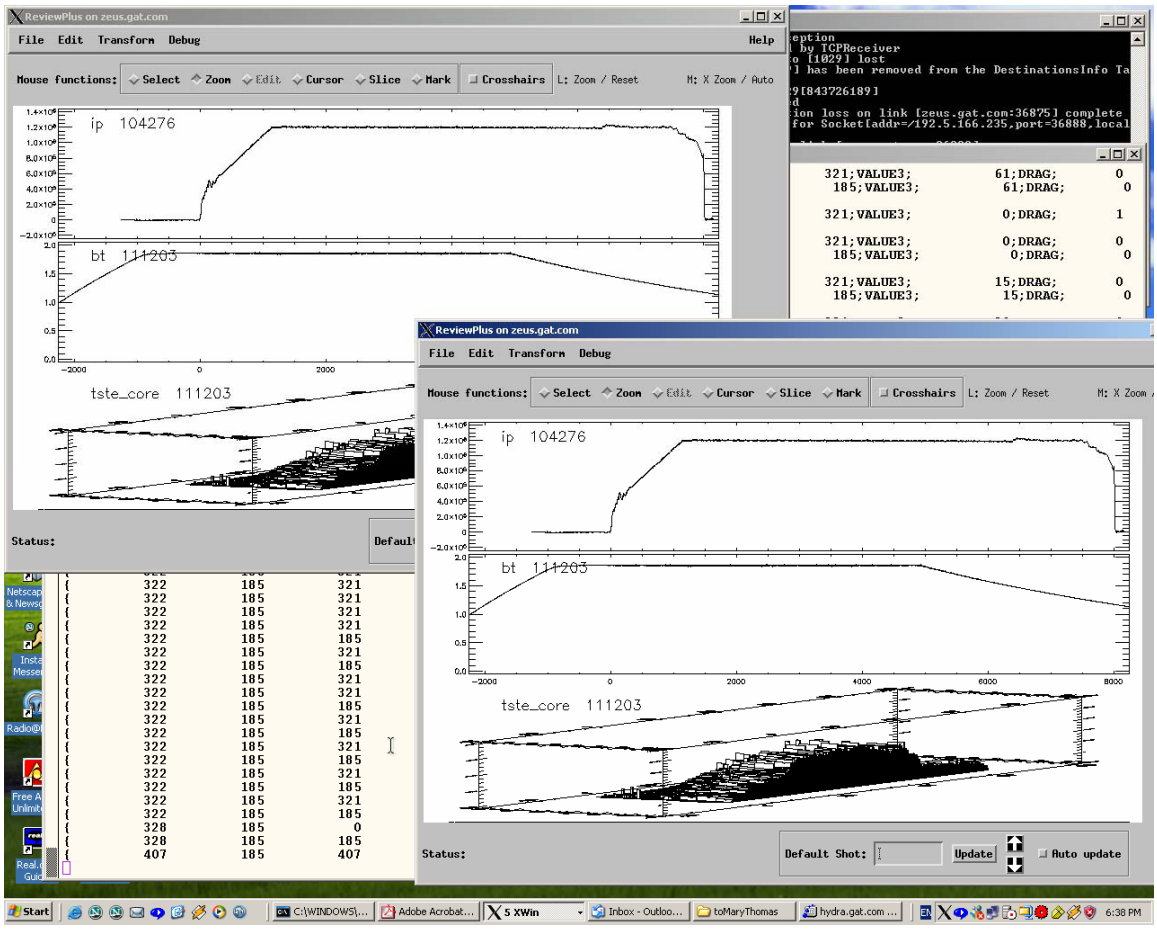



**Figure 5.11 A screen capture of the environments and the output displays of the Master client and the Participant client of the collaborative ReviewPlus applications running on a single desktop computer.**

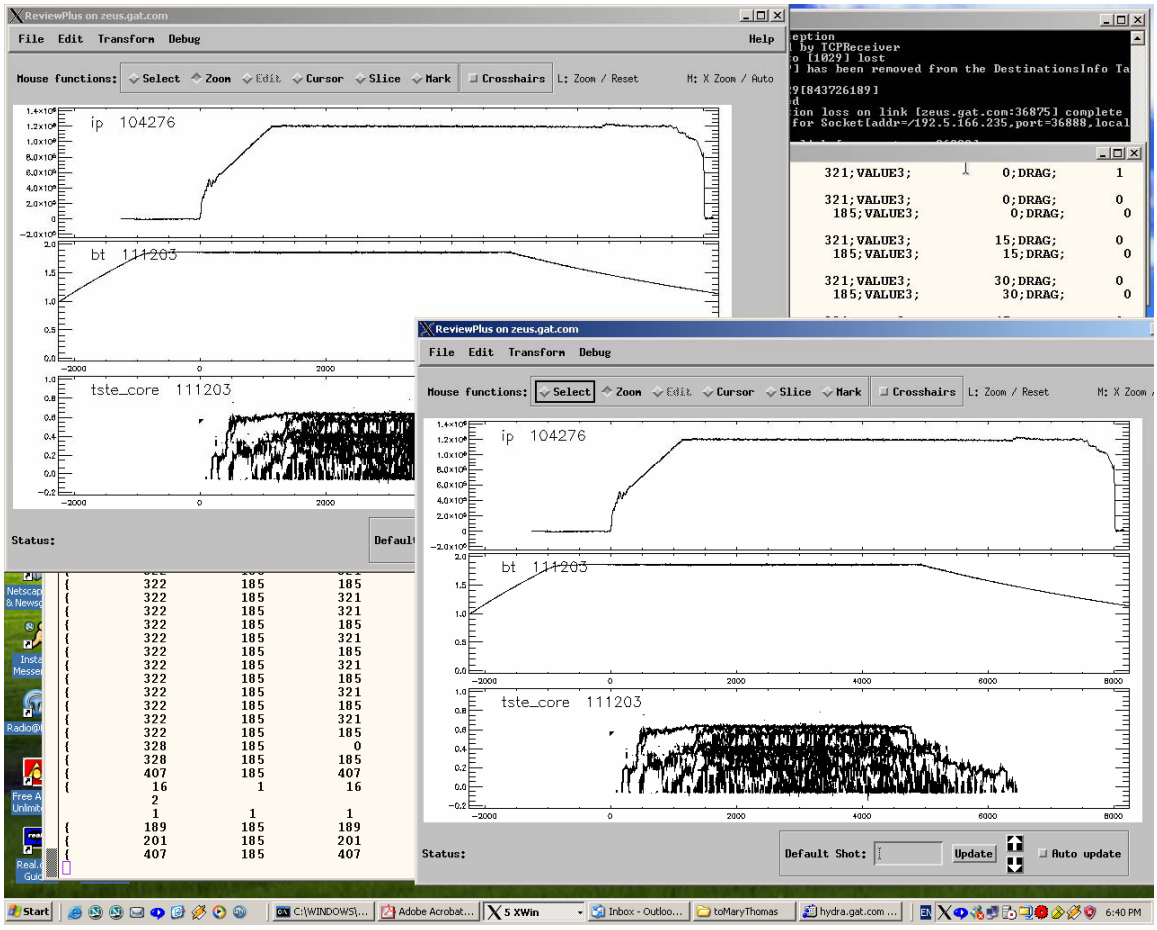



**Figure 5.12 Another screen capture of the environments and the output displays of the Master client and the Participant client of the collaborative ReviewPlus applications running on a single desktop computer.**

In short, the Master client and the Participant client collaborate to share a common DFA and reach a common state of the DFA at any collaboration step. Collaboration between the clients is all about being in a same state of the DFA at each event. The Master client and the Participant client are designed for different purposes, in different architectures, implementing mechanisms, and code shapes; they are divergent. At the same time, they collaborate on messages to present the same output displays at each

event; they have the same logic as to the state transitions on events and get to the same state of the DFA at the end of the process of each event; they are convergent.

## 5.7 Further Implementations of Research Deductions

We have illustrated the Notifying structure and the Polling structure so far in the developing of collaborative IDL applications. They are instantiations and demonstrations of the general principle – Grid-based collaboration on events. The tenet of the dissertation is Grid-based collaboration on events; the collaboration entities in a session collaborate on a Shared Event Model in Peer-to-Peer Grid computing; theoretically the collaboration entities in the session are in essence common Deterministic Finite Automata and they reach a common state at each event.

We have interest and try to find more and potentially better structures and methods for collaborations in general user IDL applications. We have deduced a Dynamic structure and an Embedded structure [Wang+TR0505] from the general principle. Both of them are research deductions and they are potential general and better solutions for collaboration in interactive IDL applications. We give analyses and reasoning with them next. We just need further implementations of the research deductions in the future in order to bring them into reality.

### 5.7.1 The Problem

The normal way to make a specific IDL application collaborative is to work on the programs of that application, delete, change, and add codes. On the Master version of it,

we write codes to catch events and send event messages to the Participant version of it for rendering. Events are caught in the event handlers on the Master; on the Participant the event structures are passed as parameters to the event handlers in the calls. However, in the overall programming, the IDL system routines and libraries are kept untouched; the abstraction is kept in the system. For instance, we have worked this way on the ReviewPlus application package and have made it collaborative. We have developed on this package and programmed mainly in event handlers and places related to event handlers to achieve collaboration, as we have described in section 5.6. We have given more such descriptions of the Implementation of the Collaborative ReviewPlus applications in *Appendix B*.

This solution proves to be workable and efficient, but for every different user IDL application we have to repeat the whole developing process again and use the same or similar technologies and skills on that specific application. It is like reinventing the wheel or building another house using the same blueprint. It just costs unnecessary time and effort; therefore it seems not a general solution for all end user IDL applications to be collaborative. To solve this problem, we have deduced two potential general solutions: the Dynamic Structure and the Embedded Structure [Wang+TR0505].

## 5.7.2 Dynamic Structure

While the Polling structure is feasible and practical for real-life IDL applications to be collaborative, we keep asking a question: is there a general structure that can dynamically generate collaborative IDL codes from any standalone, event-related IDL applications? or, is it possible to develop a general application that generates

collaborative IDL applications (that integrate and use common message brokers like NaradaBrokering), taking a standalone IDL application or applications as its input?

### 5.7.2.1 The Deduction to Generality

We have drawn a deduction of a potential general structure that serves as the initial positive answer to the above questions. We would call this potential general structure the Dynamic Structure [Wang+TR0505]; the potential general application will use this structure in its implementation. This general application takes as input any standalone IDL application(s), does lexical, syntax, and semantic analyses, and proceeds code optimization (potentially) and code generation for collaborative IDL applications. It will output either a Master client or a Participating client, or both, depending on the options the user chose before the execution of the process. It will be a specific compiler because it will go through the steps of the life-cycle of compilation theory [Aho]. Let us name it *CollabIDL_D*. This is illustrated in Figure 5.13.

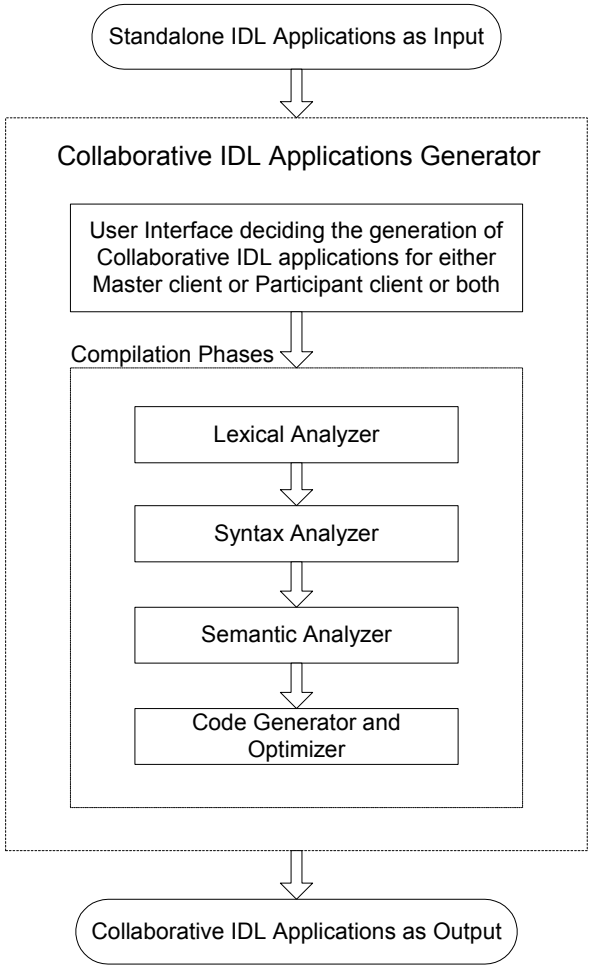



**Figure 5.13 A Dynamic structure for generating collaborative IDL applications.**

The Dynamic Structure is based on the Polling Structure. This structure can be implemented on UNIX platform, using C/C++, lex, and yacc [Levine] in the programming. lex and yacc are tools in the aids of lexical and semantic analyses.

### 5.7.2.2 Infrastructure and Superstructure

Put it another way, the "dynamic structure" is about creating an application (*CollabIDL_D*) that can generate collaborative applications from stand-alone ones. For

instance, the application (*CollabIDL_D*) will take the "ReviewPlus" application package as input, go through the life cycles of the theory of compiler, and automatically generate "collaborative ReviewPlus applications" as the one we have succeeded and demonstrated so far. ReviewPlus is a stand-alone application; we have worked on it and generated the collaborative ReviewPlus applications using the Polling structure. It works. This serves as the base case.

The application *CollabIDL_D* will generate collaborative applications from stand-alone applications using the same strategy, technology, and structure as those used in the base case to generate the Master and Participant collaboration entities; it will simulate the process of making collaborative applications from stand-alone ones (which we have succeeded on the project of collaborative ReviewPlus), and generate collaborative applications *dynamically* by taking stand-alone ones as input. That is why we call the structure of this application the dynamic structure. Our effort and experiment in the base case lay down as the foundation or the infrastructure for the superstructure of the implementation of the application *CollabIDL_D*.

### 5.7.2.3 Things to Do

There are things to do in the implementation of the application *CollabIDL_D*. The main tasks reside in the compilation phases in Figure 5.13. In the phase of lexical analyzer, all the tokens in the input IDL application(s) need to be recognized and put in a symbol table. Examples like *pro* for procedure and *func* for function are common reserved keywords in IDL. We can resort to lex for help in the programming for this. It makes the tokenizing process easier. This step is basically to make a lexer. In the phase of

syntax analyzer, we need to deal with the syntax in which the tokens are linked together in a way that the IDL allows.

Locations that are of interest to us for making collaborative applications are the places where the widgets are defined and the places where the associated event handlers are defined. We are basically working on these locations to generate collaborative applications as we have experienced in the collaborative ReviewPlus project. Let us explain with the following example.

```
x = widget_button(mEdit, value='Set Signals', $
                  event_pro='ReviewPlus_SignalDialog_event')
```

From the code above we know that this button widget has `'Set Signals'` as its value shown on its appearance and is associated with an event procedure named `'ReviewPlus_SignalDialog_event'`. When the button is clicked, the procedure is called by the IDL system. We need to remember this association and put it in an association table for later use in the code generation phase. We can resort to yacc for help in the programming for this. It makes the parsing process easier. This step is basically to make a parser. The output of lex can feed directly into yacc, or the result of the lexer can be used immediately by the parser.

In the phase of semantic analyzer, we need to check for semantic errors. Besides the common checks for many languages such as type checks and flow-of-control checks, in IDL we are also interested in doing uniqueness checks and location checks. The uniqueness check is that we need to check out that the relationship of a widget definition and an event handler is one-to-one correspondence. The location check is that we need to

check out in the source programs that the definitions of event handlers come first and the definitions of the widgets follow after.

We can organize the lexical, syntax, and semantic phases in a compilation pass, and the code optimization and generation phases in the other, if we are to use two compilation passes. Not like other language compilers that generate binary codes (to execute) in the code generation phase, the code generator in question generates collaborative IDL programs from the stand-alone IDL programs. They are all IDL source programs from the point of view of programming languages. This makes the task of code generation easier.

One thing needs to do in this phase is to connect to the NaradaBrokering (NB) message service. The code for doing this need to be put first in the main program of the generated code, and it is static: it is the same for all the generated collaborative codes from their stand-alone ones. This block of code basically connects to the NB through the IDL-Java Bridge and paves the way for message communication. It has nothing to do with the NB source code. The NB just needs to be deployed properly in the environment.

The processes for both the Master and Participant so far have been the same. From this point for the Master client, we need to go check the association table that is for the association of any named widget and its associated event handler, get the information for an association, and go to the event handler. At the beginning of the event handler, we get all the field information of the event structure of the named widget and the name of the event handler, serialize them in a delimited text string (the message for the occurrence of the widget event), and send the message to the NB for broadcasting to the Participants. We have already given a detailed example exactly suitable for this process in section 5.6 -> The Master Client Side -> Event handler. Further, the mechanism for building up the

Master client is the same as that shown in Figure 5.3. In all accounts, it implies that this approach of the implementation of application *CollabIDL_D* on the dynamic structure just follows our research, and it is a deduction.

While for the Participant client, we need to put code in the main loop of the polling structure to get an event message from the head of the linked list; then we parse the message to get all the pieces of the information, convert them into native IDL data types, build the event structure for the widget, get the name of the event handler, and finally call the event handler routine with the built event structure as its parameter. We know all the types of event structures we need to build and all the event handlers we need to call in the programming from the association table, which is for the association of any named widget and its associated event handler. We have already given a detailed example exactly suitable for this process in section 5.6 -> The Participant Client Side. Further, the mechanism for building up the Participant client is the same as that shown in Figure 5.6. In all accounts, it implies again that this approach of the implementation of application *CollabIDL_D* on the dynamic structure just follows our research, and it is a deduction.

## 5.7.3 Embedded Structure

After investigating the functionality of the IDL system and observing the routines from the system library, we realize that there is another way that can achieve the generality of collaboration in IDL applications. We address this discovery in the following parts.

### 5.7.3.1 Potential Breakthroughs

To find a general solution for user desktop IDL applications to be collaborative globally, why not explore the opposite territory? Why don't we keep the user IDL applications "untouched" (or almost "untouched") and accommodate the IDL system routines and libraries to satisfy the end user IDL applications' collaboration needs? More specifically, why can't we modify and develop the IDL system library routines such as "XMANAGER.pro," whose codes are accessible, and add new ones to mimic some system functions whose codes are private? If this way works, we can just put our time and effort here once and for all, and expect the developed package would suit every end user application's need for global collaboration over the Internet.

How could this be possible? To answer this question, let us begin with the IDL system library routine "`XMANAGER.pro`." `XMANAGER` is written in IDL and its code is open in the library of the RSI [RSI] IDL commodity software; it provides the main event loop and management of widgets created in widget programs, and registers the widget programs with it; it then takes control of the event processing until all the widgets have been destroyed in a session. `XMANAGER` is not called much; usually the statement appears once at the end of a widget program as

```
Xmanager, 'ReviewPlusSetup', self.wTLB, /no_block
```

It also orchestrates other system routines, including the function `WIDGET_EVENT`, which returns events for widget hierarchies. It is possible for us to deal with all the event handling and processing in this routine and all other related system routines to achieve

collaboration. In other words, we can develop our codes regarding collaboration issues within these system routines only; thus we can save the necessity of programming in the end user IDL applications and keep them untouched. Specifically, we can modify and develop these IDL system routines, let the `XMANAGER` orchestrate the performance, and make a version for the Master client (`XMANAGER_MASTER.pro`) and a version for the Participant client (`XMANAGER_PARTICIPANT.pro`). The Master part captures, processes, and dispatches events in messages to a common broker, while the Participant part receives messages from the broker, processes the events, and renders the displays. Then we can deploy the two versions to the end user IDL applications supposed to be Master and Participant, and replace "`XMANAGER.pro`" with "`XMANAGER_MASTER.pro`" on the Master and with "`XMANAGER_PARTICIPANT.pro`" on the Participant. Let us name the product of the two versions *CollabIDL_E*. This implies the benefit described next.

### 5.7.3.2 The Benefit

In the deployment and installation, all we have to do in any end user IDL applications is to deploy our routines of the *CollabIDL_E* with the applications and just replace the string "`Xmanager`" with "`Xmanager_Master`" in the end user programs on the Master client side and with "`Xmanager_Participant`" on the Participant client side. Usually in a simple widget program, only one "`Xmanager`" statement is called at the end. Suppose there is a huge IDL application in which there are a hundred

calls for "`Xmanager`", all we have to do (or let some utility software do it ) is to replace 100 strings for both the Master and Participant.

### 5.7.3.3 Embedded Collaboration Object

Usually, such a general solution for achieving collaborative IDL applications from standalone ones should be a standalone software application as the *CollabIDL_D* on dynamic structure we have mentioned previously. In this case of *CollabIDL_E*, however, it is "embedded," the opposite of a standalone one. Just as the name "embedded operating system" nowadays, we can analogously name it as "embedded collaboration object" [Wang+TR0505].

### 5.7.3.4 Building Blocks and Constructions

Now that we have succeeded in building the collaborative IDL ReviewPlus applications from the stand-alone ReviewPlus package, let us extract the factors for success. The success factors in the Master client include:

- Contact the NaradaBrokering message service from the IDL environment through the IDL-Java Bridge.
- Get the event handler and event structure when a widget is triggered, and serialize their information in a message string along with other necessary information.
- Send the message string to the NaradaBrokering message service for broadcasting to the Participant clients.

The success factors in the Participant client include:

- Contact the NaradaBrokering message service from the IDL environment through the IDL-Java Bridge.
- Remove a message string from the head of a linked list (where the NaradaBrokering put new messages at the tail) via the IDL-Java Bridge.
- Parse the message, get the name of the event handler, construct the event structure, and call the event handler with the event structure as parameter in IDL environment to render the same output display as the Master.

The success factors are in the forms of blocks of code in the collaborative ReviewPlus applications, which we have programmed and tested. They are working in the programs. These factors can be applied and reused in the implementation of the *CollabIDL_E* on the embedded structure, because its Master and Participant versions will basically perform the same functionality. These factors should work in the implementation of the embedded structure, because they are basically migrated (with possible modifications) from the end-user collaborative IDL applications to the IDL system library routines and the like. To the core of IDL, both parts are treated as extensions. The success factors are the building blocks for the constructions of the *CollabIDL_E*.

### 5.7.3.5 Things to Do

There are things to do in the implementation of the application *CollabIDL_E*. The main tasks reside in the programming of versions of `XMANAGER`, with `XMANAGER_MASTER.pro` for the Master client and

`XMANAGER_PARTICIPANT.pro` for the Participant client. At the beginning of the versions for both the Master and the Participant, we need to connect to the NaradaBrokering (NB) message service. The code for doing this is static. This block of code basically connects to the NB through the IDL-Java Bridge and paves the way for message communication. It has nothing to do with the NB source code. The NB just needs to be deployed properly in the environment.

For the Master client, in the main event loop of the `XMANAGER`, we need to watch for events; if an event occurs, we need to get all the field information of the event structure and the name of the event handler, serialize them in a delimited text string message, and send the message to the NB for broadcasting to the Participants. We have already given a detailed example suitable for this process in section 5.6 -> The Master Client Side. The code demonstrations and explanations there are the success factors; they can be reused in this process. It implies that this approach of the implementation of application *CollabIDL_E* on the embedded structure just follows our research, and it is a deduction.

While for the Participant client, we need to put code in the main event loop of the `XMANAGER` to poll for available event messages in the interface of NB; we get an event message if available from the head of the linked list; then we parse the message to get all the pieces of the information, convert them into native IDL data types, build the event structure for the widget, get the name of the event handler, and finally call the event handler routine with the built event structure as parameter. We have already given a detailed example suitable for this process in section 5.6 -> The Participant Client Side. The code demonstrations and explanations there are the success factors, and they can be

reused in this process. It implies that this approach of the implementation of application *CollabIDL_E* on the embedded structure just follows our research, and it is a deduction.

`XMANAGER` dispatches events to and calls the IDL system routine *WIDGET_EVENT*, where the events are handled and the event handlers are called with the events as parameters. In the development, we need to know the information about the swallowed events and the executed event handlers from *WIDGET_EVENT*, but its source code is not available in the system library, even though `XMANAGER.pro` is. Therefore, we need to add our versions for it in the development to mimic its behaviors and at the same time to satisfy the collaboration needs.

# Chapter 6

# Comparisons

In this chapter, we will make comparisons between the Collaborative PowerPoint applications, Collaborative Impress applications, and the Collaborative ReviewPlus IDL applications [Wang+SCI]. We will compare them on technologies and languages used, implementing structures, and event structures to see their differences; we will compare them on the Shared Event Model, the Grid-based Collaboration Model, and their implications to see their similarities.

Theoretically, these three cases are variant deductions from the general principle – Grid-based collaboration on events, and the collaborative applications themselves are the implementations of the deductions. These implementations are the prototypes for the Shared Event Model in Grid-based Collaboration, and the comparisons will eventually lead us to the general conclusion and the motivation for system enhancements, which will be addressed in *Chapter 8*.

## 6.1 Technologies and Languages Used

As in the Microsoft Office suite, the Collaborative PowerPoint applications use the Distributed Component Object Model (DCOM) technology [Eddon]. The technology includes Dispatch interface, Connectable Object, Connection Point, Outgoing interface, Event Sink, Type library, Wrapper class, and Automation. To communicate with the NaradaBrokering Message Service for message transmission, the Collaborative PowerPoint applications use the Java Native Interface (JNI) technology to communicate the information between two environments, because the NaradaBrokering system was written in Java language, and the Collaborative PowerPoint applications in C++. The communication is a two-way conduit, both from C++ to Java and from Java to C++.

In Open Office/Star Office and the Collaborative Impress applications, the Universal Network Object (UNO) technology [UNO] plays an important role. With it, remote communication bridge is set up between the client and the office server; component objects are instantiated in the client and server processes and communicate with each other to perform tasks across the process boundaries. The Frame-Controller-Model (FCM) paradigm plays another important role in the behaviors of the applications. The Collaborative Impress applications are in Java language, same as NaradaBrokering; so the communications between them are natural without having to overcome language boundaries.

In the Collaborative ReviewPlus applications and other IDL applications, the Graphical User Interface (GUI) programming technology and GUI Components (Widgets) take their places. The object-oriented programming is in every non-trivial application. Technologies such as the IDL-Java bridge and the Callable IDL are used. The

Collaborative ReviewPlus applications are in IDL language, and the NaradaBrokering Message Service is in Java; so the IDL-Java bridge technology is used to pave the way for message communications across the boundary between the two languages.

## 6.2 Implementing Structures

In chapters 3, 4, and 5, we have described in detail about the Collaborative PowerPoint applications, Collaborative Impress applications, and the Collaborative ReviewPlus applications. They are in themselves comparisons. We restate their implementing structures in this section to highlight the different strategies and methodologies used, which are required to tackle different environments, languages, and technologies issues in order to achieve high performance collaboration.

### 6.2.1 The Collaborative PowerPoint Applications

Microsoft Office suite is proprietary and is component object oriented. The events there are in the forms of named strings (e.g., "WindowActivate") or hexadecimal dispatch identifiers (e.g., 0x614). The office suite exposes its functionality through the standard IDispatch interface, also known as the Automation utility [Eddon]. The IDispatch interface's primary purpose is to expose the (otherwise solely user-driven) applications' functionality for other applications to use programmatically. Each exposed method or property has an associated DISPID. The events that we are concerned are special ones, which can be fired by the source objects (the connection points) and can be caught by the event handlers (in the sink).

Next, we describe the way to catch the events fired in the applications of Microsoft Office suite and the specialties in PowerPoint. Microsoft designed the Connectable Object technology that enables client and server objects to communicate with each other in both directions. The Connection Point objects are managed by the Connectable Object, where the outgoing interfaces are defined; the implementations of the outgoing interfaces are in the client event sinks. Each Connection Point is associated with only one outgoing interface. This is where the events occur and is therefore called the source interface for the client sink interface. The sink is where the event handlers are implemented.

The Client first gets a reference to the Server's *IConnectionPointContainer* interface. It then uses this reference to call method *FindConnectionPoint()* to get the connection point for the outgoing interface, where the events of interests reside. Finally, the client sets up an advisory connection/relationship with the server by calling the method *Advise()* with a pointer to its sink's *IUnknown* interface. Now the server object has a pointer to the outgoing interface of its client's sink and fires back events whenever something interesting happens in its process. The event handlers of the sink catch the events and process. This is elaborated in Figure 6.1.

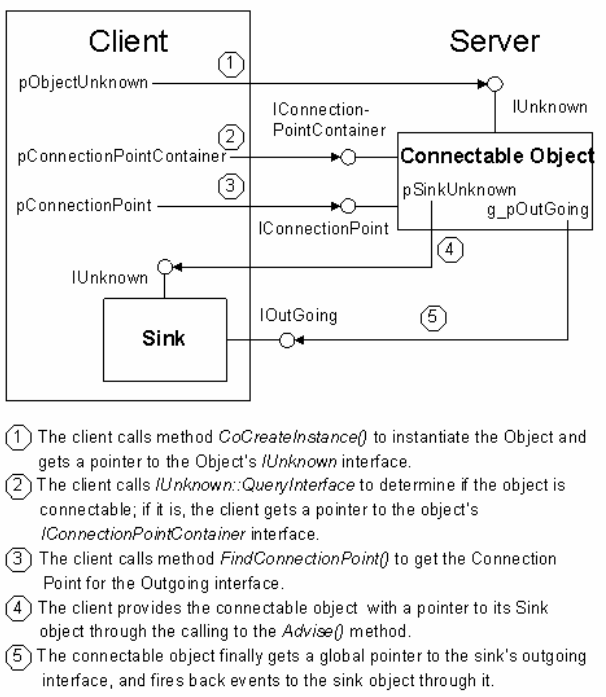



**Figure 6.1 The steps to set up an advisory connection between the client and the server so that the server's connectable object can obtain a pointer to its client's sink and fire back events.**

The Master client gets the events fired at its PowerPoint server; then it sends the event messages through the message broker to the Participants. The Participant client controls and calls the functions of its PowerPoint server under the instructions of the received event messages, generating the same displays as the Master client. The Participant client uses the Automation technology in the process. This is illustrated in Figure 6.2 (it is the same as Figure 3.2; we list it here again for convenience).

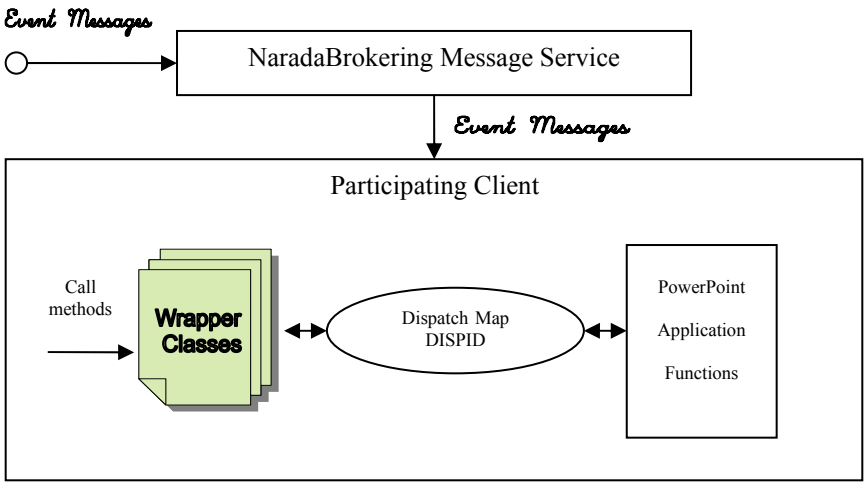



**Figure 6.2 The event messages invoke the methods of the wrapper class; the methods are then mapped to the functions of the PowerPoint application through the Dispatch Map/DISPID; the functions are executed and the result/status codes are returned.**

## 6.2.2 The Collaborative Impress Applications

The Master client connects to the OpenOffice/Star Office that serves as a server, listens to events fired there during a session, and sends the event messages to the NaradaBrokering Message Service for broadcasting to the Participating clients. The client (Master/Participant) communicates information with the office server through the TCP/IP socket. The office server listens to client TCP/IP connections using a connection URL as the parameter, which includes the hostname/IP address, the port number, and the protocol. In order to do their jobs and to work with the data located on their Office servers, both the Master client and the Participating clients need to establish a remote communication bridge with their respective Office Servers and get their servers' service managers.

After it has set up the remote bridge, the Master client takes control of the programming features via the Frame-Controller-Model (FCM) paradigm [FCM]. In FCM, the model is the document object; it has document data and also methods that access the data. The methods can change the data directly without having to use a controller object. The controller is the screen interaction with the model; it observes the changes made to the model and manages the presentation of the document. The frame is the controller-window linkage; it contains the controller for a model, and it has knowledge about the window but not the functionality of the window. That functionality is encapsulated in the underlying windows system – whatever platform it is. This decouples the specific windows implementation from the frame; thus this makes it possible to use a single frame implementation for different windows in the OpenOffice. The specific windows work with the frame to make the screen presentation.

The Master client registers listeners at the remote bridge to listen to events fired at the Office server, as in Figure 6.3 (same as Figure 4.5; we list it here again for convenience). One of the registered listeners is the "Property Change Listener," which listens to property change events of an object. The client makes the listener listen to changes of the "Current Page" of the presentation file object. Whenever a presentation slide changes in the Impress server, the listener catches the event and notifies the event handler to do further processing. The event handler gets the slide number using the method *getPropertyValue("Number")* of the *XPropertySet* interface. All the event messages are sent to the NaradaBrokering Message Service.

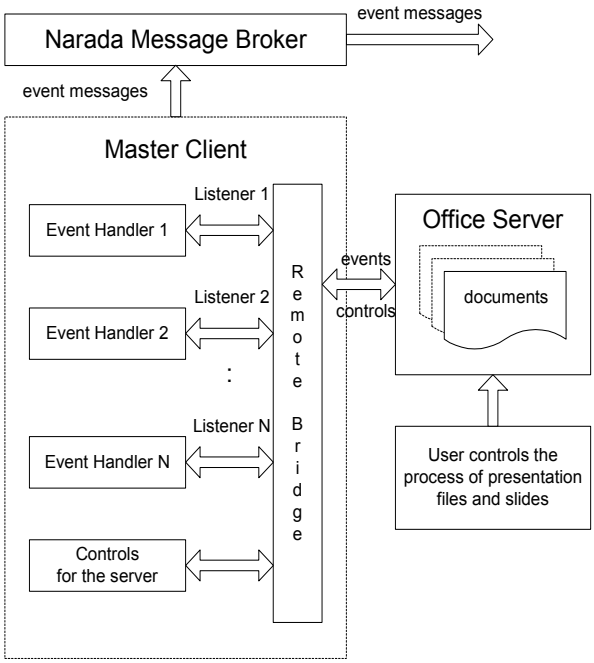



**Figure 6.3 The function structure of the Master client of the collaborative Impress applications.**

When the NaradaBrokering Message Service receives the event messages from the Master client, it notifies the Participating clients and broadcasts the messages to them, as in Figure 6.4 (same as Figure 4.6; we list it here again for convenience).

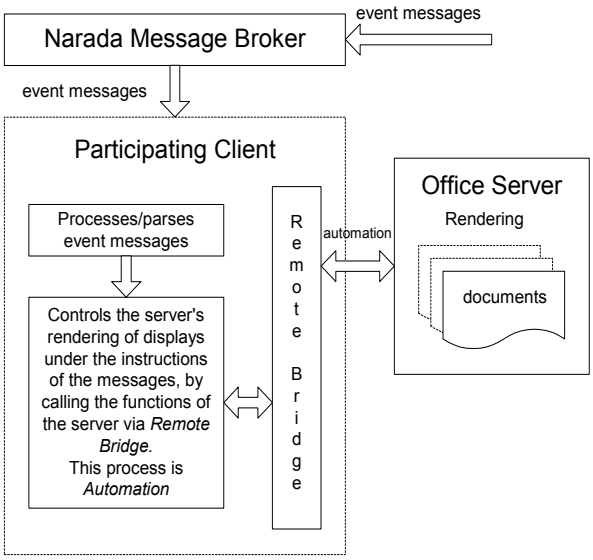



**Figure 6.4 The function structure of the Participating client of the collaborative Impress applications.**

Each Participating client connects to, controls, and makes use of the Office server. It first creates a remote bridge, gets the server's component context and service manager; then it gets control of the server's Frame, Controller, and Model, and it leverages the FCM paradigm to use the server's functionality to control the rendering process. When the client receives a message from the Narada message broker, it parses it and gets the different information parts, such as the event type, the properties, or a URL of a presentation file. It then calls the functions of the server, such as *loadComponentFromURL()*, to open/switch to a presentation; it calls the method *getDrawPages()* of the *XDrawPagesSupplier* interface, the method *getByIndex(index)* of the *XDrawPages* interface, and the method *select( xDrawPage )* of the *XSelectionSupplier* interface, to navigate to a specific slide of an opened presentation file. The event type is the key to call different processing functions, and the associated

properties are used in the functions to generate the correct presentation results. This process is automation; the functions of the Office server are called programmatically under the instructions of the event messages.

### 6.2.3 The Collaborative ReviewPlus Applications

Basically, the Master client of the collaborative ReviewPlus applications has a GUI building and managing part and an event handling part.

- It makes use of the IDL-Java Bridge, calls methods in a Java program to connect to the NaradaBrokering message broker.
- It captures events, gets event messages in the event handlers whenever a user triggers events in the GUI, such as button clicking, and sends the messages to the NaradaBrokering message broker for broadcasting to Participants.

This process is elaborated in Figure 6.5 (same as Figure 5.3; we list it here again for convenience).

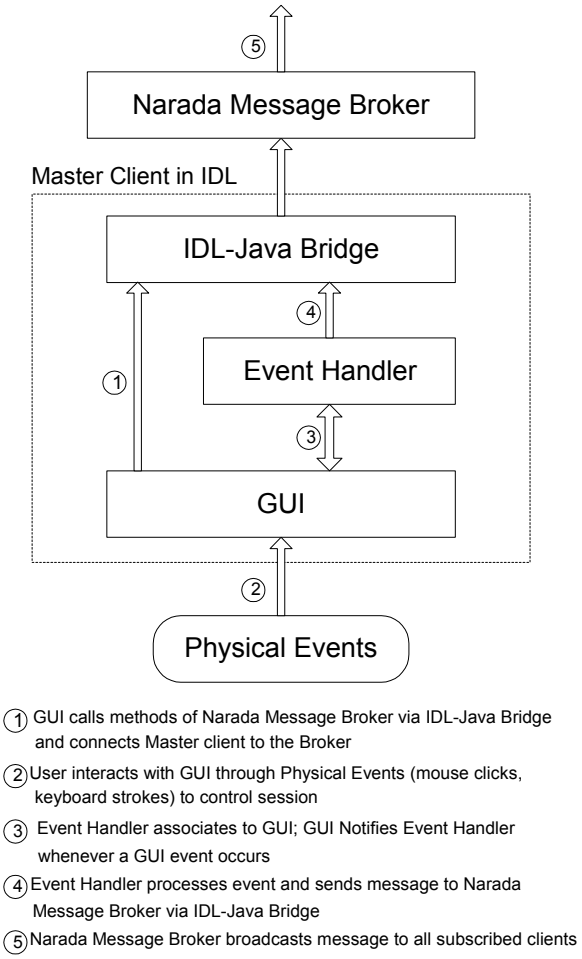
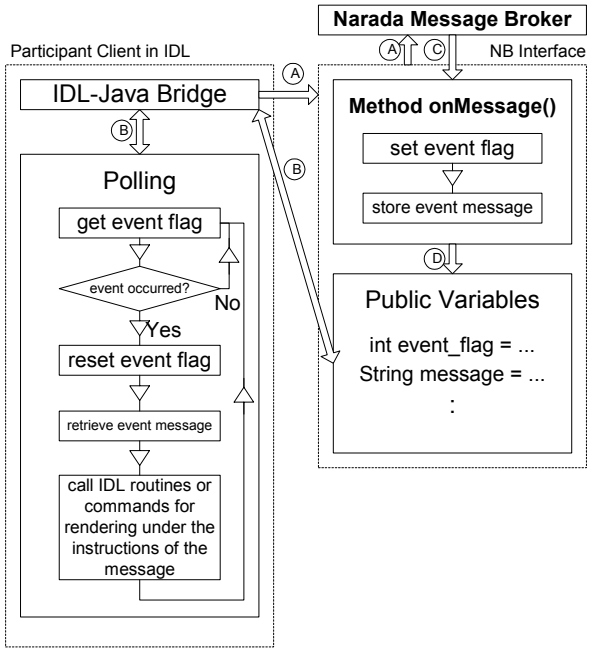



**Figure 6.5 The mechanism of the Master client of the collaborative ReviewPlus applications.**

The Participant client is implemented on a Polling Structure. It connects to the NaradaBrokering by calling the methods of the broker's interface via the IDL-Java Bridge. In a Java class, which is an interface to the NaradaBrokering, we add public global variables for the event change flag and the event message; we make the notification method *onMessage()* in the Java class update the public variables whenever the broker broadcasts event messages to the clients. The update includes increasing the

event flag and storing the event message in the variables (to the tail of a linked list). The Participating client instantiates the Java class and uses the instance. The Participating client code now has an instance of the Java class; it is constantly testing, or *Polling*, the instance variable – the event flag. If it finds that the flag is positive (indicating there is at least one event message left in the linked list), it decreases the event flag and retrieves an event message from the head of the linked list. It then follows the instructions of the message to execute the different parts of the IDL programs to generate the same displays as the Master client, as shown in Figure 6.6 (same as Figure 5.6; we list it here again for convenience).




Ⓐ Participant client connects to NB by calling methods of NB interface via IDL-Java Bridge

Ⓑ The client accesses the public variables of NB interface by calling the Bridge's methods getProperty() and setProperty()

Ⓒ The Broker invokes method onMessage() of NB interface when it has event message to broadcast

Ⓓ Method onMessage() then accesses the public variables of NB interface by setting event flag and storing message

**Figure 6.6 The mechanism of the Participant client of the collaborative ReviewPlus applications.**

## 6.2.4 Architectural Differences and Implications

On the Participant client side of the collaborative PowerPoint applications or the collaborative Impress applications, the Narada message broker notifies the client whenever it has an event message to broadcast through its interface method *onMessage()*. Inside this method we add code to parse and analyze the message; then the code calls functions under the instructions of the message to collaborate with the Master client. The Participant client in question is using a *Notifying structure*, similar to our initial prototype of a simple collaborative IDL application, as shown in Figure 5.5.

On the Participant client side of the collaborative ReviewPlus applications, a *Polling structure* is used, as we have described. We prefer this structure, because of the following implications.

1. It can avoid the potential data loss problems by using a linked list for storing the messages; therefore it can have better performance.
2. It can reduce the troubles of inter-language programming or language boundaries, as we have described and evaluated in *chapter 5* about the Notifying structure and the Polling structure; thus it can make the overall system structure simpler and clearer.
3. It can therefore save effort and time in programming for collaboration.

## 6.3 Event Structures

The event structures in the Microsoft Office suite are hexadecimal dispatch identifiers (DISPID) or meaningful named strings; each string is associated with one DISPID. Within the applications of the Office suite those DISPIDs are actually used to perform functions. It looks neat and efficient. In the OpenOffice/Star Office, the event structures are event types or short strings for methods or properties. In those Office systems, the events are mainly for interactive actions or transactions, and they are short strings and simple. This is an advantage in Grid-based collaboration as in distance education, e-Learning, and online conference. It poses little network traffic in communication between the involved clients.

The event structures in IDL are more complicated; they have the form of a structure containing hierarchical information; they are data-intensive in favor of science and engineering data analysis and computation. However, they are still short text strings in transmitting, at most several hundred characters long, as shown in table 6.3.

### 6.3.1 The Collaborative PowerPoint Applications

The events fired back and caught in the sink are in the form of hexadecimal DISPIDs. By the aid of the OLE View, we can map them to their corresponding meaningful named strings in the type library of an application; thus we know what functions we need to call later in the automation programs. In Excel, Word, etc. things go like that, but not in PowerPoint. If we open the object library of the PowerPoint (MSPPT.OLB) using the OLE View and expand the “Application” coclass, we can see there is a Dispatch event

interface called “EApplication”, which is the connection point for the event source and is associated with the outgoing interface of the event sink. Events in this interface include actions of the PowerPoint working environment and transactions of the presentation files and slides.

In this interface, however, we can not find the DISPID for each named string (which is meaningful and self-descriptive) for an event; however in programs we can only catch any event in the form of a DISPID. With the hexadecimal codes like this, you can not know their meanings and can not figure out which is which. We have done logical analysis according to the input/output of the presentation processes, and finally mapped each of the codes to its corresponding meaningful string name in the event interface of PowerPoint. This is shown in Table 6.1.

**Table 6.1 Hexadecimal codes and their corresponding text named strings for the events in the “EApplication” dispatch interface of PowerPoint**

| Hexadecimal Code | Text String |
|---|---|
| 7d1 | WindowSelectionChange |
| 7d2 | WindowBeforeRightClick |
| 7d3 | WindowBeforeDoubleClick |
| 7d4 | PresentationClose |
| 7d5 | PresentationSave |
| 7d6 | PresentationOpen |
| 7d7 | NewPresentation |
| 7d8 | PresentationNewSlide |
| 7d9 | WindowActivate |
| 7da | WindowDeactivate |
| 7db | SlideShowBegin |
| 7dc | SlideShowNextBuild |
| 7dd | SlideShowNextSlide |
| 7de | SlideShowEnd |
| 7df | PresentationPrint |
| 7e0 | SlideSelectionChanged |
| 7e1 | ColorSchemeChanged |
| 7e2 | PresentationBeforeSave |
| 7e3 | SlideShowNextClick |

## 6.3.2 The Collaborative Impress Applications

Each event listener listens to a specific event type fired at the Office Server; the event handler catches the event and translates it into a corresponding string for transmitting. These event types are for actions as well as properties. The event structures are thus decided by the types of the events; they are in the forms of single strings (short messages) in transmitting to and controlling of the Participants. We list some of the event listener interfaces and their corresponding event types that we have tried in our programs, in table 6.2.

**Table 6.2 Some event listener interfaces and their corresponding event types**

| Event listener interface | Event type |
|---|---|
| XPropertyChangeListener | PropertyChangeEvent |
| XSelectionChangeListener | EventObject |
| XFrameActionListener | FrameActionEvent |
| XKeyListener | KeyEvent |
| XMouseListener | MouseEvent |
| XMenuListener | MenuEvent |
| XWindowListener | WindowEvent |
| XContentEventListener | ContentEvent |
| XFocusListener | FocusEvent |
| XModeChangeListener | ModeChangeEvent |
| XContainerListener | ContainerEvent |

## 6.3.3 The Collaborative ReviewPlus Applications

A part of the event structures used in the IDL widget programming (as in ReviewPlus) is listed in table 6.3. They correspond to a variety of the primitive widgets in IDL, such as the Button, the Slider, the Text field, and the Draw area. The event structure for each widget is different; each one contains state information specific to that widget, e.g., flags and values. However, there are three common items in all the event

structures; they are ID, TOP, and HANDLER. They are long integers and the first three items in the structures.

4. ID is the widget ID number of the widget that generates the event.
5. TOP is the widget ID number of the top-level base that contains the widget that generates the event.
6. HANDLER is the widget ID number of the widget that is associated with an event handler.

For instance, the event structure for the BASE widget is

{WIDGET_BASE, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L}, where

X is the width of the base, and Y is the height.

**Table 6.3 A part of the event structures used in the widget programming of the Interactive Data Language**

| |
|---|
| {WIDGET_BASE, ID:0L, TOP:0L, HANDLER:0L, X:0L, Y:0L} |
| {WIDGET_BUTTON, ID:0L, TOP:0L, HANDLER:0L, SELECT:0} |
| {WIDGET_DRAW, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, X:0L, Y:0L, PRESS:0B, RELEASE:0B, CLICKS:0, MODIFIERS:0L, CH:0, KEY:0L} |
| {WIDGET_LIST, ID:0L, TOP:0L, HANDLER:0L, INDEX:0L, CLICKS:0L} |
| {WIDGET_SLIDER, ID:0L, TOP:0L, HANDLER:0L, VALUE:0L, DRAG:0} |
| {WIDGET_TABLE_CH, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, OFFSET:0L, CH:0B, X:0L, Y:0L} |
| {WIDGET_TABLE_CELL_SEL, ID:0L, TOP:0L, HANDLER:0L, TYPE:4, SEL_LEFT:0L, SEL_TOP:0L, SEL_RIGHT:0L, SEL_BOTTOM:0L} |
| {WIDGET_TEXT_STR, ID:0L, TOP:0L, HANDLER:0L, TYPE:1, OFFSET:0L, STR:''} |
| {WIDGET_TEXT_SEL, ID:0L, TOP:0L, HANDLER:0L, |

TYPE:3, OFFSET:0L, LENGTH:0L}

The Master client captures an event, gets the event structure, and serializes it in a message string, along with other information such as the name of the event handler; then the Master client sends the message out. The Participant client de-serializes the received message (from the public variables in the polling structure), rebuilds the event structure (in native IDL types), and locates the event handler; it then calls the event handler with the event structure as the parameter like this:

*ReviewPlus_SignalDialog_event, {WIDGET_BUTTON, ID:15, TOP:1, HANDLER:15, SELECT:1}*

### 6.3.4 Differences in Event Processing and Implications

There are differences in the event processing of the collaborative PowerPoint, collaborative Impress, and collaborative ReviewPlus applications. We show them in the following aspects.

**Straightforwardness of Event Structures**: The event structures in PowerPoint are hexadecimal dispatch identifiers. With the hexadecimal codes like this, you can not know their meanings and can not figure out which is which. As we have described earlier, we have done logical analysis according to the input/output of presentation processes, and finally mapped each of the codes to its corresponding meaningful named string in the event interface of the PowerPoint, as shown in Table 6.1. In Impress, the event structures are event types/short strings for methods or properties. The event types are associated with the event listeners. The event structures in ReviewPlus have the form of a structure

containing hierarchical information. Each widget has a unique event structure associated with it; the event structure is defined and described clearly in IDL.

**Certainty of the Method(s) Called**: In PowerPoint or Impress, we are not exactly sure what method(s) was (were) called in its process when an event happened. We can only guess the approximation according to the actions (e.g., change a slide), choose method(s) from the interfaces in the Master client, send the message to the Participant client, and let the Participant execute the same method(s). In ReviewPlus, the IDL associates each widget with an event handler; when the widget is triggered, the IDL system calls the event handler automatically; therefore we clearly know that the method called is exactly the event handler. We get the name of the event handler in the Master client and let the Participant client execute the same thing.

**Easiness in Getting Properties**: The property is the parameter for a method, if it has one. In the collaborative PowerPoint applications, we have to write code to get it intentionally and programmatically in the Master client. For example, if we get the event "PresentationNewSlide," we know that we have to get the property for the slide number in our programs. In Impress, the occurrence of a property is also treated as an event (PropertyChangeEvent), but is treated as an independent event, separately from the event of its associated method. We get the property separately in the related event handler, which we added for the event listener (PropertyChangeListener). Later we serialize the information of the property in the event message, along with the name of the method. In ReviewPlus, thanks to the association between the widget and its event handler and the mechanism of the IDL widget programming, we can get the property directly in the event handler, and the property happens to be the widget's event structure, as those in table 6.3.

In fact, we conveniently get the name of the method (event handler) and the property (event structure) in the same place and at the same time.

**Reliability of the Results**: For the collaborative PowerPoint or Impress applications, the methods executed in the Participant client are approximations of those executed in the Master client. The approximations depend on our understandings of the collaborative actions and the choices of the methods. Even though in our experiments and demonstrations the output displays between the Master client and the Participant client are exactly the same for the events, theoretically there could be some discrepancies somewhere. In the collaborative ReviewPlus, however, there is no need to guess because we get the method plainly in the event handler for any event. The methods called in both the Master client and Participant clients are the same and so are their properties – the event structures. Theoretically there are no discrepancies.

**Convenience in Programming**: Based on the above aspects alone, we can see that it is most convenient to make collaborative applications out of the ReviewPlus or any other IDL widget programs; it is least convenient to make collaborative applications out of the PowerPoint or any other proprietary Microsoft office applications.

**Extent of Collaboration**: The PowerPoint application is a proprietary product. The extent of collaboration in the collaborative applications is really limited to the event source (IDispatch interfaces) it supplied. The Impress is an application of the OpenOffice. Event though it is open source, we are not focusing on developing its base source code. So the extent of collaboration in the collaborative applications is limited to the event listeners and the event types it supplied. The ReviewPlus is an application developed in IDL; specially, its interactive interfaces are programmed using the widget programming

of IDL. Each widget (or event) of the interfaces is associated with an event handler, if it is well-defined and programmed. We got the chance to work on the source of the ReviewPlus. So the extent of collaboration is as the extent of the existence and soundness of the defined widgets (or events) of the ReviewPlus.

Based on the above aspects, we get the following implications. IDL interactive applications such as the ReviewPlus are the most preferable in programming for collaboration, mostly because of the good mechanisms of the widget programming and the event structures. Proprietary applications such as the PowerPoint are the most confined and limited ones in programming for collaboration, and they are the most troublesome and difficult ones, too.

## 6.4 Shared Event Model

As we have seen, each type of the collaborative PowerPoint applications, the collaborative Impress applications, and the collaborative ReviewPlus applications has a Master client peer and a Participant client peer; the peers of each type collaborate on event messages, even though the forms and sizes of the event messages are different in one type from another. We have used a Shared Event Model in the communication between peers. In this model, small text event messages are transmitted via the Grids of common message brokers and are used to coordinate the operations between the peers, so that the peers cooperate concurrently and share the same screen outputs simultaneously.

During a session, the Master client captures events in its process, deals with them, and sends the event messages to the Participant for generating the displays in the Participant's process, so that both of them can share the screen displays simultaneously [Fox+CISE+March2004]. There can be multiple Participants working with the Master client concurrently and independently. We have used the Narada message broker deployed in Grid for the message communication. The Master client captures the event, gets the event structure, and packages the information into a delimited string, as in {widget_base|id 0|top 0|handler 0|x 0|y 0}, with possibly other information such as session, source, and destination, and sends the result message string to the Narada message broker for broadcasting to the Participants. This is a serialization process. The Participant client parses the received message string, gets all the parts of the delimited information, and rebuilds the event structure by converting the sub-string sections like "id 0" to their corresponding native types of the event structure. This is a de-serialization process. The constructed event structure is then used (as a parameter) in its event handler or routine, which is invoked by the Participant client programs to generate the same event result as that happened on the Master client.

# 6.5 Grid-based Collaboration Model

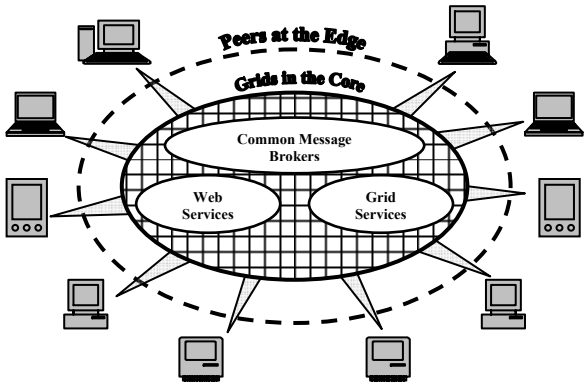



**Figure 6.7 A grid-based collaboration model.**

We have used a Grid-based Collaboration Model in the design and development of all the collaborative applications, as shown in Figure 6.7. There are two categories of systems in this model: the Grid system and the Peer-to-Peer systems. The Grid system [Foster+Kesselman, Berman+Fox+Hey, Globus] is the basis; it largely comprises stable, formal, and efficient high-functionality services such as the Web Services, the Grid Services, and the Common Message Brokers, which are deployed in the Grids on structured, well-organized, and powerful supercomputers. They are in the core of the model. The Peer-to-Peer system offers user-friendly, convenient, intuitive, and easy accessible applications and services such as the popular commodity software used daily and everywhere. They are installed on a variety of personal devices, such as desktops, laptops, PDA's, and smart phones. They are at the edge of the model.

The infrastructure of the Networks and the Internet ties up and correlates the two categories. It enables the Peer-to-Peer Grid computing to be a trend. The Peer-to-Peer Grid computing harnesses the advantages of the two categories so that they complement

each other; it also brings new opportunities and challenges to the computing in all. The Grid system offers robust, structured, and security services that scale well in pre-existing hierarchically arranged enterprises or organizations; it is largely asynchronous and allows seamless access to the supercomputers and their datasets. The Peer-to-Peer system is more convenient and efficient for the low-end clients to advertise and access the files on the communal computers; it is more intuitive, unstructured, and largely synchronous.

In our design and development of each type of the collaborative applications, we have demonstrated the Peer-to-Peer Grid computing concept. We have deployed the Narada Message Broker in the Grid and used it for message communication between the Master client and Participant clients of the type; we have deployed the Master client and Participant clients as the Peers at the edge and made them collaborate on events.

## 6.6 Implications

We have described the Grid-based Collaboration Model in the Figure 6.7. Let us zoom in and exemplify it with our collaborative applications and see the role of the Shared Event Model in collaboration. In the collaborative PowerPoint, Impress, and ReviewPlus applications, the Masters and the Participants connect to each other and communicate event messages with each other through a common message broker, which serves in Grid. Each client takes advantage of a well-known paradigm in updating, controlling, and displaying. For the OpenOffice/Star Office, it is the Frame-Controller-Model (FCM); for the others it is the Model-View-Controller (MVC) [Gamma].

We can abstract the collaboration to be collaboration between entities of paradigms linked by message; the Master entity gets the message through its paradigm, especially the Model (where the data reside) and the Controller; the Participant entity processes the received message to generate the results through its paradigm, including modifying the data in its Model and coordinating the Controller. Both the Master and the Participant entities embody the power and elegance of the paradigms. This is illustrated in Figure 6.8.

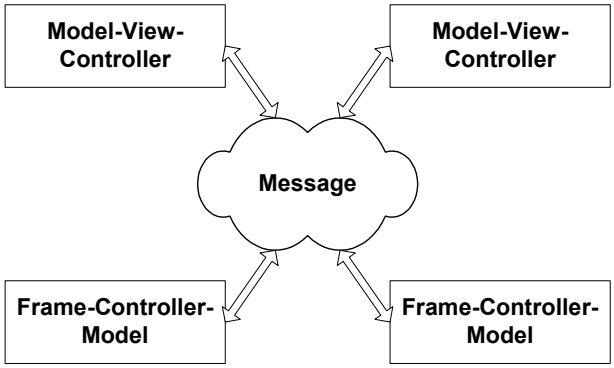



**Figure 6.8 Entities of paradigms linked by message in collaboration.**

This abstraction and the Grid-based collaboration model imply the following scenarios.

**Scenario 1**

We have addressed the collaboration of the entities of the MVC and FCM paradigms based on message. More or new entities of paradigms can be added to this picture. With message, not only the entities of paradigms of the same type collaborate with each other, but also can those of different types, e.g., the MVC with the FCM. This decouples the

types of the entities of paradigms and brings more freedom in collaboration; this also brings diversity, and diversity is important in enabling and facilitating collaboration.

**Scenario 2**

The traditional three-tier computing includes the tiers of client, server, and database. What if bring this computing model into Grid-base collaboration, using message in communication and controlling? The server and database can be deployed in Grids, taking advantage of the computing power and security of the Grid infrastructure; common message brokers can serve for the solid underlying message communication; the shared event model and message play important roles. Thus, the client and the server can be developed in different languages and run on diverse platforms; the database can be in multiple database environments as well; the message glues them together and coordinates, controls, and invokes the functions in the three tiers. We think it would result positively in performance, collaboration, and diversity.

**Scenario 3**

Let us further deduce Scenario 2. Suppose the client is in active mode, and the server is in passive mode; in other words, clients in multiple languages and platforms take control of the process of a session by sending out requests in message, and the server supplies functionality services on receiving the requests and sends the results (in message) back. This case naturally evolves into the Web Service and complies with the Web Service Architecture, with the message marked with XML tags and Web Service techniques used (such as SOAP, WSDL, and UDDI). For performance and quality of

service considerations, if the Web Service takes advantage of the Grid and common message brokers, wouldn't it be better?

Now, let us suppose the server is in active mode, and the client is in passive mode; that is, the server generates and broadcasts the message, and the client interprets and executes the received message. This case fits into the situations such as distance education, e-Learning, and online conference; it eventually becomes the structures of our collaborative applications described in this dissertation, in which the Master client is the source of the message and the Participant client is the destination. Once again, the Master and Participant entities could be in different languages, platforms, and paradigms.

## 6.7 Implementations of the Deductions from the General Principle

So far we have made comparisons between the Collaborative PowerPoint applications, the Collaborative Impress applications, and the Collaborative ReviewPlus applications. We have seen that they are different in technologies, languages used, implementing structures, and event structures. Theoretically however, they are variant deductions from the general principle, and the collaborative applications themselves are the implementations of the deductions. The general principle, also the tenet of the dissertation, is Grid-based collaboration on events; the collaboration entities in a session collaborate on the Shared Event Model in the Peer-to-Peer Grid computing; theoretically the collaboration entities in the session are in essence common Deterministic Finite Automata (DFA), and they reach a common state of the DFA at each event.

## 6.8 From Collaboration Services to Web Services

The collaborative PowerPoint applications, the collaborative Impress applications, and the collaborative ReviewPlus applications are actually Grid-based collaboration services. If they are promoted to be web services, it will enhance their usefulness, availability, and universal accessibility, among other things. We will discuss this issue next in *chapter 7* about the need, the idea, the structure scenarios, the technical problems and solutions, and the initial effort toward the implementation.

# Chapter 7

# Thin Client Collaboration Web Services

In this chapter we will introduce some collaboration applications and the needs in changing them to Web Services. We will propose and describe the idea of Thin Client Collaboration Web Services and explore some potential scenarios, in which it shows its merit and the freedom resulted in collaboration [Wang+ICIW'06]. Such a Web Service has two sets of ports: the User-facing Input/Output ports and the Resource-facing Input/Output ports. The user-facing I/O contacts a Web Service viewer, and the resource-facing I/O contacts a collaboration application. Hence, the role of the Web Service is to transcode in both directions between the two sets of ports with respect to displays and events, so that the user accesses the Web Service viewer as if the collaboration application itself. The Services are linked by messages with defined interfaces.

This chapter is notional, and substantial future work needs to be done. The architecture is consistent with the Web Service; it will associate with the Web Service features such as the Simple Object Access Protocol (SOAP) messages, the Universal

Resource Identifier (URI) endpoint, and the Web Service Description Language (WSDL) file. We will use our three collaborative applications as resources, and our SVG related projects as the indication of our initial effort toward the implementation of a general Thin Client Collaboration Web Service.

# 7.1 Introduction

There has been a lot of software developed for collaboration over the Internet. Application areas include online conferencing, distance education, e-Science, and e-Business. Applications already in industry, such as WebEx [WebEx], CollabWorx (Tango) [CollabWorx], Interwise Glance [Interwise], Groove Networks [Groove], and Placeware [Placeware], have proved to be useful, efficient, and beneficial. It means that, among other things, distance and location is no longer a barrier or restriction; it also means that time, resources, and money is saved.

The Web Service Architecture is designed to promote software's usefulness, interoperability, availability, extensibility, and so forth. If the collaboration software is made to be Web Service (WS), totally or partially, it will be more powerful and benefit everyone – different research groups, institutions, organizations, and even ordinary users.

In this chapter, we will describe some possible ways of making such software to be Web Service, propose and focus on discussing the idea of the *Thin Client Collaboration Web Services*, and explore some potential scenarios of this idea in which it shows its merit and the freedom resulted in collaboration. The idea of the *Thin Client Collaboration Web Services* is as follows: instead of packaging the whole package of a

collaboration software as Web Service, it separates the native interface from the rest of the software; it correlates the native interface (in whatever format and language) to a web service friendly user interface (such as in SVG format and Java language), with regard to the interface's screen displays and the events originated from the triggering of the widgets inside the displays; it lets the user access the resulting user interface as if it were the native interface, relying on the fact that the Thin Client Collaboration Web Service has made them one-to-one correspondence functionally, both on the corresponding parts of the displays and on the corresponding events of the widgets.

The Extensible Markup Language (XML) [XML] is an indispensable description language for Web Service, and the Scalable Vector Graphics (SVG) [SVG] is a subset of the XML. SVG has the advantages in representing screen displays because it is vector oriented. Besides all kinds of SVG viewers, the Web browsers such as the Internet Explorer can render SVG files directly inside their windows. It seems that SVG format is the right one we should choose, and Java is the right language for the purpose, because it has been designed and developed with the Web in mind. So hereafter, when we refer to WS user interface, we mean that it is in SVG format and in Java language, though others could be potential alternatives.

The Web Service in question has two sets of ports: the User-facing Input/Output ports and the Resource-facing Input/Output ports [Fox+CTS, Fox+P2PGrid]. The user interface discussed above corresponds to the user-facing I/O, and the native interface corresponds to the resource-facing I/O. Hence, the role of the Web Service is to transcode in both directions between the two sets of ports with respect to displays and events, so

that the user accesses the user-facing I/O as if he were accessing the resource – the collaboration software – itself.

We use our three collaborative projects – the collaborative PowerPoint [Wang+JDIM, Wang+ITCC'04], the collaborative Impress [Wang+KSCE], and the collaborative IDL ReviewPlus [Wang+ITCC'05] – as examples of resources. Each of these projects consists of a Master client and a Participant client. In a collaboration session, there can be multiple Participant clients. In the session, the Master client captures events and sends out the event messages through a message broker to the Participant clients for generating the same result displays. We use our SVG related projects as the indication of our initial effort toward the implementation of the *Thin Client Collaboration Web Services* idea. We point out the future work regarding to the implementation in this chapter.

## 7.2 The Problems

While it is possible to package the whole package of the collaboration software application as Web Service, the advantage of doing so really depends on situations, and there are problems with some situations, as we will discuss next. If the package is small in size, and its deployment structure is not complicated, it is good to do so. However, if the package is big in size, it would be difficult to do so. Take the collaborative OpenOffice project as an example. The developed code for collaboration is not big, but it is based on the whole underlying OpenOffice source code, making use of its fundamental functions to achieve the collaboration. Even though it is open source, the source code of the OpenOffice consists of millions of lines of code, and it has been developed by groups

of talented people with years of hard work. Theoretically, if we were to package this project as Web Service, we have to package the underlying basis, too; or we have to package at least part of it, if we were lucky enough to know which necessary parts to include, and those parts are absolutely not calling or referencing the code of the rest all the time. How hard would that be, if it is not impossible?

If the package is related to proprietary software, it would be more difficult. Take the collaborative PowerPoint project as an example. The developed code for collaboration is not big, but it is based on the functionality of the underlying Microsoft PowerPoint application, making use of the application's functionality to achieve the collaboration. It is proprietary, and it is big, too. Again, theoretically, if we were to package this project as Web Service, we have to package the functionality of this proprietary product as well. What is the possibility of success in this matter?

If the package itself is complicated in architecture and deployment, possibly involving fire walls, it would be at least difficult. Take the collaborative ReviewPlus project as an example. The ReviewPlus [ReviewPlus] itself is big; it calls functions from an independent big package called MdsPlus [MDSplus]; it also calls functions from other packages. More complicated yet, the ReviewPlus contacts with other servers, such as the event server, and exchange information with them. It is highly possible that they are behind different fire walls, as the suggestion of the ReviewPlus itself, because it is deployed behind at least one level of fire walls. Once again, theoretically, if we were to package this project as Web Service, we have to deal with all the complexity as well. How much effort would that cost? How many people and groups would be involved, given that everybody is willing to cooperate?

What is more, if the package was developed in a popular language such as Java, C++, or C#, where accompanying tools for Web Service have been developed, it is easier for the job. What if the package was developed in a language that has not had such additional tools yet, like IDL?

On all accounts, it is desirable to find another way for Web Service, which avoids the difficulty and complexity, harnesses the resource, and accesses the user interface. This leads to our next description of the *Thin Client Collaboration Web Services*.

## 7.3 Thin Client Collaboration Web Services

As we can see, the collaboration software applications are developed on different platforms, using different models, paradigms, architectures, and methodologies, and in different programming languages. One common feature is that they usually have rich user interfaces, where users can access the input/output information and achieve collaboration between peers. We refer to these interfaces as *native interfaces*, and we refer to the applications as *resources* in this text. These resources are usually developed in multi-tiers architecture, say three-tiers: the back end tier, the middle tier, and the front end tier, with databases on the back end and native interfaces on the front.

The idea of the *Thin Client Collaboration Web Services* is as follows: instead of packaging the whole package of a collaboration software as Web Service, it separates the native interface from the rest of the software; it correlates the native interface (in whatever format and language) to a web service friendly user interface (such as in SVG format and Java language), with regard to the interface's screen displays and the events

originated from the triggering of the widgets inside the displays; it lets the user access the resulting user interface as if it were the native interface, relying on the fact that the Thin Client Collaboration Web Service has made them one-to-one correspondence functionally, both on the corresponding parts of the displays and on the corresponding events of the widgets. The structure of the Thin Client Collaboration Web Service has two sets of ports: the User-facing Input/Output ports and the Resource-facing Input/Output ports [Fox+CTS, Fox+P2PGrid], as shown in Figure 7.1.

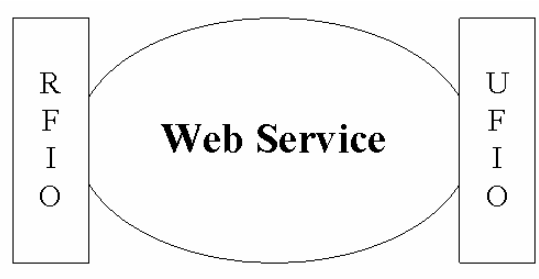



**Figure 7.1 The structure of the web service with the user-facing input/output ports and the resource-facing input/output ports.**

The WS user interface discussed above corresponds to the user-facing I/O, and the native interface corresponds to the resource-facing I/O. Hence, the role of the Web Service in question is to transcode in both directions between the two sets of ports with respect to displays and events, so that the user accesses the user-facing I/O as if he were accessing the resource – the collaboration software application – itself. Thus, the Web Service doesn't have to deal with all the difficulties and complexities of the resource; it just needs to contact with the native interface of the resource, does some transcoding between the two sets of I/O ports, and lets the rest be the encapsulation of the resource.

On one direction, the Web Service takes the output display of the resource to the input port (I) of the resource-facing I/O; then it translates the information in the native interface to the equivalence in SVG format; finally it directs the result to the output port (O) of the user-facing I/O, where the SVG viewers or the Web browsers can render the SVG file directly inside their windows for the user to view. On the other direction, the Web Service takes the event message (which is resulted from the triggering of the widget event by the user's interaction) from the SVG viewer (or Web browser) to the input port (I) of the user-facing I/O; then it translates the event message to the equivalent event (or event structure) of the resource's native interface; finally it directs the result to the output port (O) of the resource-facing I/O, where the resource gets the result and automates the execution.

We use our three collaborative projects – the collaborative PowerPoint, the collaborative Impress, and the collaborative ReviewPlus – as examples of the resources. We also have done SVG related projects: a converter that converts HyperText Markup Language (HTML) file to SVG file and a converter that converts files in Windows Metafile Format (WMF) [WMF] to SVG format. Here is a clue of implication: the OpenOffice can save its presentation file (*.sxi) as PowerPoint file (*.ppt), and the PowerPoint can save its presentation file (*.ppt) as HTML file or WMF file. The indication is that, our trial in the SVG related projects is the initial effort toward the implementation of the *Thin Client Collaboration Web Services*.

## 7.4 Thin Client Web Services in Collaboration

In this section, we give some collaboration scenarios in which the thin client web services play their roles and show their potentials. We use our three types of collaborative projects as the resources. In a session, each type has a Master client that generates event messages and at least one Participant client that consumes the event messages. The Master client and the Participant client(s) communicate and cooperate on the event messages. The thin client web services contact only with the native interfaces of the clients and access/control the clients in collaboration. The scenarios bring interoperability, flexibility, and diversity to collaboration.

### 7.4.1 Scenario 1

All the resources of our collaborative applications are developed in this manner: The Master client controls the process of a collaboration session, captures events, and sends event messages to all Participant clients through NaradaBrokering (NB) event broker [Fox+JGI, Pallickara+JDIM, Pallickara+Dissertation]; the Participants do not interfere with the process, they just receive the event messages and generate the same output displays as the Master under the instructions in the messages. This way, the states of all the clients keep the same at every event so that collaboration is achieved. This scenario follows the manner of the resources: only one instance of the Web Service is hooked up with the Master client of the resource, and the controller or the lecturer is controlling the process through a WS viewer; at least one instance of the Web Service is hooked up with

each Participant client, and the audiences or the students are viewing the displays via the WS viewers. This is depicted in Figure 7.2.

In the Figure, two participant clients are shown. In reality, it can be any number: one, two, or more (as many as the NB event broker can support); theoretically, the number is unlimited. Also in the Figure, even though only one instance of the web service is shown with each participant client, it could be multiple instances for each. What is better, the clients of the resource and the web service need not be deployed on the same location.

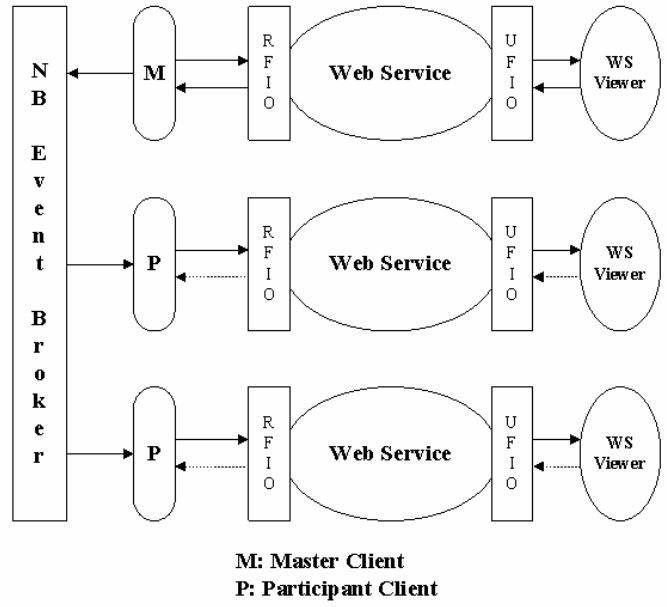



**Figure 7.2 Instances of a thin client web service in collaboration, with only one instance for the Master client and at least one instance for each of the Participant clients.**

The WS Viewers in the picture can be any SVG rendering tools: SVG viewers, Web browsers, Personal Digital Assistants (PDA), or other mobile devices [Lee+SVGopen, Lee+Dissertation], because of one factor – the output of the web service to the viewers is

in SVG format. This makes the universal access to the resources possible, and so for the universal collaborations. Since the Participant client is designed to be passive and is not allowed any input to it other than the event message from the Master client, the event input from the audiences or students (along the direction through the Web Service) has no effect on the Participant client, as indicated by the dotted arrows in the Figure.

We can originally design such a Web Service for a sole resource – the collaborative PowerPoint, the collaborative OpenOffice, the collaborative ReviewPlus, or any other collaboration application. Later on we can aggregate and relate these element Web Services to get a *general Web Service*, which will dispatch function calls to the elements on conditions so that the *general Web Service* can be used for all these resources.

### 7.4.2 Scenario 2

If we hook up two or more instances of the Web Service to the Master client in the Figure 7.2, the collaboration pattern changes to a more dynamic and democratic environment for presentation, as shown in Figure 7.3. This scenario allows two or more speakers (or lecturers) to jointly present a presentation (or lecture) to the audiences (or students), with everybody possibly on different locations. This scenario is also suitable for a study group discussing some contents online, with the rest as silent observers.

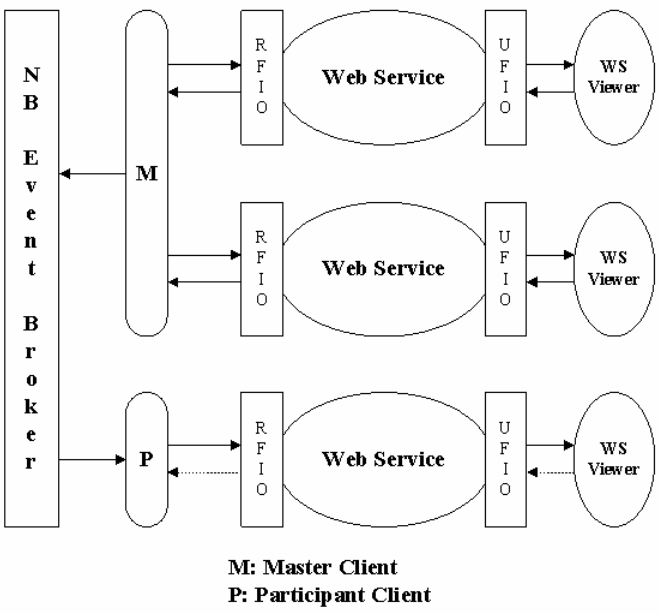



**Figure 7.3 Instances of a thin client web service in collaboration, with two or more instances for the Master client and at least one instance for the Participant client.**

It works like this: the same native interface of the Master client feeds into the input port of the resource-facing I/O of every instance hooked up with the Master client; each instance then does the transcoding job and supplies the result SVG file to its associated WS viewer through the output port of the user-facing I/O of the instance; each WS viewer then renders the same output display as the Master client. Each instance of the Web Service with the Master client takes the event message from the WS viewer (which is resulted from the triggering of the widget event by the interaction of the user) to the input port (I) of the user-facing I/O; then it translates the event message to the equivalent event/event structure of the resource's native interface; finally it directs the result to the output port (O) of the resource-facing I/O, where the Master client gets the event and automates the execution. Thus, whenever the native display of the Master client changes, the client reflects this change to each of its associated instances; every associated instance

sends some event messages to the Master client during a session, and the union of all the event messages reflects the sequence and action of the joint presentation. To the Master client, it is just like one person is controlling the process of the session.

At each step of the collaboration, the Master client and Participant client share the same state due to the communication of the event message. All the instances of the Web service hooked up with the Master client share the same state too, in the form of presenting the same resulting SVG file to the WS viewers.

### 7.4.3 Scenario 3

If we hook up multiple instances of the Web Service to the Master client only in the Figure 7.3, we get a new scenario, as shown in Figure 7.4. This scenario is a special case of scenario 2. It is adequate for a study/research group discussing some contents of their own interest online, with every one possibly in different locations. The working mechanism is the same as scenario 2. Here, the scenario just makes use of the Master client of the resource and lets the instances of the Web Service share the output displays of the Master client and control the session jointly through events.

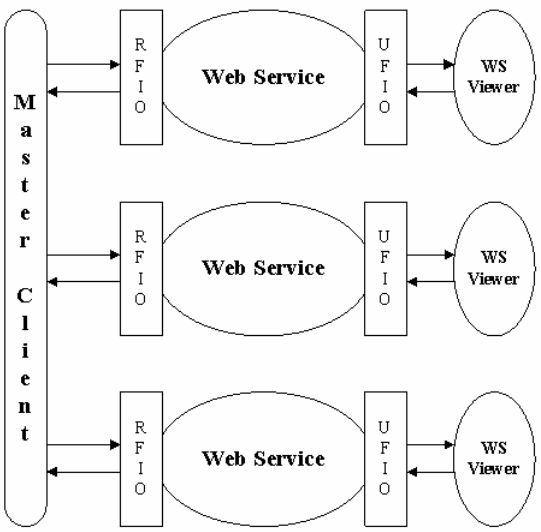



**Figure 7.4 Instances of a thin client web service in collaboration, with the instances hooked up with the Master client only.**

## 7.4.4 Scenario 4

Diverse visual aids in presentation can convey more information, make it more effective, give deep impression, and enlighten the soul. Therefore, it is a good use to bring diverse resources and Web Services together in a presentation, as shown in Figure 7.5. In this picture, the pair ($M_i$, $P_i$) represents a resource type, with $M_i$ be the Master client of the type and $P_i$ be the Participant client; the instances of the Web Service i hook up with this type of clients.

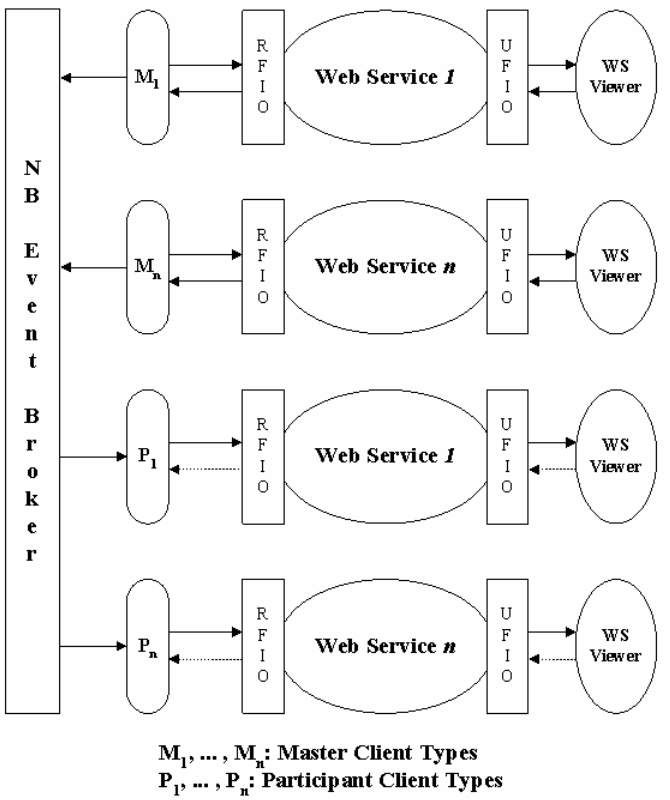



**Figure 7.5 Instances of diverse thin client web services in collaboration, with each instance type hooked up with its corresponding resource type.**

Let us conjure up an example to illustrate. Suppose that we are giving a presentation. We use the resource of the collaborative PowerPoint applications ($M_1$, $P_1$) to do the main course of the presentation; we use the resource of the collaborative Impress ($M_2$, $P_2$) to give additional information corresponding to each slide in the previous, such as references, episodes, and exhaustive details; we use the resource of the collaborative IDL ReviewPlus ($M_3$, $P_3$) to supply scientific and engineering graphs, charts, and images necessary to each slide, in 2D or 3D. This sounds more interesting and attractive.

During the presentation, the speaker controls at will the three WS viewers corresponding to the three instances of the Web Services 1, 2, and 3, which are hooked

up with the resource types 1, 2, and 3, respectively. Accordingly, the audiences use the three WS viewers to read the output displays of each type. In a simple case, the three WS viewers could be three instantiations of a Web browser on the screen of a monitor, for everyone.

As always, at each collaboration step, the states of $(M_i, P_i)$ for the Master client and the Participant client will be the same on the current event message. Any two types of resources – $(M_i, P_i)$ and $(M_j, P_j)$ – will not interfere with each other, if we put extra identifying information at the head of each event message (e.g., "ppt," "office," or "idl"); because after checking this identifying information, if the message is not supposed for a Participant client type, the client will just ignore it. Thus, the states of one resource type are independent with respect to the others.

We can originally design each Web Service for a sole resource – the collaborative PowerPoint, the collaborative Impress, the collaborative ReviewPlus, or any other collaboration application. Later on we can aggregate and relate these element Web Services to get a *general Collaboration Web Service*, which will dispatch function calls to the elements on conditions, so that the *general Collaboration Web Service* can be used for all these resources. Likewise, we can make the *general Collaboration Web Service* work for all the WS viewers, too. This makes things clearer and saves efforts in the finding and binding of the Web Service.

### 7.4.5 Potential Problems and Solutions

By now, we have focused on describing the main features of the scenarios. We can foresee some potential problems in the scenarios, and we have accordingly planed the solutions, as follows:

**Problem 1:** Usually, on the native interface (display) of the Master client of a resource, after one event is triggered (e.g., a click on a button), and before the output comes out and the display is updated, the cursor on the display window is changed to a waiting sandbox to prevent any further input; or in some special cases, even if the sandbox cursor is not shown, any further input is just ignored.

What if additional events are issued on the WS viewer in between? For instance, two consecutive button clicks – one on button 1 and one on button 2 which becomes the additional event – happened on the current display of the WS viewer. When these two events get transcoded through the Web Service and get to the native interface, this could cause trouble. Because there is a time interval between the two events, after the first one is executed by the Master client, and its native display is updated, the updated display may be a totally different one that contains no button at all. Therefore, the event of button 2 makes no sense at all to the new display.

**Solution:** Consider the behavior of the native display of the resource, we can add a similar mechanism to the Web Service. The mechanism will change the cursor on the WS viewer to a waiting sandbox after an input is put on the input port of the user-facing I/O; the mechanism will keep that status until the corresponding output has come back from

the resource and is put on the output port of the user-facing I/O. This prevents additional events from happening (by making those attempts impossible) before the current one is complete from the view of the overall system.

**Problem 2:** In scenarios 2 and 3, where multiple speakers jointly present a presentation or discuss subjects within a group, if two or more speakers issue events simultaneously or very closely, that will cause trouble as in Problem 1, since to the Master client, the result is just like one speaker is controlling. Worse yet, it could cause unpleasant results between the speakers; just imagine that, before a speaker finishes his part, another speaker issues an event intentionally or accidentally and changes the display; the rest, though innocent, also became potential "suspects" to the first one. It is at least interrupting.

**Solution:** As in some audio/video conferencing systems, we can add a resource competition mechanism to the Web Service; this resource is in the form of an icon of a microphone. When a speaker is in possess of the icon, the rest are disabled; that is, the events they issued are ignored; only the events from the current speaker can get through the Web Service and reach the Master client. After the speaker finishes his part, he releases the icon; then each speaker gets a chance to compete to get the icon and speak next. For the current speaker, problem 1 and its solution apply.

**Problem 3:** The NaradaBrokering (NB) event broker is supposed to be a common message broker deployed in the Grid for public use. It may be the case that multiple

conferencing sessions are going on using the NB as the underlying communication media, with overlaps in time with one another. How to avoid event messages from different sessions interfering with each other?

**Solution:** First, conference sessions should be advertised on a session management service, such as the GlobalMMCS [GlobalMMCS], as to schedules, titles, and unique session numbers. Second, we can arrange the Web Service to get the session number for a session and let it inform the native clients of a resource so that they agree on that session number in recognizing the supposed event messages. Any alien session number in the message will cause the message to be ignored. This session number will be added to the head of each event message. This way, each conferencing session is sifting their own messages and working on them clearly in the open Grid environment.

**Problem 4:** In scenario 4, how to avoid event messages from any two different types of resources – ($M_i$, $P_i$) and ($M_j$, $P_j$) – interfering with each other?

**Solution:** As always, at each collaboration step, the states of ($M_i$, $P_i$) will be the same on the current event message. Any two types of resources – ($M_i$, $P_i$) and ($M_j$, $P_j$) – will not interfere with each other's execution, if we make the Master client $M_i$ put extra identifying information for its type (e.g., "ppt," "office," or "idl") at the beginning of each event message (after the session number). After checking this identifying information, the corresponding Participant client $P_i$ will just ignore the message if it is not supposed for the type. Thus, the states of one resource type are independent of the others.

# 7.5 Deployment and Usage of the Collaboration Web Services

The Web Services along with the Peer-to-Peer Grids play important roles in collaboration. The Web Services enable developers and users to integrate functionality across businesses and organizations. Suppose that we have developed our general Collaboration Web Service. The information of this Web Service, such as its Universal Resource Identifier (URI) endpoint and its exposed methods, is described in the Web Service Description Language (WSDL) file; the Web Service is deployed and published to a Service Broker with this file. The users or applications can find this Web Service using the Universal Discovery, Deployment, and Integration (UDDI) technology, and then bind to the Web Service and use it via the internet [Deitel+WS], as in Figure 7.6.

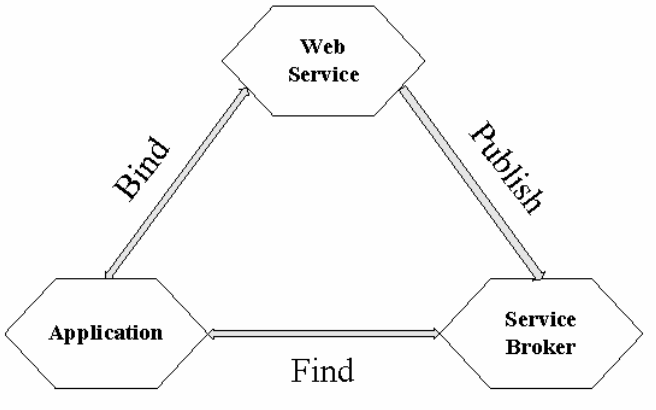



**Figure 7.6 The web service published to a service broker and used later in an application.**

On one end, a client of a resource (collaborative applications) finds and binds to the general Collaboration Web Service and cooperates with it through its resource-facing I/O; on the other end, a WS viewer performs the same procedure to bind to the general Collaboration Web Service and cooperates with it through its user-facing I/O. More specifically, as described in the scenarios before, the speaker's WS viewer is bound to the instance (of the general Collaboration Web Service) that has the Master client (of a resource) bound to it, and the audience's WS viewer is bound to the instance (of the general Collaboration Web Service) that has the Participant client (of the corresponding resource) bound to it. As always, the Master client and the Participant client(s) of each resource type collaborate on the event messages via the underlying communication of the NB event broker. Thus, collaboration is achieved through the general Collaboration Web Service.

## 7.6 Initial Effort toward the Implementation of a General Thin Client Collaboration Web Service and the Future Work

We have done SVG related projects, which serve as our initial effort toward the implementation of a general Thin Client Collaboration Web Service. They are a converter that converts HTML files to SVG files, and a converter that converts files in WMF format to SVG format. This effort is in the direction from the resource to the viewer, or from the resource-facing I/O to the user-facing I/O. It is motivated by the following clue

of implication. The Impress of OpenOffice can save its presentation file (*.sxi) as PowerPoint file (*.ppt), and the PowerPoint can save its presentation file (*.ppt) as HTML file or WMF file. The indication is that, our trial in the SVG related projects is the initial effort toward the implementation of a general Thin Client Collaboration Web Service.

Substantial work needs to be done in the implementation of the general Thin Client Collaboration Web Service that we have described, such as converting other kinds of file formats to SVG format, automating the Web Service with all kinds of resources as well as with all kinds of WS viewers, and so forth. One part of our future work resides in the transcoding of events from the viewer to the resource, or from the user-facing I/O to the resource-facing I/O. Similar experiments have been tried in the Universal CAROUSEL Access project of the Community Grid Lab at Indiana University [CAROUSEL]. In that project, the events from the SVG Viewer on a PDA are transcoded to the equivalents of the SVG Viewer on a desktop; then the user can control the SVG Viewer on the desktop wirelessly through the SVG Viewer on the PDA [Lee+SVGopen, Lee+Dissertation]. We can surely get a clue from this.

## 7.7 Conclusion

In this chapter we introduced some resources of the collaboration applications, and the needs in making them to be Web Services. We proposed and described the idea of the Thin Client Collaboration Web Services, and explored some potential scenarios of this idea in which it shows its merit and the freedom resulted in collaboration. Such a Web

Service has two sets of ports: the User-facing Input/Output ports and the Resource-facing Input/Output ports. The user-facing I/O contacts a WS viewer, and the resource-facing I/O contacts a collaborative application. Hence, the role of the Web Service is to transcode in both directions between the two sets of ports with respect to displays and events, so that the user accesses the WS viewer as if the resource itself. We used our three collaborative projects as the examples of the resources, and the SVG related projects as the indication of our initial effort toward the implementation of a general Thin Client Collaboration Web Service. We described briefly the SVG related projects. Finally we pointed out the future work.

# Chapter 8

# Conclusion, Contribution, and Future Work

This dissertation research considered key issues of the synchronous collaboration other than asynchronous Grid collaboration problems. Grids are built today totally in web service architecture and are natural for supporting collaboration as all interactions between components in terms of explicit SOAP messages. This is contrasted with the linkage of components within a traditional desktop application, which are formally messages, but their realization in such ways as method calls; this transferred information on a stack and use of non-portable pointers is hard to exploit in a collaborative environment. This can be seen in the work of this dissertation on collaborative PowerPoint and Impress applications. Our work on collaborative IDL on the other hand shows that applications built in languages where all relevant events are explicit (first class) are well suited to excellent Grid-based collaboration.

Grids are built from Web Services exchanging messages and the Community Grids Laboratory (CGL) at Indiana University has stressed the value of using Message Oriented Middleware to provide a Grid messaging substrate that can support collaborative Grids. This work provides a secure high performance messaging layer that allows for the needed message multi-cast for collaboration. Our thesis tackles complex collaborative applications that refine the requirements for the NaradaBrokering messaging system built in CGL based on the original Syracuse thesis of Dr. Pallickara's dissertation [Pallickara+Dissertation], which was partly motivated by the TangoInteractive collaboration system [CollabWorx] built at Northeast Parallel Architectures Center (NPAC) of Syracuse University.

Collaborative applications differ dramatically both in the structure and identification of the messages that are needed to define the state of the collaborating entities (agents, services). Here the dissertation makes several important contributions. First we develop a general theoretical picture – deterministic finite automata – that can be used as the formal basis for Collaborating systems. This approach applies broadly even to Audio-Video conferencing [Uyar+Dissertation] and shared display collaboration. In the first case the application is a sensor producing sound and pictures every 33 ms or so and the events are the picture changes in time produced by the sophisticated codecs like MPEG4 and H261. In the second case, the application captures the frame-buffer a few times a second and transmits the pixel changes in the defining messages. In our case we consider applications corresponding to existing applications (PowerPoint, Impress) and to those written in the IDL language. One often distinguishes between shared event and shared display cases. In our approach, everything is shared event and then one gets smaller messages in the cases

considered in this dissertation, as we are defining the state at a high semantic level such as change slide in PowerPoint or click a menu button in IDL. This can be expressed more concisely than shared display and audio/video conferencing, which essentially define the state as the pixel values in a GUI.

Peer-to-peer Grids [Hwang, Fox+ACM] exploit the observation that the entities in both P2P systems and Grids are autonomous agents (peers, services) whose state is determined by exchanging messages. Grids typically have identified "central" servers and "thin" peer clients, while P2P systems combine client and server (service) in the same machine. Our work addressed both of these cases.

Collaboration environments support Virtual Organizations (VO) or equivalently sessions with a set of authorization attributes, which specify which clients participate in which applications and their roles, such as administrative privileges and possession of the master token (floor control). Our dissertation research showed how complex applications can use this VO model.

## 8.1 Conclusion

The main tenet of this dissertation is about Grid-based collaboration between collaborating entities on event messages. We have developed different types of collaborative applications and have used them as prototypes to demonstrate this tenet. The entities in each type collaborate on event messages using a Shared Event Model. All the collaborative application types are instantiations of the Shared Event Model in Grid-based collaboration. The computing between the entities is the Peer-to-Peer Grid

computing on a Grid-based Collaboration Paradigm; the entities work on the paradigm, in which the Shared Event Model acts as the messenger and the Peer-to-Peer Grid computing acts as the basis.

These entities are finite automaton-based in a collaboration session; in essence, they are just deterministic finite automata in the session. They collaborate to share a common finite automaton in their respective instantiations and reach a common state of the finite automaton at any collaboration step. Collaboration of the entities is therefore all about being in a same state of the finite automaton at each event.

In order to increase the usefulness and universal accessibility of the collaborative applications, it is necessary to associate them with some special Web Services. We have formed and described the structure of the Thin Client Collaboration Web Services; we have explored some potential scenarios of this structure in which it shows its merit and the freedom resulted in collaboration.

There are preferences for collaboration with regard to architectures, event structures, and mechanisms of widget programming. In this dissertation all the architectures are consistent with P2P Grids. Grids today superimpose resource sharing and management on web services. Grids may or may not change message syntax but are message-based. P2P is also message-based, but lacks central services; P2P often uses different security models from Grids.

### 8.1.1 Aspect: Grid-based Messaging Scenario

In a collaboration session, there are two types of clients: the Master client and the Participant client. They collaborate on a shared event model to have the same output display at each event.

The Master client captures an event, identifies it, gets its information, and packages (or serializes) the information in a text event message, along with other necessary information. The event messages are small delimited text strings, including all the necessary information for collaboration, as to session number, application type, timing data, function to call, and property (or event structure). The Master client then sends the event message to the NaradaBrokering (NB) Message Service for broadcasting to the Participant clients.

The NaradaBrokering Message Service is a common message broker, which serves in the Grid as the underlying message communication system. It is the runtime for the communication of the event messages between the two types of clients, and it enables them to collaborate on events.

The Participant client receives, parses, and de-serializes an event messages from the NB to get all the pieces of data; it gets the name of the routine, converts the data to native types, and builds the event structure; it generates the same display as the Master client by calling the routine with the event structure as the parameter. Therefore, the Participant client renders the event message to have the same output display as the Master client.

### 8.1.2 Aspect: Shared Event Model

The entities in each type collaborate on event messages using a Shared Event Model. The entities collaborate and synchronize the states between them on the triggering of the events. Each event corresponds to a collaboration step. In a collaboration session, one entity (the Master client) in its process captures the events fired, gets the event messages, and sends them out; the other entities (the Participant clients) receive and render the event messages in their process. The event messages are small delimited text strings that contain all the necessary information for collaboration as to the session number, the application type, the timing data, the property/event structure, and the function to call. The Master client plays the role of packaging and serializing of the messages, and the Participant clients play the role of parsing and de-serializing of the messages. All the collaborative application types are instantiations of the Shared Event Model in Grid-based collaboration, and their executions are the demonstrations of the model.

### 8.1.3 Aspect: Structure of Collaborative Events

The extent of collaboration is basically decided by the extent of events that we can capture and identify in the Master clients of the collaborative applications in the research. The extent of events is decided by the events supplied and their proper definitions in the base environments, on which we are working for the collaborative applications. The ability for collaboration is limited to the extent.

We are limited in the projects of the collaborative PowerPoint and the collaborative Impress applications. Since the PowerPoint is a proprietary product, we can only rely on

the events in the dispatch interfaces provided in its type library. Other than that, we could do nothing more. Although the Impress of OpenOffice is open source, we are not developing its base code; we rely on the event types and listeners in its interfaces, and they are limited. For example, when we were trying to program the "slide show" of the Impress to be collaborative, we only found that the interfaces for this were underdeveloped.

In the collaborative IDL project, however, we are basically not limited to anything and are free to develop almost anything (as we have experienced) for collaboration. The project is a success, as we have described in Chapter 5. We owe our success mainly to the complete and proper definition of the event structures of IDL and the good mechanism of the IDL widget programming, which is on the basis of the event structures.

### 8.1.4 Aspect: Details of Collaborative IDL

In IDL, all the widgets have event structures that contain not only properties but also information as to the widget hierarchies and the associated event handlers. All the event structures have three common fields: *ID*, *TOP* and *HANDLER*. *ID* is the widget ID number of the widget that generates the event. *TOP* is the widget ID number of the top-level base that contains the widget that generates the event. *HANDLER* is the widget ID number of the widget that is associated with an event handler.

The levels of widgets include the fundamental widgets (as those in table 6.3) and the compound widgets. Take the draw widget as an example. It is a fundamental widget. It is created by the function WIDGET_DRAW in programming. We have a lot of combinations of keyword choices when we create it. If the keyword BUTTON_EVENTS

is set, pressing or releasing mouse buttons while the mouse cursor is over the draw widget causes events to be generated. If the keyword KEYBOARD_EVENTS is set, any keystroke when the draw widget has the focus of the keyboard causes an event.

Compound widgets are constructed using different types of widgets, but act as fundamental widgets. They exist for the convenience of programming and usage. IDL has some compound widgets available, such as the CW_BGROUP (button menu group), the CW_FIELD (data entry field(s)), and the CW_RGBSLIDER (RGB color value sliders). The users can develop and use their own compound widgets in programming.

We can configure some of the widgets by setting particular keywords in the widgets' creation. This allows us to adjust the accuracy of the event generation, depending on situations. Take the slider widget as an example. The function WIDGET_SLIDER has a keyword DRAG. Normally, the slider only generates position events when the slider comes to rest at its final position, and the mouse button is released; when the keyword DRAG is set, however, the slider generates events continually when it is being dragged by the user, and a large number of events can be generated.

There are two types of event handler routines associated with the widgets: the event procedures and the event functions. They are called to execute some tasks when the widgets are triggered. The difference is that, an event function can return an event (usually a different one), which is destined to a widget in the upper level of the widget hierarchy; then the event is swallowed by that widget's associated event handler to fulfill further task. This is useful in places such as the compound widgets.

### 8.1.5 Aspect: Grid-base Collaboration Paradigm

The entities work on a Grid-base Collaboration paradigm, in which the Shared Event Model acts as the messenger and the Peer-to-Peer Grid computing acts as the basis. There are two categories of systems in this paradigm: the Grid system and the Peer-to-Peer systems. The Grid is in the core and the Peers are at the edge of the paradigm. High-performance and stable services are in the Grid, such as the Web Services, the Grid Services, and the Common Message Brokers; the services of the Peers are more convenient and accessible, such as the user developed applications and the commodity applications. The infrastructure of the Networks and the Internet ties up and correlates the two categories, which complement each other. The Peer-to-Peer Grid computing harnesses the advantages of the two categories.

In our design and development of the collaborative applications, we have realized and demonstrated the Peer-to-Peer Grid computing model. We have deployed the NaradaBrokering Message Service as the broker in the Grids and used it for event message communications between the collaboration entities; we have deployed the Master client and Participant clients as Peers at the edge so that they collaborate on event messages through the Grids in the core.

### 8.1.6 Aspect: Peer-to-Peer Grid Computing

The Peer-to-Peer Grid computing is the basis of the collaboration. The Grid system largely comprises stable, formal, and efficient high-functionality services (such as the Web Services, the Grid Services, and the Common Message Brokers), which are

deployed in the Grid on structured, well-organized, and powerful supercomputers. They are in the core of the paradigm. The Peer-to-Peer system offers user-friendly, convenient, intuitive, and easy accessible applications and services, such as the popular commodity software used daily and everywhere. They are installed on a variety of personal devices, such as desktops, laptops, PDA's, and smart phones. They are at the edge of the paradigm. The Grid system offers robust, structured, and security services that scale well in pre-existing hierarchically arranged enterprises or organizations; it is largely asynchronous and allows seamless access to supercomputers and their datasets. The Peer-to-Peer system is more convenient and efficient for the low-end clients to advertise and access the files on the communal computers; it is more intuitive, unstructured, and largely synchronous.

### 8.1.7 Aspect: Theoretical Framework of Deterministic Finite Automata

Collaboration needs collaboration entities. These entities are finite automaton-based in a collaboration session; in essence, they are just deterministic finite automata in the session. Intuitively, the entities in collaboration collaborate on events to keep showing the same output displays at each step; one entity controls the process, capturing events, generating event messages, and sending them out to the other entities through a message broker; the other entities responds to the process by rendering the received event messages. Specifically, the entities collaborate to share a common finite automaton in their respective instantiations and reach a common state of the finite automaton at any

collaboration step. Collaboration of the entities is therefore all about being in a same state of the finite automaton at each event.

We also model the executions of the collaboration entities using Petri Net in *Appendix A*. The transition diagram of the DFA is much simpler and easier to understand, and, the best part for our purpose, it makes the event messages apparent by attaching them on the directed arcs, while in the Petri Net, the event messages are hidden. The main tenet of this dissertation is about collaboration between the collaborating entities on event messages. Therefore, in this dissertation, we use the DFA, even though the Petri Net representation has its own strengths and beauty in modeling the executions of the collaboration entities.

### 8.1.8 Aspect: Thin Client Collaboration Web Services

In order to promote the usefulness and universal accessibility of the collaborative applications, it is necessary to associate them with some special Web Services. We have described the structure of the Thin Client Collaboration Web Services and explored some potential scenarios in which the structure shows its merit and the freedom resulted in collaboration. Such a Web Service has two sets of ports: the User-facing Input/Output ports and the Resource-facing Input/Output ports. The user-facing I/O contacts a Web Service viewer, and the resource-facing I/O contacts a collaboration application. Hence, the role of the Web Service is to transcode in both directions between the two sets of ports with respect to displays and events, so that the user accesses the Web Service viewer as if the collaboration application itself.

### 8.1.9 Aspect: Collaborative Event-based Languages

The ability for collaboration is limited to the extent of events that we can capture and identify in the Master clients. This extent is decided by the events supplied and their proper definitions in the base environments, on which we are working for the collaborative applications. If we are to pursue high extent of collaboration, we must choose the appropriate languages and environments, such as the Interactive Data Language (IDL), where the complete and proper definitions of the event structures are supplied and the elegant mechanisms of the widget programming are guaranteed. The events are first-class; they are systematically and completely defined with respect to the structures, the levels, the configurations, and the associated types of the event handler routines.

As other things, collaboration has its preferences. First, some implementing architectures are preferable than others. While this is apparent, to actually find the better architectures becomes the point; in cases, only after we have practically tried the architectures can we tell which ones are better in general. As we have described in the dissertation and summarized in *6.2.4 Architectural Differences and Implications*, we have tried the *Notifying structure* and the *Polling structure* in our collaborative projects; we prefer the *Polling structure* because of its strengths in simpler overall system architecture, easier programming, and avoidance of data loss.

Second, well-designed event structures and good mechanisms of widget programming are of high value to collaboration. As we have described in the dissertation and summarized in *6.3.4 Differences in Event Processing and Implications*, we prefer the event structures of IDL because of the following virtues.

- They share common features: they contain hierarchical information with regard to positions of widgets in the widget trees and information of event handlers for the widgets.
- They contain property data for the widgets.
- Each widget's event structure is unique.

Likewise, we prefer the mechanism of the IDL widget programming just because it makes things easy and straightforward in the programming for the collaborative applications. As we have noticed in *6.3.4* that, IDL has distinct benefits over PowerPoint and OpenOffice in the development for collaboration with respect to Straightforwardness of Event Structures, Certainty of the Method(s) Called, Easiness in Getting Properties, Reliability of the Results, Convenience in Programming, and Extent of Collaboration. Because PowerPoint and OpenOffice are lack of these benefits, they have difficulties with event support and programming; they should have event structures that have the virtues as in IDL, and easy and straightforward mechanism of widget programming as in IDL, in order to best fit our model of Grid-based collaboration on events, as IDL does.

Third, it is preferable to work on source code than to work on binary components, in order to achieve higher extent of collaboration (if we get a chance, and if it is feasible to do so). While this is apparent again, our efforts serve as the demonstrations of this and strengthen this.

## 8.2 Contribution

During the whole dissertation research on Grid-based collaboration, we have contributed to the society as follows:

- We have developed the prototypes of Collaborative PowerPoint applications [Wang+JDIM, Wang+CATE'04], Collaborative Impress applications [Wang+KSCE], and Collaborative ReviewPlus IDL applications [Wang+ITCC'05]. They have been demonstrated mainly in the Community Grid Lab at Indiana University and can be used in e-Learning, distance education, online conference, e-Science, and more. The Collaborative Impress has also been demonstrated in England and used in a summer school in Italy for education purposes. The Collaborative ReviewPlus has been specially designed for the General Atomics and Affiliated Companies (USA), has been demonstrated to them, and will be used by them after the final release.
- We have realized the idea of Shared Event Model; the prototypes are instantiations of the Shared Event Model [Wang+SCI, Wang+KSCE] in Grid-base Collaboration.
- The prototypes are also demonstrations of the Peer-to-Peer Grid computing in the Grid-based collaboration paradigm, with the Narada Message Broker serving in Grids in the core of the paradigm and the clients of the prototypes acting as Peers at the edge of the paradigm.

- We have modeled and analyzed the prototypes with the theory of finite automaton, and this could be applied to any other such prototypes or applications with respect to collaborations on events.
- We have also shown in *Appendix A* that the prototypes can be modeled and analyzed with the theory of Petri net. Likewise, that could be applied to any other such prototypes or applications with respect to collaborations on events.
- In order to promote the collaborative applications in areas such as usefulness, availability, and universal accessibility, we have proposed and focused on discussing the idea of Thin Client Collaboration Web Services, and explored some potential scenarios of this idea in which it shows its merit and the freedom resulted in collaboration. We have used the three types of collaborative applications as resources, and projects in SVG as the demonstration of our initial effort toward the implementation of a General Thin Client Collaboration Web Service.
- We have experimented on many different methods, structures, and technologies in the research, and we are always open-minded to explore new strategies and approaches in the research of Grid-based collaboration on events. We have already had some for the future work, such as *Embedded Collaboration Object* and *Pan-event Model in Collaboration*.

# 8.3 Future Work

The future work includes branches of effort in system enhancements, exploration of implications, web services, new requirements, new categories, and new strategies and approaches.

## 8.3.1 System Enhancements

We can enhance our collaborative application systems in the following areas.

### 8.3.1.1 Extension of Event Messages

In the prototypes of our current implementations of all the collaborative application systems, the event messages of a prototype (or a type of collaborative applications) contain just enough information to make the entities of the prototype collaborate in a session. Other issues are important for collaborations of different prototypes of entities in the open environments of networks and common message brokers. They include:

1. How to prevent collaborations of different sessions from interfering with each other?
2. How to prevent different prototypes from interfering with each other?
3. How to guarantee entities of a prototype in a session share all the event messages in the same order?

The solution to all these questions is to extend the event messages to include additional information as to session numbers, type data, and ordinal numbers for each

event message. We have analyzed and reasoned how this will work in *chapter 7*. This solution will thus enable different prototypes to work jointly in a session for a purpose.

### 8.3.1.2 Continuation for Completeness

We have tried our best so far with the collaborative projects on the factors that need to be collaborative. We have relied on the following aspects.

1. Conditions allowed – Since the PowerPoint is a proprietary product, we can only rely on the dispatch interfaces provided in its type library.
2. Interfaces developed – Though the Impress of OpenOffice is open source, we are not focusing on developing its base code (it needs groups of people years of work); when we were trying to make the "slide show" of the Impress collaborative, we only found that the interfaces for this was underdeveloped, and that limited our ability to success very much.
3. Factors found – To our understanding, we have found in the ReviewPlus the factors that need to be collaborative and have done so.

We are able to continue to develop the projects if the above aspects change. For example, the change may be the new dispatch interfaces provided in the PowerPoint, the new interfaces developed in the Impress, or the new factors found in the ReviewPlus (our knowledge is always limited; we might discover more in the future, or the users might inform us).

## 8.3.2 Exploration of Implications

In section 6.6, we have described the implications of collaboration entities of paradigms linked by message in collaboration. We can explore the implications in the future. In scenario 1, we can try to let different types of collaboration entities of paradigms to collaborate with each other, e.g., MVC with FCM. This means, for instance, we can attempt to use the Master client of the collaborative PowerPoint and the Participant client of the collaborative Impress in collaboration. Thus, this decouples the types of the entities of paradigms, gives more freedom, and brings diversity in collaboration. In scenario 2, we can do research on the three-tier computing with message, where clients and servers can be developed in different languages and run on diverse platforms and databases can be in diverse database environments. The message glues the three tiers together and invokes their functions.

## 8.3.3 Web Services

We have discussed the structure of the Thin Client Collaboration Web Services in *chapter 7* and explored some potential scenarios in which the structure shows its merit and the freedom resulted in collaboration. Such a Web Service has two sets of ports: the User-facing Input/Output ports and the Resource-facing Input/Output ports. The user-facing I/O contacts a WS viewer, and the resource-facing I/O contacts a collaborative application (resource). Hence, the role of the Web Service is to transcode in both directions between the two sets of ports with respect to displays and events, so that the user accesses the WS viewer as if the resource itself. Substantial work needs to be done

in the implementation of a General Thin Client Collaboration Web Service, such as converting other kinds of file formats to SVG format, transcoding of events from the viewer to the resource (or from the user-facing I/O to the resource-facing I/O), and automating the Web Service with all kinds of resources as well as with all kinds of WS viewers.

### 8.3.4 New Requirements

Each collaborative application type consists of a Master client and a Participant client. The Master client is in active mode, capturing events, getting and sending out event messages. It is controlled by the user through its interface in a session. The Participant client is in passive mode, receiving the event messages and generating the output displays. It does not allow the user to input data or to control the process through its interface. The user is just a viewer of the output displays.

After inspecting the collaborative ReviewPlus applications, the people from the General Atomics Affiliation, Inc. of USA have suggested new requirements for the project. They need new functions developed as in the following requirements.

1. The user can control the Participant client; the user can enter data to the Participant client during a collaboration step; thus the Participant client can navigate to other displays or results during that step.
2. When the Master client moves to the next collaboration step, the Participant client should follow immediately (no matter where it navigated to during the last step), and its display should update to the same as the Master client.

This is interesting. To achieve this goal, we might have to adopt a new strategy and structure for the implementation. The method of preemption might be a good candidate. Let us call it a *preemptive structure*.

## 8.3.5 New Categories

Under the theme of Grid-based Collaboration, we have tried such areas as the PowerPoint of Microsoft Office, the Impress of OpenOffice, and the ReviewPlus of IDL applications. There are more categories we can try in the future. In the Microsoft Office suite, there are other applications such as *Access*, *Excel*, *Publisher*, and *Word*. In the OpenOffice suite, there are other applications such as *Drawing*, *Spreadsheet*, *HTML Document*, and *Text Document*. In the IDL world, there are many developed useful user applications over there. Other similar applications like IDL are of interest to us. Popular ones include Matlab, Mathematica, and Maple. We can also try applications of other categories that need collaboration through Grids.

## 8.3.6 New Strategies and Approaches

We are always open-minded to explore new strategies and approaches in our research area. Theoretically they are deductions of the general principle – Grid-based collaboration on events. We have already formed some. They are the Dynamic Structure, the Embedded Structure, and the Pan-event Model in Collaboration.

### 8.3.6.1 Compiler Application on Dynamic Structure

We have described a potential general solution for collaboration in IDL in *5.7.2 Dynamic Structure*. It is about creating a compiler application that can generate collaborative applications from stand-alone ones. For instance, that compiler application would take the "ReviewPlus" application package as input, go through the life cycles of the theory of compiler (i.e., do lexical, syntax, and semantic analyses, possibly optimize code, and generate code), and automatically generate the "collaborative ReviewPlus applications," as the one we have succeeded and demonstrated so far. It will simulate the process of making collaborative applications from stand-alone ones and generate the collaborative applications *dynamically* by taking stand-alone ones as input. We can work on this in the future.

### 8.3.6.2 Embedded Collaboration Object

We described a potential general solution to enable a desktop IDL application to collaborate between its instantiations in section *5.7.3 Embedded Structure*. We described the potential breakthroughs and benefit. We pointed out that it is a solution of *Embedded Collaboration Object*. In the section we also extracted and analyzed the factors for success in the building of the collaborative ReviewPlus applications from the stand-alone ReviewPlus package. Those factors can be applied and reused in the implementation of the embedded structure, because the corresponding versions of the Master client and the Participant client will basically perform the same functionality. Those factors should work in the implementation of the embedded structure, because they will basically be

migrated (with possible modifications) from the end-user collaborative IDL applications to the IDL system library routines and the like. To the core of IDL, both parts are treated as extensions. The success factors are the building blocks for the future work in the constructions of the embedded collaboration objects.

### 8.3.6.3 Pan-event Model in Collaboration

The common nature in all of our collaboration projects is as follows. When an event happened on the Master client, we have to figure out what major function/procedure was called and what parameters were used (or have to get the approximations); then we put the information in the event message, and let the event message direct the Participant client to generate the same thing. The ability for collaboration here is basically limited to the events that we are able to capture. We rely on the dispatch interfaces in the PowerPoint, the event listeners in the Impress, and the event handlers in the ReviewPlus. We could do nothing more in the proprietary PowerPoint; it is not our business to add or update the event listeners in the Impress or the event handlers in the ReviewPlus.

If we are to pursue higher extent of collaboration, we may have to explore new strategies and approaches in the research. We have an idea. Instead of living on the pity of the available event interfaces, we can strive to make every execution of a routine (function or procedure) an event and to get the routine's name and the parameter(s) for the event message in the collaboration. Such an implementation may require us to program on the system level. The important task is to detect the execution of any routine on the Master client, to know the routine's name and the parameter(s) used, and to make the Participant client execute the same thing. We may ignore the routines called from

within another routine, because they may be redundant for the Participant client. This idea is a deduction from the general principle of Grid-based collaboration on events.

In Java, the concept of the bytecode for portability is general, but its implementations for different platforms are different. Likewise, the implementations of this deduction for different applications on different platforms will be different, but the idea is general. Let us call it "*Pan-event Model in Collaboration*."

# Appendix A

# Applications of Event Modeling with Petri Net in Collaboration

We have modeled our collaborative projects on PowerPoint, OpenOffice and IDL using the Deterministic Finite Automata (DFA). The elements of those projects (the Master client and the Participant clients) are thought of as *Collaboration Entities* in Peer-to-Peer Grid. The elements in all the projects are just different types of entities. Those entities are finite automaton-based in a collaboration session; they are just finite automata, or deterministic finite automata in the session. They share a common finite automaton in the collaboration and collaborate to reach a same state of the automaton at each event.

In this text, we will show that the collaborative applications can also be modeled as Petri Net, which has developed for modeling a broad range of systems. Petri Net was first developed by Dr. Carl Adam Petri in his Ph.D. dissertation "Communication with Automata" [Petri+Dissertation] in 1962. It has since been developed continually through

the years by different people, including Dr. Petri himself, and research groups to expand its application categories or to focus on some aspects, such as the colored Petri Net. Nowadays people often use “Petri Nets” to refer to all the new developments and the original work in this area. Petri Net has been applied to model many different systems, including computer hardware and software; it has shown its strength and elegance in many aspects such as Parallelism or concurrency.

Informally, a Petri Net consists of Places (represented by circles), Transitions (represented by bars), an Input function, and an Output function. The Input function decides the specific places as inputs to a transition, and the Output function decides the specific places as outputs from a transition. The Places and the Transitions are linked by directed arcs. The arc pointed to a transition is from an input place, and the arc pointed from a transition is to an output place.

The Places can have tokens; in that case, the Petri Net is called a marked Petri Net. The tokens reside in the Places. The execution of the Petri Net is controlled by the number and distribution of the tokens in the Places. The Petri Net executes by firing transitions. A transition fires by removing tokens from its input places and distributing new tokens to its output places.

We can use the Places with tokens to represent the states in a collaboration session, and the Transitions to represent the events. Intuitively, the entities in the collaboration session – the Master client and the Participant clients – collaborate on events to keep showing the same output displays at each event (or collaboration step), with the Master client in controlling by capturing the events and sending out the event messages to the Participant clients through a message broker and the Participant clients in responding by

rendering the received event messages. Specifically, the entities collaborate to share a common marked Petri Net in their respective instantiations (i.e., the marked Petri Net in them are the same) and reach a common state of the marked Petri Net on any event by having the same number and distribution of the tokens in their respective Places. Collaboration of the entities is therefore all about being in a same state of the marked Petri Net at each event.

# A.1 The Petri Net Structure

A Petri Net can be represented as a four-tuple structure [Peterson]:

$$C = (P, T, I, O), \text{ where}$$

C is the name of the Petri Net;

P is the finite set of places of the Petri Net;

T is the finite set of transitions of the Petri Net;

I is an input function of the Petri Net;

O is an output function of the Petri Net.

$P = \{p_1, p_2, \ldots, p_n\}$, with $n \geq 0$. An arbitrary place of P is denoted by $p_i$. The cardinality of P is n.

$T = \{t_1, t_2, \ldots, t_m\}$, with $m \geq 0$. An arbitrary transition of T is denoted by $t_j$. The cardinality of T is m.

P and T are disjoint, $P \cap T = \emptyset$.

The input function (I) and the output function (O) relate the places and the transitions.

$I: T \rightarrow P^{\infty}$ maps a transition $t_j$ to a bag of places $I(t_j)$. The bag of places $I(t_j)$ is the input places of the transition $t_j$. A place $p_i$ is an input place of the transition $t_j$ if $p_i \in I(t_j)$. The number of occurrences of $p_i$ in $I(t_j)$ is the multiplicity of $p_i$ in the input places of the transition, which is denoted by $\#(p_i, I(t_j))$.

$O: T \rightarrow P^{\infty}$ maps a transition $t_j$ to a bag of places $O(t_j)$. The bag of places $O(t_j)$ is the output places of the transition $t_j$. A place $p_i$ is an output place of the transition $t_j$ if $p_i \in O(t_j)$. The number of occurrences of $p_i$ in $O(t_j)$ is the multiplicity of $p_i$ in the output places of the transition, which is denoted by $\#(p_i, O(t_j))$.

We give an example next to illustrate the structure of the Petri Net.

***Example 1***: A Petri Net structure.

$$C = (P, T, I, O)$$

$$P = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8\}$$

$$T = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9\}$$

$I(t_1) = \{p_1\}$ $O(t_1) = \{p_2\}$

$I(t_2) = \{p_2\}$ $O(t_2) = \{p_3, p_3\}$

$I(t_3) = \{p_3\}$ $O(t_3) = \{p_4, p_4, p_4\}$

$I(t_4) = \{p_4\}$ $O(t_4) = \{p_5\}$

$I(t_5) = \{p_5\}$ $O(t_5) = \{p_8\}$

$I(t_6) = \{p_2, p_2\}$ $O(t_6) = \{p_6\}$

$I(t_7) = \{p_6, p_6, p_6\}$ $O(t_7) = \{p_7\}$

$I(t_8) = \{p_7\}$ $O(t_8) = \{p_5\}$

$I(t_9) = \{p_2, p_3, p_6\}$ $O(t_9) = \{p_5, p_5\}$

In this example, the cardinality of P is $|P| = 8$, and the cardinality of T is $|T| = 9$. The input function (I) and the output function (O) map each $t_j$ to its bag of input places and bag of output places, respectively. For instance, $I(t_9)$ decides the bag of input places for the transition $t_9$ to be $\{p_2, p_3, p_6\}$, and $O(t_9)$ decides the bag of output places for the transition $t_9$ to be $\{p_5, p_5\}$. The multiplicity of the place $p_6$ as input to the transition $t_7$ is $\#(p_6, I(t_7)) = 3$, and the multiplicity of the place $p_3$ as output from the transition $t_2$ is $\#(p_3, O(t_2)) = 2$.

## A.2 The Petri Net Graph

While the Petri Net structure is the main method to define a Petri net, in practice, a graphical representation of the Petri Net is more convenient and useful in modeling and analysis. The Petri Net graph [Peterson] is such a graphical representation tool. The Petri

Net graph, G = (V, A), consists of a set of vertices, V = $\{v_1, v_2, \ldots, v_s\}$ ($s \geq 0$), and a bag of directed arcs, A = $\{a_1, a_2, \ldots, a_r\}$ ($r \geq 0$). The set of vertices V consists of all the places in the set P and all the transitions in the set T of the corresponding Petri Net structure. V = P U T, P ∩ T = Ø. In the graph, the places are represented as circles, and the transitions are represented as bars. An arc connects a circle and a bar.

For an arc $a_i = (v_j, v_k)$, either $v_j \in P$, $v_k \in T$, or $v_j \in T$, $v_k \in P$.

If $v_j \in P$, $v_k \in T$, then $a_i$ is an arc from the input place $v_j$ to the transition $v_k$.

If $v_j \in T$, $v_k \in P$, then $a_i$ is an arc from the transition $v_j$ to the output place $v_k$.

The graph is a multi-graph, because it allows multiple arcs from one vertex of the graph to another. It is a directed graph because its arcs are directed. It is a bipartite graph because its vertices or nodes can be partitioned into two sets: a set of places and a set of transitions. A directed arc is either from a place to a transition, or vice versa. So, the Petri Net graph is a bipartite directed multi-graph.

To convert a Petri Net structure C = (P, T, I, O) to a Petri Net graph G = (V, A), convert the places to circles and the transitions to bars, and take

$$V = P \cup T$$

$$\#((p_i, t_j), A) = \#(p_i, I(t_j))$$

$$\#((t_j, p_i), A) = \#(p_i, O(t_j))$$

The equivalent Petri Net graph representation for the Petri Net structure of *Example 1* above is shown in Figure A.1.

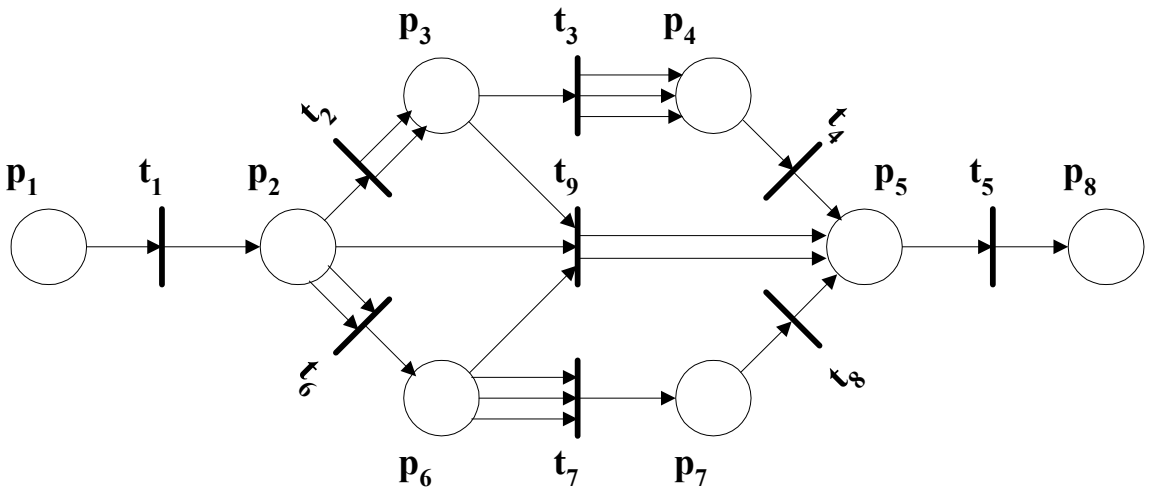



**Figure A.1 The equivalent Petri Net graph representation for the Petri Net structure of Example 1.**

For instance, in the figure,

$$\#((p_6, t_7), A) = \#(p_6, I(t_7)) = 3, \text{ and}$$

$$\#((t_2, p_3), A) = \#(p_3, O(t_2)) = 2.$$

# A.3 The Petri Net Markings

The marking μ of a Petri Net is a primitive concept of the Petri Net, just as the concepts of the places and the transitions [Peterson]. A marking of a Petri Net assigns tokens to places. The tokens can be thought of as assigned to the places and reside in the

places. The number and position of the tokens can be changed during the execution of the Petri Net. The tokens decide the execution of the Petri Net.

Formally, the marking μ can be defined as a function from places to nonnegative integers, as

$$\mu: P \rightarrow N, n = |P| \text{ and } n \geq 0.$$

At the same time, the marking μ can also be defined as an n-vector, as

$$\mu = (\mu_1, \mu_2, \ldots, \mu_n), n \geq 0.$$

The function and the vector notations are related by

$$\mu(p_i) = \mu_i, \text{ for } p_i \in P, \mu_i \in N, i = 1, 2, \ldots, n.$$

In the graph representation of a Petri Net, the tokens are indicated as small dots inside the circles for places. Since the number of tokens for a place is unbounded, N, the set of all markings for a Petri Net with n places is simply the set of all possible vectors of the n-vector on nonnegative integers, whose number is $N^n$. This set is obviously infinite, but it is numerable. For example, if there is a token in each of the places $p_2$, $p_3$, and $p_6$ in Figure A.1, the marked Petri Net looks as in Figure A.2.

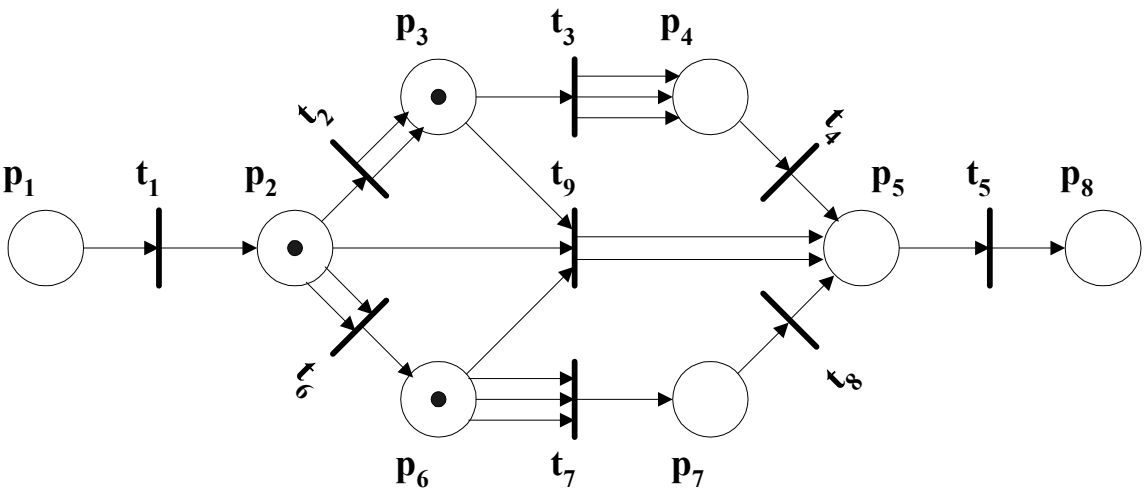



**Figure A.2 A marking of the Petri Net graph for the Petri Net structure of Example 1.**

In vector, the marking is $\mu = (0, 1, 1, 0, 0, 1, 0, 0)$.

# A.4 The Petri Net Execution Rules

The execution of a Petri Net is controlled by the number and distribution of the tokens in the places [Peterson]. The tokens reside in the places and control the execution. It can also be viewed as controlled by the markings of the marked Petri Net. Each marking corresponds to a state of the Petri Net.

Petri Net executes by firing transitions. Each transition may fire if it is *enabled*. In the graph representation, if each input place of a transition has at least as many tokens in the place as the input arcs from the place to the transition, then the transition is enabled; the tokens in the transition's input places are the *enabling tokens*.

Formally, a transition $t_j \in T$ in a marked Petri Net C = (P, T, I, O) with marking μ is enabled, if for all $p_i \in P$,

$$\mu(p_i) \geq \#(p_i, I(t_j))$$

If a transition is enabled in a Petri Net, it may fire by removing the enabling tokens in its input places, one token for each input arc, producing new tokens and depositing them into its output places, one token for each output arc. This firing of the transition creates a new marking μ', which defines the next state of the Petri Net.

Formally, if a transition $t_j \in T$ is enabled in a marked Petri Net C = (P, T, I, O) with marking μ, it may fire and create a new marking μ', which is defined by

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j)), \text{ for all } p_i \in P.$$

For instance, in Figure A.2, the transition $t_9$ is enabled, because each of its input places $p_2$, $p_3$, and $p_6$ has a token in it, and this number is equal to the number of arcs from the place to the transition, which is also 1. When the transition $t_9$ fires, it removes the enabling tokens from its inputs and deposits new produced tokens into its outputs. In this case, it removes one token from each of its input places $p_2$, $p_3$, and $p_6$ and deposits two tokens into its output place $p_5$, because $p_5$ is the only output place and its multiplicity is 2. The resulting graph is shown in Figure A.3.

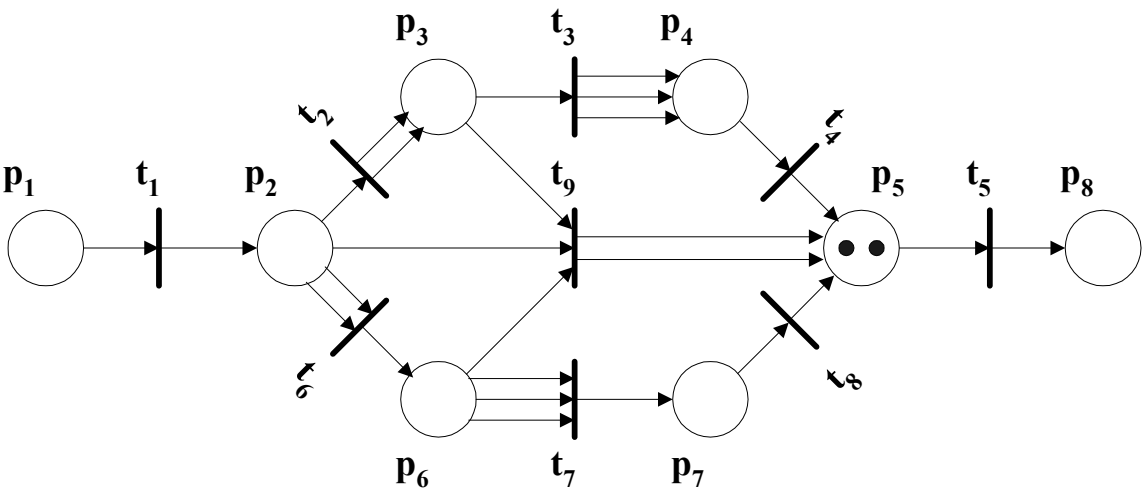



**Figure A.3 The result marking after the firing of the transition $t_9$ in the Petri Net graph of Figure A.2.**

In vector, the new marking is μ' = (0, 0, 0, 0, 2, 0, 0, 0).

# A.5 The Petri Net State Spaces

In the execution of a Petri Net, each marking corresponds to a state. The state is defined by the marking. In general, the firing of a transition results in a change in the state and a change in the marking [Peterson]. The *state space* of a Petri Net with n places is the set of all markings, whose number is $N^n$.

In a marking μ, if a transition $t_j \in T$ is enabled, it can fire, and the Petri Net will get to a new state and a new marking μ'. This can be viewed as the result of a change function – the *next-state function* δ, as $\delta(\mu, t_j) = \mu'$.

The next-state function $\delta: N^n \text{ X } T \rightarrow N^n$ for a Petri Net C = (P, T, I, O) with marking $\mu$ and transition $t_j \in T$ is defined, if and only if

$$\mu(p_i) \geq \#(p_i, I(t_j))$$

for all $p_i \in P$.

If $\delta(\mu, t_j)$ is defined, then it can fire, and $\delta(\mu, t_j) = \mu'$ with

$$\mu'(p_i) = \mu(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))$$

for all $p_i \in P$.

In a marking $\mu$, if there exists at least one transition $t_j \in T$ that is enabled, the Petri Net can execute by firing such an enabled transition $t_j$ and goes to the next state with a new marking $\mu'$. The execution of the Petri Net can keep going on as long as there exists at least one enabled transition $t_j \in T$ in the current marking $\mu$. If there is no enabled transitions in the current marking $\mu$, then $\delta(\mu, t_j)$ is undefined, and the execution of the Petri Net must halt.

Along with the execution of a Petri Net, two sequences of information are produced: the *sequence of markings* ($\mu^0, \mu^1, \mu^2, \ldots$) and the *sequence of transitions* ($t_{j0}, t_{j1}, t_{j2}, \ldots$). The relationship between the two sequences is $\delta(\mu^k, t_{jk}) = \mu^{k+1}$, for k = 0, 1, 2, ….

Given the initial marking $\mu^0$ and the sequence of transitions in the execution of a Petri Net, we can derive the corresponding marking sequence; except for some degenerate cases, given the sequence of markings, we can derive the corresponding transition sequence. Both sequences provide a record of the execution of a Petri Net.

For a Petri Net C = (P, T, I, O), a marking μ' is *immediately reachable* from a marking μ, if there exists some $t_j \in T$, such that $\delta(\mu, t_j) = \mu'$.

If μ' is immediately reachable from μ, and μ" is immediately reachable from μ', then μ" is *reachable* from μ.

For a Petri Net C = (P, T, I, O) with marking μ, the *reachability set* R(C, μ) is the smallest set of markings, which is defined by the following conditions:

1. $\mu \in R(C, \mu)$.

2. If $\mu' \in R(C, \mu)$, and $\delta(\mu', t_j) = \mu''$ for some $t_j \in T$, then $\mu'' \in R(C, \mu)$.

Given the initial marking μ and a transition sequence $t_{j0}t_{j1}\ldots t_{jk}$, we can derive the final new marking μ' by firing first $t_{j0}$, then $t_{j1}$, and so on until $t_{jk}$ is fired. Each transition must be enabled when it is to be fired.

It is often convenient to extend the next-state function δ, and let it take a transition sequence instead of a single transition as the parameter, when describing the execution of a Petri Net. This leads to the concept of the *extended next-state function*.

The extended next-state function $\Delta: N^n \times T^* \rightarrow N^n$ for a Petri Net C = (P, T, I, O) with initial marking μ, is the mapping of the pair of μ and a transition sequence $\sigma \in T^*$ into a new marking μ', as in

$$\Delta (\mu, \sigma) = \mu'$$

It can be defined by induction on the length of the transition sequence σ, as follows.

**BASIS**: $\Delta (\mu, \varepsilon) = \mu$. That is, if the Petri Net is in marking μ, and there is no transition sequence fired from there, the Petri Net is still in marking μ.

**INDUCTION**: Suppose that $\sigma = t_j t_{j1} \ldots t_{jk}$, $x = t_{j1} \ldots t_{jk}$, and $a = t_j$, we can write $\sigma = ax$, in which "a" is the first transition of the sequence σ, and "x" is the rest of the sequence. Then,

$$\Delta (\mu, \sigma) = \Delta (\delta (\mu, a), x)$$

# A.6 Characteristics of the Petri Net for Collaboration Entities

As we have mentioned at the beginning of this text, the execution of an entity of a type can be modeled as a marked Petri Net; all the instantiations of the entities of that

type have the same Petri Net in their respective instances in a collaboration session; the collaboration between them is all about having the same state or marking at any collaboration step, with each step corresponding to an event. The modeling of the collaboration entities as a special marked Petri Net is described below.

- The places P of a Petri Net C = (P, T, I, O) are used to represent all the states in an instantiation of an entity (of a type of collaborative applications), with a place and a state in one-to-one correspondence.
- The transitions T of a Petri Net C = (P, T, I, O) are used to represent all the events in an instantiation of an entity (of the type of collaborative applications), with a transition and an event in one-to-one correspondence. Events fired from different states are treated as different events, even if they have the same semantic meaning. For example, the event message "next slide" is treated differently for slide 1 and slide 2. Therefore, a transition $t_j$ conveys not only the semantic event but also the locality of the event between states.
- The invocation of an entity is dedicated as the initial state, and events can only be fired from here. The exit of the execution of the entity is dedicated as the final state, and no events can ever be fired from there.
- Based on the deterministic property of the collaboration entity and the above assignments, each state may associate with many events (e.g., state slide 2 may fire event "next slide" or "previous slide"), and each event can only be fired from that associating state and result in the next determined state. In other words, a transition $t_j$ takes only one place as input and connects only one place as output; that is, $|I(t_j)| = |O(t_j)| = 1$.

- In the Petri Net graph, the place corresponding to the initial state has one and only one token when the entity is invoked; this is indicating that an instantiation of the entity is ready. Later, this token is moving around the places with the execution, indicating which state the execution is in. It will finally come to the place corresponding to the final state when the execution is exited and will remain there forever.
- According to the execution rules of the Petri Net, a place (state) in this case can only fire one of its associated transitions (events) and disable the others. There is only one token in the place; when one of its associated transitions is fired, the token is removed from the place and is deposited in the transition's output place; now the place has no token any more and therefore none of the other transitions is enabled. This is illustrated in Figure A.4.
- There is one and only one token in the whole marked Petri Net.

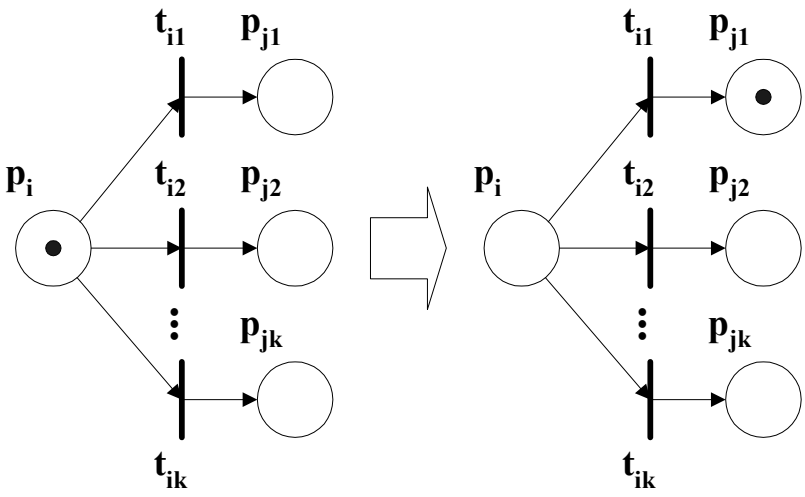



**Figure A.4 The transitions $t_{i1}$, $t_{i2}$, …, $t_{ik}$ are enabled by the token in $p_i$. After the firing of the transition $t_{i1}$, the token is removed from $p_i$ and is deposited into $p_{j1}$ by**

**the Petri Net execution rules. By now, none of the other transitions $t_{i2}$, …, $t_{ik}$ is enabled.**

In computer hardware devices, *safeness* is a very important property. The hardware devices can be modeled as Petri Nets. A place is safe if the number of tokens in the place never exceeds one. A Petri Net is safe if all the places are safe.

Formally, a place $p_i \in P$ of a Petri Net $C = (P, T, I, O)$ with initial marking $\mu$ is *safe*, if for all $\mu' \in R(C, \mu)$, $\mu'(p_i) \leq 1$. The Petri Net is safe if all the places are safe.

In the implementation of a hardware device, a safe place can be implemented as a single flip-flop.

The modeling of the collaboration entities as a marked Petri Net described above shares this safeness property, because there is only one token in the marked Petri Net moving around among the places as the execution of the Petri Net is going on, so the requirement for the net to be safe is satisfied.

Petri Net can be used to model resource allocation systems to describe the requests, the allocations, and the releases of the resources. The tokens in the net represent the resources. In a stable system, the number of the resources is fixed; therefore, the total number of the tokens in all the places of the net is fixed at any step; the tokens are neither created nor destroyed. Hence, in this type of systems, conservation is a very important property.

A Petri Net C = (P, T, I, O) with initial marking μ is *strictly conservative*, if for every $\mu' \in R(C, \mu)$, $\sum \mu'(p_i) = \sum \mu(p_i)$, $p_i \in P$ (i = 1, 2, 3, …).

The modeling of the collaboration entities as a marked Petri Net described previously is strictly conservative, because there is only one token in the marked Petri Net moving around among the places as the execution of the Petri Net is going on, so the summation of all the tokens in the net at any marking remains constant, 1.

## A.7 Unification of the Collaboration Entities with Petri Net

If we observe the collaboration entities in a project – the Master client and the Participant clients – on lower levels (e.g., design and implementation), they are different things, with respect to strategies, architectures, languages and technologies used, and roles supposed. However, if we consider the entities on high level as to changes of the markings (state transitions) of the Petri Net in question, they share the same state of logic (same marking) at any step in a collaboration session; therefore, in essence, they have the same Petri Net in their respective instantiations and collaborate to have the same marking at the end of each firing of a transition, which is equivalent to an event.

In practice, the entities of the Master client and the Participant client are created for different purposes; they are binary. In theory, they follow the same logic of the collaboration and manage to share the same marking of a common Petri Net at a step in a session; they are unity. They are binary so that they serve the special needs and satisfy

the special requirements as to the capturing of events in the Master client and the rendering of the event messages in the Participant client. They are unity so that they have the same logic state (marking) at any collaboration step in the form of the same output displays.

In more detail, on the entity of the Master client, the user controls the process of a session by physically controlling the interfaces of the entity using mouse clicks, keystrokes, etc., which we can call physical events. The entity responds to these physical events and navigates through each of the corresponding states. At the same time, for each of these events, it builds up an event message regarding information about the event such as the function to call and the property (or event structure). The event message is a delimited text string and the intermediate representation of the event for transmission or broadcasting via the message broker. On the entity of the Participant client, the entity parses the delimited text string after receiving it; based on the information, the entity arranges which function to call, converts all the types of data represented in string to the system's interior representations, and builds up the native event structure. The entity then automates to get to the same state as in the Master client by calling a function, mostly with the property or the event structure as the parameter. The entity of the Participant client is controlled programmatically during a collaboration session, which is called automation.

Considered logically, both the Master and the Participant entities can be modeled as a Petri Net in a collaboration session. They maintain the same sets of places and transitions and collaborate on events to be in a same marking at any event. The next-state

function δ takes the current marking μ and a transition $t_j$ (which corresponds to an event message) as parameters and transits to the next state by resulting in a new marking.

In Object-oriented Programming languages like C++, polymorphism is used to refer to the objects of classes in different shapes, builds, and configurations yet performing the same logical functions using the same interfaces, such as the "print" objects for different devices of the printing hardware and the monitor screens. In Peer-to-Peer Grid computing, we can use polymorphism in higher level to reference the instantiations of the collaboration entities (the Master client and the Participant clients), which are different in shapes, builds, and configurations, but are same in logic of the unity of the marked Petri Nets. This is the Unification of the collaboration entities.

## A.8 A Petri Net Example for the Collaborative PowerPoint and the Collaborative Impress

Let us use a Petri Net example suitable for the collaborative PowerPoint and the collaborative Impress applications to demonstrate the execution of the collaboration entities with the Petri Net. Suppose that there is a presentation file either in the format of .ppt for PowerPoint or .sxi for Impress, and there are three slides in this file, slides 1, 2, and 3. Accordingly, the finite set of places P of the Petri Net C = (P, T, I, O) is:

$$P = \{p_0, p_1, p_2, p_3, p_4\}, \text{ where}$$

$p_0$ is the place for the state when the application is started;

$p_1$ is the place for the state when the application is in slide 1;

$p_2$ is the place for the state when the application is in slide 2;

$p_3$ is the place for the state when the application is in slide 3;

$p_4$ is the place for the state when the application is ended.

All the possible event messages in the execution are as follows.

$$\Sigma = \{a_0, a_1, a_2, a_3, a_4, a_5, a_6\}, \text{ with}$$

$a_0$ = "Openfile;C:/file1.ppt" (or file1.sxi, meaning open this file);

$a_1$ = "Goto;1" (meaning go to slide 1);

$a_2$ = "Goto;2" (meaning go to slide 2);

$a_3$ = "Goto;3" (meaning go to slide 3);

$a_4$ = "Exit" (meaning the application exits);

$a_5$ = "Previous" (meaning go to the previous slide);

$a_6$ = "Next" (meaning go to the next slide).

Accordingly, the finite set of transitions T of the Petri Net C = (P, T, I, O) is:

$$T = \{t_{01}, t_{04}, t_{11}, t_{12}, t_{13}, t_{14}, t_{15}, t_{16}, t_{21}, t_{22}, t_{23}, t_{24}, t_{25}, t_{26}, t_{31}, t_{32}, t_{33}, t_{34}, t_{35}, t_{36}\}$$

$t_{01}$ is the transition that results in the change of state from place $p_0$ to $p_1$.

$t_{04}$ is the transition that results in the change of state from place $p_0$ to $p_4$.

$t_{11}$ is the transition that results in the change of state from place $p_1$ to $p_1$.

$t_{12}$ is the transition that results in the change of state from place $p_1$ to $p_2$.

$t_{13}$ is the transition that results in the change of state from place $p_1$ to $p_3$.

$t_{14}$ is the transition that results in the change of state from place $p_1$ to $p_4$.

$t_{15}$ is the transition that results in the change of state from place $p_1$ to $p_1$.

$t_{16}$ is the transition that results in the change of state from place $p_1$ to $p_2$.

$t_{21}$ is the transition that results in the change of state from place $p_2$ to $p_1$.

$t_{22}$ is the transition that results in the change of state from place $p_2$ to $p_2$.

$t_{23}$ is the transition that results in the change of state from place $p_2$ to $p_3$.

$t_{24}$ is the transition that results in the change of state from place $p_2$ to $p_4$.

$t_{25}$ is the transition that results in the change of state from place $p_2$ to $p_1$.

$t_{26}$ is the transition that results in the change of state from place $p_2$ to $p_3$.

$t_{31}$ is the transition that results in the change of state from place $p_3$ to $p_1$.

$t_{32}$ is the transition that results in the change of state from place $p_3$ to $p_2$.

$t_{33}$ is the transition that results in the change of state from place $p_3$ to $p_3$.

$t_{34}$ is the transition that results in the change of state from place $p_3$ to $p_4$.

$t_{35}$ is the transition that results in the change of state from place $p_3$ to $p_2$.

$t_{36}$ is the transition that results in the change of state from place $p_3$ to $p_3$.

The start place is $p_0$, which is the place for the state when the application is started. So, we can add an initial token to this place to indicate this situation. The graph of the marked Petri Net for this example is shown in Figure A.5.

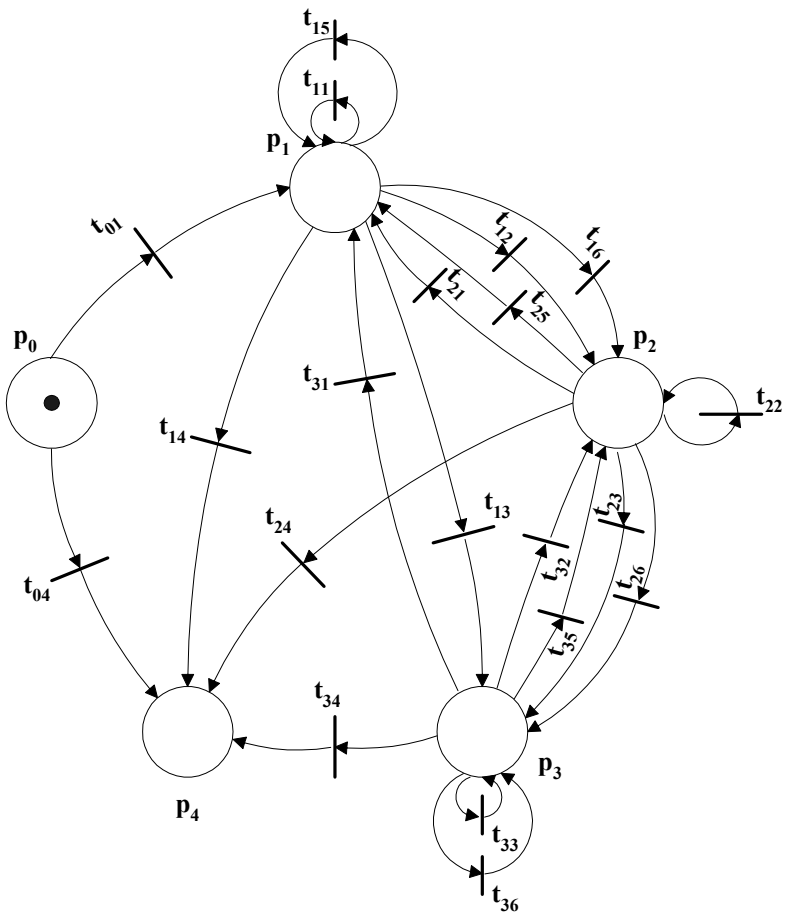



**Figure A.5 The graph of the marked Petri Net for the collaboration entities working on a presentation file in PowerPoint or Impress of OpenOffice.**

## Explanation of the Example

After one collaboration entity is instantiated, it is in the start state, which is represented by place $p_0$. The initial marking for places $p_0$, $p_1$, $p_2$, $p_3$, and $p_4$ is $\mu = (1, 0, 0, 0, 0)$, which indicates that there is only one token in the whole net, and it is in place $p_0$. From here, the instance can exit immediately without doing anything by going to place $p_4$ after the firing of transition $t_{04}$ (on event $a_4$ = "Exit"), or it can go to place $p_1$ after the firing of transition $t_{01}$ (on event $a_0$ = "Openfile;C:/file1.ppt"), at which the presentation

file is opened and by default is on slide 1. Because in the graph of the modeling, each transition $t_j$ has exactly one input and one output, by the execution rules of the Petri Net, the token in place $p_0$ can transit to either place $p_4$ or place $p_1$, with either the new marking $\mu' = (0, 0, 0, 0, 1)$ or $\mu' = (0, 1, 0, 0, 0)$.

From place $p_1$, the instance can go to place $p_2$ that is the state for slide 2 ($\mu' = (0, 0, 1, 0, 0)$), after the firing of transition $t_{12}$ (on event $a_2$ = "Goto;2") or after the firing of transition $t_{16}$ (on event $a_6$ = "Next"), or it can go to place $p_3$ that is the state for slide 3 ($\mu' = (0, 0, 0, 1, 0)$), after the firing of transition $t_{13}$ (on event $a_3$ = "Goto;3"), or it can remain in place $p_1$ ($\mu' = (0, 1, 0, 0, 0)$), after the firing of transition $t_{11}$ (on event $a_1$ = "Goto;1") or after the firing of transition $t_{15}$ on event $a_5$ = "Previous" (because for slide 1, there is no previous slide for it, so it stays), or it can go to place $p_4$ that is the state when the collaboration entity is killed ($\mu' = (0, 0, 0, 0, 1)$), after the firing of transition $t_{14}$ (on event $a_4$ = "Exit").

From place $p_2$, the instance can go to place $p_1$ that is the state for slide 1 ($\mu' = (0, 1, 0, 0, 0)$), after the firing of transition $t_{21}$ (on event $a_1$ = "Goto;1") or after the firing of transition $t_{25}$ (on event $a_5$ = "Previous"), or it can go to place $p_3$ ($\mu' = (0, 0, 0, 1, 0)$), after the firing of transition $t_{23}$ (on event $a_3$ = "Goto;3") or after the firing of transition $t_{26}$ (on event $a_6$ = "Next"), or it can remain in place $p_2$ ($\mu' = (0, 0, 1, 0, 0)$), after the firing of transition $t_{22}$ (on event $a_2$ = "Goto;2"), or it can go to place $p_4$ ($\mu' = (0, 0, 0, 0, 1)$), after the firing of transition $t_{24}$ (on event $a_4$ = "Exit").

From place $p_3$, the instance can go to place $p_1$ ($\mu' = (0, 1, 0, 0, 0)$), after the firing of transition $t_{31}$ (on event $a_1$ = "Goto;1"), or it can go to place $p_2$ ($\mu' = (0, 0, 1, 0, 0)$), after the firing of transition $t_{32}$ (on event $a_2$ = "Goto;2") or after the firing of transition $t_{35}$ (on event

$a_5$ = "Previous"), or it can remain in place $p_3$ ($\mu'$ = (0, 0, 0, 1, 0)), after the firing of transition $t_{33}$ (on event $a_3$ = "Goto;3") or after the firing of transition $t_{36}$ on event $a_6$ = "Next" (because for slide 3, there is no next slide for it, so it stays), or it can go to place $p_4$ ($\mu'$ = (0, 0, 0, 0, 1)), after the firing of transition $t_{34}$ (on event $a_4$ = "Exit").

Place $p_4$ is the state when the collaboration entity is ended. It is not an input for any of the transitions in the graph. So, when the token comes here, or when the new marking becomes $\mu'$ = (0, 0, 0, 0, 1), the execution of the Petri Net is done. Nothing will ever happen from here.

## Properties of the Example

This example has followed the Petri Net execution rules and has demonstrated several properties of the Petri Net for modeling of the collaboration entities. The properties of the net for the example include the following areas.

- It is **safe**.
- It is **strictly conservative**.

It is safe, because there is only one token in the marked Petri Net moving around among the places as the execution of the Petri Net is going on, so the requirement for the net to be safe is satisfied. It is strictly conservative, because there is only one token in the marked Petri Net moving around among the places as the execution of the Petri Net is going on, so the summation of all the tokens in the net at any marking remains constant, 1.

# A.9 Further Discussion on Petri Nets for Collaboration Entities

The execution of a Petri Net is run by the next-state function δ, which takes the current marking μ and an enabled transition $t_j$ in the marking as parameters and results in a new marking μ', as $\delta(\mu, t_j) = \mu'$. In the example of section A.8, the initial marking is μ = (1, 0, 0, 0, 0), and the transition $t_{01}$ is enabled; so it can fire and reach the new marking μ' through the next-state function $\delta(\mu, t_{01}) = \mu'$, where μ' = (0, 1, 0, 0, 0).

Along with the execution of the Petri Net, two sequences of information are produced: the sequence of markings ($\mu^0$, $\mu^1$, $\mu^2$, …) and the sequence of transitions ($t_{j0}$, $t_{j1}$, $t_{j2}$, …). The relationship between the two sequences is $\delta(\mu^k, t_{jk}) = \mu^{k+1}$ (for k = 0, 1, 2, …). Both sequences provide records of the execution of the Petri Net.

From the example of section A.8, we have noticed that, the markings of the Petri Net for the collaboration entities have a special characteristic: there is only one position in each marking that is non-zero; it is 1, and it indicates that the only token is in the corresponding place; the rest of the positions in the marking have the value 0. So, for clarity, we might as well say that the token is in certain place instead of giving the marking, in order to identify the state of the net. Further more, we can use a more succinct representation, say, $L_0$ for (1, 0, 0, 0, 0), $L_2$ for (0, 0, 1, 0, 0), and so on. However, the sequence of markings in an execution when written down in column or row becomes a sparse matrix. Existent algorithms on sparse matrices can be used to find out some patterns.

On one hand, given the initial marking $\mu^0$ and the sequence of transitions ($t_{j0}$, $t_{j1}$, $t_{j2}$, …) in the execution of a Petri Net, we can derive the corresponding marking sequence ($\mu^0$, $\mu^1$, $\mu^2$, …); on the other hand, given the sequence of markings ($\mu^0$, $\mu^1$, $\mu^2$, …), we can derive the corresponding transition sequence ($t_{j0}$, $t_{j1}$, $t_{j2}$, …), except for some degenerate cases. The degenerate cases in the graph of Figure A.5 are the reflexive circles on the places, or the transitions on the reflexive circles, such as $t_{11}$ and $t_{15}$ on place $p_1$, $t_{33}$ and $t_{36}$ on place $p_3$, and $t_{22}$ on place $p_2$. Take $t_{11}$ as an example. The marking that enables it is $\mu$ = (0, 1, 0, 0, 0). After the firing of $t_{11}$ according to the execution rules, the new marking is still the same, and the token that indicates the state is still in place $p_1$. Both the firings of $t_{11}$ and $t_{15}$ will produce the same result; therefore, based on the sequence of markings, we can not be sure which transition was fired. However, the degenerate cases can still be meaningful in situations as follows.

Suppose that a user is navigating through the slides in a presentation file using the "Previous" (or "Next") button without noticing that the current slide is the first (or the last); then the firing of the transition $t_{15}$ (or $t_{36}$) can keep the current slide without causing a logical error.

Again, suppose that a slide is being annotated in a collaboration session, and the speaker wish to erase all the annotations made on the slide after some time; the speaker can click on the thumb-nail for the slide on the outline view of the presentation file and lets the slide go to itself so as to refresh the slide.

An execution of a Petri Net can be recorded as a transition sequence $\sigma' \in T^*$. Any prefix of $\sigma'$, denoted as $\sigma$, contains enough information to get to a specific state of the net from the initial state. Hence, with the initial marking $\mu$ and the prefix transition sequence $\sigma$, the extended next-state function $\Delta: N^n \times T^* \rightarrow N^n$ for a Petri Net $C = (P, T, I, O)$ can be used to map the pair $(\mu, \sigma)$ into a new marking $\mu'$ (or a new state of the net), as in

$$\Delta (\mu, \sigma) = \mu'$$

It can be used in direct access, sequential access, and reverse indexing access of information, based on its induction definition described previously. It is similar to the description we have made in the part of DFA in *chapter 2*, so we do not bother to elaborate it here again. The transition sequences $\sigma' \in T^*$ in all the executions of a Petri Net form a language. It is similar to the description we have made in the part of DFA in *chapter 2*, so we would like to leave it alone.

# A.10 Comparisons

We have so far described the using of the DFA and the Petri Net in modeling the collaboration entities. Both of them can be used to illustrate the state transitions and the synchronization of states between the entities of a type in a collaboration session. The DFA can be represented as a five-tuple structure, a transition diagram, or a transition table. The Petri Net can be represented as a four-tuple Petri Net structure or a Petri Net graph.

The transition diagram of a DFA has circles representing the states and directed arcs (with event messages on the arcs) representing the state transitions. The Petri Net graph has a set of places and a set of transitions, uses the firings of transitions to represent the events' occurrences (and hence the changes of states), and uses the markings of tokens to represent the states of the net. In our example for the modeling of the collaboration entities, the only token in a place indicates the current state, similar to a circle indicating the current state in the DFA. The token makes the current state apparent.

The Petri Net graph has the bipartite representation of the places and the transitions, and the transitions can suggest more information such as the locality of the events. The markings can suggest detailed resource allocations, and the inputs and the outputs of the transitions can suggest detailed resource requests and releases. However, as we have noticed in the graph of Figure A.5, the Petri Net graph is much more complicated than the equivalent transition diagram of the DFA that we have described earlier in Figure 2.1 in *chapter 2* (we have used the same example in the two contexts to have implicit comparisons).

The transition diagram of the DFA is much simpler and easier to understand, and, the best part for our purpose, it makes the event messages apparent by attaching them on the directed arcs, while in the Petri Net, the event messages are hidden. The main tenet of this dissertation is about collaboration between the collaborating entities on event messages. Therefore, in this dissertation, we would like to use the DFA in all the texts, even though the Petri Net representation has its own strengths and beauty in modeling the executions of the collaboration entities.

# Appendix B

# A Description of the Implementation of Collaborative ReviewPlus

In this text, we use the implementation of the Collaborative ReviewPlus as an example to describe some issues of the collaborative applications, mainly focusing on event and logic with the applications. We will see that the Master client and the Participant clients share a common Deterministic Finite Automaton (DFA) in a collaboration session, have the same logic with regard to the state transitions, and converge on the same state on each event. We describe the issues as follows.

# B.1 Units and Unity

We have developed the collaborative ReviewPlus applications – the Master and Participant collaboration entities – from the original ReviewPlus application, without changing the overall logic related to state transitions. So the applications have the same logic with regard to the state transitions on events. The logic corresponds to the transition function δ or the extended transition function Δ of the DFA. The logic is composed of many IDL routines – procedures and functions with unique names. We can think of the routines as the building blocks or *units* of the logic and the logic as the *unity* of the routines. So, routines are the units, and δ or Δ is the unity of the units. On an event, only one or some routines are executing to do the transition; in other words, only one part or some parts of the unity are actually functioning, but we can indistinguishably say that δ or Δ is reacting on the event and transiting to the next state.

# B.2 Divergence and Convergence

The Master and Participant collaboration entities are designed for different purposes, in different architectures, implementing mechanisms, and shapes of codes; they are divergent. At the same time, they have the same logic as to the state transitions on events and get to the same state at the end of the process of each event; they are convergent.

Let us describe this in more detail using the implementation of the collaborative ReviewPlus applications as an example (It is similar for the others). On the Master client, each widget that fires event is associated with an event handler – either a procedure or a function – in the widget construction programs, which are registered at the end of the

constructions with the IDL system routine “xmanager.pro.” This system routine is managing the life-cycle of the widgets and listening for events from them. Whenever a widget is triggered by the user through the interface, the system automatically gathers the information for the event, fits the information in the event structure, and invokes the event handler with the event structure as the only parameter.

We add the code for collaboration here at the beginning of each event handler to capture the event and get the information of it for every field of the event structure, convert them into flat strings, serialize them into a delimited string along with the names of the event structure and the event handler, and send this result string to the NaradaBrokering (NB) message broker for broadcasting to the Participant clients. The NB broadcasts the string to the Participant client and saves the string in a public variable (which is one element of a synchronized linked list added in one of the NB’s interface class) and also updates the event flag variable (which reflects the number of strings saved).

The Participant client is developed using a Polling Structure. It has a main loop that is constantly polling the public variable – testing the event flag – to see if it is non-zero; if it is, then the client removes a string from the head of the linked list to do further process. In the process, the client parses the string on the delimiter to get all the field pieces, the event structure name (or widget name), and the event handler name, converts the field pieces to native type values of the event structure, constructs the event structure using these values (according to the event structure name), and finally renders the display by calling the event handler routine with the event structure as the parameter (according to the event handler name). As to the interactive input value to an input field such as the text

field, on the Master client, the user input the value physically; on the Participant client, after it gets the value, it sets the value in the input field programmatically.

From the descriptions above, we can see that the entities of the Master client and the Participant client diverge in the shapes of codes, architectures, implementing mechanisms, and purposes. They are in diversity under the goal of collaboration. However, on each event, they have the same input value in the input field (if there is one), call the same routine of the event handler with the same event structure as the parameter, and have the same output displays at the end of the process of the event; in other words, they converge on the same state of the DFA on each event, from the start state to the final accepting state, which is a well-defined session.

## B.3 Collaboration on Event and Transition Function

We now describe the collaboration between the entities of the Master client and the Participant client using pieces of code from the collaborative ReviewPlus applications, mainly focusing on event and the transition function. Instead of using exhausting enumeration of each and every widget cases in the interfaces, we give some typical and interesting ones that sufficiently illustrate the idea of collaboration in terms of convergence on the same state of the DFA at the end of the process of each event, with the transition function doing the real job of state transitions. Since the output displays of both the Master client and the Participant client are the same at each event, we just show a single set of image captures in the demonstration at each step. We begin with the

invocation of the collaboration entities, as shown in Figure B.1. This corresponds to the start state $q_0$ of the DFA.

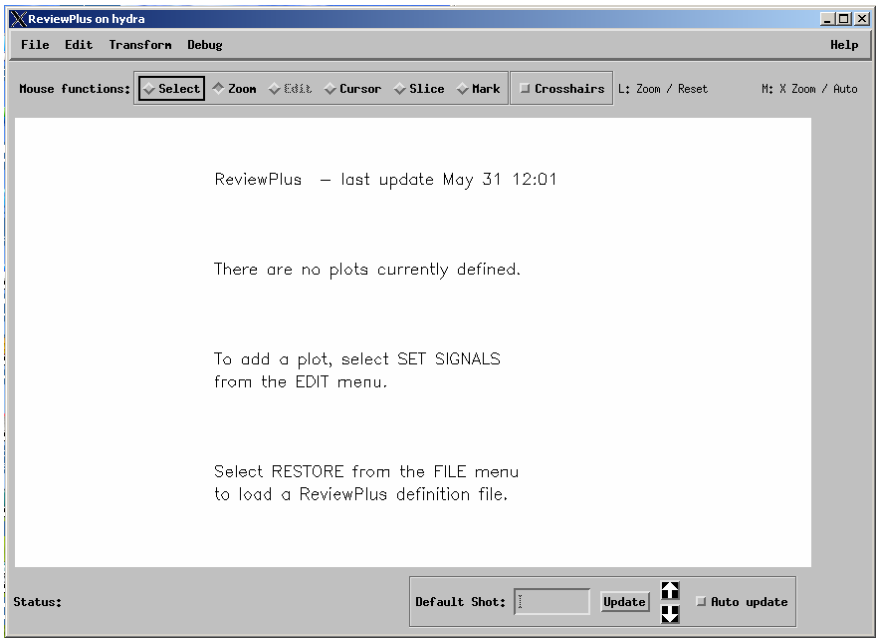



**Figure B.1 The initial interface and display of ReviewPlus.**

From this interface on the Master client, if we click on the "Edit" item from the main menu, a sub-menu will appear, as shown in Figure B.2.

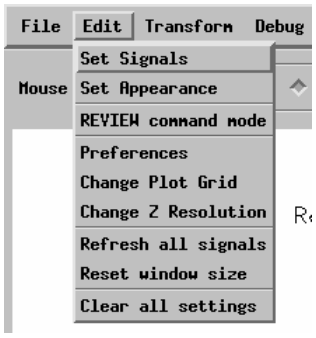



**Figure B.2 A sub-menu from the main menu of ReviewPlus.**

If we then click on the “Set Signals” item from the sub-menu, an event is fired. This is a button widget, and an event handler routine is defined for the event. We describe the pieces of code for both the Master client and the Participant client in achieving collaboration in response to this event, as follows.

**The Master Client Side**

- Widget creation

```
x = widget_button(mEdit, value='Set Signals', $
                  event_pro='ReviewPlus_SignalDialog_event')
```

From the code above, we know that this button widget has `'Set Signals'` as its value shown on its appearance and is associated with an event procedure named `'ReviewPlus_SignalDialog_event'`. When the button is clicked, the procedure is called by the IDL system.

- Definition of event structure for widget

Here is the definition of the event structure for widget button:

```
{WIDGET_BUTTON, ID:0L, TOP:0L, HANDLER:0L, SELECT:0L}
```

It has a name WIDGET_BUTTON and 4 fields – ID:0L, TOP:0L, HANDLER:0L, and SELECT:0L, each with a field name, a colon, and a type value. In this case all the values of the fields are of `long` type indicated by the suffix letter L.

SELECT: If the button is pressed, the value is 1; if it is released, the value is 0.

- Event handler

```
pro ReviewPlus_signaldialog_event,event
;;;;;;;; collaboration code added ;;;;;;;;
eventMessage = "ReviewPlus_signaldialog_event;"+"WIDGET_BUTTON;"+"ID;"$
  +string(event.ID)+";TOP;"+string(event.TOP)+";HANDLER;"$
  +string(event.HANDLER)+";SELECT;"+string(event.SELECT)
  COMMON BROKER, joChat2
  joChat2 -> writeMessage, eventMessage
;;;;;;;; end of collaboration code ;;;;;;;;
  widget_control,event.top,get_uvalue=info
  info.oReview->SignalDialog
end
```

From the code above we can see that the collaboration code captures the event and gets its field data from `event.ID, event.TOP, event.HANDLER,` etc., converts the data into strings, and serializes the strings into a semicolon delimited string, along with the event structure name `"WIDGET_BUTTON"` and the event handler name `"ReviewPlus_signaldialog_event"`. This result string is the event message and is sent to the NB broker for broadcasting to the Participant client.

**The Participant Client Side**

- Parsing of event message

```
            result = STRSPLIT(uval, ';', COUNT=count, /EXTRACT,
/PRESERVE_NULL)
            which_event = result[0]
            which_widget = result[1]
```

The next event message string for the Participant client to process is saved in the variable `uval`. The IDL system function `STRSPLIT` is called to parse it with `';'` as the delimiter. All the pieces of information around the delimiter are extracted and saved in the array `result` with the null string preserved as a piece, and the total number of them is saved in the variable `count`. The event handler name is in `result[0]` or `which_event`, and the event structure name (or widget name) is in `result[1]` or `which_widget`. The rest of the pieces are all for the fields of the event structure and are saved in the rest elements of the array starting with `result[2]`.

- Conversion to IDL native types

```
            FOR i=2, count-1, 2 DO BEGIN
               IF (result[i] EQ 'ID') THEN BEGIN
                  id_name = 'ID'
                  id_value = long(result[i+1])
               ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
                  top_name = 'TOP'
                  top_value = long(result[i+1])
               ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
                  handler_name = 'HANDLER'
```

```
      handler_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'SELECT') THEN BEGIN
      select_name = 'SELECT'
      select_value = long(result[i+1])
                        :
   ENDIF
ENDFOR
```

The code above converts the information (in string) of the fields of the button event structure to the IDL native types; each pair of the strings, i.e., those stored in `result[i]` and `result[i+1]`, decide the field's value and the type of the value, with the former indicating the name and type of the value (due to the unique association of a name with a type, the name alone can also indicate a type, e.g., `ID` is a `long` type) and the latter the value in string. In this case, all the values of the fields are of `long` type; therefore the strings are converted to the IDL type `long`.

- Construction of event structure

```
IF (which_widget EQ 'WIDGET_BUTTON') THEN $
   event_structure = {WIDGET_BUTTON,id:id_value,$
   top:top_value,handler:handler_value,select:select_value}$
ELSE IF ...
```

The code above constructs the widget button event structure using the converted native values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`.

- Invocation of the routine of event handler

```
...
ELSE IF (which_event EQ 'ReviewPlus_signaldialog_event') THEN BEGIN
           ReviewPlus_signaldialog_event, event_structure
     ENDIF ELSE IF ...
```

The code above calls the routine of the event handler `ReviewPlus_signaldialog_event` with the constructed event structure `event_structure` as the only parameter.

**Step Summary**

In the process of the event, both the Master client and the Participant client call the same routine – the event handler `ReviewPlus_signaldialog_event` – that is a unit of the transition function δ, with the event structure as the only parameter. The event message acts as the messenger, the information source, and the coordinator. With $\delta (q_0, a_0) = q_1$, the Master client and the Participant client converge on the same state $q_1$ of the DFA (on the event message $a_0$) at the end of the process of the event; therefore they have the same output displays, as in Figures B.3 and B.4, which are two parts of a big interface.

Note that, inside an event handler, other routines can be called in any sequence and order, which we don't have to worry about but just think of the whole as the encapsulation and abstraction of the event handler.

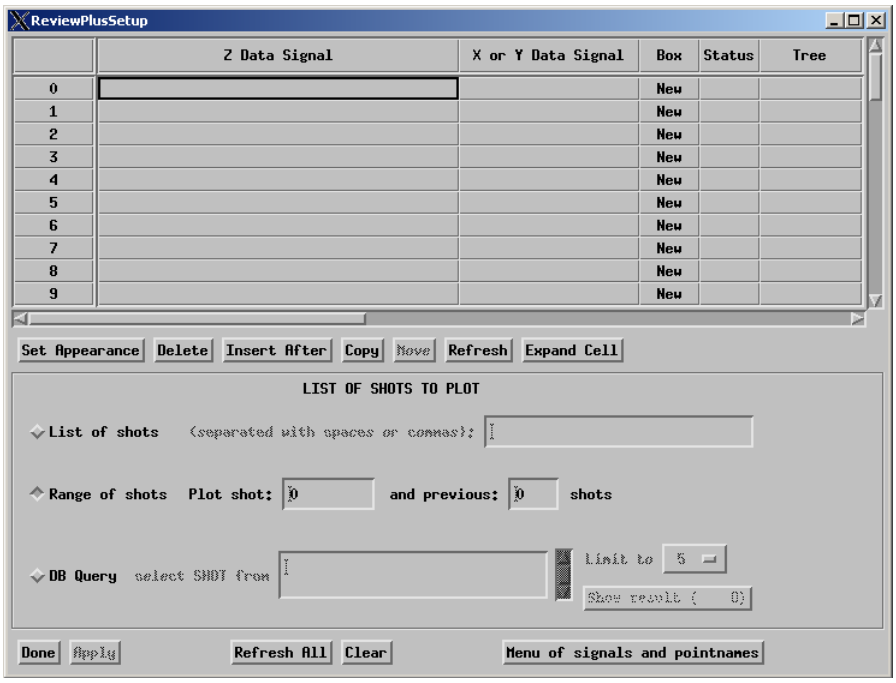



**Figure B.3 A part of a big interface in ReviewPlus for setting up and managing of signals.**

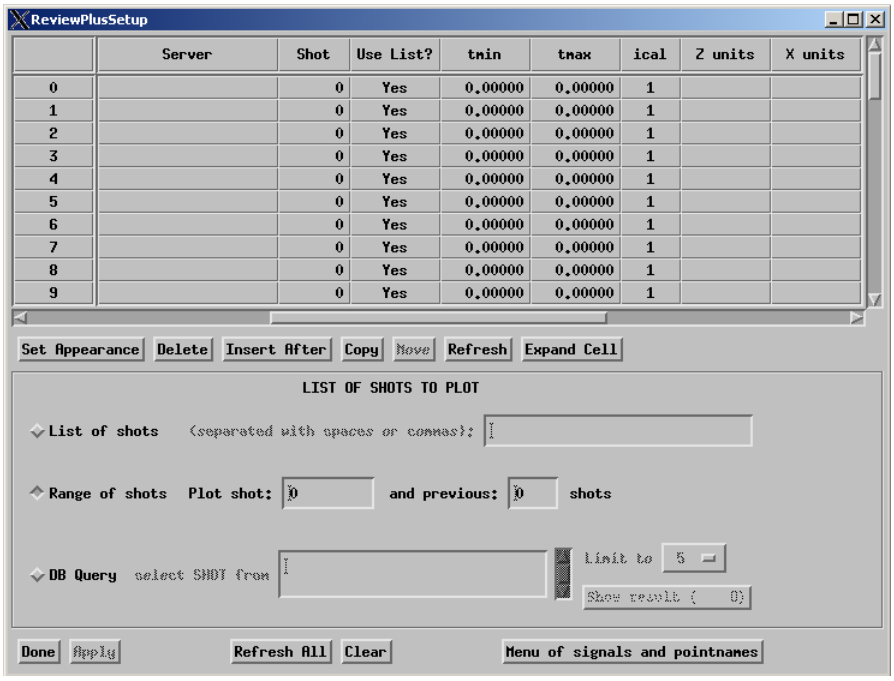



**Figure B.4 Another part of the big interface in ReviewPlus for setting up and managing of signals.**

At the upper part of this interface there is a widget table. Different types of events can happen there, such as *Insert Single Character, Insert Multiple Characters, Delete Text, Select Text, Select Cell, Change Row Height,* and *Change Column Width*, each with a different definition of event structure. Some of the events are of time sequence priority. For example, in order to enter a single character or multiple characters, we must first select the cell, and there must be some contents in the cell before we can delete or select the text. However, the process for each event is similar. So, not loosing generality, we can just choose to describe the *Insert Single Character* event.

Suppose we enter some data in row 0 of the widget table. Let us input “ip” in the cell under the column “Z Data Signal” in Figure B.3 and input “104276” in the cell under the column “Shot” in Figure B.4. In this case, each character entered including the line feed

and the carriage return will cause an event and therefore a state transition in the DFA. We just describe one of them in representation because the rest are the same in process.

**The Master Client Side**

- Widget creation

```
  oColumns = objarr(17)
  oColumns[0] = obj_new("GATableColumnEdit", label="Z Data Signal",
width=300)
...
  oColumns[6] = obj_new('GATableColumnEdit', label="Shot", width=60,
alignment=2)
...
  oTable = obj_new("GATable", self.wTLB, oColumns, Y_SCROLL_SIZE=10,
X_SCROLL_SIZE=5, /RESIZEABLE_COLUMNS)
...
  info = self->_GetColumnInfo()
  self.wID = widget_table(self.wParent, $
                          column_labels=info.label, $
                          /edit, /all_events, $
                          ...
                          event_pro='GATable_event', uvalue=self)
```

From the code above, we can see that this table widget is associated with an event procedure named `'GATable_event'`, and whenever an event is occurred, the procedure is called by the IDL system; we can see that this table widget is editable, which is indicated

by the keyword `/edit`; we can see that an event is fired whenever the contents of the text field of a cell has changed, which is indicated by the keyword `/all_events`. This table consists of 17 columns; each is an object of a module, or class. The listed one that is of interest to us is `GATableColumnEdit`. The table itself `oTable` is an aggregated object consisting of the column objects; during its instantiation `obj_new("GATable", ...)`, it calls `widget_table(...)` to build up the widget.

- Definition of event structure for widget

Here is the definition of the *Insert Single Character* event structure for widget table:

{WIDGET_TABLE_CH, ID:0L, TOP:0L, HANDLER:0L, TYPE:0, OFFSET:0L, CH:0B, X:0L, Y:0L}

It has a name WIDGET_TABLE_CH and 8 fields. The field CH:0B is of `byte` type indicated by the suffix letter B; the field TYPE:0 is of `int` type indicated by a value only; the others are of `long` type.

TYPE: The event type from the widget table.

CH: The ASCII value of the inserted character.

OFFSET: The zero-based character position after insertion.

X: The zero-based column address of the cell in the widget table.

Y: The zero-based row address of the cell in the widget table.

- Event handler

```
pro GATable_event, ev
```

```
;;;;;;;; collaboration code added ;;;;;;;;
   COMMON BROKER, joChat2
   case (ev.TYPE) of
      0 : BEGIN
            eventMessage =
              "GATable_event;"+"WIDGET_TABLE_CH;"+"ID;"+string(ev.ID)$
             +";TOP;"+string(ev.TOP)+";HANDLER;"+string(ev.HANDLER)$
             +";TYPE;"+string(ev.TYPE)+";OFFSET;"+string(ev.OFFSET)$
             +";CH;"+string(ev.CH)+";X;"+string(ev.X)+";Y;"+string(ev.Y)
            joChat2 -> writeMessage, eventMessage
          END
      ...
   endcase
;;;;;;;; end of collaboration code ;;;;;;;;
  widget_control,ev.id,get_uvalue=oSelf
  oSelf->_HandleEvent,ev
end
```

From the code above, we can see that the collaboration code captures the event and gets the field data from `ev.ID, ev.TOP, ev.HANDLER,` etc., converts the data into strings, and serializes the strings into a semicolon delimited string, along with the event structure name (`"WIDGET_TABLE_CH"`) and the event handler name (`"GATable_event"`). This result string is the event message and is sent to the NB broker for broadcasting to the Participant clients. The event handler handles different types of widget table events, such as *Insert Single Character, Insert Multiple Characters, Delete Text,* and *Select Cell*,

depending on the event structure's field value `ev.TYPE` to decide the current case. We just list the one in question (*Insert Single Character*) above.

**The Participant Client Side**

- Parsing of event message

Same as previously described.

- Conversion to IDL native types

```
FOR i=2, count-1, 2 DO BEGIN
   IF (result[i] EQ 'ID') THEN BEGIN
      id_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
      top_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
      handler_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TYPE') THEN BEGIN
      type_value = fix(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'OFFSET') THEN BEGIN
      offset_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'CH') THEN BEGIN
      ch_value_array = byte(result[i+1])
      ch_value = ch_value_array[0]
   ENDIF ELSE IF (result[i] EQ 'X') THEN BEGIN
      x_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'Y') THEN BEGIN
      y_value = long(result[i+1])
```

```
                                    :
        ENDIF
    ENDFOR
```

The code above converts the information (in string) of the fields of the *Insert Single Character* event structure to the IDL native types; each pair of the strings, i.e., those stored in `result[i]` and `result[i+1]`, decide the field's value and the type of the value, with the former indicating the name and type (due to the unique association of a name with a type, the name alone can also indicate a type, e.g., `ID` is a `long` type) and the latter indicating the value in string. In this case, the field `TYPE` is of `int` type and the string is converted to the IDL type `fix`; the field `CH` is of `byte` type and the string is converted to the IDL type `byte`; then the first element of the byte array is pulled out for the character; the other fields are of `long` type and therefore the strings are converted to the IDL type `long`.

- Construction of event structure

```
IF (which_widget EQ 'WIDGET_TABLE_CH') THEN $
    event_structure = {WIDGET_TABLE_CH,id:id_value,$
                       top:top_value,handler:handler_value,$
                       type:type_value,offset:offset_value,$
                       ch:ch_value,x:x_value,y:y_value}
```

The code above constructs the *Insert Single Character* event structure of widget table using the converted native values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`.

- Invocation of the routine of event handler

```
...
ELSE IF (which_event EQ 'GATable_event') THEN BEGIN
   GATable_event, event_structure
ENDIF ELSE IF ...
```

The code above calls the routine of the event handler `GATable_event` with the constructed event structure `event_structure` as the only parameter. The event handler `GATable_event` calls the following procedure indirectly to insert the new character programmatically into the contents of the current cell of the widget table.

```
pro GATableColumnEdit::_HandleInsertCharacter, cell, ev
    ...
    self->_AddToBuffer, string(ev.ch), ev.offset
    cell = {column:ev.x, row:ev.y}
    s = self.oTable->GetBuffer()
    self.oTable->SetCellValue, cell, s
    ...
end
```

**Step Summary**

In the process of the event, both the Master client and the Participant client call the same routine – the event handler `GATable_event` – which is a unit of the transition function δ, with the event structure as the only parameter. The event message acts as the messenger, the information source, and the coordinator. With $\delta(q_i, a_i) = r$, the Master client and the Participant client converge on the same state r of the DFA (on event message $a_i$) at the end of the process of the event; therefore they have the same output displays. In this case, whenever a character is entered on the Master side, it is immediately reflected on the Participant side, so that the entire detailed entering process is showing on both sides. We show the final display about this in Figure B.5.

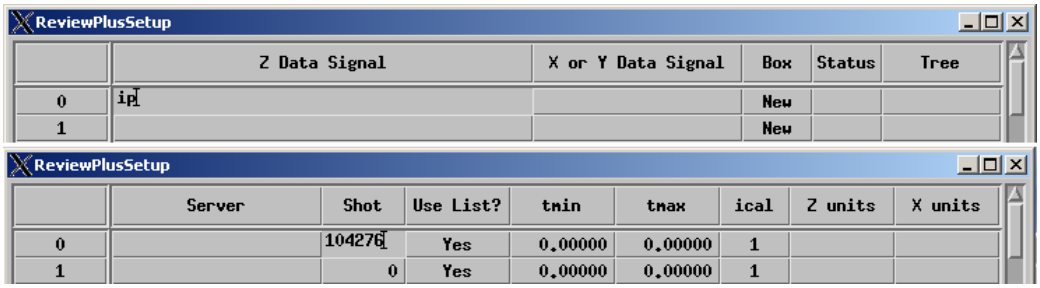



**Figure B.5 Data entered in the big interface in ReviewPlus for setting up and managing of signals.**

At this point, if we click on the widget button "Done" at the lower part of the interface in Figure B.4, we will get the result as shown in Figure B.6. We can omit the description of this event process because we have done button event previously, except in this case the routine of event handler `ReviewPlusSetup_done_event` is called on both sides. So, they converge on the same state r of the DFA.

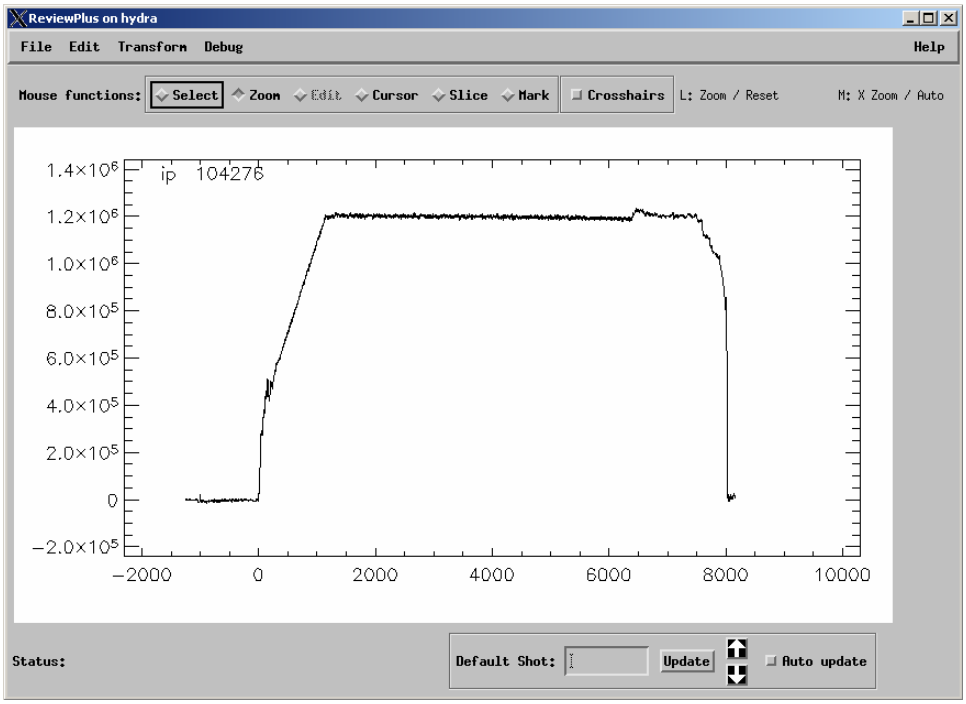



**Figure B.6 The display of a signal in ReviewPlus.**

If we click on the “Change Plot Grid” item from the sub-menu of Figure B.2, an event is fired. This is a button widget. The Master client and the Participant client will converge on a same state r of the DFA and have the same displays, as shown in Figure B.7.

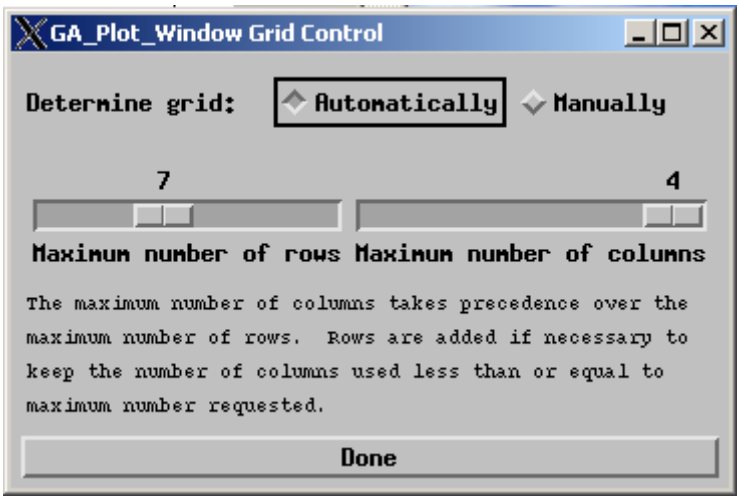



**Figure B.7 An interface of ReviewPlus for grid control of plot window.**

Let us click on the exclusive button “Manually” in Figure B.7, and we will get the display as follows in Figure B.8.

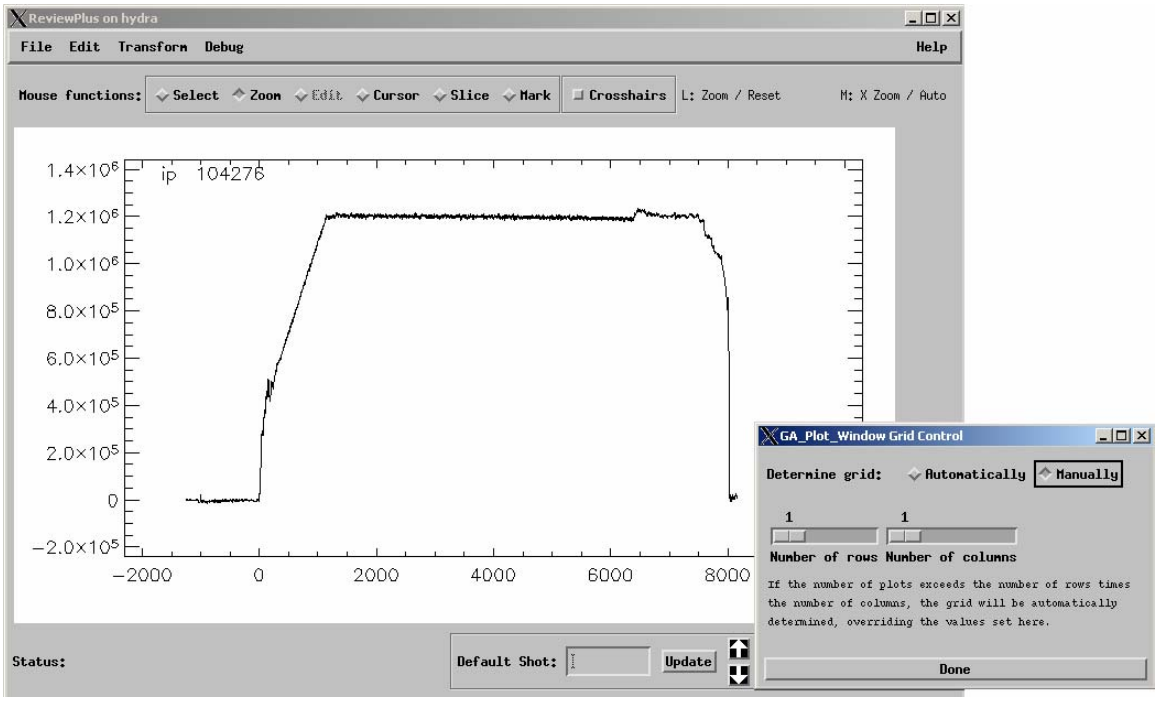


**Figure B.8 The display of a signal in the main interface of ReviewPlus and another interface of it for grid control of plot window.**

Some of the widgets in this interface for grid control are widget sliders. We describe one of them now. Let us drag the widget slider "Number of rows", change its value from 1 to 2, and see how this action will also affect the graphics output in the main interface.

**The Master Client Side**

- Widget creation

```
...
baseMan = widget_base(bboard, /column, map=0,$
             event_pro='ga_plot_window_griddialog_man_event')
r = widget_base(baseMan, /row)
slrow = widget_slider(r, title="Number of rows", min=1, max=16)
...
```

From the code above, we can see that this widget slider is associated with an event procedure named `'ga_plot_window_griddialog_man_event'` through the slider's ancestor widget in the widget hierarchy. The two base widgets determine the layout of the grid for component widgets. Whenever an event is occurred to the widget slider, it bubbles up through the hierarchy to the widget base with the event handler, and the event procedure is called with the event by the IDL system. The widget slider has the title `"Number of rows"` and the range of values with `min=1, max=16`.

- Definition of event structure for widget

Here is the definition of the event structure for widget slider:

{WIDGET_SLIDER, ID:0L, TOP:0L, HANDLER:0L, VALUE:0L, DRAG:0}

It has a name WIDGET_SLIDER and 5 fields. The field DRAG:0 is of `int` type indicated by a value only, and the others are of `long` type.

VALUE: The new value of the widget slider.

DRAG: The value that indicates whether the event is generated during or at the end of a slider drag, with the value 1 or 0.

- Event handler

```
pro ga_plot_window_griddialog_man_event,ev
;;;;;;;; collaboration code added ;;;;;;;;
   tag = tag_names(ev,/str)
   if (tag eq 'WIDGET_SLIDER') then begin
      eventMessage = "ga_plot_window_griddialog_man_event;"$
                     +"WIDGET_SLIDER;"+"ID;"+string(ev.ID)$
                     +";TOP;"+string(ev.TOP)+";HANDLER;"$
                     +string(ev.HANDLER)+";VALUE3;"+string(ev.VALUE)$
                     +";DRAG;"+string(ev.DRAG)
      COMMON BROKER, joChat2
      joChat2 -> writeMessage, eventMessage
   endif
```

```
;;;;;;;; end of collaboration code ;;;;;;;;
  widget_control,ev.top,get_uvalue=u
  widget_control,u.slrow,get_value=row
  widget_control,u.slcol,get_value=col
  u.self->UserSetGrid,[col,row],/draw
end
```

From the code above, we can see that the collaboration code captures the event and gets the field data from `ev.ID, ev.TOP, ev.HANDLER,` etc., converts the data into strings, and serializes the strings into a semicolon delimited string, along with the event structure name (`"WIDGET_SLIDER"`) and the event handler name (`"ga_plot_window_griddialog_man_event"`). This result string is the event message and is sent to the NB broker for broadcasting to the Participant clients.

**The Participant Client Side**

- Parsing of event message

Same as previously described.

- Conversion to IDL native types

```
FOR i=2, count-1, 2 DO BEGIN
   IF (result[i] EQ 'ID') THEN BEGIN
      id_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
      top_value = long(result[i+1])
```

```
      ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
         handler_value = long(result[i+1])
      ENDIF ELSE IF (result[i] EQ 'VALUE3') THEN BEGIN
         value3_value = long(result[i+1])
      ENDIF ELSE IF (result[i] EQ 'DRAG') THEN BEGIN
         drag_value = fix(result[i+1])
                              :
      ENDIF
   ENDFOR
```

The code above converts the fields' data (in string) of the widget slider event structure to the IDL native types; each pair of the strings, i.e., those stored in `result[i]` and `result[i+1]`, decide the field's value and the type of the value, with the former indicating the name and type (due to the unique association of a name with a type, the name alone can also indicate a type, e.g., `ID` is a `long` type) and the latter indicating the value in string. In this case, the field `DRAG` is of `int` type, and the string is converted to the IDL type `fix`. The other fields are of `long` type, and the strings are converted to the IDL type `long`. The field `VALUE` is of `long` type in this event structure, but in some other event structures, the field `VALUE` is used for `int` or `float`. To resolve this conflict in programming, we use `VALUE3` to exclusively indicate that it is of `long` type.

- Construction of event structure

```
...
ELSE IF (which_widget EQ 'WIDGET_SLIDER') THEN $
```

```
        event_structure = {WIDGET_SLIDER,id:id_value,$
                           top:top_value,handler:handler_value,$
                           value:value3_value,drag:drag_value}$
    ELSE IF ...
```

The code above constructs the event structure of widget slider using the converted native values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`.

- Invocation of the routine of event handler

```
...
ELSE IF (which_event EQ 'ga_plot_window_griddialog_man_event')$
     THEN BEGIN
          ReviewPlus_widget_slider_event, event_structure
          ga_plot_window_griddialog_man_event, event_structure
ENDIF ELSE IF ...
```

The code above first calls the routine of `ReviewPlus_widget_slider_event` with the constructed event structure `event_structure` to set up the value of the widget slider programmatically as the Master client; then it calls the routine of the event handler `ga_plot_window_griddialog_man_event` with the event structure to have the function as the Master client. The routine of `ReviewPlus_widget_slider_event` is listed below.

```
pro ReviewPlus_widget_slider_event,event
```

```
  widget_control,event.id,set_value=event.value
end
```

**Step Summary**

In the process of the event, both the Master client and the Participant client call the same routine – the event handler `ga_plot_window_griddialog_man_event` – which is a unit of the transition function δ, with the event structure as the only parameter. On the Participant client, an extra routine `ReviewPlus_widget_slider_event` is called to set up the value of the slider in order to simulate the slider drag on the Master client, so that the sliders on both the Master client and the Participant client have the same value and appearance. We can think of this extra routine as another unit of the transition function δ; hence in this process of the event, two units are called to get to the same state. With $\delta (q_i, a_i) = r$, the Master client and the Participant client converge on the same state r of the DFA (on event message $a_i$) at the end of the process of the event; therefore they have the same output display – in this case, the same appearance of the widget slider and the look of the output in the main interface, as shown in Figure B.9.

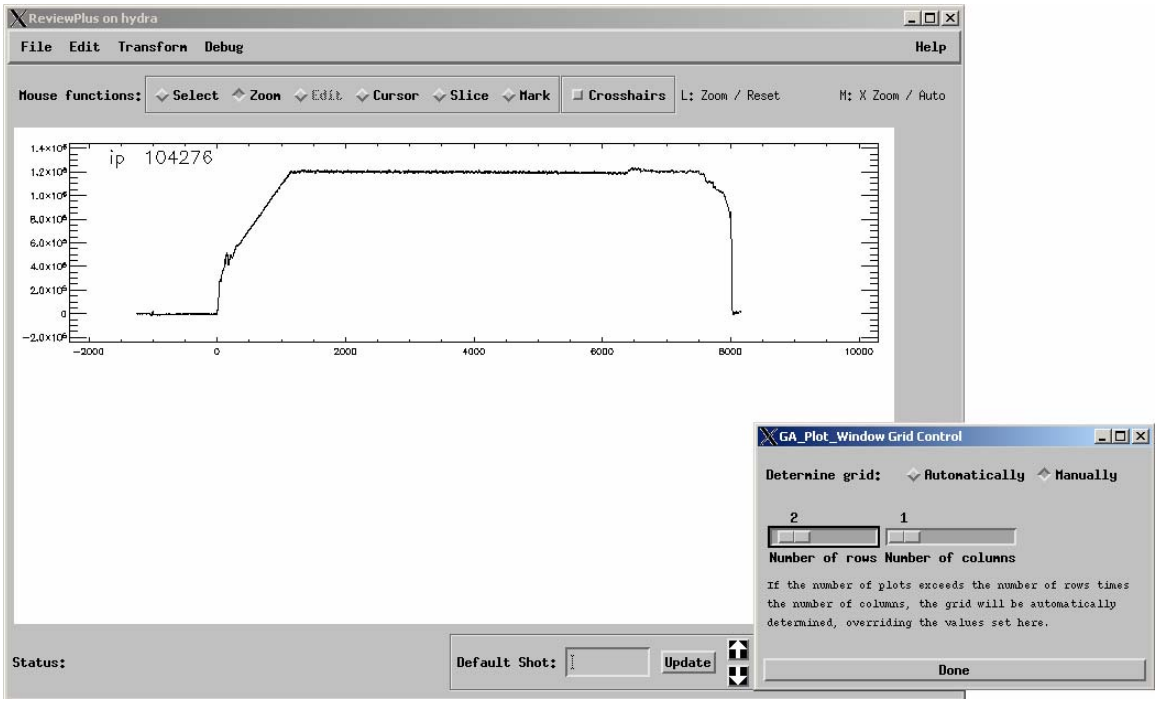



**Figure B.9 The effect to the graphics output in the main interface of ReviewPlus when dragging the widget slider "Number of rows" in the interface of grid control of plot window and changing its value from 1 to 2.**

If we click on the "Preferences" item from the sub-menu of Figure B.2, an event is fired. This is a button widget. The Master client and the Participant client will converge on a same state r of the DFA and have the same display, as follows in Figure B.10.

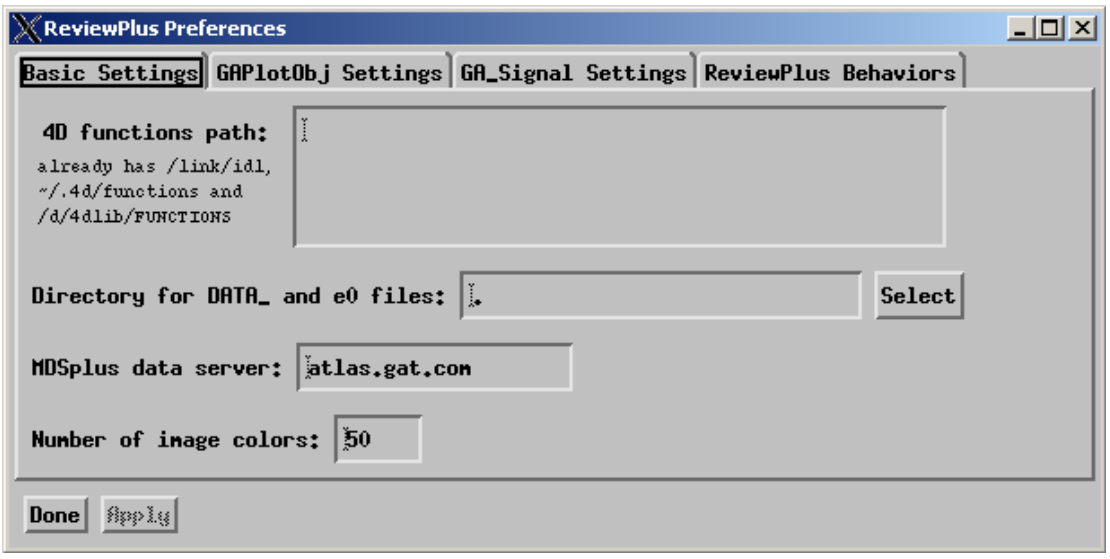



**Figure B.10 The ReviewPlus Preferences interface that mainly contains a tab widget.**

This interface mainly contains a new type of widget – tab widget. Tab widget usually contains multiple pages, each with a tab for labeling. Only one page can be displayed at a time in the widget's display area. We switch to a new page by selecting a different tab. The titles of the tabs above are "*Basic Settings*," "*GAPlotObj Settings*," "*GA_Signal Settings*," and "*ReviewPlus Behaviors*." Let us select the tab with the title "*GAPlotObj Settings*" and describe the process of this event in the collaboration entities. This action actually triggers two events – one related to the widget tab and the other related to a widget base in the upper level of the widget hierarchy.

A function event handler is associated with the tab widget. A function event handler not only processes the event happened to its associated widget, but also returns the event, possibly with modifications, to the parent widget in higher level of the widget hierarchy. The event keeps bubbling up through the hierarchy until it is consumed or swallowed by an associated event handler or by the IDL system at last otherwise. The function event

handler in charge of the widget tab processes the event; then it returns the event as a new event to the event handler associated with the widget base for further process to change the result of the output.

**The Master Client Side**

- Widget creation

```
...
wBase = widget_base(title="ReviewPlus Preferences", /column,$
               /floating, group_leader=self.wTLB, tlb_frame_attr=3)
wTabs = WIDGET_TAB(wBase, event_func='ReviewPlus_widget_tab_event')
wBasic = WIDGET_BASE(wTabs, TITLE='Basic Settings', /COLUMN)
self.oWindow->PreferencesDialog,wTabs,TITLE='GAPlotObj Settings'
wSigs = WIDGET_BASE(wTabs, TITLE='GA_Signal Settings', /COLUMN, $
                                                    /ALIGN_LEFT)
wBehaviors = WIDGET_BASE(wTabs, TITLE='ReviewPlus Behaviors',$
                                                      /COLUMN)
...
```

From the code above, we can see that this tab widget is associated with an event function named `'ReviewPlus_widget_tab_event'` and has 4 tabs with the title names "*Basic Settings*," "*GAPlotObj Settings*," "*GA_Signal Settings*," and "*ReviewPlus Behaviors*," respectively; we can see that the tabs are the children of the tab widget, and the tab widget is the child of the base widget `wBase` in the widget hierarchy.

- Definition of event structure for widget

Here is the definition of the event structure for widget tab:

{WIDGET_TAB, ID:0L, TOP:0L, HANDLER:0L, TAB:0L}

It has a name WIDGET_TAB and 4 fields. All the fields are of `long` type.

TAB: The zero-based index of the selected tab in the tab widget.

- Event handler

```
function ReviewPlus_widget_tab_event,event
   tag = tag_names(event,/str)
   if (tag eq 'WIDGET_TAB') then begin
      eventMessage = "ReviewPlus_widget_tab_event;"+"WIDGET_TAB;"$
                     +"ID;"+string(event.ID)+";TOP;"+string(event.TOP)$
                     +";HANDLER;"+string(event.HANDLER)$
                     +";TAB;"+string(event.TAB)
      COMMON BROKER, joChat2
      joChat2 -> writeMessage, eventMessage
   endif
   return, event
end
```

From the code above, we can see that the collaboration code captures the event, gets the field data from `event.ID, event.TOP, event.HANDLER,` etc., converts the data into strings, and serializes the strings into a semicolon delimited string, along with the event

structure name (`"WIDGET_TAB"`) and the event handler name (`"ReviewPlus_widget_tab_event"`). This result string is the event message and is sent to the NB broker for broadcasting to the Participant clients.

The main purpose of this event function is to capture the event and to get the event message when the user clicks on the tab such as "*GAPlotObj Settings*" on the Master client. Later on the event message acts as the messenger and the information source and coordinates the Participant client to change to the same tab programmatically. The second purpose of this event function is to return the event, without modification, to its parent widget, which is a base widget with the widget ID `wBase` and is associated with an event handler named `reviewplus_preferences_apply_event`, as listed below.

```
    xmanager, 'ReviewPlus_preferencesdialog', wBase, $
        event_handler='reviewplus_preferences_apply_event', /no_block
```

The event acts as a new event and is further processed and consumed by this event handler.

```
pro ReviewPlus_preferences_apply_event,event
;;;;;;;; collaboration code added ;;;;;;;;
  tag = tag_names(event,/str)
  if ...
     ...
  endif else if (tag eq 'WIDGET_TAB') then begin
    eventMessage = "ReviewPlus_preferences_apply_event;"+"WIDGET_TAB;"$
                  +"ID;"+string(event.ID)+";TOP;"+string(event.TOP)$
```

```
                    +";HANDLER;"+string(event.HANDLER)$
                    +";TAB;"+string(event.TAB)
  endif else ...
  COMMON BROKER, joChat2
  joChat2 -> writeMessage, eventMessage
;;;;;;;; end of collaboration code ;;;;;;;;
...
;;; other statements and commands
...
end
```

From the code above, we can see that the collaboration code captures the event, gets the field data from `event.ID, event.TOP, event.HANDLER,` etc., converts the data into strings, and serializes the strings into a semicolon delimited string, along with the event structure name (`"WIDGET_TAB"`) and the event handler name (`"ReviewPlus_preferences_apply_event"`). This result string is the event message and is sent to the NB broker for broadcasting to the Participant clients.

**The Participant Client Side**

- Parsing of event message

Same as previously described.

- Conversion to IDL native types

```
FOR i=2, count-1, 2 DO BEGIN
```

```
   IF (result[i] EQ 'ID') THEN BEGIN
      id_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
      top_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
      handler_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TAB') THEN BEGIN
      tab_value = long(result[i+1])
                        :
   ENDIF
ENDFOR
```

The code above converts the fields' data (in string) of the widget tab event structure to the IDL native types; each pair of the strings (i.e., those stored in `result[i]` and `result[i+1]`) decide the field's value and the type of the value, with the former indicating the name and type (due to the unique association of a name with a type, the name alone can also indicate a type, e.g., `ID` is a `long` type) and the latter indicating the value in string. In this case, all the fields are of `long` type; therefore the strings are converted to the IDL type `long`.

- Construction of event structure

```
...
ELSE IF (which_widget EQ 'WIDGET_TAB') THEN $
   event_structure = {WIDGET_TAB,id:id_value,top:top_value,$
                      handler:handler_value,tab:tab_value}$
```

```
        ELSE IF ...
```

The code above constructs the event structure of widget tab using the converted native values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`.

- Invocation of the routines of event handlers

```
...
ELSE IF (which_event EQ 'ReviewPlus_widget_tab_event') THEN BEGIN
          ReviewPlus_widget_tab_event, event_structure
ENDIF ELSE IF (which_event EQ 'ReviewPlus_preferences_apply_event')$
      THEN BEGIN
          ReviewPlus_preferences_apply_event, event_structure
ENDIF ELSE IF ...
```

The Master client sends out the event messages `'ReviewPlus_widget_tab_event…'` and `'ReviewPlus_preferences_apply_event…'` in that order. The Participant client processes them in the same order (we have a mechanism to guarantee this order). Therefore, the code above (within a loop) first calls the routine of `ReviewPlus_widget_tab_event` with the constructed event structure `event_structure` to change to the tab programmatically as that of the Master client; then it calls the routine of the event handler `ReviewPlus_preferences_apply_event` with the event structure to have the same function as that of the Master client. This is to reflect to the routine any changes in the previous page (from which we have switched by

clicking on the tab associated with the current page). The routine of `ReviewPlus_widget_tab_event` is listed below.

```
Pro ReviewPlus_widget_tab_event,event
  widget_control,event.id,set_tab_current=event.tab,/show
end
```

**Step Summary**

In the processes of the events, both the Master client and the Participant client call the same routines (with the event structure as the only parameter) in the same order – the event handlers `ReviewPlus_widget_tab_event` and `ReviewPlus_preferences_apply_event`, which are two units of the transition function δ. As we have noticed, the event handler `ReviewPlus_widget_tab_event` is of different shape and purpose on the Master client from those on the Participant client; on the Master client, it is a function, and it is for capturing the event, sending out the event message, and returning the event to the upper widget, while on the Participant client, it is a procedure, and it is for changing to the page of the desired tab. Nevertheless, it has the goal to coordinate both sides to have the same function and appearance at the end of the process of the event; in other words, it has the goal to coordinate both sides to converge on the same state r of the DFA. With $\delta(q_i, a_i) = r$, the Master client and the Participant client converge on the same states r of the DFA (on event messages $a_i$) at the end of the processes of the events; therefore they have the same output display – in this case, the same appearance of the widget tab and the same output in the main interface. We list the appearance of the widget tab in Figure B.11.

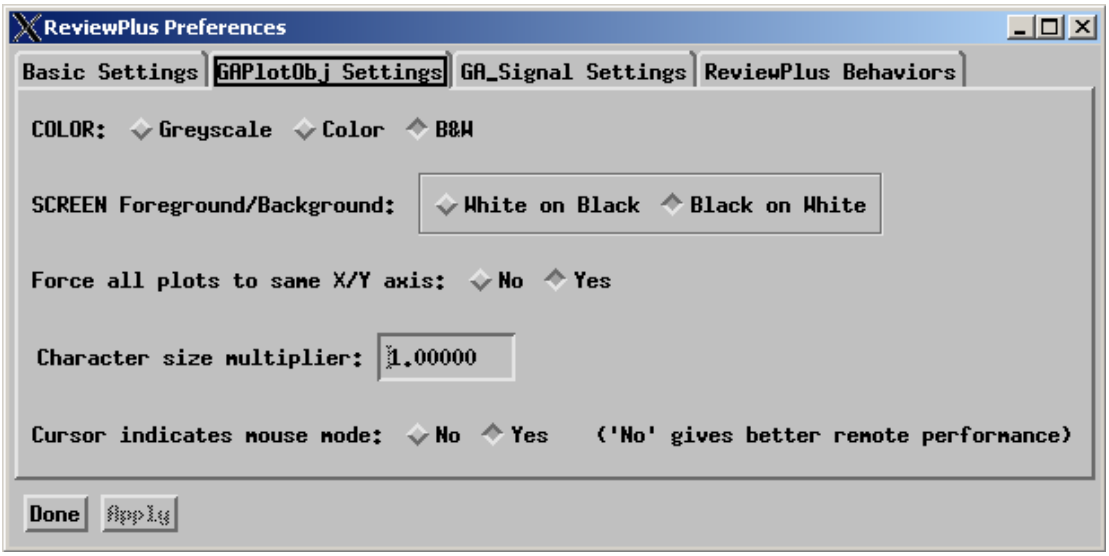



**Figure B.11 The ReviewPlus Preferences interface that currently shows the page under the tab of "GAPlotObj Settings."**

Now, in this interface, if we click on the button "*White on Black*" from the exclusive button group beside the title "*SCREEN Foreground/Background*," the button will be set; at the same time the other one "*Black on White*" will be unset; the colors of the foreground and the background of the output in the main interface will be exchanged. As the previous step that we have described, this action triggers several events in sequence; it is related to event functions and the bubbling up of the event through the widget hierarchy. We describe it below.

**The Master Client Side**

- Widget creation

```
pro ga_plot_window::PreferencesDialog,parent, title=title
   ...
   tlb = widget_base(parent, /column, frame=1, title=title, $
                     pro_set_value='ga_plot_window_preferences_set',$
                     event_func='ga_plot_window_pref_applyfunc_event')
   ...
   baseReverse = widget_base(tlb, /row,$
                   event_func='ReviewPlus_cw_bgroup_index_func_event')
   x = widget_label(baseReverse, value="SCREEN Foreground/Background: ")
   butsRev = cw_bgroup(baseReverse,['White on Black', 'Black on White'],
                       /exclusive, /row, /frame, /no_release)
   ...
```

From the code above, we can see that this compound widget CW_BGROUP is associated with an event function named `'ReviewPlus_cw_bgroup_index_func_event'` through its parent widget `widget_base` with ID `baseReverse`; we can see that the group of buttons in its base are exclusive (indicated by the keyword `/exclusive`) and are laid horizontally (indicated by `/row`). The parent of widget `baseReverse` is `widget_base` with ID `tlb`, and the parent of widget `tlb` is the widget with ID `parent`. If we list again some piece of the code we have described in the last step as follows, we can see their relationships.

```
    ...
    wBase = widget_base(title="ReviewPlus Preferences", /column,$
                   /floating, group_leader=self.wTLB, tlb_frame_attr=3)
    wTabs = WIDGET_TAB(wBase, event_func='ReviewPlus_widget_tab_event')
```

```
...
self.oWindow->PreferencesDialog,wTabs,TITLE='GAPlotObj Settings'
...
```

We can see that ID `parent` and ID `wTabs` are equivalent algebraically in execution, and the parent of widget `wTabs` is the base widget with ID `wBase`. For clarity, we list the related widgets in question along the path from the upper to lower levels of the widget hierarchy showing part of the relationships of the family, with each item having the format of *widget ID*, *widget type*, and *associated event handler*.

```
wBase, widget_base, pro ReviewPlus_preferences_apply_event,event
                                    ↑
  wTabs, WIDGET_TAB, function ReviewPlus_widget_tab_event,event
                                    ↑
tlb, widget_base, function ga_plot_window_pref_applyfunc_event,event
                                    ↑
                baseReverse, widget_base, function
            ReviewPlus_cw_bgroup_index_func_event,event
                                    ↑
                        butsRev, cw_bgroup,
```

- Definition of event structure for widget

Here is the definition of the event structure for CW_BGROUP widget:

```
{ID:0L, TOP:0L, HANDLER:0L, SELECT:0L, VALUE:0}
```

It has 5 fields; unlike most of the widgets, it doesn't have a structure name. The field VALUE is of `int` type, and the rest are of `long` type.

SELECT: The value passed through from the button event, with 1 indicating that the button is set and 0 otherwise.

VALUE: The value of the button. It can be "INDEX" (the index of the button in the base), "ID" (the widget ID of the button), "NAME" (the name of the button), or "BUTTON_UVALUE" (the user value for the button).

- Event handlers

Whenever an event happens to the CW_BGROUP widget `butsRev` (e.g., a button is set or unset in the base), it automatically bubbles up to the parent of `butsRev`, which is `baseReverse`, because `butsRev` doesn't have its own event handler. The `baseReverse` has an event function as follows to deal with the event.

```
function ReviewPlus_cw_bgroup_index_func_event,event
   tag = tag_names(event,/str)
   if (tag ne 'WIDGET_BASE') then begin
      eventMessage = "ReviewPlus_cw_bgroup_index_event;"$
                  +"CW_BGROUP_INDEX;"+"ID;"+string(event.ID)$
                  +";TOP;"+string(event.TOP)+";HANDLER;"$
                  +string(event.HANDLER)$
                  +";SELECT;"+string(event.SELECT)$
                  +";VALUE_CW_BGROUP;"+string(event.VALUE)
      COMMON BROKER, joChat2
```

```
      joChat2 -> writeMessage, eventMessage
   endif
   return, event
end
```

From the code above, we can see that the collaboration code captures the event, gets the field data from `event.ID, event.TOP, event.HANDLER,` etc., converts the data into strings, and serializes the strings into a semicolon delimited string, along with the event structure information string `"CW_BGROUP_INDEX"` (it is a CW_BGROUP widget and is created using `INDEX`, which is used to refer to a button in the base) and the event handler name (`"ReviewPlus_cw_bgroup_index_event"` to be called on the Participant client). This result string is the event message and is sent to the NB broker for broadcasting to the Participant clients.

The main purpose of this event function is to capture the event and to get the event message when the user clicks on a button from the button group in the base on the Master client. Later on the event message acts as the messenger and the information source and coordinates the Participant client to set or unset the same button programmatically as the Master client. The second purpose of this event function is to return the event, without modification, to its parent widget, which is a base widget with the widget ID `tlb` and is associated with an event handler named `ga_plot_window_pref_applyfunc_event`, as listed below.

```
function ga_plot_window_pref_applyfunc_event,event
  return,event
end
```

This function does nothing but returns the event to the widget in the upper level of the hierarchy. It contributes nothing to the state transitions of the DFA as long as the Participant client is concerned. So we would just leave it alone. From here, the event continues to bubble up to the parent of `tlb`, which is `WIDGET_TAB` with the widget ID `wTabs` and is associated with the function event handler named `ReviewPlus_widget_tab_event`. We have described previously the process and the action happened within the function event handler in the last step. We list again the part of interest to this step as follows.

```
function ReviewPlus_widget_tab_event,event
   tag = tag_names(event,/str)
   if (tag eq 'WIDGET_TAB') then begin
      eventMessage = ...
      ...
   endif
   return, event
end
```

Because CW_BGROUP widget is not a `WIDGET_TAB`, the mechanism of conditional test guarantees that no event message will be generated and sent out to the NB broker from this function event handler, but the `event` will be returned untouched to the parent widget. Once more, the event continues to bubble up to the parent of `wTabs`, which is `widget_base` with the widget ID `wBase` and is associated with the procedure event handler named `ReviewPlus_preferences_apply_event`. The event gets processed and

finally gets swallowed by this event handler. We list this procedure once again as follows only with the relevant part.

```
pro ReviewPlus_preferences_apply_event,event
;;;;;;;; collaboration code added ;;;;;;;;
    ...
    endif else begin
      eventMessage = "ReviewPlus_preferences_apply_event;"$
                   +"CW_BGROUP_INDEX;"+"ID;"+string(event.ID)$
                   +";TOP;"+string(event.TOP)+";HANDLER;"$
                   +string(event.HANDLER)$
                   +";SELECT;"+string(event.SELECT)$
                   +";VALUE_CW_BGROUP;"+string(event.VALUE)
    endelse
    ...
  COMMON BROKER, joChat2
  joChat2 -> writeMessage, eventMessage
;;;;;;;; end of collaboration code ;;;;;;;;
...
;;; other statements and commands
...
end
```

From the code above, we can see that the collaboration code captures the event, gets the field data from `event.ID, event.TOP, event.HANDLER,` etc., converts the data into strings, and serializes the strings into a semicolon delimited string, along with the event structure information string `"CW_BGROUP_INDEX"` (it is a CW_BGROUP widget and is

created using `INDEX`, which is used to refer to a button of the button group in the base) and the event handler name (`"ReviewPlus_preferences_apply_event"` to be called on the Participant client). This result string is the event message and is sent to the NB broker for broadcasting to the Participant clients.

**The Participant Client Side**

- Parsing of event message

Same as previously described.

- Conversion to IDL native types

```
FOR i=2, count-1, 2 DO BEGIN
   IF (result[i] EQ 'ID') THEN BEGIN
      id_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'TOP') THEN BEGIN
      top_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'HANDLER') THEN BEGIN
      handler_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'SELECT') THEN BEGIN
      select_value = long(result[i+1])
   ENDIF ELSE IF (result[i] EQ 'VALUE_CW_BGROUP') THEN BEGIN
      cw_bgroup_value = result[i+1]
                        :
   ENDIF
ENDFOR
```

The code above converts the fields' data (in string) of the CW_BGROUP widget event structure to the IDL native types, except the `VALUE` field. Because it can be "INDEX" (the index of the button in the base), "ID" (the widget ID of the button), "NAME" (the name of the button), or "BUTTON_UVALUE" (the user value for the button), depending on how the widget is created. We need further information from the event message to decide its type. For example, if we get the event structure information string `"CW_BGROUP_INDEX,"` then we know it is an index, and so its type is of `int`. So, for the moment, we just save it in a variable `cw_bgroup_value` and wait for further process in the construction of the event structure.

- Construction of event structure

```
IF (which_widget EQ 'CW_BGROUP_INDEX') THEN BEGIN
   bgroup_value = fix(cw_bgroup_value)
   event_structure = {CW_BGROUP_INDEX,id:id_value,$
                      top:top_value, handler:handler_value,$
                      select:select_value, value:bgroup_value}
ENDIF
```

The code above constructs the event structure of CW_BGROUP widget with the `value` field in `INDEX` type, using the converted native values for each field, with the field name followed by a colon and then by the value, as in `id:id_value`. The type of the `value` field is finally decided with the information indicated in `'CW_BGROUP_INDEX'`, and the value is converted from string to integer via `fix()`.

- Invocation of the routines of event handlers

```
...
IF (which_event EQ 'ReviewPlus_cw_bgroup_index_event') THEN BEGIN
      ReviewPlus_cw_bgroup_index_event, event_structure
      ...
...
IF (which_event EQ 'ReviewPlus_preferences_apply_event') THEN BEGIN
      ReviewPlus_preferences_apply_event, event_structure
      ...
```

The Master client sends out the event messages `'ReviewPlus_cw_bgroup_index_event…'` and `'ReviewPlus_preferences_apply_event…'` in that order. The Participant client processes them in the same order (we have a mechanism to guarantee this order). Therefore, the code above (within a loop) first calls the routine of `ReviewPlus_cw_bgroup_index_event` with the constructed event structure `event_structure` as the parameter to set or unset the button of the CW_BGROUP widget programmatically as that of the Master client; then it calls the routine of the event handler `ReviewPlus_preferences_apply_event` with the event structure `event_structure` as the parameter to have the same function as that of the Master client. This is to exchange the colors of the foreground and the background of the output display within the main interface. The routine of `ReviewPlus_cw_bgroup_index_event` is listed below.

```
pro ReviewPlus_cw_bgroup_index_event,event
  widget_control,event.id,set_value=event.value
end
```

**Step Summary**

In the processes of the events, both the Master client and the Participant client have the same functionality of the routines by executing them in the same order (the event handlers `ReviewPlus_cw_bgroup_index_event` and `ReviewPlus_preferences_apply_event`, which are two units of the transition function δ) with the event structure as the only parameter. As we have noticed, the event handler `ReviewPlus_cw_bgroup_index_event` is of different shape and purpose on the Master client from the Participant client. On the Master client, it is a function, and it is for capturing the event, sending out the event message, and returning the event to the upper widget. On the Participant client, it is a procedure, and it is for setting or unsetting the button of the CW_BGROUP widget programmatically as that of the Master client. Nevertheless, it has the goal to coordinate both sides to have the same function and appearance at the end of the process of the event; in other words, it has the goal to coordinate both sides to converge on the same state r of the DFA. The `ReviewPlus_preferences_apply_event` in this case exchanges the colors of the foreground and the background of the output display. With $\delta (q_i, a_i) = r$, the Master client and the Participant client converge on the same states r of the DFA (on event messages $a_i$) at the end of the processes of the events; therefore they have the same output display – in

this case, the same appearance of the CW_BGROUP widget and the same output in the main interface. We list the appearance of them in Figure B.12.

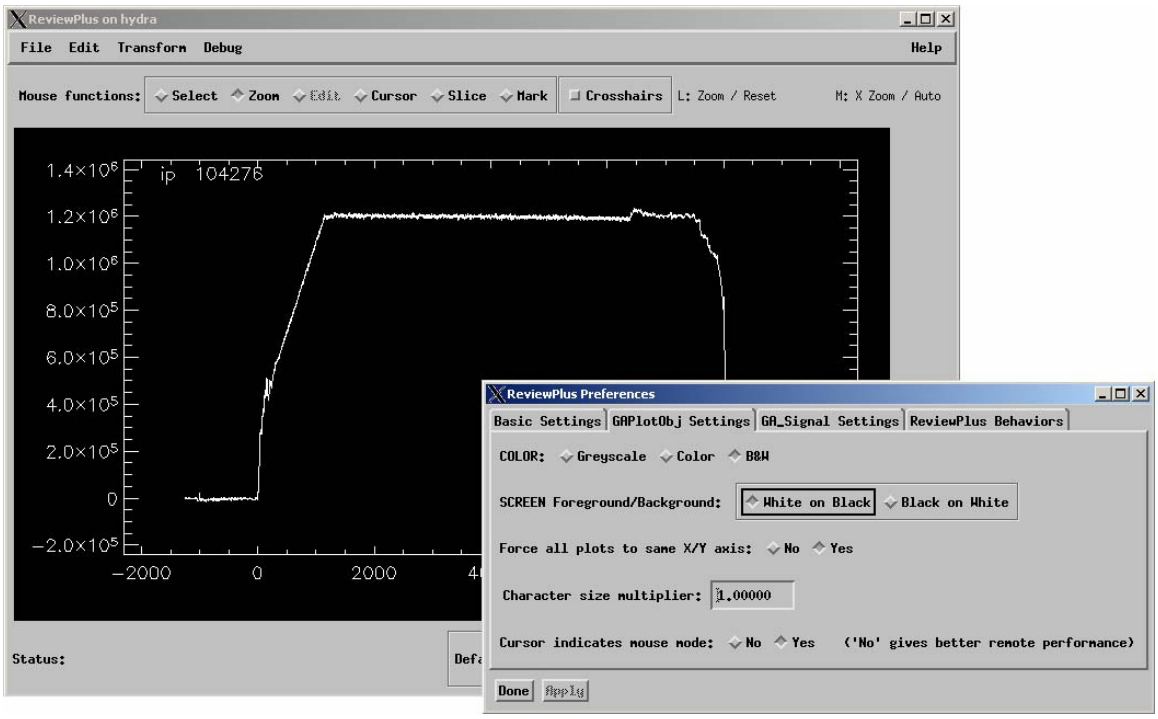



**Figure B.12 The effect to the output display in the main interface of ReviewPlus when setting up the "White on Black" for the screen foreground/background colors in the ReviewPlus Preferences interface.**

Now, in the interface of Figure B.11, if we click on the CW_FIELD widget beside the title "*Character size multiplier*," input the value 2, and hit the carriage return, the outputs on both the Master client and the Participant client will be displayed as in Figure B.13.

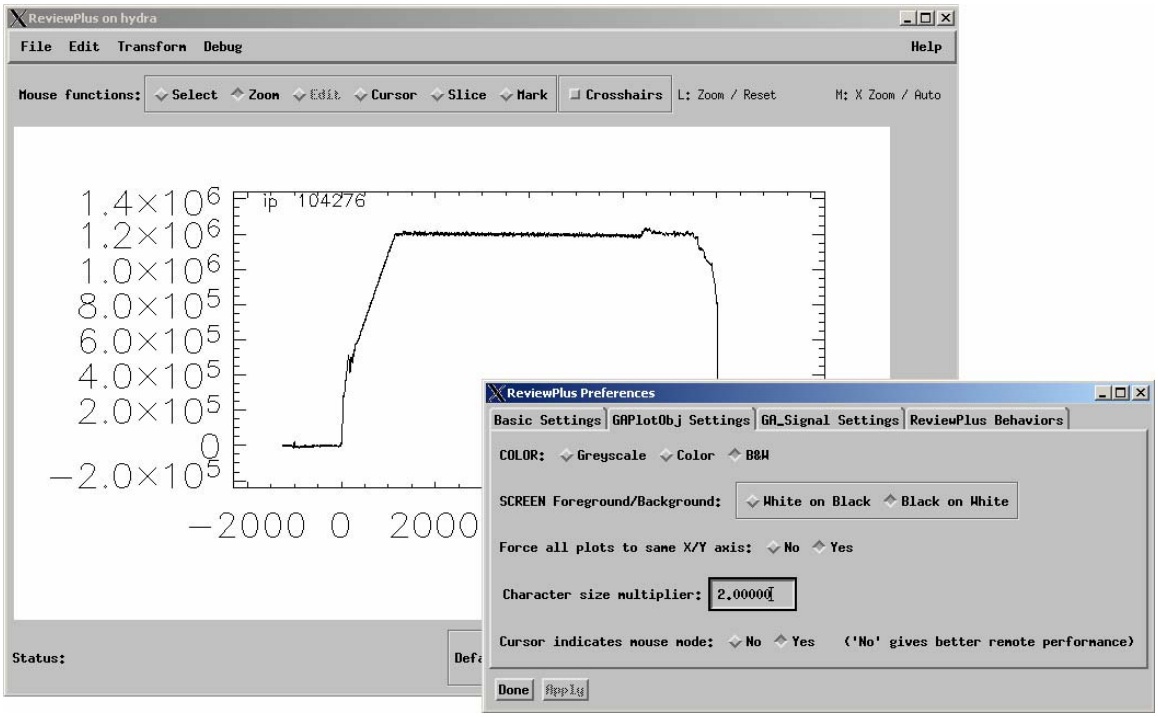



**Figure B.13 The effect to the output display in the main interface of ReviewPlus when changing the value of the CW_FIELD widget beside the title "Character size multiplier" in the ReviewPlus Preferences interface.**

The process of this step is very similar as that of the last step, so we would not bother to describe it. The only difference is that, this is a CW_FIELD widget. We would like to list the event structure and the specialties below. Here is the definition of the event structure for CW_FIELD widget:

{ID:0L, TOP:0L, HANDLER:0L, VALUE:'', TYPE:0, UPDATE:0}

It has 6 fields; unlike most of the widgets, it doesn't have a structure name. The field VALUE is of `string` type; the fields TYPE and UPDATE are of `int` type.

VALUE: The value of the field.

TYPE: The type of the data in the field, with 0=string, 1=floating point, 2=integer, and 3=long integer.

UPDATE: 0 if the field has not been updated, 1 if it has.

The widget CW_FIELD is not defined with keyword `event_pro` or `event_func`; therefore it has to associate with an event handler through its parent or ancestor widget, normally a base widget. As before, at this step, both the Master client and the Participant client converge on the same states r of the DFA and have the same output as in Figure B.13.

There are a lot more interfaces and widgets in ReviewPlus, such as WIDGET_DRAW, WIDGET_DROPLIST, WIDGET_LIST, WIDGET_TEXT, and CW_FORM. The descriptions of their processes of events are similar to those we have done so far. So we would like to get off the stage and leave the imaginations to the reader.

# Bibliography

[AG] Access Grid Conferencing Environment from Argonne National Laboratory

http://www.accessgrid.org/

[Aho] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, USA, 1988.

[Berman+Fox+Hey] Fran Berman, Geoffrey Fox, and Tony Hey, editors. *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley & Sons Ltd, Chichester, West Sussex PO19 8SQ, England, 2003.

See also http://www.grid2002.org

[Blackboard] Blackboard Learning System

http://www.blackboard.com

[Bulut+CIIT] Hasan Bulut, Geoffrey C. Fox, Shrideep Pallickara, Ahmet Uyar and Wenjun Wu. Integration of NaradaBrokering and Audio/Video Conferencing as a Web Service. *Proceedings of IASTED International Conference on Communications, Internet, and Information Technology*, pp. 401-406, St. Thomas, Virgin Islands, USA, 2002.

http://grids.ucs.indiana.edu/ptliupages/publications/AVOverNaradaBrokering.pdf

[CAROUSEL] Community Grid PDA project

http://grids.ucs.indiana.edu/ptliupages/projects/carousel/

[Centra] Centra Collaboration Environment

http://www.centra.com/

[CollabWorx] CollabWorx Integrated Web Collaboration Solutions

http://www.collabworx.com/

[Deitel+WS] H. M. Deitel, P. J. Deitel, J. P. Gadzik, K. Lomeli, S. E. Santry, and S. Zhang. *Java Web Services for Experienced Programmers*. Pearson Education, Inc., Upper Saddle River, NJ, USA, 2003. ISBN 0-13-046134-2

[Deitel+XML] H. M. Deitel, P. J. Deitel, T. R. Nieto, T. M. Lin, and P. Sadhu. *XML How to Program.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2001.

[DOM] W3C Document Object Model (DOM)

http://www.w3.org/DOM/

[Eddon] G. Eddon and H. Eddon. *Inside Distributed COM.* Microsoft Press, One Microsoft Way, Redmond, Washington, 1998.

[FCM] OpenOffice.org, Developer's Guide, Chapter 6: Office Development

http://api.openoffice.org/docs/DevelopersGuide/OfficeDev/OfficeDev.htm

[Foster+Kesselman] Ian Foster, and Carl Kesselman, editors. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., San Francisco, CA 94104-3205, USA, 1999.

[Fox+ACM] Geoffrey Fox, Shrideep Pallickara, Xi Rao. Towards Enabling Peer-to-peer Grids. Concurrency and Computation: Practice & Experience, Volume 17 , Issue 7-8 (June 2005) Pages: 1109 - 1131. 2002 ACM Java Grande–ISCOPE Conference Part II

http://grids.ucs.indiana.edu/ptliupages/publications/CCPE-P2PGrids.pdf

[Fox+CISE+July2004] Geoffrey Fox. Grids of Grids of Simple Services. *CISE Magazine*, July/August 2004.

http://grids.ucs.indiana.edu/ptliupages/publications/Cisegridofgrids.pdf

[Fox+CISE+March2004] Geoffrey Fox. The Rule of the Millisecond. *CISE Magazine*, March/April 2004.

http://grids.ucs.indiana.edu/ptliupages/publications/cisejano4.pdf

[Fox+CTS] Geoffrey Fox, Hasan Bulut, Kangseok Kim, Sung-Hoon Ko, Sangmi Lee, Sangyoon Oh, Shrideep Pallickara, Xiaohong Qiu, Ahmet Uyar, Minjun Wang, and Wenjun Wu. Collaborative Web Services and Peer-to-Peer Grids. *Proceedings of 2003 Collaborative Technologies Symposium (CTS'03)*, Orlando, USA, 2003.

http://grids.ucs.indiana.edu/ptliupages/publications/foxwmc03keynote.pdf

[Fox+CTS'06] Geoffrey Fox. Collaboration and Community Grids. *Proceedings of IEEE 2006 International Symposium on Collaborative Technologies and Systems (CTS 2006)*, IEEE Computer Society, Las Vegas, Nevada, May 14-17, 2006, pp 419-428.

http://grids.ucs.indiana.edu/ptliupages/publications/CTSMay06_cgc-03.pdf

[Fox+Distance] Collection of Resources on Distance Education, by Geoffrey C. Fox.

http://grids.ucs.indiana.edu/ptliupages/publications/disted/

[Fox+Education] Geoffrey C. Fox, Education and the Enterprise with the Grid, in F. Berman, G.C. Fox and A.J.G. Hey (Ed.), *Grid Computing: Making the Global Infrastructure a Reality, 43* (Chichester, West Sussex, England: John Wiley & Sons Ltd, 2003) 963-976.

[Fox+IC'02] Geoffrey C. Fox, Wenjun Wu, Ahmet Uyar, and Hasan Bulut. A Web Services Framework for Collaboration and Audio/Video Conferencing. *Proceedings of 2002 International Conference on Internet Computing (IC'02)*, Las Vegas, Nevada, USA, 2002.

http://grids.ucs.indiana.edu/ptliupages/publicatios/avwebserviceapril02.pdf

[Fox+JGI] Geoffrey Fox, Shrideep Pallickara, and Xi Rao. A Scalable Event Infrastructure for Peer to Peer Grids. *Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference*, pp. 66-75, Seattle, Washington, November 3-5, 2002. http://grids.ucs.indiana.edu/ptliupages/publications/ScaleableEventArchForP2P.doc

[Fox+P2PGrid] G.C. Fox, D. Gannon, S. Ko, S. Lee, S. Pallickara, M. Pierce, X. Qiu, X. Rao, A. Uyar, M. Wang, and W. Wu, Peer-to-peer Grids, in F. Berman, G.C. Fox and A.J.G. Hey (Ed.), *Grid Computing: Making the Global Infrastructure a Reality, 18* (Chichester, West Sussex, England: John Wiley & Sons Ltd, 2003) 471-490.

[Fox+Pallickara+IC'02] Geoffrey C. Fox and Shrideep Pallickara. JMS Compliance in the Narada Event Brokering System. *Proceedings of 2002 International Conference on Internet Computing (IC'02)*, pp. 391-397, Las Vegas, Nevada, USA, 2002. http://grids.ucs.indiana.edu/ptliupages/projects/NaradaBrokering/papers/JMSSupportInNaradaBrokering.pdf

[Fox+Pallickara+PDPTA'02] Geoffrey C. Fox and Shrideep Pallickara. The Narada Event Brokering System: Overview and Extensions. *Proceedings of 2002 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'02)*, pp. 353-359, Las Vegas, Nevada, USA, 2002. http://grids.ucs.indiana.edu/ptliupages/projects/NaradaBrokering/papers/naradaBrokeringBrokeringSystem.pdf

[Fox+WS] Geoffrey Fox, Sangmi Lee, Sunghoon Ko, Kangseok Kim, and Sangyoon Oh. CAROUSEL Web Service: Universal Accessible Web Service Architecture for Collaborative Application. November 2002. http://grids.ucs.indiana.edu/ptliupages/publications/Carousel_PerCom03.doc

[GA] General Atomics and Affiliated Companies

http://www.ga.com/

[Gamma] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Pearson Education Corporate Sales Division, 201 W. 103rd Street, Indianapolis, IN 46290, USA, 2002.

[Gannon] D. Gannon, J. Alameda, O. Chipara, M. Christie, V. Dukle, L. Fang, M. Farrellee, G. Fox, S. Hampton, G. Kandaswamy, D. Kodeboyina, S. Krishnan, C. Moad, M. Pierce, B. Plale, A. Rossi, Y. Simmhan, A. Sarangi, A. Slominski, S. Shirasuna, and T. Thomas. Building Grid Portal Applications from a Web-Service Component Architecture. *Special Issue of IEEE Distributed Computing on Grid Systems*.

http://grids.ucs.indiana.edu/ptliupages/publications/portal-apps-arch.pdf

[GlobalMMCS] Global Multimedia Collaboration System

http://www.globalmmcs.org/

[Globus] The Globus Alliance

http://www.globus.org

[Groove] Groove Desktop Collaboration Software

http://www.groove.net

[Gumley] Liam E. Gumley. *Practical IDL Programming: Creating Effective Data Analysis and Visualization Applications*. Morgan Kaufmann Publishers, San Francisco, CA 94104-3205, USA, 2002.

[Hopcroft] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Boston, MA, USA, 2001.

[Hwang] Professor Kai Hwang's research and work

http://gridsec.usc.edu/Hwang.html

[IDL+Lib1] The IDL Astronomy User's Library

http://idlastro.gsfc.nasa.gov/

[IDL+Lib2] The JHU/APL/S1R IDL Library from the Applied Physics Laboratory at Johns Hopkins University

http://fermi.jhuapl.edu/s1r/idl/s1rlib/local_idl.html

[IDL+Lib3] IDL Libraries Browser at Department of Astronomy, University of Washington

http://www.astro.washington.edu/deutsch/idl/htmlhelp/

[IDL+Link] Fanning Consulting

http://www.dfanning.com/documents/idllinks.html

[IVOA] International Virtual Observatory Alliance

http://www.ivoa.net/

[Interwise] Interwise Enterprise Communications Platform

http://www.interwise.com/

[JMS] Sun Microsystems Java Message Service

http://java.sun.com/products/jms

[JXTA] Sun Microsystems JXTA Peer to Peer technology

http://www.jxta.org

[Jabber] Jabber Instant Messenger

http://www.jabber.org/

[Jetspeed] Jetspeed Enterprise Portal from Apache

http://portals.apache.org/jetspeed-1/

[Lee+Dissertation] Sangmi Lee, "A Modular Data Pipelining Architecture (MDPA) for Enabling Universal Accessibility in P2P Grids", Ph.D. dissertation, Florida State University, 2003.

http://grids.ucs.indiana.edu/ptliupages/publications/Sangmi_Lee_thesis.pdf

[Lee+SVGopen] Sangmi Lee, Geoffrey Fox, Sunghoon Ko, Minjun Wang, and Xiaohong Qiu. Ubiquitous Access for Collaborative Information System using SVG. *Proceedings of SVGopen conference*, Zurich, Switzerland, July 2002.

http://grids.ucs.indiana.edu/ptliupages/projects/carousel/pagers/draft.pdf

[Levine] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc., Sebastopol, CA 95472, USA, 1995.

[Liang] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification.* Addison Wesley Longman, Inc., One Jacob Way, Reading, Massachusetts 01867, USA, 1999.

[MDSplus] MDSplus: Introduction

http://www.mdsplus.org/intro/index.shtml

[MQSeries] MQSeries

http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci214524,00.html

[MWPR] Microsoft Corp. *Microsoft Windows Programmer's Reference*. Redmond: Microsoft Press, 1990.

[Microsoft+KB] Microsoft Knowledge Base

http://support.microsoft.com/

[Monson-Haefel] Richard Monson-Haefel and David A. Chappell. *Java Message Service*. O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472, USA, 2001.

[NetMeeting] Microsoft Windows NetMeeting

http://www.microsoft.com/windows/netmeeting/

[OASIS] OASIS Web Services for Remote Portals (WSRP) and Web Services for Interactive Applications (WSIA).

http://www.oasis-open.org/committees/

[OpenOffice] OpenOffice.org

http://www.openoffice.org/

[Oram] Andy Oram, editor. *PEER-TO-PEER: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA 95472, USA, 2001.

[Pallickara+Dissertation] Shrideep Bhaskaran Pallickara, "A Grid Event Service", Ph.D. dissertation, Syracuse University, Syracuse, New York, USA, 2001.

[Pallickara+JDIM] Shrideep Pallickara and Geoffrey Fox. Efficient Matching of Events in Distributed Middleware Systems. *Journal of Digital Information Management*, June 2004, Volume 2, Issue 2, pp. 79-87.

[Peterson] James L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Inc., Englewood Cliffs, N.J. 07632, USA, 1981.

[Petri+Dissertation] Carl Adam Petri, "Kommunikation mit Automaten" (In German), Ph.D. dissertation, University of Bonn, Bonn, West Germany, 1962.

[Placeware] Placeware Collaboration Environment

http://www.placeware.com

[Polycom] Polycom Conferencing Environment

http://www.polycom.com

[Qiu+dissertation] Xiaohong Qiu, "Message-based MVC Architecture for Distributed and Desktop Applications", Ph.D. dissertation, Syracuse University, Syracuse, New York, USA, 2005.

http://grids.ucs.indiana.edu/ptliupages/publications/qiuPhDthesis.pdf

[RSI] Research Systems Inc.

http://www.rsinc.com/

[ReviewPlus] ReviewPlus Data Visualization Software User Manual

http://web.gat.com/comp/analysis/uwpc/reviewplus/manual/

[Russell+Norvig] Stuart J. Russell and Peter Norvig, editors. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458, USA, 1995.

[SVG] Scalable Vector Graphics (SVG) 1.0 Specification

http://www.w3.org/TR/2001/PR-SVG-20010719/

[Santo] M. Di Santo, N. Ranaldo, D. Villacci, and E. Zimeo. Performing Security Analysis of Large Scale Power Systems with a Broker-based Computational Grid. *Proceedings of ITCC 2004 Conference*, pp. 77-82, Las Vegas, Nevada, USA, April 2004.

[Snelling+GGF16] Standards, Industry, and the Roadmap to Grid Adoption

http://www.ggf.org/GGF16/materials/GGF16_Snelling_KeynoteA.ppt

[Tanimoto] Steven L. Tanimoto. *The Elements of Artificial Intelligence Using Common Lisp*. W. H. Freeman and Company, 41 Madison Avenue, New York, New York 10010, USA, 1995.

[Tannenbaum] A. Tannenbaum. *Metadata Solutions.* Addison-Wesley, One Lake Street, Upper Saddle River, NJ, USA, 2002.

[TeraGrid] TeraGrid

http://teragrid.org/

[UNO] OpenOffice.org, Developer's Guide, Chapter 3: Professional UNO

http://api.openoffice.org/docs/DevelopersGuide/ProfUNO/ProfUNO.htm

[Uyar+Dissertation] Ahmet Uyar, "Scalable Service Oriented Architecture for Audio/Video Conferencing", Ph.D. dissertation, Syracuse University, Syracuse, New York, USA, 2005.

http://grids.ucs.indiana.edu/ptliupages/publications/UyarThesisFinal.PDF

[VNC] Virtual Network Computing

http://www.uk.research.att.com/archive/vnc/

[VOMS] VOMS

http://hep-project-grid-scg.web.cern.ch/hep-project-grid-scg/voms.html

[WMF] Microsoft Windows Metafile

http://wvware.sourceforge.net/caolan/ora-wmf.html

[Wang+CATE'04] Minjun Wang, Geoffrey Fox, and Shrideep Pallickara. New Tools in Education. *Proceedings of the 7th IASTED International Conference on Computers and Advanced Technology in Education (CATE 2004)*, Kauai, Hawaii, USA, August 16-18, 2004.

http://grids.ucs.indiana.edu/ptliupages/publications/428-047_wang.doc

[Wang+ICIW'06] Minjun Wang, Geoffrey Fox, and Marlon Pierce. Thin Client Collaboration Web Services. *Proceedings of IEEE International Conference on Internet*

*and Web Applications and Services (ICIW'06)*, Guadeloupe, French Caribbean, February 23-25, 2006.

http://grids.ucs.indiana.edu/ptliupages/publications/wangm-Thin2.pdf

[Wang+ITCC'04] Minjun Wang, Geoffrey C. Fox, and Shrideep Pallickara. A Demonstration of Collaborative Web Services and Peer-to-peer Grids. *Proceedings of IEEE ITCC2004 International Conference on Information Technology*, Las Vegas, USA, April 5-7, 2004.

http://grids.ucs.indiana.edu/ptliupages/publications/wangm_collaborative.pdf

[Wang+ITCC'05] Minjun Wang, Geoffrey Fox, and Marlon Pierce. Grid-based Collaboration in Interactive Data Language Applications. *Proceedings of IEEE International Conference on Information Technology*, Las Vegas, Nevada, April 4-6, 2005.

[Wang+JDIM] Minjun Wang, Geoffrey Fox, and Shrideep Pallickara. Demonstrations of Collaborative Web Services and Peer-to-Peer Grids. *Journal of Digital Information Management*, June 2004, Volume 2, Issue 2, pp. 93-96.

[Wang+KSCE] Minjun Wang and Geoffrey Fox. Design of a Collaborative System (for Impress of Open Office). *Proceedings of IASTED KSCE 2004 Conference*, US Virgin Islands, November 2004.

http://grids.ucs.indiana.edu/ptliupages/publications/OpenOfficeCollaborativeSystemFINAL.pdf

[Wang+SCI] Minjun Wang, Geoffrey Fox, and Marlon Pierce. Instantiations of Shared Event Model in Grid-based Collaboration. *Proceedings of the 9th World Multi-*

*Conference on Systemics, Cybernetics and Informatics (SCI 2005)*, Orlando, Florida, July 10-13, 2005.

http://grids.ucs.indiana.edu/ptliupages/publications/InstantiationsGridCollab.pdf

[Wang+TR0505] Minjun Wang, Geoffrey Fox, and Marlon Pierce, "General Collaboration Structures for Interactive Data Language Applications", Technical Report, May 2005.

http://grids.ucs.indiana.edu/ptliupages/publications/GeneralCollabIDLApp.pdf

[Wang+TR0705] Minjun Wang, "A Description of the Implementation of Collaborative ReviewPlus", Technical Report, July 19, 2005.

http://grids.ucs.indiana.edu/ptliupages/publications/Generalization_ReviewPlus.pdf

[WebCT] WebCT Learning System

http://www.webct.com/

[WebEx] WebEx Collaboration Environment

http://www.webex.com

[XML] Extensible Markup Language (XML) 1.0 (Third Edition)

http://www.w3.org/TR/REC-xml/

# VITA

NAME OF AUTHOR: Minjun Wang

PLACE OF BIRTH: Zibo, P.R. China

DEGREES AWARDED:

M.S. in Computer Engineering, May 1998, Syracuse University, U.S.A

B.S. in Computer Science, July 1989, Jilin University, P.R. China