Modeling distributed asynchronous processes using events and calculus of communicating systems.

Hon Chun Patrick. Chan

University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

https://scholar.uwindsor.ca/etd/2752

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
NOTE TO USERS

This reproduction is the best copy available.

UMI
Modeling Distributed Asynchronous Processes
Using Events and
Calculus of Communicating Systems

By

Patrick HonChun Chan

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree
of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2000

© Patrick H. Chan
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.
Abstract

Remote object invocation is one of the distributed communication approaches, which usually deals with the synchronous communication. However, this type of communication approach does not fully capture the decoupled communication behavior. As the need for decoupled communication between objects in distributed applications becomes critical, the request of asynchronous process notification is highly on demand. This thesis addresses the design of a distributed model for asynchronous communicating processes. This model is using the CORBA event services to enable decoupling of processes. It also supports event registration and notification features. Processes receive only the up-to-date notification in which they are interested. Our design allows an arbitrary number of producers and consumers to be specified in the system. The relationship between producer processes and consumer processes are defined by our prototype using embedded specification formalism. The semantics of the specification is based on Milner's Calculus of Communicating Systems. Using our prototype, software designers do not need to know the implementation details of the system under development. Future directions of research are discussed highlighting the potential benefits of modeling distributed asynchronous processes using events and calculus of communicating systems.
Keywords: Distributed System, Asynchronous, CORBA, Event Services, and Calculus of Communicating Systems.

To My Parents
Acknowledgements

I was extremely fortunate to have the exceptional supervision of Dr. Indra Tjandra. First, I would like to express my appreciation to Dr. Tjandra, for his helpful guidance and generous donation of his time. My research was greatly improved by his suggestions and by the high standards with which he conducts his own work. Throughout, Dr. Tjandra has been a stimulating and an enthusiastic collaborator as well as my mentor caring my future and ensured that I never lost sight of my goals. He sent me to conferences where I gained a strong theoretical knowledge and he sent me to training where I gained a high degree of technical skills in building distributed applications. I also benefited from my internal reader Dr. Richard Frost for not only his suggestions, but also quick and comprehensive feedback. Furthermore, I would like to express appreciation to my external reader, Dr. W.C. Miller, for his feedback and interest in my work.

The seven years I spent in Windsor have left me with many friends and colleagues whom I would like to express sincere gratitude. I would like to thank to the members of distributed system group, my friends in optical network group and my friends in speech group. From start to finish, I had so many friends always there to solve problems with, to debate with and to share the many good and bad times. A special thanks goes to Mr. K. Achan and his wife, Anita for their encouragement, understanding and support. Also, I would like to thank my former colleagues at University Windsor IT and Purolator IS. They were remarkable kind.

Finally and most importantly, my deepest appreciation goes to my family and my love. Their enduring love and support made the rough spots smooth. I could not have succeeded without their constant encouragement and grace.
ABSTRACT

DEDICATION

ACKNOWLEDGEMENTS

1. INTRODUCTION
   1.1 OVERVIEW
   1.2 REMOTE PROCESS COMMUNICATION
   1.3 COMMUNICATION IN CLIENT/SERVER ARCHITECTURE
   1.4 NEW TRENDS OF SOFTWARE DEVELOPMENT
   1.5 OBJECTIVES AND SCOPE OF THE THESIS WORK
   1.6 ORGANISATION OF THE THESIS

2. PROCESSES COMMUNICATION MODELS
   2.1 COMMUNICATION PROTOCOLS
      2.1.1 Bulk Data Transfer
      2.1.2 One-to-Many Communication
      2.1.3 Atomic multicast
      2.1.4 Reliable multicast
      2.1.5 Totally ordered multicast
      2.1.6 Continuous media
      2.1.7 Remote Process operations
   2.2 PROCESSES COMMUNICATION MODELS
      2.2.1 Request/Response Communication
      2.2.2 Event-based Communication

3. BASIS OF CORBA
   3.1 THE OBJECT MANAGEMENT ARCHITECTURE (OMA)
   3.2 OBJECT REQUEST BROKER
   3.3 THE COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA)
      3.3.1 The CORBA Computing Model
      3.3.2 The CORBA Distribution Model
      3.3.3 CORBA Event Service
      3.3.4 CORBA Naming Service

4. THE CALCULUS OF COMMUNICATING SYSTEM

5. THE INVESTIGATION OF MODELING ASYNCHRONOUS PROCESSES
    COMMUNICATION WITH NEW SOFTWARE DEVELOPMENT TREND
   5.1 THE ESSENTIAL CONSIDERATION OF OUR PROTOTYPE
      5.1.1 Asynchronous communication with CORBA Event Service
      5.1.2 Java is the implementation language
      5.1.3 Analysis of concurrent systems
   5.2 INTEGRATE ASYNCHRONOUS COMMUNICATION MODEL WITH NEW DEVELOPMENT
      METHODOLOGIES
      5.2.1 Firms form alliance to rent software over the Internet
      5.2.2 Integrating purchased objects component with our generated systems
      5.2.3 Automated System Generator
   5.3 ARCHITECTURE OF OVERVIEW

6. THE FRAMEWORK OF DISTRIBUTED APPLICATIONS CODE GENERATION SYSTEM
   6.1 REQUIREMENT ANALYST
1. Introduction

This chapter gives an introduction of this thesis work. It also elicits the reasons for working on this thesis work. An organization of the thesis is provided as well.

1.1 Overview

As computing technology continues to develop, the capacity and power of personal computers are improving significantly. In the past few years, there is a trend that personal desktop computers are transforming into programmable workstations integrated with mainframe computers and formed local networks. These local network systems usually are combined by a number of different types of computers with different application software residing on them. Since computers allow sharing of resources within the same network system, the hardware and software operational and moving operations from an expansive mainframe computer to cheap workstations can dramatically reduce development costs. These kinds of network systems are called distributed systems. The goal of an ideal distributed system is not only to provide a way of resources sharing, but also to increase application development productivity.

A distributed system consists of a number of processes. These processes are connected by communication links [MK96]. The interprocess communication architectures are growing in popularity as more and more organizations install networks to share both hardware and software resources in distributed environments. The delivery requirements for interprocess communication services include minimum latency, maximum throughput and, in the case of continuous media (audio and video), they should minimize jitter and irregularities in latency [Mul95]. Therefore, a high-speed network is required for distributed environments. Fortunately, the technology for computer networks continues to grow in both capability and speed. Information can be transferred in reasonable time even across wide area networks by increasing bandwidth and capacity of computer network. Separate networks are used to carry voice, data and video information. A
number of researches on the fast networking technique have been done in order to gain multitude services, flexibility and ability of transmission.

Each process in a distributed system can be written in any language and perhaps, for different platforms. With the continued growth of interest in distributed systems, designers of processes communication for distributed systems are turning their attention to Remote Procedure Invocation (RPI). RPIs provide a way for local processes to invoke operations of remote applications. Do not forget that we must develop an intuition. We design distributed systems that perform as we expect. This way, we have a good understanding of existing distributed systems and modification of the system will be much easier [Sch92]. Moreover, an important characteristic of larger computer networks is heterogeneous in nature such as the Internet, the World Wide Web (WWW), and corporate Intranets [Vin97]. RPI is the basic technology that solves two major problems which distributed system application developers are facing today. These two problems are applications integration and scalability. RPI has moved to the centre stage of many new IT developments.

Communication between sending and receiving processes can be synchronous or asynchronous. In a synchronous mode of communication, the calling process sends a request to a remote process and waits for the response. However, this calling process must stop processing or it is blocked from proceeding until the remote process issues a response. In the asynchronous form of communication, the process is unblocked from the proceeding one. Program can send request and continue its processing before a response is issued. Although synchronous mode of communication is widely been used in interprocess communication, a few systems were chosen to use asynchronous mode. This is because the client might need to take back the control to perform other activities while waiting for a response. The asynchronous mode of communication provides an advantage of taking a much shorter time than its counterpart [JZ93][KVR83]. In parallel mode of communication, processes call their stub simultaneously and the stub responds to the request very quickly. With asynchronous call, system performance can be
increased if there is no server reply message. It is because the client does not need to wait for the reply.

1.2 Remote Process Communication

There are two major techniques of RPI, they are Remote procedure call (RPC) and Object Request Broker (ORB). RPC is the ordinary procedure-based model. It invokes individual procedures in an application to run on remote or local machines. RPCs use a familiar programming concept, the procedure call. RPCs are extremely suitable for the network request/reply model. However, ordinary RPCs are written in the traditional programming languages. Therefore, they do not support object oriented concept and lack other capabilities. We are going to address the synchronous and asynchronous RPC in this survey. The Object Management Group (OMG) defined the standards for definition of a distributed software platform. These standards adhere to the object-oriented programming paradigm [See98][Sig96][Sig98][Vin97]. The Common Object Request Broker Architecture (CORBA) is the standard that we are referring to. CORBA adheres to the RPI paradigm for supporting distributed processing. It is one of the best worldwide standards to allow interoperability-distributed objects in heterogeneous, distributed environment with transparent location.

1.3 Communication in Client/Server Architecture

Traditionally, most of client/server applications are implemented using request/reply model for communication. However, this model has some shortcomings for widely distributed systems with a larger numbers of data passing. Thus, a new software revolution has been taking place over the past few years. There is growing usage on Publish/Subscribe model for communication. Publish/Subscribe model is an Event Based model for solving the shortcomings of request/reply model. The CORBA decoupling communication, which is the feature of its Event Service, provides an asynchronous process communication [See98][Sig96][Sig98][Vin97]. This way, we can achieve better performance on both the systems and processes. However, two problems are raised. First,
the CORBA specification is very complex and rather difficult to learn. Secondly, design mistakes are often happened in the design stage, especially when building a distributed application which processes are separated in a geographical area and access to all relevant data within the same enterprise. This kind of mistakes sometimes could not be able to find out until the application is released and being use in the production. Company might lose a huge amount of profit and a large number of customers because of above mistakes.

1.4 New trends of software development

With the emergence of innovative, strategies employed in software development is changing in many aspects. Software automation is the central theme of such strategies. With software automation, developers are able to improve quality of the software product, reduce the development cost and time. However, improvement of the quality of software product should not only emphasize on the performance and quality of services, but also equally on software accuracy and formal semantic. Visual programming, frameworks, patterns and component methodology are the popular software automation techniques that are widely used in the industry. However, industry is interested in techniques that build a complex system very rapidly to prove the accuracy of systems design. In particular, they recognize the need to solve complex problems efficiently. These complex problems include data communication between a local and remote system, i.e. local system calling a remote system to perform a complex task and a local system sharing data with Web Wide Web application. The data communication between two systems could be very complex, and can involve different business rules and technical restrictions.

The modern business has realized the need for the usage of modeling and simulation and this can be seen in the growing dependence of these techniques for software development. A model represents a system using function or logic. It also defines the system. A model provides ideas that are modifiable. It describes the series steps involved, decisions made, and allows checking for rigor and robustness. An imitation of a system
or an incomplete representation of reality are not models, however, they are simulations. In most cases, the more complex the system is, the more importance is a good modeling technique. Therefore, rigorous modeling becomes an essential factor for building applications successfully. Formal semantic is one of the best ways to prove the application system. It behaves not only like a simulation but also as a verification of the design.

1.5 Objectives and Scope of the Thesis Work

This dissertation focuses on using technologies, which permits efficient development of software application. This dissertation is a comprehensive review of how asynchronous processes communication can be used effectively to maintain the advantages of resource sharing, data transformation and notification. However, the prime objective is to provide a new development technique together with a formal model that can be used to specify the design of a geographical distributed event-based system using asynchronous model for communication. With this development technique, designer is able to observe the behavior of the system in its design stage and prove the system with its formal semantics.

The major research contributions achieved by this dissertation include:

- Designing architecture and specification of code generator, which include:
  - Algorithms and specification of automated code generator for building distributed system applications. The result system could be infinite deep, in a hierarchical-based processes communication system.
  - Specification and practical evaluation of rules of code generator for mapping the input of software designer to CCS semantics.
  - Definitions and practical evaluation of rules of using CORBA event service to build publish/subscribe process models with Java.
  - Definitions and mechanisms for verification of input CORBA IDL.
• Address the benefits of asynchronous processes communication using CORBA as the specification and Java as the implementation language.
• Classify and evaluate the existing processes communication.
• Analysis of the current trends and predict the future direction of software development technique in the design and building a fault tolerance system.

1.6 Organisation of the thesis

The organisation of the remainder thesis is as follows: chapter 2 reviews background material on modeling, different types of processes communication models, CORBA even services and Calculus of Communicating System, Chapter 3 discusses the reasons for integrating the technologies highlighted in chapter 2. Chapter 4 and 5 discuss implementations and demonstrates the prototype for the automatic component generation and presents a number of techniques aimed to achieve a better performance. Chapter 6 presents how the formal semantics of the specification can predict and monitor the behaviour of the developing system. A case study demonstrating how the prototype can apply on the real world is also presented in the same chapter. Possible future developments in the modeling asynchronous process communication are discussed in Chapter 7. Chapter 8 presents the conclusion of this thesis.
2. Processes Communication Models

Distributed system supports concurrency and fault tolerance. It allows more than one process running at the same time. If one process fails, it might still be possible to look for another process to do the same task. Shared services that are provided transparently by the cooperative effort of both local and remote processes are the most common use of distributed systems [KKM93]. The method taking care of the above operations is called interprocess communication mechanism. Interprocess communication mechanism allows processes in a distributed system to communicate and shield one process from failures of another [Mul95]. It also provides transparency communications and allows static or dynamic reconfiguration.

Three major boundaries of the construction of interprocess communication are:

1. The operating system interfaces and the network interfaces.
2. Hardware mechanism, such as network.
3. Memory management and protocol machinery.

An ideal interprocess communication mechanism must be able to deliver information with maximum throughput and minimum latency. Moreover, it should be authenticated and secure as well.

2.1 Communication protocols

Bulk data transfer, one-to-many communication, continuous media, and Remote operations are the four broad classes of communication protocol. Each of these classes is tailored for a specific category of applications.
2.1.1 Bulk Data Transfer

Data storage, transfer and manipulation in geographical information systems (GIS), remote sensed applications and many other applications require the handling of massive amounts of data.

2.1.2 One-to-Many Communication

One-to-One communication means sending data from one end to the other end. It is a basic communication between two processes. However, if a sender needs to send the same data to a group of processes, using one-to-one communication is not efficient. Therefore, one-to-many communication broadcast communication protocols is established. The terms broadcast communication and multicast communication are interchangeable in interprocess communication. Broadcast and multicast mean sending information to all hosts on the network, and sending to a selected subset of hosts respectively [Mul95]. In fact, replication protocol is a multicast protocol.

2.1.3 Atomic multicast

In an atomic broadcast protocol, each member stores the update that it receives in a local buffer until it learns that the update is stable, i.e. when all members have received the update [WMK93]. Therefore, transmitted messages are either received by all the processes or by none. Failed process must be removed from the membership list of the multicast group.

2.1.4 Reliable multicast

Reliable multicast transport is considerably more complex than reliable unicast. Different applications usually have different reliability requirements and operation types [SKK92]. Therefore, it is difficult to build a generic reliable transport protocol for multicast. However, building a generic transport protocol for unicast is much easier [WCD+97]. Reliable multicast method does its best to transmit messages to all of the
processes of receiving groups. However, it does not guarantee that every one will receive it.

2.1.5 Totally ordered multicast

Totally ordered multicast is one of the best solutions that take care of the above problem. It guarantees that all the members of the multicast group receive the same order of messages. All the communicating machines must agree on the same total order of delivery without considering the fact that messages may be transmitted concurrently from different sites. Moreover, a message may take arbitrarily long to arrive at different sites. Therefore, the agreement on a total order usually bears a cost in performance. Totally ordered, reliable broadcast and multicast protocol are the approaches which have been widely used. They set up the rules and provide mechanisms for programming distributed system and distributed database [Cha84]. Computer supports cooperative work programs, groupware system and shared tools. Some other new applications are gaining advantages by using these approaches [WMK93].

2.1.6 Continuous media

It requires low-latency and constant-rate data transport and high-speed network. The capacity of the network must be large and traffic does not interfere with others. ATM network is suitable for such multimedia transmissions.

2.1.7 Remote Process operations

The last, but not the least topic that we discuss in this section is remote operations. The basic form of communication in a distributed system is remote operation [CDK94][Mul95]. The general explanation of remote operation is that a process sends a request to another process and asks that process to perform a task. The requisition side is called client and the responding side is called server. After the server receives a request, it processes the request, and then sends the acknowledgement back to the client if it is
needed. Processes can be both server and client at the same time. Remote operation is used for communication between not only local processes, but also processes within the same distributed system. This is the one of the core concept of this dissertation.

2.2 Processes Communication models

Processes communication is an important element to achieve the goal of resource sharing. One process passing information on another process provides a mean for remote processes invocation as well as a mechanism for information transformation.

2.2.1 Request/Response Communication

Request/Response communication is the traditional approach for transfer information from one application to other applications. In client/server architecture, the client application calls the server application to perform a specific task. With request/response communication, a remote operation looks like a local procedure call. Request/Response communication has been effectively used to interact with a computational or database server. This semantic of request/response communication is similar to the ordinary procedure call, wherein, caller gives up the control to the called procedure and the called procedure returns the result when the operation is done. Request/response communication model is also characterized as “demand-driven” communications. It is based on polling methodology, that polling requested process to request a specific information at an automatic time interval. In order words, the process will continuously issue demands information while an application is running.

The main drawback of this communication approach is, if a process needs to invoke a number of different operations, the client process must call each affected process individually to ensure that all the execution are performed. This approach creates a large amount of traffic and reduces the performance of the system, especially in the situation of cross-domain remote processes communication. It is because this approach needs a point-to-point communication. A typical example is stock prices scenario: Read-time response of stock prices is a must requirement of a stockbroker application. If we are
using request/response communication approach, the client process must poll the server frequently in order to get the latest prices of stocks. The more accurate data (up-to-date) that client wants to get, the more requests it needs to issue. This will exponentially increase both the server load and network traffic. Due to the limitation of distributed environment, this kind of overload access could cause dead lock, lack of memory, communication data missing, slower response time or all of above. Thus, it is nearly impossible to allow a large number of clients accessing a single server without a break.

Therefore, a new communication approach is needed to solve the “real-time-monitoring” requirements.

2.2.2 Event-based Communication

Most approaches make the application difficult or impossible to implement technique to automatically notify users or other processes. Therefore, many applications do not provide this feature. In the case of stockbroker scenario we discussed above, user needs to periodically check the system to see if the stock price was changed. A significant improvement will be reached if the system could automatically and immediately send a new price information to indicate that a stock price has been changed.

To support the requirements we discussed above, most of the current researchers on distributed processes communication are concerned with the new approach, that is, the event-based communication [RW97]. Obviously, events are the key design consideration in this new approach.

An Event monitors the activity in the system. It represents the changed information of the system. Research considers events as activities explicitly generated by the user program or implicitly generated by the system [WWW+93]. The different between an event and a message is, a message is sent from one object to a single object and an event is sent from one object to any number of objects. Therefore, message is one-to-one communication while event is one-to-many communication [Figure1]. A process-to-process interaction
model can be built using events to provide the benefits of integration and automation to local and remote process.

![Diagram of event and message flow](image)

**Figure 1** With events, a process can notify all interested parties automatic.

The built-in trigger capability in current database management system does trigger a process to handle local operations properly but it does not perform well on a geographically based distributed system. Processes in Event-based communication model are automatically triggered when event occurs. This process-to-process communication model is based on the Publish/Subscribe methodology. Publisher, subscriber and broker are the three parties that involve in this communication. Publishers trigger the data/process, generate events and notify their occurrence to their brokers. Subscribers register for events that they are interested, and then wait for the new events to be computed. Brokers act as managers and middle-person that keep track of the registration list of the events and broadcast the new event to the subscribers. There is always more than one subscriber registered for the same event and one subscriber may registers many events.

Before, we discuss the middleware infrastructure of event based architectural styles, we should look at types of process in distributed system environment. The types are synchronous process and asynchronous process. In the synchronous form of communication, the calling process sends a request to a remote process and waits for the response. However, this calling process must stop processing or it is blocked from proceeding until the remote process issues a response. In the asynchronous form of communication, the process is unblocked from the proceeding one that is waiting for
response. An asynchronous process sends request and continues its processing before a response is issued. The synchronous and blocking are the nature of request/response model.

Software architect should at least consider the following issues before he/she can make the decision of the type of process to be used in communication:

1. The purpose of the application - Does the application require real time respond?
2. The system performance - the capability of the system handles large amount of processes.
3. The system resources - the capability of the database support concurrence access.
4. The network bandwidth - the amount of traffic that the network can support.
5. The cooperation of the new application with other existing applications.
6. The development tools - Does the currently developed tools support the development technologies?

In order to have a better understanding of these two types of processes, we should look at a real example. When we call toll-free customer services numbers for customer support, most of us have a long time waiting experience. Usually it is really hard to determine the period between the call is connected and the moment we can speak to a representative. It depends on the number of customer calling and the number of representatives working. We can categorize callers in two groups:

Group 1: Callers just wait and do nothing till there is representative available to talk to them.
Group 2: Callers do something else (such as watching TV, read magazine) while they are waiting.

The first group of callers obviously is performing synchronous mode of operation. They make the calls, wait till representative responses. They cannot do anything else except wait. However, no one knows how long customers need to wait. Second group of people chooses to perform other tasks while they are remaining in queue. They are more flexible on the activities that they are doing. This way, they do not waste too much of their time. Thus, they are performing asynchronous mode of operation. The asynchronous
processing works incredibly well in the situation if unknowing how long a process will take. However, people in the second group must be ready to interrupt their asynchronous tasks at anytime.

The design issue for using synchronous process and asynchronous process are the same in terms of choosing the communication style. Choosing what kind of process to be used in an application is based on the purpose of the application. The consideration also include the following:

For synchronous process:
Features: High-speed response time.
Programming and system architecture are less complex.
Suitable for request/response communication model.
Tradeoff: Process may be blocked and idle the system.
Less flexibility on making operations.
Communication dead lock might occur easily.

For asynchronous process:
Features: More flexibility on execution of operations, concurrence operations are allowed
Communication dead lock is less lightly occurred.
Maximizes the application performance
Tradeoff: Slow response time
Application is more difficult to design

As the number of computer networks is increasing rapidly, communication between processes play a much important role than before. Selecting the suitable communication process for a particular communication is one of the key factors in controlling the performance of an application. Both synchronous and asynchronous processes communication has its strengths and weaknesses. Nowadays computer networks are designed in different architectures, that is, heterogeneous networks [MLB+97]. It is always difficult to predict process response time in heterogeneous network due to their complexity in design, mixed of types of communication protocols and a variety of data
resources. Comparing with synchronous, asynchronous processes communication provide more flexibility for the application. Therefore, asynchronous processes communication becomes an important technology in software development.

In this dissertation, we investigated the event-based style in CORBA with our own technology. Therefore, in next chapter we provide a basis for CORBA, and its event service and naming service.
3. Basis of CORBA

Many companies are currently considering using Object Request Broker (ORB) as a mean of integrating the data and processing their heterogeneous computer systems [Sig98]. ORB's remote operation call falls into two categories, those products that conform to the Common Object Request Broker Architecture (CORBA) standard from the Object Management Group (OMG), and Microsoft's OLE/COM technology [Omg98].

3.1 The Object Management Architecture (OMA)

OMA [Omg98] is a high-level vision of a complete distributed environment. Object Model and Reference Model are the two essential components of OMA. The object model defines how objects distributed across a heterogeneous environment can be described and the reference model characterizes interactions between those objects [Vin97]. OMA provides the environment for applications to provide their basic functionality via a standard interface [Sig98]. As figure 2 illustrates [Vin97], four categories of software interface are located inside OMA: CORBA services; CORBA facilities, CORBA domains and ORB.

**CORBA services:** These specify the basic middleware services for the development and deployment of distributed applications. However, some of these services are essential for all applications and some of them are for specific type of applications. The following are CORBA services with a short description about each service.

- Naming service -- allows client to find objects based on names.
• Event service -- enables event posting and dissemination.

• Transaction service -- provides transactional semantics to run business objects.

• Concurrency Control -- allows the client to access the share resources without creating any trouble.

• Relationships -- provides the capability for managing objects relationships.

• Externalization -- defines externalizing and internalizing objects protocols.

• Life Cycle -- defines the rules for object creating, deleting, copying and moving.

• Persistence -- association between persistent and stored objects’ state.

• Trading Service -- provides matchmaking service.

• Security -- provides the authentication, authorization, integrity, and privacy to degree, and using mechanisms that are yet to be determined.

**CORBA facilities:** They consist of Horizontal CORBA facilities and Vertical CORBA facilities. The Horizontal CORBA facilities are user-oriented while Vertical CORBA facilities are more specific for application domains.

**CORBA domain:** It represents vertical market area, such as banking, airlines, telecommunications and health-care [Dol97]. ORB is responsible for facilitating communication between objects and clients.
3.2 Object Request Broker

ORB is the central component of CORBA. An ORB is the middleman that allows the client and server objects to communicate with each other [Dol97]. The implementation of ORB does not depend on the CORBA specifications. It is based on the CORBA compiler vendors. Irrespective of who implements the ORB, all the ORBs should accomplish the initial communication that happened between objects. The way ORB works is very similar to RPC.

3.3 The Common Object Request Broker Architecture (CORBA)

In 1991, the OMG published revision 1.1 of the CORBA specification containing a concrete description of the interfaces and services that must be provided by compliant ORBs [Omg97][Omg98][Vin93]. CORBA is composed of the following major components:
**ORB Core:** It is used to deliver requests to correspondent object and return the response to clients. It provides a way of transparency to both the programmers and clients. Therefore, the clients do not need to know the details of the network objects. Either IDL stub or Dynamic Invocation Interface is used for the communication between client and ORB Core. ORB uses object reference to identify and locate objects.

**Interface Definition Language (IDL):** Interface Definition Language describes objects and uses object reference type [SKK92]. It specifies the operations and types that the object supports. It also defines the requests that can be made on the object.

**Interface Repository (IR):** It is a CORBA object, which provide a way for the applicant to traverse a hierarchy OMG IDL information. The primary function of IR is to provide the type of information necessary to issue request using the Dynamic Invocation Interface. IR can be used to discover programmatically type information (type checking) at run-time.

**Static Invocation Interface (SII):** It is a stub-based interface, which allows client programs to use services on application objects. SII is very similar to an RPC interface.

**Dynamic Invocation Interface (DII):** It supports dynamic client request invocation [Sig96]. It provides a way to allow calls on objects without having compile-time knowledge of their interfaces. Therefore, it does not need any static stub. Because of runtime interpretation of request parameters and operation identifiers; DII becomes a generic
client-side stub at run-time be. DII provides flexibility but it is expensive because a number of calls have to make through it.

**Object Adapters (OA):** It cooperates with CORBA object implementation and ORB itself [Vin97]. Object Adapters keep the ORB as simple as possible by including Object registration, object reference generation, server process activation and object activation, request demultiplexing and object upcalls. Basic Object Adapter (BOA) allows objects to be implemented as separate programs. Library Object Adapter (LOA) is used with lightweight object implementations of client applications [Vin93]. Object-Oriented Database Adapter (OODA) is used as an interface between object-oriented database and the ORB.

Having discussed CORBA's underlying theoretical concepts, we will now emphasize on the practical aspects so as to understand how CORBA exactly works.

### 3.3.1 The CORBA Computing Model

RPC supports both local procedure invocation and remote procedure invocation. Similarly, CORBA allows request passing within the same process and between distributed processes. Figure 3 shows a request passing from client to an object implementation in the CORBA architecture. Both the client and Server objects interfaces are defined in OMG IDL. Therefore, the client is not able to know the implementation details of other objects. The only thing client needs to know is the object’s interface, which is provided in the IDL. This way, the client does not need to worry about the
implementation behind the interface. Thus, a plug-and-play component environment is established. ORB acts as a manager managing the request passing and objects invocations. However, this does not mean that ORB is another process running on CORBA environment. In fact, it links clients and object implementations into an executable module through library routines. In order to make the ORB layer to be lightweight and fast, the implementation of ORB must be sophisticated.

Figure 3 Request is passed from a client to an object implementation in CORBA architecture. A process contains Client, Object, IDL stub and skeleton, and ORB.

The IDL specifics object operations, input and output parameters of different operations of the object. Client objects use this interface definition to build and dispatch invocations as the object implementation uses it to receive and respond [See98]. Client sends an invocation request to an object via its interface. However, this request does not pass from client to object implementation directly. ORB delivers the request to the correspondent object implementation IDL interface. This object implementation IDL interface defines the operations of the object provided. The object developer must implement these operations specified in the IDL interface. The developer has the right to choose the language he/she wants to use in the implementation. Most major programming languages
mapping standard such as for C, C++, Java, COBOL are already specified by OMG [See98]. Non-standardized mappings for other languages can also be used to implement interfaces. OMG’s COM/CORBA inter-working specification defines the non-standardized Visual Basic mapping [Sig98]. Client can use either Static or dynamic invocations for calling the operations of another object. In the static mode, this is similar to RPC, a stub and a skeleton are generated after the successful compilation of the IDL interface as showed in figure 4. A stub creates and issues client request while a skeleton delivers request from ORE to the CORBA object implementations [Vin97]. Stub and skeleton perform marshalling and unmarshalling, if necessary. IDL compilers provide an automated compiler optimization environment and eliminate careless programming errors [GS96].

![Diagram of IDL specification and components](image)

**Figure 4** The components that are generated from IDL
In the dynamic mode, the client uses a set of ORB functions to form dynamic invocations at run-time. With the DII, a client application can invoke requests on any object without having compile-time knowledge of the object's interfaces. One of the advantages of using dynamic mode is what it is much more flexible than static mode. However, a single DII request could require several remote invocation calls. In the past, CORBA provides synchronous communication and only DII supports deferred synchronous request invocation capability. Recently, new OMG asynchronous/messaging services [Omg97] are embedded in CORBA. These new services allow deferred-synchronous in request invocation via the static stub.

3.3.2 The CORBA Distribution Model

The distributed model requires ORB-to-ORB communication. Figure 5 [Sig98] shows the data flow of a request when client is on one machine while target object is on the remote machines. As we can see from the figure, the initial step of the request process is the same as we just discussed (local invocation). ORB determines the local and remote object invocation by checking the object references together with the request in the client code. If it is a remote invocation, ORB examines the object reference and then passes the request invocation to the target object through the network. However, the local ORB is not able to route the request invocation directly, instead, the ORB of the target object will pick up the request and perform communication with the target object. In order to ensure that all ORBs on the network have access to IDL interface definitions of all the objects, sharable Interface repositories (IR) with name and unique identifier are used.
Figure 5 Request is passed from a client to a remote object. ORB handles network communications between the client and target object reside on different machines.

The IR provides interoperability between different ORB implementations. Assuming the ORB of the target packs up the invocation, the ORB will extract the object ID that is encapsulated within the object reference and check with the active object Map. Portable Object Adapter (POA) routes the invocation after mapping Objects ID to servant. POA is the new standard object adapter defined in the CORBA 2.2 specification [Omg98] while the programming language implementations are called servants. Applications built using CORBA version 1.0 was small enough to reply on a single ORB service. As applications got bigger and more distributed, CORBA version 2.0 specification comprised the standard General Inter-ORB Protocol (GIOP) and its specification for implementation over TCP/IP, the Internet-ORB Protocol (IIOP). POA is a new feature that is being added to CORBA version 3.0. Figure 6 shows a request flowing from a client into a server application. The following are the steps for request dispatching:

1. Client invokes the request with an object reference. The object reference contains an object key, which is used to identify the target object within the server application.
2. Server ORB receives request.
3. Server ORB dispatches request to the proper POA (more than one POA could be inside the same application) by using object key.

4. POA grabs the object ID from the object key and use it to determine the association between the target object and a programming language servant.

5. POA dispatchers request to servant.

6. Servant processes request and sends results back to the POA

7. POA sends result back to server ORB

8. Server ORB passes result to network

9. Client ORB receives the result and sends back to the client.

3.3.3 CORBA Event Service

Standard CORBA operation invocation models are based on synchronous execution of an operation provided by an object as showed in figure 6. They are too restrictive for building a more sophisticated distributed application. The restrictions are, client in standard model is blocked until response returns, and only un-cast communication is allowed and client and server must be present. This restriction is alleviated by using one of the CORBA Object Service (COS), that is the CORBA Event service. COS Event Service model is based on the “publish/subscribe paradigm. CORBA Event Service provides a standards-based event mechanism that allows asynchronous message delivery and decoupage communications between objects. Event data can be delivered from suppliers to consumers without requiring these participants knowing each other explicitly [DFG97]. Thus, an asynchronous communication model with multiple suppliers and consumers can be established.
Figure 6 Standard CORBA method invocations provides only synchronous execution of an operation by an object.

Supplier, Consumer and Event Channel are the three participants in COS Event service. Supplier generates event data. Consumer process event data. Event channel in figure 7 acts as a mediator from which consumers can obtain event. Push and pull models are the participant collaboration models present in COS Event Service architecture. In the push model, supplier pushes an event to consumers. Supplier takes the initiative in the push model. In the pull model, consumer solicits event from suppliers. Consumer takes the initiative in the pull model [Omg00].

Figure 7 Supplier-consumer communication Model
In COS Event Service architecture, there are four communication models collaborating between Consumers and Suppliers:

- **The Canonical Push Model**: Active Suppliers push event to passive Consumers through Event Channels.

- **The Canonical Pull Model**: Active Consumers pull event from passive suppliers explicitly through Event Channels.

- **The Hybrid Push/Pull Model**: Active Consumers pull event deposited by active Suppliers explicitly through Event Channel. In this model, Event Channels play the role of Queue.

- **The Hybrid Pull/Push Model**: Active Event Channels first pull event from passive Suppliers then push event to passive Consumers.

The role of each COS Event Model component is summarised below:

- **EventChannel**: This interface defines administrative operations for adding consumers (ConsumerAdmin) and suppliers (SupplierAdmin). It also has the operation for destroying the channel.

- **SupplierAdmin**: This interface creates supplier proxy objects for connecting suppliers to event channels

- **SupplierAdmin**: This interface creates consumer proxy objects for connecting suppliers to event channels.
- **PullConsumer**: It provides factory operations for consumer in the pull model, such as disconnecting from a consumer.

- **PushSupplier**: It provides factory operations for supplier in the push model, such as disconnecting from a supplier.

- **PullSupplier**: It provides factory operations for supplier in the pull model, such as transmitting event to a consumer in pull model.

- **PushConsumer**: It provides factory operations for consumer in the push model, such as receiving event from a supplier.

- **ProxyPullSupplier**: This interface acts as a proxy for the supplier from which event channel pulls events from suppliers. Consumers use this interface for controlling the connections with event channel in pull model.

- **ProxyPushSupplier**: This interface acts as a proxy for the supplier for pushing events to event channel. Consumers use this interface for controlling the connections with event channel in push model.

- **ProxyPullConsumer**: This interface acts as a proxy for the consumer that allows event channel pulls events from consumers. Suppliers use this interface for controlling the connections with event channel in pull model.

- **ProxyPushConsumer**: This interface acts as a proxy for the consumer that allows event channel pushes events to consumers. Suppliers use this interface for controlling the connections with event channel in push model.

The following figure shows an overview of Objects of COS Event Service.
3.3.4 CORBA Naming Service

CORBA Naming Service plays the role of binding database, which holds names and objects references. Binding is the association between a name and an object [CPS]. Thus, Naming Service provides a way to associate one or more logical names with object implementation and store those names in a namespace. Naming Service allows a name with Object reference to be bound. By resolving the name within the Naming Service, we can get a particular Object reference. In other words, a server or an implementation can register with Naming Service with its object reference and a particular name, so that other parties on the same system will be able to find this object [LSN+99].

Binding a name to an object reference, resolving a name to get an object reference, unbinding a name to delete a binding and listing names with a naming context are the four major operations that COS Naming Service provides [Ion98]:

Name Service stores name contexts. Theses name contexts are administrated in a hierarchy structure. Figure 9 is an example of name context graph.
Figure 9 A Naming Context graph

John, Peter and manger are the nodes of the graph. They represent the naming context. The “Company-Marketing-Manger” names an object Manger and Manger based on the same naming context as “Company-Marketing-John”. With this tree like structure, searching the same group of context names will become easy.

An advantage of the Naming Service is that the names associated with objects are independent of any properties of the objects they refer to. In particular, a name is an independent of object interface, server or host name.
4. The Calculus of Communicating System

The behavior of processes communication controls the activities of dynamic systems. A complex dynamic system usually hosts an environment that has both transient faults and dynamic changes. On one hand, asynchronous caller process takes back the control before response is issued, thereby gaining more flexibility in inter-communication when compared to asynchronous mode of communication. On the other hand, concurrency provides the flexibility in data communication by allowing more than one pair process communication at the same time.

Asynchronous and concurrence communication always occurs in complex systems, especially in real-time systems [RS84]. Because of the asynchronous processes running at different speeds [DKM93], the behavior of the system is hard to predict. The Calculus of Communicating System is a well-defined technique in modeling the systems of communicating processes. Each system unit has its own identity, which we call agents. An agent is intended to represent the action and behavior of a process. Even though an agent represents only part of the system, it identifies the behavior succinctly and accurately. A complex system is formed from the meaning of a new agent and some means of combing processes [Mil89]. Agents perform two types of interactions. The first type is an interaction with its neighboring agents [Mil98]. The second type is an internal interaction within an agent itself. This kind of internal communication is considered as an independent action because the interaction happens between two agents within the same computer. A sender, a receiver and a medium are the three basic entities involved in
information transmission processes. Information is sent from sender to receiver through medium. Messages are the normal information being transmitted.

Based on specific character and pattern of behavior of sending and receiving occurred, media is classified into unbounded ether, bounder ether and shared memory. The following describe the role of sender and receiver under an Unbound ether discipline of information transmission.

**Sender:** *may send messages to Ether at any time.*

**Receiver:** *may receive messages from non-empty Ether.*

**Message Order:** *The receiving message order may not be the same as the sending message order.*

This type of Ether allows sender sending to send messages continuously regardless of the size of the Ether. Definitely, this is a problem because Ether may be full. Bounded Ether resolves this problem by allowing the sender to send message if and only if the bounded Ether is not full. However, these two types of Ether contain an unordered set of messages. In order to preserve message, we use buffer as medium:

**Sender:** *may send a message to Buffer at any time.*

**Receiver:** *may receive a message from non-empty Buffer.*

**Message Order:** *The receiving message order is identical to the sending message order.*

Just like the Ether, we can produce a buffer with bounded discipline. As in Bounded Ether, sender is allowed to send message to bounded buffer if and only if the bounded buffer is not full.
Based on the above disciplines, we understand that the properties of media are the essential considerations in modeling communication. In fact, more disciplines of transmission can be proceeding. To sum up, Sender and Receiver are the active agents and media is the passive entity. Buffer is one of the media that has been used in this dissertation. The basic operators on agents are prefix, composition, summation, restriction and recursion [Mil80].

**Prefix ( . )**:  
Prefix is the most basic operator. Suppose there is a process, which performs from the initially event described by 'in' to become the process described by A. We use in.A to represents the process. For the prefix operation, we always have a label on the left, an agent on the right and a '.' in the middle. Let the other label of agent A be called 'out', Then, we have the following diagram.

![Diagram of Prefix Operator](image)

**Figure 10 The prefix operator**

When A is empty, A may accept an item at the left-hand port. When A holds an item, A may deliver the holding item to right-hand port. Then we can define the behavior of agent A as:

\[ A \overset{\text{def}}{=} \text{in}(x). A'(x), \ x \text{ is the item being transferred} \quad \text{(I)} \]
We might define the second behavior of agent A as following:

\[ A' \overset{\text{def}}{=} \text{out}(x).A \text{, where } x \text{ is the item being transferred} \quad \ldots\text{(II)} \]

**Summation Operator (+):**

Summation operator is used to conjoin the capabilities of both sides of agents. In other words, given Agents P and Q, we form the composite agent \( P + Q \) that demonstrates the capabilities of both agent P and agent Q. Therefore, This agent behaves like P or Q.

**Recursion Operator ( = ):**

By combining II, and I we get a recursion behavior of agent A, which defines the behavior of C. C repeatedly takes a value at the in port and outputs it at the port \( \text{out} \). This mutually recursive definition could be rewritten as the following recursive definition:

\[ A \overset{\text{def}}{=} \text{in}(x).\overset{\text{---}}{\text{out}}(x).A \]

**Composition Operator ( | ):**

Parallel composition can be used to describe the possibility of communication of two labels. Suppose we have an agent B, defined as:

\[ B \overset{\text{def}}{=} \text{receive}.\overset{\text{--}}{\text{in}}(x).B \]
We use \( \mid \) as the constructor for composition operation. \( A \mid B \) denotes that agent A and B may proceed independently and they may communicate with each other. Hence, we may define a new agent BA with the following behavior:

\[
BA \overset{\text{def}}{=} A \mid B
\]

Pictorially, we can represent agent BA as:

![Diagram of BA]

\textbf{Figure 11} The composition operator

Through the internal communication between agent A and B, BA may accept values at label "come" and release values at label "\( \overline{\text{out}} \)". If we compose a new agent C with AB, we have agent BAC, where \( BAC \overset{\text{def}}{=} C \mid BA \). BAC enables the communication between CB and BA. BAC can be represented pictorially as:

![Diagram of BAC]

\textbf{Figure 12} The composition system

It is clear that by adding a new agent C into the above composition system, more communication between new agent and A can be established. However, we may need
want to isolate a pair of connected ports so that no future connection is allowed. For this, we need another constructor, the restriction operator \( \backslash \).

**Restriction Operator** \((\backslash)\): we use \( P \backslash l \) to represent that \( P \) is restricted on \( l \). \( P \backslash l \) is an agent which behaves like \( P \). However, this agent cannot to communicate through both label \( l \) and \( \bar{l} \) ports, but \( P \) can have such a capability. Removing the restricted labels pictorially can represent restriction. For example \( BAR \) is defined by

\[
BAR \overset{\text{def}}{=} BA \backslash \text{in}
\]

and can be represented as

![Diagram of BAR](https://via.placeholder.com/150)

**Figure 13** The restriction operator

By adding agent \( C \), we have this definition: \( CBAR \overset{\text{def}}{=} C \mid BAR \),

Thus, \( CBAR \) is pictured as below:

![Diagram of CBAR](https://via.placeholder.com/150)

**Figure 14** The restriction operator in a composition system

36
5. The Investigation of Modeling Asynchronous Processes

Communication with New Software Development Trend

The focus of this dissertation is not only on how to apply asynchronous communication on distributed processes, but also in the general the software development methodologies. In this chapter, we throw light on modeling asynchronous processes from a theoretical standpoint. We also discuss implementation details and review the requirements of our prototype in the next chapter.

5.1 The Essential Consideration of Our Prototype

Distributed asynchronous processes communication is another channel to achieve the goal of resource sharing. In the design of our prototype, we need to consider on the following issues:

- How do we implement the asynchronous processes communication?
- How do we define the communication rules?
- How do we achieve the resource sharing and adopt the new software development methodologies.
- How do we model the system?

5.1.1 Asynchronous communication with CORBA Event Service

One of the most efficient ways to gain the asynchronous processes behavior is by using the subscribe/publish communication model [BBH+96]. To implement a subscribe/publish model, we need events [AS96]. Event represents the changes of our communication system. By combining with event, data transformation and notification,
we have our event model [CDR+96]. How do we build our event model? We use CORBA specification. CORBA is a standard for distributed object communication [Sch98]. CORBA defines the processes communication behaviors. CORBA allows applications to request a remote operation to be performed by a distributed application [Sig96]. Its Event Service specifies the general rules for event based communication model. Thus, we build our asynchronous communication model with CORBA Event Service. Three different communication parties are defined in our modeling system; they are push-client, push-server and push-client-push-server. Therefore, we have a Push-model system. In our push model, supplier processes control the data flow by pushing events to consumers through event channel. Push model provides the best model for monitors server’s events [Omg00]. It is much suitable for building business application which requests instant response or monitoring. However, consumers might not be able to get the real time notification. The notification time depends on the system speed and network traffic [SMF+97]. However, it is a nearly real-time communication.

Figure 15 A push communication model
5.1.2 Java is the implementation language

We chose Java as the implementation language because perfectly complements CORBA. CORBA supports many existing languages. CORBA also allows mixing these languages within a single distributed application. By leveraging the strengths of Java programming language with the robustness of CORBA, we can build distributed asynchronous processes applications using our prototype. Java being a platform-independent programming language allows portability of applications to nearly any platform [KR99]. The wide platform of supporting Java applications reduces platform "lock-in". Java supports object-oriented development paradigm development, therefore its applications can be maintained easily and be reused [Boo93]. Java also provides the scalability, which can handle a large number of users. Though the execution time for Java may not be the fastest, with its abilities, our prototype can produce more productive systems, which can be to run in any operation system within the network.

5.1.3 Analysis of concurrent systems

Systems that built by our prototype are currency system. In our generated system, two pair of processes communication are able to perform simultaneously. As we have mentioned in last chapter, we use CCS to prove the semantic of our generated systems [Str91]. In order to manipulate and analyze the CCS syntax, we use the Edinburgh Concurrency Workbench (CWB). CWB is an automated tool which allows for various equivalence, preorder and model checking using a variety of different processes
semantics. An agent is a local unit in our generated system. CWB can be used to simulate the behaviour of our agents.

5.2 Integrate Asynchronous Communication model with New Development Methodologies

The new development methodologies that we address in this dissertation are modeling technique and integrating purchased objects into other software environment.

5.2.1 Firms form alliance to rent software over the Internet

According to International Data Corp (Canada) Ltd, on-line software rental is forecast to grow next year to $200 million in Canada. The research firm Dataquest Inc. of San Jose, Calif. forecasts the same sector to be worth $23-billion (U.S.) by 2003 [Law99]. This sector is called Application Service Providers (ASP).

One of the biggest ASP in Canada is an alliance formed by Nortel, which bundles its network equipment with hardware from Hewlett-Packard Co., and software from HP and three other firms. Starting in the first quarter of 2000, this alliance will sell the packages to Internet service providers and other carriers. Other carriers include the group of companies, which wants to rent software on-line.

With the services from ASP, customers relieve the task of upgrading computer software. Software rental company can keep these tasks at a central site and rent or distribute them through the Internet or private lines.
ASP alliance develop communications and E-commerce applications aimed at small-sized and medium-sized businesses, which will rent the software for a monthly fee from an Internet service provider or an individual. With the CORBA and other distributed application communication technologies, we predict that the ASP will provide more services, especially those that require heavy processing resources and a large amount of development time.

5.2.2 Integrating purchased objects component with our generated systems

Historically, application software has been built from scratch. On one hand, software architects designed all the models of software applications. On the other hand, developers implemented the application requirements. This was because the software development techniques were not grow-up at that moment:

In the past:

1. Most of the applications were written in different programming languages, therefore, applications were incompatible and difficult to communicate with others.
2. Software architects did not consider the software re-use concepts.
4. Recourse sharing methodologies were not defined properly.
5. Data transmission speed was low within the network and networks bandwidth was small.
6. Software application were designed for specify needs and used to solve problems only on specify.
7. Lacked of programming standards and processes communication standards for heterogeneous distributed computing.
Nowadays:

1. Distributed systems concepts have been widely used to provide a resource-sharing environment.

2. Many geographical networks have been built by combining small and medium networks together. This way, a local process in Windsor might be able to talk to a remote process in France through Internet.

3. Data transmission speed has been increased significantly.

4. Object technology provides software re-uses. Thus, an application contains a number of small modules instead of a big one.

5. Processes communication specifications have been standardize. Local process is able to talk to remote process easily. Object Management Architecture (OMA) is the architecture that allows applications to communicate. CORBA is the specification language that supports OMA.

The software development trends have been changed because of the existence of CORBA object interoperability. Enterprise integration needs strong accomplishment. This accomplishment is not focus on the connections of applications, but on the connections of objects. By combining the capabilities of the connecting of objects and the IDL, a plug and play environment can be established.

The world of business is changing rapidly. The trick to withstand global competition is finding ways to reduce development cost and time. Building software from scratch involves large investments and time. Today, software development moves to the year of challenge. With the completion of the notion ‘Y2K’ problem, developments in computing technologies are bound to speed up.
The research reveals that preference for code reuse than build systems from scratch, such reuse of components and integration with new modules will largely reduce development lost and time. However, compatibility and lack of standards proved to be blocking stones for such integration. These hindrances can be overcome with CORBA, which provides the Inter-ORB communication feature [Hen98]. With Inter-ORB feature, client and object implementation can reside on different platforms, different operating systems, different networks and different vendors' ORBs [Sig98]. Moreover the client and object implementation could be written in different programming languages [Vin98]. With the ability of integrating a purchased object component, programmer from the user frame does not need to write the object implementation. This programmer only needs to write a client object accessing the remote object based on the IDL files. IDL files contain descriptions of operations. Thus, user can easily integrate existing software with his/her system by accessing services in the provider's remote server. Also, user can purchase object component and installs it to his/her own server to shorten data travel time. This way, programmer does not need to build software application from scratch. Companies can purchase the basic software components that implement common knowledge in their area and extend the functionality of the purchased modules or build the modules interacting with the basic modules. Figure 16 shows the integration of a purchase object in a consumer's software environment. When a consumer purchases a software module, the vender will provide IDL files. There are two types of purchase. The first type comes with the executable object implementation. Consumer needs to install the executable object implementation on his/her server node. After the executable object implementation is installed, an object referencing the implementation object will be
generated. This object reference may be placed in a file on share drive or associated to naming service. Client objects communicating with the implementation object residing on server node will use this object reference. The second type of purchase is the provider providing services on his/her own server node. Consumer responds by installing the client part of the application on his/her server. Also the consumer needs the object reference of the executable object implementation on the provider’s server. This way, consumers might reduce the cost of software and hardware expenses. However, the drawback is that the object communication speed might be slow due to the traveling distance. Of course, services providers can provide the same service to more than one consumer to access his/her service node at the same time. Moreover, services providers are able to restrict the number of accesses for consumers. Programmer working for the consumer’s company can compile the IDL file with the desired IDL compiler. A client stub and implementation skeleton will be generated. Programmer writes client applications based on client stub. Again, programmer could use any programming language that he/she desires. To access the ORB, client applications use the client stub. To invoke operations on implementation objects, client applications use the object reference storing in a file or registered with the naming services. The auto-generated skeleton code can be thrown out because client application does not need it.
Figure 16 Producing an IDL file and object implementation, then integrating a purchased object into consumer's system.
5.2.3 Automated System Generator

Building complex application system involves so many steps and procedures. Usually, the more complex of the system are, larger the amount of development time [Dil95]. Changes in entities and modules over time enhance tremendous complexity to business data models. This usually involves introducing additional entities and modules [GHJ+95]. Therefore, addition, deletion and modification of the existing systems may involve re-designing the system to adopt to changes. Re-designing a system is time consuming since we need to prove the correctness of the new system. Moreover, the former solution might be outdated or we might have re-structured the whole existing system during updating. Therefore, building systems using automated generating technique as done in this dissertation allows software designers to build an accurate system on the fly. The only task for system designers is to provide the design specifications, so those modules of systems will be automatically generated. This way, system designer focuses on the design and does not need to worry about the programming or interrogating issues. With the automated generating tool, changing the existing system is just re-designing the system and the new system will be up and running in a short period of time [LB88]. By joining the component modeling techniques and automated technologies, we can manage a complex system in the easiest way. We believe the automatic generation technique fulfill industry needs. Thus, we introduce our model. Our model is named Distributed Application Automatic Generator (DAAG). DAAG is designed to respond to these exacerbated architectural problems. It also reduces
cost and time-to-market by building high quality asynchronous processes communication system.

5.3 Architecture of Overview

The completed architectural prototype is presented in next chapter. In next section, we describe the general concept of the workflow of our modeling system.

The Java programming language and CORBA concepts warps the final application generated by DAAG. Software designer uses our prototype showed on figure 15 to generate the final applications and the semantic of the applications. Therefore, we do not require software design to have any CORBA or Java knowledge. However, software designer needs to specify the type of event model that agent will be used to communicate with another agent. Figure 17 shows the design framework using our prototype. After the input specification is entered, our modeling system will do the computation and perform sequence operations. Actually input specification can be used to represent in a graph structure. Input specification will be analysed by our computation unit and perform syntax checking on input IDL files. After verifying input IDL files, our system will generate the session of environments for CWB, which includes agents, action sets, propositions, and macros. These environments will be analysed by CWB to determine the behaviour of each agent in the system. If the specification passes the CWB checking, a number of modules are generated. The auto-generated modules will be compiled by Java and IDL compiler, and then be linked together to be distributed to the right node on the network. User can use the application after the above processes are performed.
Figure 17 Design framework of the prototype
6. The Framework of Distributed Applications Code Generation System

Combing with the formal syntax, semantics and CORBA services, we have the fundamental base for the distributed system. In this chapter, we look into how DAAG automates the processes of building distributed applications. From the architecture overview that we discussed in the last chapter, we demonstrated that the core responsibility for the DAAG is to reduce the complexity on developing distributed applications. In other words, DAAG is a tool, which helps developers with their code writing task. In fact, DAAG actually does more than that. In addition to reducing developer's workload, the finally applications can be built with no developer involved. This shows DAAG is the concept of DAAG is vigorous.

6.1 Requirement Analyst

We want the software from ASP to be "rented" through a private network line or Internet. In the system that we build, we allow each individual agent to make external calls. The action for calling an external process is an instruction to ask remote process to perform a particular task. This allows the system generated by DAAG to gain advantages by passing some tasks to a high-performance server. The result of the call is based on the method name as well as the input parameters.

As the word "automation" suggests, we want to significantly reduce the developmental. Our system should develop software with only little inputs from users.

We identified several requirements for DAAG:
• It should be a clearly designed and user friendly system. The high level concept of the system design concept should be easy to understand.

• It should support the sharing of common data files and services.

• It should adopt a distributed architecture. The system should support multiple application communication, wherein applications are distributed and executed on different machines connected by a high-bandwidth private network or Internet.

• It should support component-based software development approach and the Object-Oriented methodology.

• The input data shall be read from an input data file with a critical syntax validation before the whole result system is built.

• It should allow the system containing arbitrary levels.

• It should contain the naming service for identifying the location of remote processes and using CORBA event service for performing decouple communication [DFG+97].

• All the applications within the system should be run on any machines regardless of the type of their operation systems.

• It should incorporate the renting software technology over the Internet.

• It should allow integration and customization.

DAAG meets above requirements. We choose CORBA as the middleware to provide naming service, event service and communication between internal applications and outside world. Java is the implementation language that we used to build DAAG. Our
goal is to allow the system generated by DAAG system to work in Microsoft Windows system and UNIX system. With Java, we can achieve the platform independent ability.

6.1.1 The IDL of event channel

The event channel that we are using in DAAG is the OBEEventChannel Factory. OBEEventChannel Factory is ORBacus Event Service with optional, proprietary interfaces for event channel administration. With OBEEventChannel a single event service is able to manage multiple channels [Omg00] [Sch97].

The following is the IDL for OBEEventChannel:

```java
interface EventChannelFactory
{
    CosEventChannelAdmin::EventChannel create_channel(in ChannelId id)
        raises(ChannelAlreadyExists);
    CosEventChannelAdmin::EventChannel get_channel_by_id(in ChannelId id)
        raises(ChannelNotAvailable);
    ChannelIdSeq get_channels();
    void shutdown();
};
```

This IDL defines the operations for event channel. The first operation creates and returns an event channel. The second operation retrieves an event channel object. The third operation returns a sequence of channel-ids, which is used to identify available event-channels. The last operation is used to shut down the event service.

The channel identifier, hostname and port number forms the unique index that is used to locate a channel. For example:
iioploc://149.99.92.40:7022/DefaultEventChannelFactory

‘DefaultEventChannelFactory’ is the channel identifier while ‘149.99.92.40’ is the hostname (IP). ‘7022’ is the port number.

6.2 Syntax And Semantics Of The Design Specification

The DAAG is used to assist in building distributed system. This result system contains a number of processes. There are relationships between these processes. Before we begin the construction of a distributed system using DAAG, we should make some simplifying assumptions and come up with mathematical representations of the underlying processes. With the proven design, the resulting system will be much more reliable. In this section we will describe mathematical models that simplify the complex processes that occur in nature.

The distributed processes described in the previous sections are described again in this section. This time we glance at them in terms of how they are represented as a mathematical model. In mathematical model, we use DS to represent our auto-generated distributed system. The overall distributed system contains a collection of agent applications. External processes, IDL specifications and machines form a completed agents application system. The symbols for representing external processes, IDL specifications and machines in our mathematical model are P, J and M respectively. In order to maximize the system performance, we only use Push operation. In fact, we could use the Push, and Pull operation for the decoupled communication. Figure 18 shows the
structure of auto-generated system built by DAAG. In this figure, we can clearly identify the relationship between DAAG components.

Figure 18 An example of the design architecture

As we can see form the graph, each of agents performs one task only. This task could be a remote process or a local process. For the first agent, its task is accepting input from the standard input. For the last task, it is displaying a text message. The information passing between agents is through CORBA event channel. In fact, each of event channels contains buffers, which are used to store the incoming objects. Queue is the data structure that CORBA event channel uses to store incoming objects. More details of the queue logic will be discussed in the last section.

6.2.1 The DAAG Mathematical Models

Ordered pairing constructor is one of the most basic ways of building new objects in mathematics and programming. For example, in mathematics one builds rational numbers
as pairs of integers and complex numbers as pairs of real numbers. In programming, a data processing record might consist of a name paired with basic information such as student name and student ID. For DAAG, the symbol DS is used to represent a completed distributed system that is generated by DAAG. DS consists a pair with node and operation between nodes. Thus, we have

\[ DS = \langle N, \xi \rangle, \text{ where } N \text{ is the node and } \xi \text{ is the operation between nodes.} \]

In DAAG, the finite set of processes is represented with symbol \( P \), the finite set of IDL specifications is represented with symbol \( J \) while the finite set of machines is represented with symbol \( M \). The direct product of three sets \( P, J \) and \( M \) are defined to be the set of all points \((p, j, m)\), where \( p \in P, j \in J \) and \( m \in M \). The Cartesian product is denoted \( P \times J \times M \), which is the set of all directed relationship from elements \( p \) of \( P \) to elements \( j \) of \( J \) to elements \( m \) of \( M \). Node \( N \) is a subset of the above Cartesian product. Thus, we have the following expression

\[ N \subseteq P \times J \times M \]

The about components can be viewed as

\( P \) to be the set of finite processes = \{ Process_0, Process_1, ..., Process_{N-1} \}

\( J \) to be the set of finite IDL specifications = \{ IDLSpecification_0, IDLSpecification_1, ..., IDLSpecification_{N-1} \}
\( M \) to be the set of finite machines = \{Machine_0, Machine_1, ..., Machine_{N-1}\}

For the relationship between nodes, we have
\[ \xi = N \times \{\text{Push, Pull}\} \times D \rightarrow N \times \{\text{Push, Pull}\}, \text{ where } D \text{ is a set of data} \]

As we have mentioned before, each node could have two types of operations. These operations are either Push or Pull. A supplier node can use Push or Pull operation to pass events to the event channel. A consumer node can use the Push or Pull operation to communicate with the same event channel. However, in this dissertation, DAAG only uses the push operation to achieve the faster communication speed.

The aim of DAAG system is to provide system automation for designers. In most of the cases, there are human users at each end of the system only, and the rest of processes are done by a remote or local implementation. In fact, the users at each end can be a person or a machine, such as satellite or vehicle-tracking detector.

The structure of CCS includes Node and services. Their symbols are \( v \) and \( x \) respectively. Therefore, we have the followings:

\[ v : N \rightarrow \text{Agent}, \text{ where Agent is a set of agent variables.} \]

\[ x : D \rightarrow \text{Act}, \text{ where Act is a set of actions.} \]

The whole system contains four major different components, they are Agent -- Agent, Act -- Tasks Node, EC -- Event Channel, DB -- Database. Thus, the Distributed System
DS is the final system containing the above components. More precisely, the DS is represented by a set of Agents, \( \text{Agent}=\{\text{Agent}_1, \text{Agent}_2, \text{Agent}_3, \ldots, \text{Agent}_n\} \), a set of Tasks Nodes, \( \text{Act}=\{\text{Act}_1, \text{Act}_2, \text{Act}_3, \ldots, \text{Act}_n\} \), a set of Event Channel, \( \text{EC}=\{\text{EC}_1, \text{EC}_2, \text{EC}_3, \ldots, \text{EC}_n\} \). Depending on the needs of system services, DS could contain a set of Databases, \( \text{DB}=\{\text{DB}_1, \text{DB}_2, \text{DB}_3, \ldots, \text{DB}_n\} \). For the example services of DAAG, we specified one service accessing database, thus, a database is needed for demonstrate our example. Also, we have a central naming server to manage the location of system services. The system also contains a set of Data, \( \text{D}=\{\text{D}_1, \text{D}_2, \text{D}_3, \ldots, \text{D}_n\} \). Each of agents and service nodes is an individual process that could be running on the same machine or different machines across the heterogeneous distributed network.

In order to understand the mathematical model easier, we take figure 18 as an example. The system architecture showed in the figure 18 is generated by DAAG. From the design, we clearly see that we have four different services running in the distributed environment. These services provide the activities that agents are required to perform. Each of the different services is a CORBA object implementation that runs as an independent process in a particular machine. These services are managed by CORBA naming services and hence even though they are running in different machine, they are still able to transparent to their users. Moreover, they have the ability to serve more than one system in the network contemporaneously. Thus, the decision for placing service processes in what type of machine or the location of the machine becomes very important. If the service is a heavy workload service or frequently used process, it should be placed in a high-speed server and close to users.
6.2.1.1 The Communication Paths

From the design specification of the example that we presented, we can identify that there are four processes running concurrently. These four processes are the tasks that are waiting for new incoming event. Once new event arrives, they invoke certain service to process the event and generate an output event. The output event will be passed to an event channel. The above scenario is always true except that the agents at each ends of hierarchy. Agents at the beginning of hierarchy take inputs from users while agents at the end of hierarchy return responses to users. The set Act is a collection of task processes, which composes Act₀, Act₁, Act₂, Act₃ and Act₄ in the design. There are two event channels that have been used in the example and it represents the set of event channel, EC contains EC₀, EC₁. Each EC has at least one queue those stores events from the producer process and passes the event to subscribed consumer processes. DAAG’s event channel supports one-to-many communication. A CORBA event service manages the set of EC.

6.2.1.2 Modeling Distributed System

In order to model distributed system, we define and evaluate the relationships of components within the system.

The process communications among these components are modeled as a set of Actions, Act = {Req, Res, ToQ, FromQ, WriteDB}. where Req = {Req₀,Req₁,Req₂,Req₃,Req₄}, Res = {Res₀,Res₁,Res₂,Res₃,Res₄}, ToQ = {ToQ₁, ToQ₂, ToQ₃, ToQ₄}, FromQ =

57
\{\text{FromQ}_1, \text{FromQ}_2, \text{FromQ}_3, \text{FromQ}_4\}. The actions that we outline are the critical ones. In fact, we are not going to look at these minor methods and actions precisely. Since we have five agents in the example above, the distributed system generated by the DAAG can be represented as below.

\[
\text{System} = (\text{Agent} \mid \text{Act} \mid \text{EC} \mid \text{DB}) \setminus L, \text{ where Agent} = \{\text{Agent}_1, \ldots, \text{Agent}_n\}, \text{ Act} = \{\text{Act}_1, \ldots, \text{Act}_n\}, \text{ EC} = \{\text{EC}_1, \ldots, \text{EC}_n\}, \text{ and DB}_i; \ L \subseteq \text{Act}.
\]

Based on the above expression and the interaction illustrated on the figure 18, we have the behavior of agents and their sequence of operations. Agent_0 accepts data from standard input and then sends the request to Act_0 for performing a special task. Act_0 responses back Agent_0 with the result. Agent_0 receives the result that it sends the data to event channel_0. After the data is sent to event channel, Agent_0 goes back to its beginning stage and wait for new data to come. Therefore, Agent_0 has the following behavior:

\[
\text{Agent}_0 = \text{AcceptData} \cdot \underline{\text{RequestAction}} \cdot \underline{\text{ReceiveAction Result}} \cdot \underline{\text{SendData}} \cdot \text{Agent}_0
\]

The Agent_1 has similar behavior as agent_0, except Agent_1 deals with Act_1 instead of Act_0. In this design, the standard keyboard is the device that provides input to Agent_0 and Agent_1. By modifying our application, we could make the input coming from other input device (such as bar code scanner). The following expression defines the behavior of Agent_1.

\[
\text{Agent}_1 = \text{AcceptData} \cdot \underline{\text{RequestAction}} \cdot \underline{\text{ReceiveAction Result}} \cdot \underline{\text{SendData}} \cdot \text{Agent}_1
\]
Unlike Agent\textsubscript{0} and Agent\textsubscript{1}, Agent\textsubscript{2} receives event from the event channel Queue\textsubscript{0} instead of from the user-input device. Same as Agent\textsubscript{0} and Agent\textsubscript{1}, Agent\textsubscript{2} passes the event to predetermined action process. When the result from action arrives, Agent\textsubscript{3} pushes the event to the Queue\textsubscript{1}. This is a repeatedly behavior.

Agent\textsubscript{2} = Receive\textsubscript{Data} \bullet Request\textsubscript{Action} \bullet Receive\textsubscript{Act Result} \bullet Send\textsubscript{Data} \bullet Agent\textsubscript{2}

The Agent\textsubscript{3} and Agent\textsubscript{4}, have the reverse behaviors as Agent\textsubscript{0} or Agent\textsubscript{1}. They wait for event to be pushed into their receiver units. Then they call action for performing the per-determined tasks. Once the tasks are completed, they return the result to the receivers. In our design, the receivers are standard output. Similar to the input device, the receiver device could be any output device, such as a paging system.

Agent\textsubscript{3} = Receive\textsubscript{Data} \bullet Request\textsubscript{Action} \bullet Receive\textsubscript{Act Result} \bullet Output\textsubscript{Result} \bullet Agent\textsubscript{3}
Agent\textsubscript{4} = Receive\textsubscript{Data} \bullet Request\textsubscript{Action} \bullet Receive\textsubscript{Act Result} \bullet Output\textsubscript{Result} \bullet Agent\textsubscript{4}

The expressions above illustrate the behaviors of agents. The next step is to determine the behavior of our Actions.

Act\textsubscript{0} accepts request, passes the appended string to the receiver then returns itself back to the state of Act\textsubscript{0}.

Act\textsubscript{0} = Request\textsubscript{Action}(X) \bullet Receive\textsubscript{Act Result} (message + X) \bullet Act\textsubscript{0}
\( \text{Act}_1 \) accepts request. Then it appends the input with a constant message and returns the value of two times input. Then returns itself back to the state of \( \text{Act}_1 \).

\[ \text{Act}_1 = \text{RequestAction}(X) \cdot \text{ReceiveActionResult}(2 \times X) \cdot \text{Act}_1 \]

\( \text{Act}_2 \) accepts request, writes the input value \( x \) into database and passes the input value back to sender as an acknowledgment then returns itself back to the state of \( \text{Act}_2 \).

\[ \text{Act}_2 = \text{RequestAction}(X) \cdot \text{WriteToDatabase}(X) \cdot \text{ReceiveActionResult}(X) \cdot \text{Act}_2 \]

\( \text{Act}_3 \) and \( \text{Act}_4 \) accept request, send the value of input value \( x \) to the dispatch server and pass the input value back to sender as an acknowledgment. Then returns itself to their original states.

\[ \text{Act}_3 = \text{RequestAction}(X) \cdot \text{Dispatch}(X) \cdot \text{ReceiveActionResult}(X) \cdot \text{Act}_3 \]

\[ \text{Act}_4 = \text{RequestAction}(X) \cdot \text{Dispatch}(X) \cdot \text{ReceiveActionResult}(X) \cdot \text{Act}_4 \]

6.2.1.3 Decoupling using Event Channels

The event channels receive events which are pushed by their producer, then store incoming events into their Queue and then push the events to the consumers. The set of event channels in DAAG is represented using \( EC = \{ EC_0, \ldots, EC_q \} \), where \( q \) is 1 in our example. EConSupp means event for connecting suppliers while EConCon standards for event for connecting consumers. Both connecting parties include their bind event channel operation. EC is defined recursively. The behaviors of event channels are illustrated as the following:
EC=$\{EC_0,EC_q\}$, where $EC = EConSupp \cdot EC_+ EConCon \cdot EC$

The $EConSupp$ is a recursive operation which does the binding to event channel, get supplier administration, get a consumer proxy, add the supplier to the Event Channel or data transfer.

\[
EConSupp_{i,k} = \text{BindEC}_k \cdot + \text{GetSupplierAdminForSupplier}_{j,k} \cdot EConSupp_k \\
+ \text{ObtainProxyForPushConsumer}_{j,k} \cdot EConSupp_k \\
+ \text{ObtainProxyForPullConsumer}_{j,k} \cdot EConSupp_k \\
+ \text{ConnectPushSupplier}_{j,k} \cdot EConSupp_k \\
+ \text{ConnectPullSupplier}_{j,k} \cdot EConSupp_k \\
+ \text{PushEvent PushEvent}_k \cdot EConSupp_k \\
+ \text{PullEvent}_k \cdot EConSupp_k + \text{TryPullEvent}_k \cdot EConSupp_k
\]

\[
\sum_{i=1,2,...,n} EConCon_{l,j} = \text{BindEC}_j \cdot EC_i + \text{GetASupplierAdminForConsumer}_{i,j} \cdot EConCon_l \\
+ \text{ObtainProxyForPullPushSupplier}_{i,j} \cdot EConCon_l \\
+ \text{ObtainProxyForPullSupplier}_{i,j} \cdot EConCon_l \\
+ \text{ConnectPushSupplier}_{i,j} \cdot EConCon_l \\
+ \text{ConnectPullSupplier}_{i,j} \cdot EConCon_l \\
+ \text{PushEvent}_j \cdot EConCon_l \\
+ \text{PullEvent}_j \cdot EConCon_l + \text{TryPullEvent}_j \cdot EConCon_l
\]
The above expressions basically defined the general behavior of the event channels.

However, we know that we have two event channels in the example system. Thus, we define the Event channel as:

\[ EC_i = \sum_{k=1,2,...,n} EconSupp_{i,k} \cdot EC_i + \sum_{j=1,2,...,m} EconCon_{i,j} \cdot EC_i \]

The next step is to define the data within the distributed system. Combining the behaviors of agents in system with the definition of data, we will have a precise picture of the communication processes of the dynamic system. We have three groups of data flowing within the system, the first group is the data that flows between processes and actions. The second group is the data flowing between processes and queues. The last group of data flows between the end-users and actions.

Event Channel\(_i\): Proxy Push Consumer\(_i\) + \(\sum_{i=1,2,...,n}\) ProxyPushSupplier\(_i\) + SupplierAdmin\(_i\) + ConsumerAdmin\(_i\) + BindEventChannel\(_k\) + DiconnectPushSupplier\(_i\) + DiconnectPushConsumer\(_k\)

Transfer Data from the **Proxy Push Supplier** to the **Proxy Push Consumer**.

SupplierAdmin\(_i\): ObtainConsumerProxy\(_i\)

ConsumerAdmin\(_i\): ObtainSupplierProxy\(_i\)
The Event Channel has no suppliers or consumers at the time when it is created. Four steps have been used to connect and use an event channel by Suppliers and Consumers.

1. Binds the object to Event Channel.
2. Obtains the right proxy object based on the reference of administrative object from the event channel
3. Connects the object with the proxy object
4. Transfers or receives data through the proxies.

The following illustrates the above steps in CCS.

Supplier₁: BindEventChannel₁ + ObtainConsumerProxy₁ + DisconnectPushSupplier₁ + 

SendDataToProxyPushConsumer₁

**Push Supplier** sends data to its **Proxy Push Consumer** in response to events that it is monitoring. When the event channel is being destroyed, event channel calls the **DisconnectPushSupplier** method to disconnect the supplier. Same scenario happens with Push Consumers.

ProxyPushConsumerᵢ: SendDataToProxyPushConsumerᵢ + 

SendDataToProxyPushSupplierᵢ,ⱼ

\[ \sum_{i=1,2,...,n} \text{ProxyPushSupplier}_{i,j}: \text{SendDataToProxyPushSupplier}_{i,j} + \]

\[ \sum_{i=1,2,...,n} \text{SendDataToPushConsumer}_{i,j,k} \]
\[ \text{Consumer}_k: \text{BindEventChannel}_k + \text{ObtainSupplierProxy}_k + \text{DisconnectPushConsumer}_k + \sum_{i=1,2,\ldots,n} \text{SendDataToPushConsumer}_{i,j,k} \]

**Push Consumer** spends most of its time in an event loop, waiting for data to arrive from the **Proxy Push Supplier**.

6.2.1.4 The Queue

The number of Queue in the DAAG is based on the number of push Consumer. Each of the push Consumers contains a queue for storing events that will be pushed to the Consumer client. There is a queue located in each of the ProxyPushSupplier. This is because each of the consumers has a separate queue. Therefore, Event channel ECi has SUM Q q,i associated with it, where q = 1 to K.

Queue stores outstanding event objects to be held for each consumer. If the consumer cannot keep up with the rate of message from the supplier, a large amount of event objects will be accumulated in the Queue. Out-of-memory could easily occurs with this design, thus, we used the MAX_QUEUE_LENGTH attributes in the Event Channel to limit the number of event objects that could stores inside the Queue. Actually the storage stores the event inside a queue called cell. MAX_QUEUE_LENGTH defines the number of cell inside queue. The oldest event object in the queue will be removed if there is a new event object attempting to add to a queue and the queue is full. Since the CORBA
event service specified that each individual consumer has separate queue. Therefore, a slower consumer may lose message due to the queue filling up quickly. The DAAG is using the push supplier and push client model. The Push Supplier Push consumer model allows DAAG to achieve a near real-time de-couple communication between supplier and consumer. With the push supplier and push consumer model, we do not really need to store a large number of cells inside their queues. The next step is to define the relationship between queue and cells.

\[ Q_{q,i} = \{ \text{Cell}_{q,i,0} | \ldots | \text{Cell}_{q,i,n-1} \} \], where n is equal to the maximum number of cell inside queue.

The following defined the behaviors of cells inside a queue.

\[ \text{Cell}_{q,i,0} = \text{receiveEvent} \bullet \text{ShiftNextElement}_{q,i,1} \bullet \text{insertElementAt}_{q,i,0} \bullet \text{Cell}_{q,i,0}^+ \]

\[ \text{getCell}_{q,i,1} \bullet \text{removeElementAt}_{q,i,j-1} \bullet \text{Cell}_{q,i,0} \]

\[ \text{Cell}_{q,i,1} = \text{ShiftNextElement}_{q,i,1} \bullet \text{ShiftNextElement}_{q,i,2} \bullet \text{getCell}_{q,i,1} \bullet \text{removeElementAt}_{q,i,1} \bullet \text{setSizeOfQueue}_{q,i,1} \]

\[ \text{Cell}_{q,i,j-1} = \text{ShiftNextElement}_{q,i,j-1}^+ \bullet \text{getCell}_{q,i,j-1} \bullet \text{removeElementAt}_{q,i,1} \bullet \text{setSizeOfQueue}_{q,i,j-1} \bullet \text{removeElementAt}_{q,i,1} \]
6.3 Architectural Overview

Figure 19 A conceptual view of the DAAG

Figure 19 illustrates a conceptual view of the DAAG architectural for generating distributed applications. Library module at the top level of the graph stores templates for creating Push Consumer, Push Supplier, Push Supplier- Push Consumer and Naming Service Register and other tools. The IDL interface definition and their implementations are stored in the Library model as well. Code Generator model in the middle level performs code generation. At the bottom level, we have the Java applications and tools that are ready to execute and form the distributed enterprise.

6.3.1 The Library module

The library module includes two different categories of template files. They are Java templates and Tools Templates. Both types of template files are used as an archetype for creating java application files.
6.3.1.1 Java template file

The Java template files are basically for building application that perform agents’ activities and Naming Service registration. We have three kinds of agents. They are PushServer, PushClient, PushClientPushServer.

6.3.1.1.1 PushServer

The role of PushServer is to act as an agent, which pushes events into event channel. PushServer is a java application that accepts input from user. User types in an input string. The PushServer obtains input string from the user and then pushes the event to the correct event channel. The event channel will push the event to the subscribers. The subscribers could be a PushClientPushServer or a PushClient in DAAG. Thus, agent PushServer performs two tasks, they accept text input from user and passes input test to event channel as a new event.

PushServer.java contains the Java code for Push supplier. Push Supplier creates an ORB to get the object reference for event service. Then it retrieves the Proxy for PushConsumer and connects the implementation to ProxyPushConsumer.

```java
public class PushServer {
    public static void main(String args[]) {
        Properties props = System.getProperties();
        props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
        props.put("org.omg.CORBA.ORBSingletonClass", "com.ooc.CORBA.ORBSingleton");
        System.setProperties(props);
```
try
{ ORB orb = ORB.init(args, props);

try
{ obj = orb.resolve_initial_references("EventService"); }
catch(org.omg.CORBA.ORBPackage.InvalidName e) { }

EventChannel e = EventChannelHelper.narrow(obj);
SupplierAdmin supplierAdmin = e.for_suppliers();
ProxyPushConsumer consumer = supplierAdmin.obtain_push_consumer();
PushSupplier_impl supplier = new PushSupplier_impl(consumer, 0, ecName,"channelName2", agentName);

try
{ consumer.connect_push_supplier(null); }
catch(AlreadyConnected ex) {} }

PushSupplier_impl.java

The PushServer creates an PushSupplier_impl object.

6.3.1.1.2 PushClient

The role of PushClient is to act as an agent, which receives events from event channel. PushClient is a java application that performs a specific task. This task could be defined by designer or rented from ASP. PushClient receives the event from event channel to pass it to the defined task implementation. The result will be sent back to the PushClient. PushServer is the starting point of system process while PushClient is the end point of system process.

6.3.1.1.3 PushClientPushServer

Unlike PushClient and PushServer, the PushClientPushServer receives events from event channel and pushes events to another event channel immediately. It is a two-ends
communication that communicates with at two event channels. The event channel in the
front of PushClientPushServer pushes event to PushClientPushServer. The event channel
at the back of PushClientPushServer receives events from PushClientPushServer. Same
as PushClient, PushClientPushServer performs tasks assigned by software designers. The
PushClientPushServer is always in between of the PushServer, PushClient or another
PushClientPushServer.

6.3.1.1.4 PushConsumer_Impl.java

The PushConsumer_impl.java has the implementation of PushConsumer. It controls the
behavior of a PushConsumer defined by the actions that PushConsumer will be
performed. It also calls the remote process to perform a task. The method for
disconnecting push consumer is also defined in this program.

6.3.1.1.5 PushSupplier_Impl.java

Unlike PushConsumer_Impl.java, PushSupplier_impl gets input from standard input if
the flag is set. The flag is set only when this Supplier is a root agent. PushSupplier_impl
pushes incoming event to event channel. It does not invoke other external services. It also
defines the method for disconnecting push supplier.

6.3.1.1.6 ReadFile.java

This template is used to produce a Java application that is able to read our database. In
fact, the Database that we are using is a union text file. It is not possible to open this file
with regular text editor. This is because Java does store data in its own format, the union code.

6.3.1.2 Tools Templates

It consists of templates for producing makefile.mak, ob.config:

6.3.1.2.1 Makefile.mak

The Makefile.mak will copy from the Daag\Tools directory to the destination directory specified by software designer. Makefile.mak file will be used to compile all the java files as well as the IDL files. With the Makefile, software designer or can compile the code files easily. Therefore, designers can focus on the input specification file only.

6.3.1.2.2 ob.config

The ob.config file stores the IP address and port number for the Naming services and the Event Channel.

6.3.1.3 The Naming File

The NamingSericeReg.java is the only Java template that belongs to this group. The naming Server template contains the code that need to register for an IDL object. The Naming Service is used to store the object reference of IDL implementation, which are the action that will be performed by agents. This object reference can be pointed to a remote rental service offering by ASP.
6.3.1.4 The IDLs and their implementation files.

This Library contains two types of template, they are the interface definition language file templates and the implementation Java files templates. If we are using the ASP service, then we do not need to have the implementation file provided by ASP. In fact, we do need the IDL in order to make a successful distributed process communication. Since IDL needs to be compiled to generate the subs and skeletons. A sub is used for the client side to perform marshalling/unmarshalling and skeleton is used on the server side to perform the marshalling/unmarshalling.

The following are the services that we are using to test DAAG. These services have IDL files and their associated implementation file.

**AppendText**: It takes an input parameter and append it with a ‘*’.

**CallCourierForPickup**: It returns a message to the caller to pretend that it pages courier.

**FilterPickup**: It filters pickup requests and see if it belong to the local service port.

**SendString**: It returns a simple string.

**WriteDataToDataBase**: It writes input parameter into database.

Let us consider WriteDataToDataBase services for illustration. For the WriteDataToDataBase service, we need its IDL file since IDL interface defines the contract between the client and server parts of WriteDataToDataBase application. IDL also specifies the operations and attributes that are available.
The IDL file for WriteDataToDataBase is not complicated. Its interface has a single operation. Usually there are three steps for writing IDL:

   Step 1: Declaring the CORBA IDL Module
   Step 2: Declaring the Interface
   Step 3: Declaring the Operations

Since DAAG have the ability to use services provided by ASP, we do not need to have module in the IDL. A CORBA module is a namespace that groups the related interfaces and declarations together. Our first task for building IDL is to declare the Interface.

CORBA IDL declares the API contract that an object has with other objects. In our case, each interface statement in the IDL maps to a Java interface statement when we perform the mapping. Interface statement of the Java code will be generated when we compile this statement in IDL.

```
interface WriteDataToDataBase
{
};
```

Next, we need to declare operation(s). IDL operations define the behavior that servers will be performed on behalf of clients. Each operation statement in the IDL generates its corresponding method statement in the Java interface. This action is performed when we compile the IDL files with IDL compiler. WriteDataToDataBase has a single operation that is writeDataToDataBase. The input parameter for this operation is a String data type.

Below is IDL interface that describes the WriteDataToDataBase program.

72
interface WriteDataToDataBase
{
    void writeDataToDataBase(in string inString);
};

The compilation of IDL file performs the mapping from WriteDataToDataBase.idl from
IDL to Java. Depending on the options chosen on the command line, IDL compiler
generates a number of files. The following are the files generated by IDL compiler:

_WriteDataToDataBase_ImplBase.java
    It is an abstract class that acts as server skeleton. It provides CORBA
    functionality for server. It has the WriteDataToDataBase.java interface.

StubForWriteDataToDataBase.java
    It is a client stub that provides CORBA functionality for client. It implements the
    StubForWriteDataToDataBase.java interface.

WriteDataToDataBase.java
    It is a Java version of IDL interface. It has the single method
    writeDataToDataBase. It provides standard CORBA object functionality by
    extending org.omg.CORBA.Object.

WriteDataToDataBase.Helper.java
    It is a class providing auxiliary functionality and notably the narrow method
    required casting CORBA object references to their proper types.

WriteDataToDataBaseHolder.java
    It is the class that holds a public instance member of type WriteDataToDataBase.
    It provides operations with in and out arguments.

73
WriteDataToDatabase_impl.java

IDL compiler produces Java files stub and skeleton files. We also need to develop the implementation file for WriteDataToDatabase. This implementation file implements the operation that is defined in WriteDataToDatabase.idl. The following is the essential part of the implementation file.

class WriteDataToDatabase_impl extends _WriteDataToDatabaseImplBase {
    private RandomAccessFile output;
    public void writeDataToDatabase(String inString)
    {
        openFile();
        if ( output != null ) {
            addRecord(inString);
            closeFile();
        }
    }

    private void openFile()
    {  // Open the file
    }

    private void closeFile()
    {
        //Close Data File
    }

    public void addRecord(String inString)
    {
        //create an PuroRecord and add it to the data file
        PuroRecord record = new PuroRecord();

        try {
            record.setInfoString(inString);
            output.seek( output.length() );
            record.write( output );
        } catch ( IOException io ) {
            closeFile();
        }
    }
}
Once server invokes the method `writeDataToDataBase`, method `writeDataToDataBase` will open the data file. After the file is opened, it creates an object for `PuroRecord` and assigns the input parameter to the object. The `PuroRecord` will write its attributes into data file.

`PuroRecord` is the business object that we use to capture the information to be inserted into database. Since `PuroRecord` is implemented with Object Oriented methodology, it is more flexible to change attributes of the class in the future [SHA95]. It also contains method that can be used to write the information to the data file. It also has operation for returning the value of attributes. In our case, `PuroRecord` has only one string attribute called `infoStringToWrite`.

`PuroRecord` contains an attribute, which is a String. This String basically stores all the information that needs to be read from or written to the database. Even though the `PuroRecord` also has the ability to write its record information to the specified RandomAccessFile, we use the RandomAccessFile to simulate the flexible behaviours of a database. The `SetInfoString` and `GetInfoString` methods in `PuroRecord` sets and returns the string attribute of `PuroRecord` respectively.

### 6.3.2 The Middle Layer

The middle layer of DAAG architecture is the Core Java application, that is, the `Daag.java`. Before we talk about the `DAAG.java`, we first look at our input prototype for
our system. This way, it will be much easier to understand the reason why the DAAG application is associated with two Java objects. These two Java objects are Node and Edge.

6.3.2.1 Input prototype

The input prototype is designed to be simple and clear structured. It contains the following:

\[ N \] : represents Node of system graph, which contains a finite set of agent names and their action. ie. \{(Agent name, Action)\}

\[ E \] : represents Edge of system graph, which contains a finite set of pair of agents. ie. \{(Agent name, Agent name)\}

**EeventService** : represents the IP address of the machine that CORBA Event Service is going to run from. If its value is ‘default’, it assumes that the Event Service will be executed on the machine using to generate final system.

ie. default, 137.127.160.2, mantra.lam2f.uwindsor.ca

**NamingService** : represents the IP address of the machine that CORBA Naming Service is going to run. If its value is ‘default’, it assumes that the Naming Service will be executed on the same machine using to generate final system.

ie. default, 137.127.160.2, mantra.lam2f.uwindsor.ca

**InterfaceRepository** : is the IP address of the machine that CORBA Interface Repository Service is going to run. If its value is ‘default’, it assumes that the Interface Repository Service will be executed on the machine using to generate final system.

ie. default, 137.127.160.2, mantra.lam2f.uwindsor.ca

76
Definitions of the interfaces to objects can be defined statically or dynamically. Statically, Interfaces can be defined in an interface definition language, called the OMG Interface Definition Language (OMG IDL). As we stated before, IDL defines the types of objects according to the operations that may be performed on them and the parameters to those operations. Dynamical, Interface Repository service provides a way to allow adding interfaces during execution time. In other words, this service represents the components of an interface as objects, permitting runtime access to these components. In any ORB implementation, the Interface Definition Language and the Interface Repository have the same abilities.

**DAAGRootDirectory**: indicates the Destination Root Directory for the final product.

`ie C:\DAAG`

6.3.2.2 The DAAG logic and its Java components

DAAG.java, Node.java and Edge.java are the core Java application and classes for building the DAAG system. Before we discuss the DAAG.java, we first look at its supplementary classes.

6.3.2.2.1 Node Class

The Node.java is a java class that stores information for a node. A node class represents an agent. It contains the following attributes and methods:

**Attributes:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>member access modifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>agentName</td>
<td>String</td>
<td>private</td>
<td>Name of Agent</td>
</tr>
<tr>
<td>agentService</td>
<td>String</td>
<td>private</td>
<td>name of action that the agent performs</td>
</tr>
<tr>
<td>Attribute</td>
<td>Type</td>
<td>Access</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------</td>
<td>---------</td>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>inDeg</td>
<td>int</td>
<td>private</td>
<td>number of input agent</td>
</tr>
<tr>
<td>outDeg</td>
<td>int</td>
<td>private</td>
<td>number of output agent</td>
</tr>
<tr>
<td>inChannelName</td>
<td>String</td>
<td>private</td>
<td>the name of channel pushes event to node</td>
</tr>
<tr>
<td>outChannelName</td>
<td>String</td>
<td>private</td>
<td>the name of channel that the node pushes event to</td>
</tr>
<tr>
<td>inPush</td>
<td>boolean</td>
<td>private</td>
<td>set if input channel pushes event to the node</td>
</tr>
<tr>
<td>inPull</td>
<td>boolean</td>
<td>private</td>
<td>set if the node pulls event from input channel</td>
</tr>
<tr>
<td>outPush</td>
<td>boolean</td>
<td>private</td>
<td>set if the node push event to output channel</td>
</tr>
<tr>
<td>outPull</td>
<td>boolean</td>
<td>private</td>
<td>set if output channel pull event from the node</td>
</tr>
</tbody>
</table>

**Methods:**

<table>
<thead>
<tr>
<th>Name</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>getAgentName</td>
<td>sets the name of agent</td>
</tr>
<tr>
<td>getAgentService</td>
<td>sets the name of agent action</td>
</tr>
<tr>
<td>setInChannelName</td>
<td>sets the name of input channel</td>
</tr>
<tr>
<td>setOutChannelName</td>
<td>sets the name of output channel</td>
</tr>
<tr>
<td>getInChannelName</td>
<td>returns the name of input channel</td>
</tr>
<tr>
<td>getOutChannelName</td>
<td>returns the name of output channel</td>
</tr>
<tr>
<td>isInPush</td>
<td>returns true if the input channel push data to the node</td>
</tr>
<tr>
<td>isInPull</td>
<td>returns true if node pull data from input channel</td>
</tr>
<tr>
<td>isOutPush</td>
<td>returns true if node pushes data to output channel</td>
</tr>
<tr>
<td>isOutPull</td>
<td>returns true if output channel push data from node</td>
</tr>
<tr>
<td>setInPush</td>
<td>sets the node to push client mode</td>
</tr>
<tr>
<td>setInPull</td>
<td>sets the node to push client mode</td>
</tr>
<tr>
<td>setOutPush</td>
<td>sets the node to push server mode</td>
</tr>
<tr>
<td>setOutPull</td>
<td>sets the node to pull server mode</td>
</tr>
<tr>
<td>displayAgentInfo</td>
<td>displays the node information</td>
</tr>
</tbody>
</table>

As we can see from the attributes and methods tables, Node object basically contains the name of agent, the role of agent, the name of its input and output event channels and the communication mode between event channels. It also has methods to set and retrieve those attributes.
6.3.2.2.2 Edge Class

The Edge.java is a java class that stores the relationship between agents. An edge class represents a line between two nodes. It contains the following attributes and methods:

Attributes:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>member access modifier</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent1Name</td>
<td>String</td>
<td>private</td>
<td>Name of Agent</td>
</tr>
<tr>
<td>Agent2Name</td>
<td>String</td>
<td>private</td>
<td>name of action that the name performs</td>
</tr>
<tr>
<td>Agent1Action</td>
<td>int</td>
<td>private</td>
<td>number of input agent</td>
</tr>
<tr>
<td>Agent2Action</td>
<td>int</td>
<td>private</td>
<td>number of output agent</td>
</tr>
<tr>
<td>ChannelName</td>
<td>String</td>
<td>private</td>
<td>the name of channel pushes event to node</td>
</tr>
</tbody>
</table>

Methods:

| Name              | Action                                                        |
|-------------------|                                                              |
| setChannelName    | sets the name of event channel                                |
| getFirstAgentName | sets the name of first agent name                             |
| getFirstAgentAction| sets the delivery operation between agent 1 and channel       |
| getSecondAgentName| sets the name of output channel                               |
| getSecondAgentAction| sets the delivery operation between agent 2 and channel       |
| displayAgentInfo  | displays the agent information                                |

6.3.2.2.3 DAAG Logic

DAAG.java is a Java application that produces the final distributed system based on the input specification file. Thus, the first task for DAAG is to obtain the name of input specification file. After that DAAG will open input file and extract the information.

```java
try {
    //Read input
    BufferedReader stdinn = new BufferedReader(new InputStreamReader(System.in));
    System.out.print("Please enter input specification file name: ");
    String inputFileName = stdinn.readLine();
    System.out.println("Reading input specification file" + "+" + inputFileName + ".");

    //Create Data stream for input
    BufferedReader readinputFile = new BufferedReader(new InputStreamReader
```
6.3.2.2.3.1 Building Distributed Infrastructure

In fact, the extracted information such as the values of E and N are used to build the Edge and Node of the distributed system graph. Once again, we use a graph to represent our distributed system in mathematical form. A node is an agent that performs certain action. An edge is a relationship between two agents with an event channel in between. How do we capture the information of a Node and an Edge? We use Java objects Node to store node information and Java object Edge to hold edge data. The following program outlines the action for building the system graph. When the flat is 1, we are building Node objects. When the flat is 2, we are building Edge objects.

```java
StringTokenizer stDetails = new StringTokenizer(tempString,"","");
while (stDetails.hasMoreTokens()) {
    if (flat==1) { // Building Node Objects
        tempString= stDetails.nextToken();
        tempString2=stDetails.nextToken();
        tempString2=RemoveBracket(tempString2);
        if (tempString2.indexOf(".") > -1)
            tempString2=extractIDLmethods(tempString2);
        else
            System.out.println("***** "+tempString2);
        Node tAgent=new Node(tempString,tempString2);
        Agents.addElement(tAgent);
    } else { // Building Edge Objects
        tempString=stDetails.nextToken();
        tempString2=stDetails.nextToken();
        ...
    }
}
```

80
Node and Agent objects are stored in a Java vector instead of an array. Conceptually, vector and array are both collections of memory locations. Each location holds the same type of data. The major difference between an array and a vector is their expandability. Array length is fixed and must be indicated at the time when it is initialized. However, the length of a vector can grow as needed. For DAAG, we do not know how many Node and Edge will be used in the System. Actually, DAAG have high degree of ability in openness and Scalability. Thus, we decide to use vector to keep track of the Nodes and Agents. Actually when we build the Nodes and Edges, we also extract the IDL files and assign the services name and other information to the correct location. More details of extracting IDL information will be discussed when we are introducing the Node.java and Edge.java.

The next step is to assign the IP address to the services variables. Services represent all the CORBA services that are running within the system. They are Event Service, Naming Service, and Interface Repository Service. Since the input specification already provide the IP addresses for these services, we can simply assign the IP address provided by designer to the corresponding variable. However, if designer uses default as service IP, we assume the machine used to generate the DAAG applications is where the services will be run. Thus, we assign the local IP to those services with a default value. The way
we retrieve local host IP is by creating a Java InetAddress object, which is under the java.net package. The java.net package provides the classes for implementing networking applications. The InetAddress class represents an Internet Protocol (IP) address. The getLocalHost() method of InetAddress class returns the local host, and hence, we have the value for default IP address.

```java
public static void verifyIPs() {  
    String DefaultIP = null;  
    try{  
        DefaultIP = getDefultIP(); 
    } catch (UnknownHostException u) {  
        System.out.println("ip address of Naming Service is missing... default IP has been used");  
        NamingService = DefaultIP; 
    }  
    else 
        NamingService = NamingService.trim();

    if ( NamingService.compareToIgnoreCase("default") == 0) { 
        NamingService = DefaultIP; 
    }  

    ...............  
}
public static String getDefultIP() throws UnknownHostException {  
    InetAddress local = InetAddress.getLocalHost();  
    System.out.println("IP Address of Local Machine (default) ->" + local.getHostAddress());  
    return local.getHostAddress();
}
```

6.3.2.2.3.2 Calculate Event Channel Information

In this step, we are going to create the name for event channels. Then set the event channel name for agents that use that event channel. In order to find out how many event channels are needed, we look at how many InPush variables are set to true in nodes
vector. The variable InPush tells us if there is an event channel pushed event in a node. For DAAG architecture, an event channel only pushes event to one node but an event channel could receive event from more than one node. Therefore, variable InPush and event channel is a one-to-one relation. A node could have no input event channel. However, an event channel must have an output agent. Once we have the entire event channel for output agents, we can start checking the relationship between Edge and agents. For example, let an agent A be communicating with another agent B. We retrieve the input event channel name for agent B. We set the output event channel name of agent A same as the agent B' input event channel name. Thus, event channel decouples the communication between agent A and agent B. By doing the above two steps repeatedly until all the Edges have been verified, we have the system communication part done. At this time, we can start generating DAAG code files.

```java
for (int i=0; i<Agents.size(); i++) {
    tempNode1 = (Node) Agents.elementAt(i);
    if (tempNode1.isInPush())
    
        tempNode1.setInChannelName("Ch"+counter);
        counter++;
    
}

for (int p=0; p<Edges.size(); p++) {
    tempe = (Edge) Edges.elementAt(p);
    tempNameAgent1 = tempe.getFirstAgentName();
    tempNameAgent2 = tempe.getSecondAgentName();

    for (int i=0; i<Agents.size(); i++) {
        tempNode1 = (Node) Agents.elementAt(i);
        if (tempNameAgent2.compareTo(tempNode1.getAgentName()) == 0)

            channellName = tempNode1.getChannelName();
        for (int j=0; j<Agents.size(); j++) {
            tempNode2 = (Node) Agents.elementAt(j);
            if (tempNameAgent1.compareTo(tempNode2.getAgentName()) == 0)

                tempe.setChannelName(channellName);
        }
    }
```
6.3.3 The Lower Layer

The lower layer consists the application files module, batch files module and tools module.

6.3.3.1 Clearing the Destination Directory

Designer needs to specify the destination file directory in an input specification file. If default has been entered, we assume that destination DAAG root directory is the location where DAAG is executing. If the directory is not found, user will be prompted if he/she wants to create one. If the directory is already existing, user will be prompted asking if he/she wants to overwrite the existing directory. This adds more security and safety to user environment.

6.3.3.2 Building Java Application Programs

After the execution of the steps above, vectors node and edge store all the information for agents and their communication paths, DAAG can start generating Java programs now. When DAAG is building Java program, it also needs the information stored in our library. DAAG opens library files, and goes through each line, if needed. DAAG scans library templates. If it finds DAAG action tags, it will call the conversion routines. If
none of DAAG expression control identifiers is in the statement, DAAG will just simply copy the statement to the output file.

There are two kinds of DAAG action identifiers. They are action expression control tags and action name replacing tags.

Action expression control tags:

- **Block Start**: it indicates the beginning of DAAG block.

- **Block End**: it indicates the end of DAAG block

  When the while statement is executed, DAAG begins by evaluating the expression. DAAG executes the statement block. After completing the statement block, it returns to the beginning of the loop to test the expression again. DAAG repeats this process until the number of executed time is equal to the number of services used in the final system. For the current DAAG design, the number of services is the number of different operations defined in the input specification files.

- **Control**: it indicates a control statement

  DAAG evaluates control statement. Control statement is an expression that will be passed to BuildDaagFile parsing routine. BuildDaagFile is a method that converts the following name replacing identifiers into a suitable name.

Action name replacing tags:

- **EventService**: IP address for the CORBA Event Service

- **NamingService**: IP address for the CORBA Naming Service

- **InterfaceRepository**: IP address for the CORBA Interface Repository Service

- **ServiceName**: an element in the ServiceName array
- **MethodName**: an element in the MethodName array

- **Indication**: a cursor for the iteration that has the value between 1 and number of Services

When we are replacing the ServiceName and MethodName, we use the local cursor to locate the right element in the array. We also use the value of local cursor to replace the value of indication. This local cursor will be incremented by 1 after each loop. Its value is in between one and the number of services that is specified in the input specification file.

Here is a sample code from library template:

```java
*Block Start*
*Control*  case -Indication- : inst-ServiceName- = -ServiceName-Helper.narrow(cObj);
*Control*    if(inst-ServiceName- == null)
              throw new RuntimeException();
*Control*    System.out.println("Resolved `inst-ServiceName-` in naming context `nc2" +
                     "in naming context `nc`");
*Control*    break;
*Block End*
```

If the ServiceName ={ AppendText, WriteDataToDataBase} and noOfService is 2, the sample generate code could be as following:

```java
case 1 : instAppendText = AppendTextHelper.narrow(cObj);
          if (instAppendText == null)
              throw new RuntimeException();
          System.out.println("Resolved `instAppendText` in naming context `nc2" +
                               "in naming context `nc`");
          break;
case 2 : instWriteDataToDataBase = WriteDataToDataBaseHelper.narrow(cObj);
          if (instWriteDataToDataBase == null)
              throw new RuntimeException();
          System.out.println("Resolved `instWriteDataToDataBase` in naming context `nc2" +
                               "in naming context `nc`");
          break;
```
The second example is one of the statements in ob.conf template for building a tool file - ob.conf:

```
*Control*ooc.service.EventChannelFactory=iiop://-EventService-:7022/DefaultEventChannelFactory
```

DAAG parsing should return the following statement if the EventService IP is equal to 10.128.233.172

```
```

A few of DAAG program are duplicated from template library without any modification. Those files are PushServe, BuildSupplierImp, services IDL files and their implementations, the tool file ReadFile.java, the Makefile for UNIX and makefile.mak for WINDOWS. The DAAG BuildDaagFile parsing routine examines the rest of them.

### 6.3.3.3 Building DAAG batch files

Once those java programs, IDL files configuration files and makefile are built, DAAG generates Batch files. A batch file is a text file that contains a sequence of commands. A set of commands is batched into a single file. This is the reason why it is called a batch file. In UNIX-based operating systems, batch file is called shell script with an executable mode. In the DOS Operating system, a batch file ends with the file name extension ".BAT". The reason that DAAG produces batch file is because we want to simplify the complexity of execution of DAAG applications. Otherwise, a long command line with many arguments would have to be presented to the system interactively from a keyboard one at a time.
The following are two batch files. The first one is to bring Agent1 (Agent1.bat) up and the other one is for invoking Naming Service (NamingServer.bat)

Agent1.bat:

```
java project.PushServer Ch1 -ORBconfign ob.conf Agent1
```

NamingServer.bat

```
```

With the batch files, users or application operators do not need to have any CORBA programming concept. They do not need to find out the IP addresses for each of services as well. A high level of location transparency is given with this technique.

6.3.3.4 Building Tools Files

The makefile.mak and ob.conf are also being generated in this stage.

6.4 A complex example

Figure 20 describes a more complex system that DAAG is able to generate. It is a package pick-up system. This pick-up system allows call centre representatives and Internet customers to schedule pick-up. Pick-up data will be stored into database. Local couriers will be paged to pick-up customers’ packages. The only requirement for producing this system is to provide the input specification file of the design.
The following is the input specification file, which is designed by software designer or architect.

\[ N = \{(CA1,Root),(CA2,Root),(PA1,Root),(PA2,Root),(IC,Root),(CMS,AppendText.idl),(CCDC,AppendText.idl),(WDC,AppendText.idl),(CCSS,AppendText.idl),(WCSS,AppendText.idl),(DS,AppendText.idl),(DD,WriteDataToDataBase.idl),(TorontoD,FilterPickup.idl),(WindsorD,FilterPickup.idl),(TCP,CallCourierForPickup.idl),(WCP,CallCourierForPickup.idl)\} \]
\[ E = \{(CA1, CMS),(CMS, CCSS),(CCSS, DS),(DS, DD),(DS, TorontoD),(DS, WindsorD),(TorontoD, TCP),(WindsorD, WCP),(CA2, CMS),(PA1, CCDC),(CCDC, CCSS),(PA2, CCDC),(IC, WDC),(WDC, WCSS),(WCSS, DS)\} \]
\[ NamingService=default \]
\[ EventService=default \]
\[ InterfaceRepository=default \]
\[ DAAGRootDirectory=C:\DAAG \]

The design specification defines the behaviour of the nodes and relationships between nodes. We have five starting agents. They are CA1, CA2, PA1, and PA2. CA1 and CA2 are the Cyber call centre agents. PA1 and PA2 are the customer representatives at the regular call centre. IC is an Internet customer. CMS, DCDC, WDC, CCSS, WCSS and DS represent the Monitoring Systems, Call Centre Data Centre, Web Data Centre, Call
Centre Services Server, Web Services Server and Dispatch Server respectively. These services perform the same task, which displays input events. DD stands for the Dispatch Database, which writes event into database. Toronto Dispatch (TorontoD) and Windsor Dispatch (WindsorD) filter events that are not belonging to them. TCP and WCP are the Paging systems for Toronto and Windsor. Paging system pages local couriers to pick-up packages.

The edge E defines relationships between agents. This example uses the default IP for the IP address of Naming Service, Event Service and Interface Repository. DAAGRootDirectory stores the destination of the resulting system. In DOS or Unix prompt, we type `java DAAG` to start the DAAG application. DAAG will prompt designer for the name of input design specification.

```
C:\DAAGBuilder\Tools>java Daag
Please enter input specification file name: Courier.dat
```

Since designer user ‘default’ for the IP address in this design specification file, the local IP address is determined by DAAG before producing the resulting system. After this, DAAG builds the Edge-Node tree and determines how many event channels are needed.

The following shows the relationship between nodes and event channels

```
.... Collecting Edge Information
  (<CA1)   Act:(Push)>---(Ch1)---<(<CMS)  Act:(Push)>
  (<CMS)   Act:(Push)>---(Ch4)---<(<CSSS)  Act:(Push)>
  (<CSSS)  Act:(Push)>---(Ch6)---<(<DS)  Act:(Push)>
  (<DS)    Act:(Push)>---(Ch7)---<(<DD)  Act:(Push)>
  (<DD)    Act:(Push)>---(Ch7)---<(<TorontoD)  Act:(Push)>
  (<TorontoD) Act:(Push)>---(Ch9)---<(<WCP)  Act:(Push)>
  (<WindsorD) Act:(Push)>---(Ch9)---<(<CMS)  Act:(Push)>
```
If the destination directory already existed in machine, DAAG asks designer if he/she wants to overwrite the directory. Then DAAG produces Java files for the resulting system.

The Directct C:\DAAG\Daag\ already exists. Do you want to overwrite it? y

... Clearing DAAG Directory (C:\DAAG\Daag)        
... Creating DAAG Directory File (C:\DAAG\Daag\DAAG.mak)       
... Creating DAAG IDL Implementation File (C:\DAAG\Daag\PushSupplier_impl.java)
... Creating DAAG IDL Implementation File (C:\DAAG\Daag\PushConsumer_impl.java)
... Creating DAAG Application File (C:\DAAG\Daag\PushClient.java)
... Creating DAAG Application File (C:\DAAG\Daag\PushServer.java)
... Creating DAAG IDL File (C:\DAAG\Daag\AppendText.idl)
... Creating DAAG IDL Implementation File (C:\DAAG\Daag\WriteDataToDataBase.idl)
... Creating DAAG Application File (C:\DAAG\Daag\PeruRecord.java)
... Creating DAAG IDL File (C:\DAAG\Daag\FilterPickup.idl)
... Creating DAAG IDL Implementation File (C:\DAAG\Daag\CallCourierForPickup_impl.java)
... Creating DAAG Application File (C:\DAAG\Daag\ReadFile.java)
... Creating DAAG MakeFile File (C:\DAAG\Daag\makefile.mak)
... Creating DAAG Configuration File (C:\DAAG\Daag\job.conf)
... Creating DAAG Batch File (C:\DAAG\Daag\CA1.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\CA2.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\PA1.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\PA2.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\IC.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\CMS.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\CCDC.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\WDC.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\CSS.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\WSSS.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\WDC.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\WDC.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\EventServer.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\NamingServer.bat)
... Creating DAAG Batch File (C:\DAAG\Daag\RegNameService.bat)

At the same time, DAAG builds the batch files, IDL files implementation files, configuration and makefile.mak for the resulting system. After the DAAG completes its tasks, we need to go to the subdirectory under the destination directory and compile the
IDL and java files. In order to perform the compilation, we type the following command to execute the makefile.mak generated by DAAG.

```
C:\DAAG\Daag>nmake /f makefile.mak
    jidl -package project -output-dir generated *idl
    set CLASSPATH=..\lib..\lib\lib;.;c:\Job3.2\OB.jar;c:\Job3.2\OBEvent.jar;c:\Job3.2\OBNaming.jar;c:\Job-3.2\OBProperty.jar;c:\Job-3.2\OBTest.jar;c:\Job-3.2\Job-3.2\lib\lib;c:\Job-3.2\Job3.2 \event\lib;c:\Job-3.2\Job-3.2\naming\lib;c:\DAAG\Daag\classes
    javac -d classes *.java generated\project\*.java
```

Before we start running the resulting system, we need to bring up the event server, naming server by executing their batch files.

```
C:\DAAG\Daag>EventServer
C:\DAAG\Daag>java com.ooc.CosEventServer.Server -ORBconfig ob.conf
```

```
C:\DAAG\Daag>NamingServer
```

Then we need to register the Services in the Naming server.

```
C:\DAAG\Daag>regnameservices
C:\DAAG\Daag>java project.Server -ORBservice NameService iioop://149.99.116.253:5000/
NameService
AppendText is registered
WriteDataToDataBase is registered
FilterPickup is registered
CallCourierForPickup is registered
```

We start the agents by typing their batch file, such as typing CA1 for staring first Cyber Agent CA1 and CMS for string the Cyberagent Monitoring system.

```
C:\DAAG\Daag>CA1
C:\DAAG\Daag>java project.PushServer Ch1 -ORBconfig ob.conf CA1
'Channel Ch1' was created
CA1 is ready for actions
Please enter the shipment information
>
```

```
C:\DAAG\Daag>CMS
C:\DAAG\Daag>java project.PushCPushS 1 Ch1 Ch4 -ORBconfig ob.conf CMS -ORBservice
NameService iioop://149.99.116.253:5000/NameService
'Channel Ch4' was created
```
Resolved `instAppendText' in naming context `nc2'in naming context `nc'

By typing the entire batch files for the agents, we have the pick-up system running and able to schedule pick-ups for customers.
7. Thesis Evaluation and future work

In summarizing the work set forth in this thesis, it is useful to analyze the original stated goal for this research. As presented in Chapter 1, the aim of this research was to evaluate asynchronous process communicating and define a methodology for building distributed applications with asynchronous process communicating.

To achieve the above goals, the CORBA middleware was thoroughly analyzed and used to develop a prototype that can shorten the development and design time for budding distributed application

7.1 Evaluation

Seven objectives were defined for this research. These objectives that we have discussed in Chapter 1, will be reviewed and analyzed in order to evaluate the work presented.

7.1.1 Requirements

The first objective was:

To analyze the processes communication and Client/Server Architectures and address the new trends of software development.

This objective was met in Chapter 1 and Chapter 2. Chapter 1 introduced the general concepts related to client/server Architectures and process communication. This chapter also addressed the new trends of software development. We concluded that the software
and process-time renting would play an important role in the near future of distributed system architecture design.

7.1.2 Background

The second objective was:

To analyze the different types of communication protocols and their features

Chapter 2 pointed out the different communication protocols. We also discussed their features and weaknesses. The processes communication models, such as the Request/Response communication model and event-based communication model are also addressed in the same chapter in detail. With the discussion of the above two communication models, we are confident that we have justified the reasons for choosing these communication methodologies and applying them in this thesis.

7.1.3 Current Work in the Field

The third objective was:

To analyze the concepts of object management architecture and CORBA computing and distribution models, its Event service and naming service. To analyze the Calculus of Communication System and demonstrate its usage.

In Chapter 3, we discussed the concept of OMA. The features of CORBA, and its computing and distributed models are also addressed in the same chapter. The CCS was
introduced in chapter 4 together with the demonstrations of the usage of the basic CCS notation. We mentioned the new software development trend in Chapter 5.

7.1.4 Implementation

The fourth objective was:

To design the architecture and specification for the DAAG Code generator that is applying the asynchronous processes communication.

The algorithms and specification of automatic code generator for building distributed system applications were discussed in Chapter 6. The resulting systems that DAAG produces can have infinite layers, in a hierarchical-based processes communication system. The DAAG system models asynchronous process communication with the new software development trend that allows adopting rental processes through the distributed network.

7.15 Correctness

The fifth objective was:

To prove the correctness of the system that is generated by the DAAG.

In Chapter 6, we prove the DAAG by using the CCS. By passing the process notation to CWB, we will able to model the behavior of DAAG. Therefore, system designer is able to predict the activities of the system before running the system in production.
7.1.6 Demonstration

The sixth objectives was:

To demonstrate that DAAG achieved the goals of this thesis.

In the Chapter 6, we demonstrate the example of the DAAG, which rapidly produces a courier application for handling shipments. This application allows call center agents to schedule a pick up and then dispatch the pick up request to a local courier.

7.1.7 Future Work

The seventh objective was:

To proceed further development and research based on the knowledge, examination and evaluation of the theories of asynchronous process communication with using CORBA event services.

Although DAAG can be considered as a final or complete product, it can only be ranged as the first comprehensive layer of functionality for the asynchronous process communication model providing rapid prototype for software development. Obviously, the functionality provided by DAAG can be augmented by prolongation, or by enhancement.

7.2 Areas of Future Development

This section is an account of the future development issues for DAAG. As explained through this work, the implementation provides a well-defined functionality to process
communication. Consequently, future developments are aimed to enhance such functionality either by improving the techniques applied to their inner mechanisms, or by extending them with new functionality.

7.2.1 Extending Functionality

Missing the Pull operation is one of the major limitations of DAAG. The major aim of this thesis is to develop an application, which provides high-speed of data communication, however, in some situations, we may want to use the Pull operation in order to achieve the best processes performance or fulfill the needs of operations. Therefore, the specialization of existing functionality to perform pull operations is the extension of functionality. This specialization is achieved by adding classes and methods to support operations specific to consumers and suppliers, and by modifying the input specification to allows designers take advantage of these functionalities.

7.2.2 Improving Functionality

Refining algorithms and classes to increase their performance and usability is the core in functionality improvement. Adding new enhancements are also part of the improvement.

- Gaining the True Real-time process communication

  Apply the real-time process communication methodologies to allow the current DAAG to produce true real-time processes communication application system. This can be done by researching the current technologies in both hardware and software discipline.

- The Graphical User Interface
The current version of DAAG is able to read input design from the data file only. It lacks the abilities for visual design. Creating a design prototype, evaluating the prototype, and then repeating the process several times is the value of iterative design. It is one of the reasons why applications with using GUI are usually more robust than the one without using it.

- Automation on uploading final applications into the desired locations

Manually loading applications is required in the current version of DAAG. In order for system design to gain advantages of distributed systems, we always need to run processes in different machines. The loading mechanism can be performed by a ftp program or client/server application to dispatch DAAG components into the desired destinations.

- Single producer with MULTI-Event Channel

Each producer in the current version of DAAG is associated to a single event channel. Allowing producer to communicate to more than one event channel will reduce the restrictions on the DAAG.

- MISCELLANEOUS IMPROVEMENTS

The miscellaneous improvements include the following:

1. Add more services to the existing DAAG for the designer to use.

2. Improve the existing methods for catch fetal input IDL files.

3. Introduce the problem log to capture the run time exceptions.
8. Conclusion

The information technology is moving to a new decade, more and more applications will be developed. Applications are much more complex that ever. Building complex application requires a large amount of cost to develop, maintain, and upgrade. In this thesis, we introduced a new methodology for building distributed application. We named it DAAG. We also predict the future trend of application development.

With the DAAG, the service processes that we defined are available to all client applications. Developer can code complex business logic, access data stored in a database, and call out to external systems with a simple call. The DAAG service processes are needed to make functionality of service objects available to clients. Service process is a separated process, usually anchored on a remote server and its location is transparent to their clients. Thus, many clients are able to share the same service provided by a service processes located anywhere within the same distributed network.

We know that adding new features to the existing application always runs the risk of introducing new, sometimes intractable bugs and endangers backward compatibility with earlier versions. Moreover, most applications are monolithic, providing rich sets of features but no easy way to add missing features or remove unneeded ones. DAAG allows software design adding new features easily without adding risk. It is because all the DAAG services processes should be well tested before it is using in the production. Also, the formal model of DAAG proves the designs of new application system. In
addition, applications built by DAAG are longer monolithic, it is very easy to remove unneeded features in DAAG application. DAAG is designed to provide the flexibility to be used to design a variety of distributed systems over a spacious range of industries.

We believe renting software over the Internet is going to be one of the major future trends in software usage. This trend also impacts the software development methodology significantly. DAAG is fully adapted to this trend. In addition, designer has the choice to rent services or build and maintain their own service processes. CORBA technology provides location transparency and processes communicating abilities to DAAG [HS98]. As a result, DAAG processes running in entirely different vendors’ machines with different architectural designs are able to work together successfully.

Overall, this thesis has presented the DAAG prototype for software designers to design and build legacy applications within a short period of time. By modeling distributed asynchronous process using events and CCS, we introduced the methodology for integrating new development trends and rapid prototyping for building N-Tier distributed systems applications. The ability to move processes across platforms and domains easily is also one of the strength of DAAG. It is because software designer can use our prototype defining and changing the relationship between communicating processes. Moreover, designer can observe the behavior of the system in the design stage. The DAAG applications have been demonstrated to support the elicitation of the concept of software re-use and efficiency in distributed process communication.
Bibliography


VITA AUCTORIS

Patrick H. Chan was born in 1973 in P.R.China. He completed his High School in Toronto, Ontario in 1993. From there he went on to University of Windsor where he obtained a Bachelor of Computer Science with honours co-op in 1997. He is currently a candidate for the Master’s degree in Computer Science at the University of Windsor and will graduate in the fall of 2000.