WEB APPLICATION FRAMEWORK IN ERASMUS

Duo Peng

A thesis

in

The Department

of

Computer Science & Software Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Master of Computer Science

Concordia University

Montréal, Québec, Canada

March 2012

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Duo Peng**

Entitled: **Web Application Framework in Erasmus**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

______________________ Dr. R. Jayakumar Chair

______________________ Dr. C. Constantinides Examiner

______________________ Dr. Joey Paquet Examiner

______________________ Dr. Peter Grogono Supervisor

Approved ______________________

Chair of Department or Graduate Program Director

__________ 20 ____ ______________________

Dr. Robin A.L. Drew, Dean

Faculty of Engineering and Computer Science

Abstract

Web Application Framework in Erasmus

Duo Peng

This thesis introduces a new framework in Erasmus which is specially designed for web application. Through analysis of this framework, it shows that the new language Erasmus has potential to be able to improve the efficiency of web application development and the performance of the web service. Compared with the traditional object oriented language, as a process oriented language and concurrent language, Erasmus is expected to be more suitable for web application development.

Erasmus, as a process oriented language, developed by Peter Grogono at Concordia University, Canada and Brian Shearing at The Software Factory, UK, is based on communicating processes. Comparing the popular Object Oriented languages such as Java, ASP, C++, which make it difficult to write multithreaded programs that use many concurrent processors efficiently, Erasmus can easily and clearly depict multi-process scenarios based on new concepts: protocol, port, cell and processes. In Erasmus all the processes are linked with the roles: Client and Server. A process can be a client of one process and also be a server of other processes. The processes communicate with each other using a protocol via server and client ports no matter whether they are local or remote. So, Erasmus is not only a new language, furthermore, it brings us a new view to observe multi-process systems and it gives us a new method to describe the system.

Why is Erasmus suitable for web development? First, a web Application is a typical client-server system in which browsers are clients and web applications are servers; Second, a web Application is a typical multi-process system in which concurrency is a basic status. Third, a web Application is always running on distributed environment. Comparing the characteristics of Erasmus and web Application, we reach the conclusion that Erasmus is really suitable for web application development. The advantages of Erasmus are realized in web application development.

In this thesis, we firstly give detail of the language Erasmus and explain the new concepts: Cell, Process, Protocol and Port which is introduced by Erasmus. Then, we introduce the broker which is a proxy to communicate between the processes on different machines. After the introduction of broker, we describe how to dispose HTTP protocol which is the basic protocol for web application with Broker. Next, we depict the architecture and the resource management of the Erasmus web application including database access and dynamic process creation. At the end, we give an example of a simple web application with simple Erasmus language and with an Erasmus MVC framework.

My web application compiler is build on the Eramus compiler provided by Dr. Peter Grogono. Using this compiler, we successfully developed a basic web application which can provide basic service, for instance, getting web page, accessing database and distributed communication. But, this framework now is only for simple web applications to prove the feasibility of the ideas. For complicated web application in real life, we still have a lot of work to implement it.

Acknowledgments

I thank my supervisor Dr. Peter Grogono for his advice and support. I have gained a lot of knowledge through his wisdom and insights. The Erasmus compiler which I am using to develop my web application is built on his Erasmus compiler. His advice has also been invaluable throughout this thesis work.

I also thank all the faculty members and staff of the Computer Science Department particularly the Graduate Advisor, Ms. Halina Monkiewicz for her advice.

I am grateful to the Faculty of Engineering and Computer Science, Concordia University for supporting in part this thesis work.

I thank all colleagues in our lab. I shared wonderful and memorable moments with them. The most import thing is that when I have some difficulties in my research, they always discuss with me and give me a lot of suggestions.

Finally, I would like to thank my family, my beloved wife, Ying Zhan, her support, encouragement and understanding makes this thesis be possible.

Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface
ASP	Active Server Pages
CGI	Common Gateway Interface
CMS	Content Management System
CRLF	Carriage Return and Line Feed
CRUD	Create Read Update Delete
DBI	Database Interface
DNS	Domain Name System
DSL	Domain-Specific Language
EJB	Enterprise JavaBeans
HTTP	Hypertext Transfer Protocol
IE	Internet Explorer
IETF	The Internet Engineering Task Force
IPC	Inter-Process Communication
J2EE	Java 2 Platform, Enterprise Edition
JDBC	Java Database Connectivity
JNDI	Java Naming and Directory Interface
JMS	Java Message Service
JSP	Java Server Pages
MVC	Model-View-Controller
OOL	Object-Oriented-Language
ORM	Object Relational Mapping

OS	Operating System
PRM	Process Relational Mapping
RFC	IETF Request For Comments
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
TCP/IP	Transmission Control Protocol/Internet Protocol
W3C	World Wide Web Consortium
WSDL	Web Services Description Language

Chapter 1

Introduction

A web application is an application that is accessed over a network such as the Internet or an intranet. Because of the ubiquity of web browsers, and the convenience of using a web browser as a client, from the 1990s, the World Wide web has experienced a dramatic growth. Nowadays, internet has become the main resource for people to get information. We can say, we are living on Internet.

The structure of web applications is called thin client because web application client is always a web browser: IE, Firefox or other browsers. For a web application, we can update and maintain web applications without distributing and installing software on potentially thousands of client computers. The browsers understand HTML and JavaScript, which makes the web application to be cross-platform compatible. Common web applications include webmail, online selling, online search, wikis and many other functions.

To meet the needing of web application development, Object-Oriented-Language (OOL) for Web-based software, such as J2EE, C#, Ruby, emerged and soon becomes the main software developing tool.

Web-based applications are really different from other traditional applications because they are concurrent, distributed systems which provide services for millions clients all over the world. The difference brings us a lot of challenges in developing web applications, for example, the securities, the response time, the complexity of system etc.

Traditional applications consist only of 1 tier because the application usually run on one machine, but a web application always is an n-tiered system. The most common structure is the three-tiered application which includes presentation, application and storage tiers. So, a web browser is the first tier (presentation), a language using some dynamic web content technology (such as ASP, ASP.NET, CGI, ColdFusion, JSP/Java, PHP, Perl, Python, Ruby on Rails or Struts2) is the middle tier (application logic), and a database is the third tier (storage). The web browser sends requests to the middle tier, which services them by making queries and updates against the database and generates a user interface. For more complex applications, a 3-tier solution may not be enough and more tiers are needed [8].

At the application side, we define more than 2 tiers. For example, according to MVC pattern, we divide application tier into 3 tiers: Controller, Model and View. To fulfill this structure, we have many of frameworks (Spring, EJB, etc.) to ease our development. These frameworks meet our needs, but at the same time, it makes our applications more complex and harder to maintain.

Nowadays, the popular web application development languages are PHP, ASP, JSP and Perl [29]. These languages are all Object-Oriented-Languages because Object is such a familiar paradigm for general purpose programming tasks that it is hard to imagine another paradigm replacing it. But software systems are becoming ever more complex and hard to maintain [35].

In substance, a Web-Based Application, as a concurrent, distributed and rich-clients application is a concurrent multi-processes application. From the definition, obviously, the concept of process is the core of the web Application. If we can have a Process-Oriented-Language to describe the web application, it should be more efficient and easier to understand than using OOL language.

The language Erasmus is just this kind of language. It is being developed by Peter Grogono at Concordia University, Canada and Brian Shearing at The Software Factory, UK, is based on protocols, ports, cells and processes. Concurrency in Erasmus is based on communicating processes [21]. It provides us with a brand new viewpoint to develop the Web-based applications.

With traditional object oriented languages, it is difficult to write multithreaded programs to use many concurrent processors efficiently. But Erasmus is a special process oriented language focusing on concurrent scenarios. By the concepts of Cell, Process, Protocol and Port, Erasmus is potentially better suited for concurrent tasks [27].

This thesis describes issues related to the implementation of Erasmus especially in developing web applications. It focuses on four aspects. First, we answer the question: is the language Erasmus is suitable to design web applications? Second, Erasmus is based on communicating process, in the thesis, we describe how to communicate among processes in distributed, concurrent environment and how to deal with HTTP requests in Erasmus. Third: For a web application, management of resources is a critical problem. In this thesis we explain in detail about the access of the most important resources: database access, file access and process creating in web application environment. Fourth: we introduce a simple web application designed in Erasmus and give an example of MVC structures in Erasmus.

Erasmus is not only a new language, it is but also a new concept; it brings us a brand new view point to design our distributed software system. With Erasmus, we expect system to become more scalable, more flexible and more efficient. At the end of the thesis, I compared the Erasmus with other web languages and get a conclusion that Erasmus is suited for web application development.

Of course, our Erasmus is under development, we still have a long way to make it practical and perfect. But, with the development of the hardware, increasing number of CPUs are introduced to one PC, more benefits of the Erasmus, as a process oriented language, will be realized.

Chapter 2

Background

This chapter introduces some background knowledge which helps us to understand Erasmus and ‘applications. In the first part of this chapter, we explain background knowledge related to our project: concurrent program, distributed environment and web application system. In the second part of this chapter, we depict the architecture of a web application and the protocols we are using for web service. At the end of this chapter, we introduced the MVC framework which is an application framework that implements the model-view-controller (MVC) pattern.

2.1 Concurrent Programme

A concurrent program is a set of processes which are executed in parallel [9]. Traditionally, the word parallel is used for systems in which the executions of several programs overlap in time by running them on separate processors. The word concurrent is reserved for potential parallelism, in which the executions may, but need not, overlap; instead, the parallelism may only be apparent since it may be implemented by sharing the resources of small number

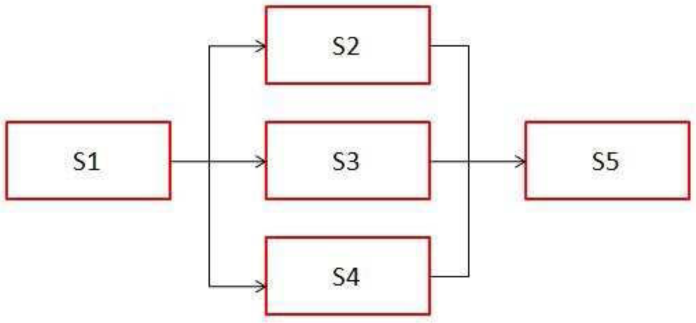

Figure 1: Concurrent Processes Structure.

of processors.

Concurrency is the execution of two or more independent, interacting processes in the same time. Dijkstra first proposed a notation for parallel processes [13].

begin S1; parbegin S2; S3; S4; parend; S5 end.

In this notation, the statement S1 is executed first, then the parallel processes S2, S3 and S4 are executed concurrently. After S2, S3 and S4 are all finished, S5 then is executed. From this notation, we can have the diagram for the concurrent processes structure (Figure 1).

As a example, we define:

S1: n =0;

S2: n=n+1;

S3: n=1;

S4: n=n+3;

S5: print s;

In sequential programs, running a program with same input will always give same result, so, it makes sense to debug a program. But for a concurrent program, it is nonsense to debug the program because there are many possibilities:

$S1 \rightarrow S2 \rightarrow S3 \rightarrow S4 \rightarrow S5 => n = 4;$

$S1 \rightarrow S3 \rightarrow S2 \rightarrow S4 \rightarrow S5 => n = 5;$

$S1 \rightarrow S2 \rightarrow S4 \rightarrow S3 \rightarrow S5 => n = 1;$

$S1 \rightarrow S3 \rightarrow S4 \rightarrow S2 \rightarrow S5 => n = 5;$

$S1 \rightarrow S4 \rightarrow S2 \rightarrow S3 \rightarrow S5 => n = 1;$

$S1 \rightarrow S4 \rightarrow S3 \rightarrow S2 \rightarrow S5 => n = 2;$

In real life, there are various possible computer architectures which are concurrent systems [10].

Multitasking systems: Concurrent program is being executed by multitasking by sharing the resources of one computer.

Multiprocessor computers: a multiprocessor computer with more than one CPU. The memory is physically divided into banks of local memory and global memory.

The communication among processes is called Inter-Process Communication(IPC). The most common method of IPC is shared memory. Multiple processes exchange data by reading and writing a shared memory. Shared memory is very efficient, but in a condition that all the processes can access the same memory which means that normally all the processes should be on the same machine. Another method of IPC is Socket which connects two processes directly. The processes that communicate via socket can be on one machine or on different machines. Figure 2 shows three processes communicate through sockets and shared memory.

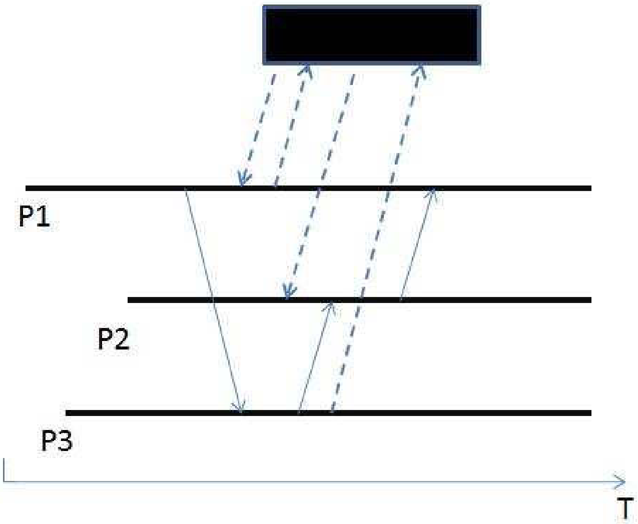

Figure 2: Communication among processes

Since the booming of Internet, concurrent programming becomes more and more important because of many reasons:

- Concurrent programming facilities are notationally convenient and conceptually elegant for writing system in which many events occur concurrently, for example, operating systems, real-time systems and database systems.

- Inherently concurrent algorithms are best expressed when concurrency is stated explicitly; otherwise, the structure of algorithm maybe lost.

- Efficient utilization of multiprocessor architectures requires concurrent programming.

- Concurrent programming can reduce program execution time even on uniprocessors, by allowing input/output operations to run in parallel with computation [24].

Erasmus is a language specially designed for concurrent systems. Concurrency in Erasmus is based on communicating processes [20]. Except the communication among processes, the processes in Erasmus are independent. The goal of the Erasmus project is to make it easier to develop scalable concurrent systems.

2.2 Distributed System

A distributed system is a collection of autonomous computers linked by a computer network that appear to the users of the system as a single computer. From the definition of distributed system, we know that:

1. All the computers are autonomous. This means that the computers, in principle, can work independently

2. The distributed system is working as a single system to solve a certain problem and the computers are linked by certain kind of network [18].

From the definition of distributed system, we also can identify three primary characteristics of distributed systems [32]:

- Multiple Computers: A distributed system contains more than one physical computer.

- Interconnections: Some of the I/O paths will interconnect the computers.

- Shared State: The computers cooperate to maintain some shared state. Put another way, if correct operation of the system is described in terms of some global invariants, then maintaining those invariable requires the correct and coordinated operation of multiple computers.

The three primary characteristics tell us that distributed systems should be considered to be distinct from concurrent systems. Because in a concurrent system implemented by

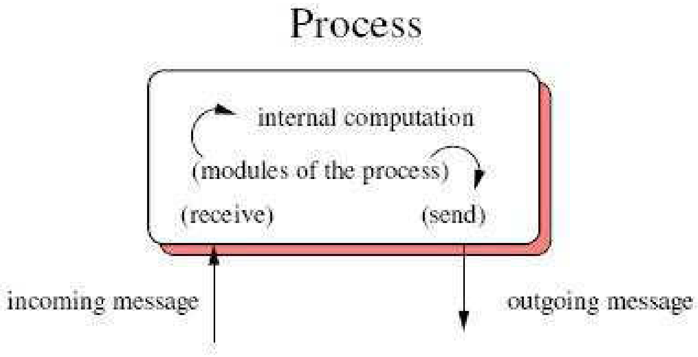

Figure 3: Step of a process

multitasking or multiprocessing, the global memory is accessible to all processors and each one can access the memory efficiently. However, in a distributed system, the nodes may be geographically distant from each other [10].

To reason about distributed systems, we represent the underlying physical system by two abstractions: processes and links [22].

The processes of a distributed program abstract the active entities which perform computations. A process represents a computer, a processor with a computer, or simply a specific thread execution within a processor. The processes exchange messages using some communication network. A distributed algorithm is viewed as a collection of distributed automata, one per process. The automation at a process regulates the way the process executes its computation steps. A process step consists in receiving a message from another process, executing a local computation and sending a message to some process (Figure 3) [10] .

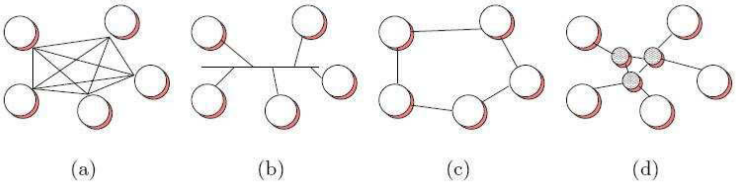

Figure 4: The link abstraction

Links abstract the physical and logical network that supports communication among processes. The link abstraction is used to represent the network component of the distributed system. Every pair of processes is connected by a bidirectional link, a topology that provides full connectivity among the processes (Figure 4) [10].

There are a lot of advantages of distributed system, for example, the distributed system can provide higher performance than centralized computer; the distributed system can involve many applications; the distributed system provides more reliability because even if some machines crash, others survive; the distributed system provides better scalability because it is easy to add new computers into the system. But the disadvantage of distributed system is also obvious. The distributed systems will have an inherent security issue and if the network gets saturated then problems with transmission will affect the performance of the system.

A web application is an example of distributed system. Through all kinds of protocol such as SOAP, REST, the web application can run on thousands of computers anywhere in the world. Normally, a web application involves more than one database server, many application servers to provide better performance. It is difficult to imagine that when thousands of users attack the web application at the same time, only one machine is dealing with all the requests. So, if we want Erasmus as a web application language, Erasmus should first to be a distributed system language.

2.3 Web Application

Internet began with point-to-point communication between mainframe computers and terminals in 1950s, then it expanded to point-to-point connections between computers and then early research into packet switching. In 1982 the protocol TCP/IP was standardized and the concept of a world-wide network of fully interconnected TCP/IP networks called the Internet was introduced. Since the mid-1990s the Internet has had a drastic impact on culture and commerce, including Email, instant messaging, VoIP, two-way interactive video calls, and the World Wide Web with its discussion forums, blogs, social networking, and online shopping sites. The Internet continues to grow and goes into every family, driven by ever greater amounts of online information and knowledge, commerce, entertainment and social networking.

Web applications include two components: web servers and web clients.

Web server refers to the application which is running on the server side and delivers web pages or web services through the Internet to the clients all around the world. Via web pages, the web server can deliver HTML documents and any additional content that may be included by a document, such as images, style sheets and JavaScripts. Via web service, the server can provide all kinds of services, for example, weather, stock price, audio dictation etc. Every web server has an IP address and a domain name. Through the IP address and domain name, the client sends a request to the server, and the server then finds the page or calculates the result and sends it to the clients. Users typically visit web sites by clicking on a hyperlink that will bring them directly to a sites URL in the address bar of a browser or get web service through SOAP protocol. Any computer can be turned into a web server by installing server software, and connecting your computer to the Internet.

There are many different web server applications available for use on your computer:

- Apache HTTP Server

  It is the most popular web server in the world developed by the Apache Software Foundation. Apache web server is an open source software and can be installed on almost all operating system including Unix, Linux, Windows, FreeBSD, Mac OS X. About 60% of the web server machines run the Apache Web Server. With tomcat module, Apache Web server can support JSP and J2EE components [2].

- Internet Information Services (IIS)

  The Internet Information Server is a high performance web server developed by Microsoft. This web server runs only on Windows NT/2000 and 2003 platforms. IIS comes bundled with Windows NT/2000, 2003 and later; Because IIS is tightly integrated with the operating system so it is relatively easy to administer it. IIS support .net framework [2].

- lighttpd

  The lighttpd, pronounced lighty is a free web server that is distributed with the FreeBSD operating system. This open source web server is fast, secure and consumes much less CPU power. Lighttpd can also run on Windows, Mac OS X, Linux and Solaris operating systems [2].

- Sun Java System Web Server

This web server from Sun Microsystems is suited for medium and large web sites which runs on Windows, Linux and Unix platforms. Though the server is free it is not open source. The Sun Java System web server can support various languages, scripts such as JSP, Java Servlets, PHP, Perl, Python, Ruby on Rails, ASP and Coldfusion etc [2].

- Jigsaw Server

  Jigsaw is developed by the World Wide Web Consortium. It is open source and free and can run on various platforms like Linux, Unix, Windows, Mac OS X Free BSD etc. Jigsaw is written in Java and can run CGI scripts and PHP programs [2].

- JBoss Application Server

  JBoss Application Server (JBoss AS) is a free software/open-source Java EE-based application server. An important distinction for this class of software is that it not only implements a server that runs on Java, but it actually implements the Java EE part of Java. Because it is Java-based, the JBoss application server operates cross-platform: usable on any operating system that supports Java. JBoss AS was developed by JBoss, now a division of Red Hat [2].

- IBM Webphere Application Server

  IBM WebSphere Application Server (WAS) is built using open standards such as Java EE, XML, and Web Services. It is supported on the following platforms: Windows, AIX, Linux, Solaris, i/OS and z/OS. Beginning with Version 6.1 and now into Version 8, the open standard specifications are aligned and common across all the platforms [2].

The web browser, the client side of web applications, is a software application for retrieving, presenting, and traversing information resources on the World Wide Web. The information resource is identified by a Uniform Resource Identifier (URI) and may be a web page, image, video, or other piece of content. Hyperlinks present in resources enable users to navigate their browsers to resources which they are interested in. A web browser can also be defined as an application software to enable users to access, retrieve and view documents and other resources on the Internet. Of course, the browser can also be used to access information provided by web servers in private networks or files in file systems.

There are different web browsers that are available and in use today. Some of the available web browsers include Internet Explore (IE), Navigator, FireFox, Safari and Opera. They are all free.

Web browsers come with some features. Some common features of web browsers are spell checkers, search engine toolbars, download managing, password managing, bookmark managing, as well as form managing. Accessibility features that may be included with many web browsers include pop-up blocking, page zooming, tabbed browsing, ad filtering, incremental finding, HTML access keys, voice controls, mouse gestures, spatial navigation, text to speech, and caret navigation.

Web browsers support frames, Java, XSLT, XForms, RSS, Atom, SVG, WML, VoiceXML, MathML, and XHTML. Also, with many web browsers, support for different languages include English, Slovak, Arabic, German, Dutch, Turkish, Swedish, Chinese, French, Spanish, Thai, Hebrew, Italian, Greek, Russian, Polish, Welsh, as well as hundreds more.

A web session is a sequence of web transactions of web requests and web responses. First, The browser extracts the name of the server and sends his local DNS to resolve the

IP address of the server. The local DNS obtains the IP address by eventually contacting some intermediate name servers or even a root of the domain name server. Then, With the IP address, the browser establishes the TCP/IP conection to the web server. After the completion of the TCP connection, the browser sends a HTTP request to the server to fetch a web page. The server gets the content of the web page and send back to the client. Once the browser gets the object, it parses it and presents it. If the browser finds that there are embedded objects, it sends separate requests for each object. At last, after all the objects and embedded objects are received, the whole session is finished.

One of the major advantages of the web application is that the user doesn't need to install any programme when he needs to use some web services except the browser.

Furthermore web application is easy to deploy compared with conventional client-server applications. The web application just needs to be installed and updated on the web side. Another advantage of the web application is that because all the client sides are running on browsers, with the same interfaces, the user doesn't need to spend a lot of time to learn how to use the application.

A major difficulty of web applications is that they are based on the asymmetric design of the HTTP Protocol and the fact that it is stateless [36]. The server is unable to send updates to the client and has to wait for incoming requests. Moreover, the web server is unable to control the navigation facilities in web browsers, like the back- and forward-buttons or the capability to open new windows of the same page. These navigation facilities lead to synchronization problems with the state of the server and its clients. This means that the server has to deal with the fact that for one question asked (e.g., filling in an order form) there may be more than one answer. This happens if the user uses the back button

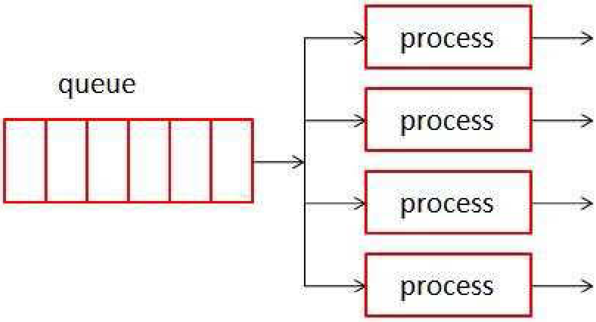

Figure 5: Process-based architecture

or clones a window and submits the form a second time. Thus a user can follow several paths of interaction in a session at the same time [40].

Another major difficulty is that the web application should have capability to deal with thousands of request in busy time. It means that thousands of processes are running on the server side concurrently. The processes share resources including hardware devices and data. The web application server has to give clients a correct response in a reasonable time. For example, online game is a kind of web application. From the online records, we can figure out how many processes are running on the server side at the same time. EVE Online, the world's largest game universe, records 45,000 simultaneous players in 2004 [3]. Furthermore, Microsoft is trying to develop 300,000 player online games.

2.3.1 Architecture of Web Application

Web application is typical a concurrent, rich client software. The basic architecture of web application is consisting of multiple single-threaded processes, each of which handles one request at a time. It is called process-based architecture (Figure 5). The Apache 1.3 is using such kind of architecture [44].

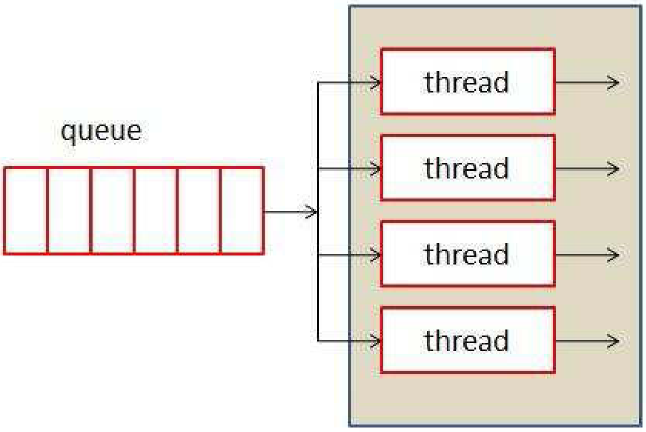

Figure 6: Thread-based architecture

Another basic architecture is thread-based architecture, the server consists of a single multithreaded server; each thread handles one request at a time. The Apache 2.0 Worker MPM implements an example of this type of approach (Figure 6).

Comparing thread architecture, the advantage of process-based architecture is the stability because any crash of one process will not influence other processes, but the drawback of this architecture is related to the performance: creating and killing processes overloads the web server. The thread-based architecture is not as stable as a process-based one but memory requirements are much smaller than for a process-based architecture.

The hybrid architecture combines the advantages of both architectures (Figure 7) and reduces their disadvatages. For example, suppose that a web server has p processes with n threads each. So, up to $p \times n$ requests can execute simultaneously. If a thread crashes, it can bring down the process in which it runs, but all other $(p-1) \times n$ processes continue to process their requests. Less memory is required in this approach to handle $p \times n$ concurrent requests than to run the same number of requests in the process-based architecture.

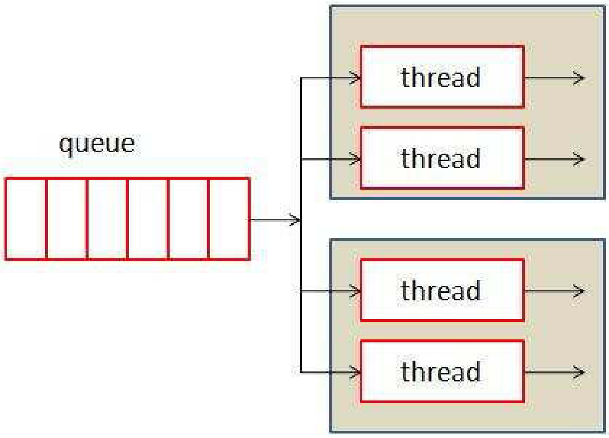

Figure 7: Hybrid architecture

In the diagram of concurrent system (Figure 1), if we set the process S5 to null, we can get exact same diagram of the process-based architecture. So, the language for concurrent process system is very suitable to develop the web server software. Normally, web application is running in distributed environment, because the web browser: IE, Firefox and Safari are at everywhere of the world but the server side normally is hold on a powerful server somewhere. Figure 8 illustrates that a web application is a concurrent system running on distributed environment.

2.3.2 HTTP Protocol

HTTP stands for short for Hyper Text Transfer Protocol, the underlying protocol used by the World Wide Web, which defines the format of the web messages and how to transmit over the internet. HTTP was designed to support HyperText Markup Language (HTML), which defines a set of special textual indicators (markups) which specify how a web page's texts and images should be displayed in the web browser [23].

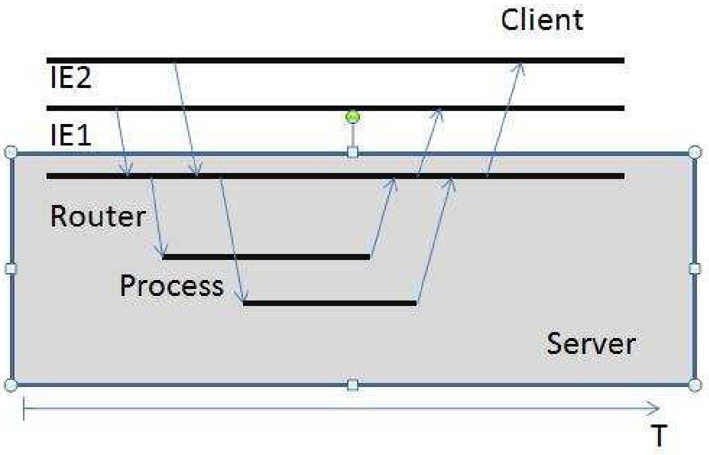

Figure 8: Web Application Working Flow

HTTP is running on top of TCP/IP protocol. But in fact, TCP/IP is not really one protocol, but a set of protocols, a protocol stack, as it is most commonly called. Its name, for example, already refers to two different protocols, TCP (Transmission Control Protocol) and IP (Internet Protocol). There are several other protocols related to TCP/IP such as FTP, HTTP, SMTP and UDP. TCP is a reliable, connection-oriented protocol that provides error checking and flow control through a virtual link that it establishes and finally terminates. So, the HTTP can guarantee the information transmitting between web servers and web clients without any loses. The primary function of HTTP protocol is to establish connection with the server and send HTML page back to the user's browser.

- HTTP relies on the URI naming mechanism and uses URIs in all its transactions to identify resources on the web [25].
- HTTP is connectionless: After the client connects to the server, it sends a request then disconnects from the server and waits for a response. When the server wants to send the response back to the client, it has to re-establish the connection.

- HTTP is media independent: HTTP can send any type of data, no matter whether it is text or an image file. How content is handled is determined by the MIME specification.
- HTTP is stateless: This is a direct result of HTTP's being connectionless. Any HTTP request is independent of the former requests. For this reason neither the client nor the browser can retain information between different requests across the web pages. The client can use cookies or parameters to tell the current state to the web server.

The format of the HTTP request and response message are similar, they both have four parts: an initial line, header lines, a blank line, an optional message body [43].

Request

$Request = Request - Line;$

$*((general - header;$

$\mid Request - header$

$\mid entity - header)\ CRLF)$

$CRLF$

$[Message - body]$

When a client wants to communicate with the server, the client sends a request to the server. A request at least has a request-line which consists of three parts. They are separated by spaces:

- An HTTP Method Name Method= "OPTIONS"

  $\mid$ *"GET"*

  $\mid$ *"HEAD"*

| "$POST$"

| "PUT"

| "$DELETE$"

| "$TRACE$"

| "$CONNECT$"

| $Extension - method$

- The local path of the requested resource.
- The version of HTTP being used: The HTTP version always takes the form "HTTP/ x.x"

GET $Get/Path/Concordia/index.htmlHTTP/1.0$

GET is the most common HTTP method. The path is the part of the URL after the host name. This path is also called the request Uniform Resource Identifier (URI). A URI is like a URL, but more general. The GET method means to retrieve whatever information from the server according to the Request-URI.

HEAD The HEAD method is identical to the GET method but the response of HEAD must not contain a message body. The HEAD request normally is used for testing hypertext links for validity, accessibility, and recent modification.

POST The POST method is to submit data to be processed to the identified resource. The data is included in the body of the request. This may result in the creation of a new resource or the updates of existing resources or both.

POST Path/Concordia/calculate.asmx HTTP/1.1

With POST method, normally, it will have a message body like this:

licenseID=string&content=string&/paramsXML=string

PUT The PUT method requests that the enclosed entity be stored under the supplied Request-URI. If the Request-URI refers to an already existing resource, the enclosed entity should be considered as a modified version of the one residing on the origin server. If the Request-URI does not point to an existing resource, and that URI is capable of being defined as a new resource by the requesting user agent, the origin server can create the resource with that URI. If a new resource is created, the origin server must inform the user agent via the 201 (Created) responses. If an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes should be sent to indicate successful completion of the request [42].

DELETE The DELETE method requests that the origin server delete the resource identified by the Request-URI.

A HTTP request also contains other important information: the IP address of you and/or your HTTP proxy; which document you requested; which version of the browser; preferred languages and cookies. HTTP is a request/response protocol, which means a request is sent to web server and then the web server sends back a response.

Response

$Response = Status - Line$

$*((general - header$

$| response - header$

$| entity - header)CRLF)$

$CRLF$

$[message - body]$

The initial response line, called the status line, also has three parts separated by spaces:

- The version of HTTP being used.
- A response status code that gives the result of the request.
- An English reason phrase describing the status code.

Here is an example of initial line for Response Message.

$HTTP/1.0$ 200 OK

$HTTP/1.0$ 404 $Not\ Found$

Header Lines

Head Lines provide information about the request or response, or about the object sent in the message body. The header lines are in the usual text header format, which is: one line per header, of the form “Header-Name: value”, ending with CRLF (Carriage Return and Line Feed). It’s the same format used for email and news postings, defined in RFC 822. Any number of spaces or tabs may be between the “:” and the value. Header lines beginning with space or tab are actually part of the previous header line, folded into multiple lines for easy reading.

Here is an example of header lines

$Host : api.opencalais.com$

$Content - Type : text/xml; charset = utf - 8$

$Content - Length : length$

$SOAPAction :$ "$http : //clearforest.com/Enlighten$"

The Message Body

An HTTP message may have a body of data sent after the header lines. In a response, this is where the requested resource is returned to the client (the most common use of the message body), or perhaps explanatory text if there's an error. In a request, this is where user-entered data or uploaded files are sent to the server.

2.3.3 Service-Oriented Architectures

SOA(Service-Oriented Architecture) is an architecture concerned with dynamic loose coupling and dynamic binding between services. Service is available at an endpoint in the network, and it receives requests and sends responses according to its specification. The service is deployed at the endpoint and has specific function whose interface are defined in a special document, for example, WSDL file [39].

The basic principle of SOA is that it needs to provide an abstract definition of the service, including the details of the service that allow the person who wants to use this service to bind to this service. Second, the service provider should publish his service information for the users to understand the service. Third, those who need the service should have a way to find what services are available and meet their requirements.

Web Service is a kind of SOA architecture over internet. W3C defined web service as follows: "A Web Service is a software system designed to support interoperable machine-to-machine iteraction over a network. It has an interface described in a machine-processable format (specifcally WSDL). Other systems interact with the web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards" [34].

To enable applications to communicate via Remote Procedure Calls (RPCs) by a simple network of standard types on XML/HTTP, SOAP was introduced in 1999. SOAP is a specification for a simple but flexible XML protocol which is focused on the common aspects of all distributed computing scenarios. it provides the following [45].

- A mechanism for defining the unit of communication.
- A processing modal.
- A mechanism for error handling.
- An extensibility modal.
- A flexible mechanism for data representation.
- A convention for representing Remote Procedure Calls (RPCs) and responses as SOAP messages.
- A protocol binding framework.

A typical SOAP message look likes this (Program 1):

WSDL is a specification that defines how to describe web services in XML grammar, WSDL describes four critical pieces of data [12]: Interface information describing all available functions. Data Type information for all requests and responses. Binding information about the transport protocol. Address information for locating the specified services (Program 2).

Program 2 GreetingService WSDl

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="’GreetingService"
targetNamespace="http://www.ecerami.com/wsdl/GreetingService.wsdl"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:tns="http://www.ecerami.com/wsdl/greetingServices.wsdl"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<message name="SayHelloRequest">
<part name="firstName" type="xsd:string" />
</message>

<message name="SayHelloResponse">
<part name="greeting" type="xsd:string" />
</message>

<portType name="Greeting_PortType">
<operation name="sayHello"
<input message="tns:SayHelloRequest" />
<output message="tns:SayHelloResponse" />
</operation>
</portType>

<binding name="Greeting_Binding" type="tns:Greeting_PortType" >
<soap:binding type="rpc"
transport="http://schemas.xmlsoap.org/soap/http" />
<operation name="sayHello">
<input>
<soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:examples:greetingservice"
use="encoded" />
</input>
<output>
<soap:body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
namespace="urn:examples:greetingservice"
use="encoded" />
</output>
</operation>
</binding>

<service name="GreetingService">
<documentation>WSDL FILE for HelloService</documentation>
<port binding="tns:Greeting_Binding" name="Greeting_Port">
<soap:address
location="http://localhost:8080/soap/servlet/rpcrouter" />
</port>
</service>
</definition>
```

Program 1 SOAP Message

```
Content-Type: application/soap+xml; charset=utf-8

<?xml version="1.0" encoding="UTF-8" ?>
<soapenv:Envelope
   xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema "
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

 <soapenv:Body>
 <doCheckResponse
 soapenv:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
 <rpc:result xmlns:rpc="http://www.w3.org/2003/05/soap-rpc">return</rpc:result>
 <return xsi:type="xsd:boolean">true</return>
   </doCheckResponse>
 </soapenv:Body>
</soapenv:Envelop>
```

2.4 Model-View-Controller(MVC) Framework

Web application, compared with traditional programs, has some special features, or we can say, problems [33]:

- User interface logic tends to change more frequently than business logic, especially in Web-based applications.
- In some cases, the application displays the same data in different ways.
- Designing visually appealing and efficient HTML pages generally requires a different skill set than does developing complex business logic. Rarely does a person have both skill sets. Therefore, it is desirable to separate the development effort of these two parts.
- User interface activity generally consists of two parts: presentation and update. The

presentation part retrieves data from a data source and formats the data for display. When the user performs an action based on the data, the update part passes control back to the business logic to update the data.

- In web applications, a single page request combines the processing of the action associated with the link that the user selected with the rendering of the target page. In many cases, the target page may not be directly related to the action.
- User interface code tends to be more device-dependent than business logic.
- Creating automated tests for user interfaces is generally more difficult and time-consuming than for business logic. Therefore, reducing the amount of code that is directly tied to the user interface enhances the testability of the application.

To solve these problems, MVC, as an architectural pattern, was introduced into web application program. MVC maps the traditional architectural pattern: input–processing–output into the GUI realm as an architectural pattern: Model–View–Controller.

The pattern isolates "domain logic" (the application logic for the user) from the user interface (input and presentation), permitting independent development, testing and maintenance of each (separation of concerns). Following this pattern, the MVC framework separates the modeling of the domain, the presentation, and the actions based on user input into three separate classes [16].

- *Model*. The model manages the behaviour and data of the application domain, responds to requests for information about its state and responds to instructions to change state.
- *View*. The view manages the display of information.

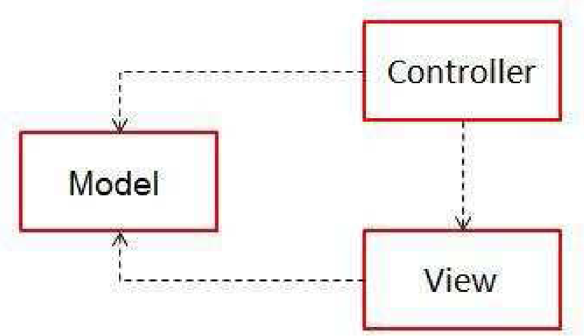

Figure 9: MVC pattern

- *Controller*. The controller interprets the action inputs from the user, for example, mouse and keyboard inputs and then informs the model and/or the view of these actions.

2.4.1 Passive Model and Active Model

Steve Burbeck described two variations of MVC in his article: a passive model and an active model [11].

The passive model is employed when one controller manipulates the model exclusively. The controller modifies the model and then informs the view that the model has changed and the view should be refreshed. The model in this scenario is completely independent of the view and the controller, which means that there is no means for the model to report changes in its state. The HTTP protocol is an example of this. There is no simple way in the browser to get asynchronous updates from the server. The browser displays the view and responds to user input, but it does not detect changes in the data on the server. Only when the user explicitly requests a refresh is the server interrogated for changes.

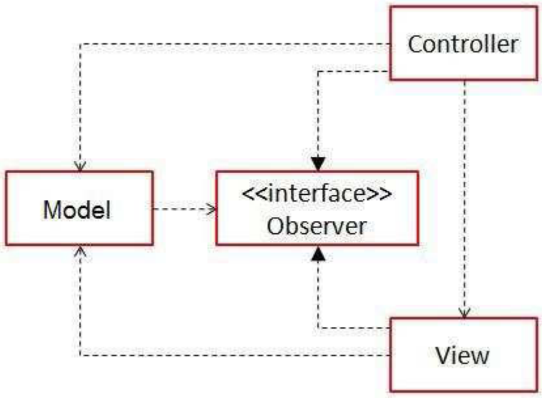

Figure 10: Active model

The active model is used when the model changes state without the controller's involvement. This can happen when other sources are changing the data and the changes must be reflected in the views. Consider a stock-ticker display. We receive stock data from an external source and want to update the views (for example, a ticker band and an alert window) when the stock data changes. Because only the model detects changes to its internal state when they occur, the model must notify the views to refresh the display (Figure 10).

Here we will show a calculator as an example of the MVC architecture which is written in C# . The form will house the view and events will be passed to the controller which will then call methods on our model such as Add/Subtract/NumberPress. The model takes care of all the work and it holds the current state of the calculator. The controller takes an interface to the view and model. It is important to know that the view will typically interact with the controller if it needs notification of events which are fired via the view (such as a button click). In this program, we have the controllers constructor pass a reference to itself to the view class (Program 3) [38].

Program 3 Controller

```
static class Program
{
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
// Note: The view should not send to the model but it is often useful
// for the view to receive update event information from the model.
// However you should not update the model from the view.
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        frmCalcView view = new frmCalcView();
        CalculatorModel model = new CalculatorModel();
        CalcController controller = new CalcController(model, view);
        Application.Run(view);
    }
}

/// <summary>
/// The controller process the user requests.
/// Based on the user request, the Controller calls methods in the View and
/// Model to accomplish the requested action.
/// </summary>
class CalcController : IController
{
    ICalcModel model;
    ICalcView view;

    public CalcController( ICalcModel model, ICalcView view)
    {
        this.model = model;
        this.view = view;
        this.view.AddListener(this); // Pass controller to view here.
    }

    public void OnClick( int number )
    {
        view.Total = model.SetInput(number).ToString();
    }

    public void OnAdd()
    {
        model.ChangeToAddState();
    }
}
```

The view does not interact with the model, it receives update requests from the controller and the view also passes click events on to the controller (Program 4).

The Model does all the computation of the calculator: Add and Subtract and it handles the state (Program 5).

The MVC framework brings us benefits. The most important benefit is that it supports multiple views. Because the view is separated from the model and there is no direct dependency from the model to the view, the user interface can display multiple views of the same data at the same time. Another benefit is that it accommodates change. User interface requirements tend to change more rapidly than business rules. Users may prefer different colors, fonts, screen layouts, and levels of support for new devices such as intelligent cell phones or IPADs. Because the model does not depend on the views, adding new types of views to the system generally does not affect the model. As a result, the scope of change is confined to the view.

However, MVC also brings a liability: Complexity. The MVC framework introduces new levels of indirection and therefore increases the complexity of the solution slightly. It also increases the event-driven nature of the user-interface code, which can become more difficult to debug.

Compared with the liabilities, we get more benefits from the MVC framework. Therefore, MVC becomes more and more popular with evolvement of the web applications. Almost every web development language provides its own MVC framework. When we design the web application framework in Erasmus, we also provide our Erasmus MVC framework to ease the development of the web application.

Program 4 View

```
/// <summary>
/// Windows Form is our view class.
///
/// </summary>
public partial class frmCalcView : Form, ICalcView
{
    IController controller;
    public frmCalcView( )
    {
        InitializeComponent();
    }
    /// <summary>
    /// The view needs to interact with the controller to pass the click events
    /// This could be done with delegates instead.
    /// </summary>
    /// <param name="controller"></param>
    public void AddListener( IController controller )
    {
        this.controller = controller;
    }
    private void lbl_Click(object sender, EventArgs e)
    {
        // Get the text out of the label to determine the letter and pass the
        // click info to the controller to distribute.
        controller.OnClick((Int32.Parse(((Label)sender).Text)));
    }
    private void lblPlus_Click(object sender, EventArgs e)
    {
        controller.OnAdd();
    }

#region ICalcView Members
    public string Total
     {
        get
        {
            return textBox1.Text;
        }
        set
        {
            textBox1.Text = value;
        }
      }
#endregion
}
```

Program 5 Model

```
/// <summary>
    /// Calculator model, The model is independent of the user interface.
    /// It doesn’t know if it’s being used from a text-based, graphical, or web inter
    /// This particular model holds the state of the application and the current valu
    /// The current value is updated by SetInput
    /// </summary>
    class CalculatorModel : ICalcModel
    {
        public enum States { NoOperation, Add, Subtract };
        States state;
        int currentValue;
        public States State
        {
            set { state = value; }
        }
        public int SetInput ( int number )
        {
            if (state == States.NoOperation)
            {
                currentValue = number;
            }
             else if (state == States.Add)
            {
                currentValue = Add(currentValue , number );
            }
            return currentValue;
        }
        public void ChangeToAddState()
        {
            this.state = States.Add;
        }
        public int Add( int value1, int value2 )
        {
            return value1 + value2;
        }
        public int Subtract(int value1, int value2)
        {
            throw new System.ApplicationException(" Not implemented yet");
        }
}
```

Chapter 3

Related work

In this Chapter, we study three popular programming languages (Perl, Ruby, and J2EE) especially their framework for web applications. In fact, some original concepts and ideas in our project are gotten from these languages because their popularity proves that their frameworks are suitable for web environment. Comparing Perl, Ruby and J2EE, Erasmus is a new language and a process oriented language. But as a program language, it is also necessary to learn from other languages to improve Erasmus.

3.1 Perl

Perl is an interpreted language optimized for scanning arbitrary text files, extracting information from text files, and printing reports based on that information. It combines the features of C language, shell scripting, sed and awk. Perl is easy to use, flexible and efficient. Perl was originally developed by Larry Wall in 1987 and then became more and more popular. It provides powerful text processing facilities for easy manipulation of text files. It is also used for graphics programming, system administration, network programming, applications that require database access and CGI programming on the Web.

The overall structure of Perl derives broadly from C. Perl is procedural in nature, with variables, expressions, assignment statements, brace-delimited blocks, control structures, and subroutines.

Perl also takes features from shell programming. All variables are marked with leading sigils, which unambiguously identify the data type (for example, scalar, array, hash) of the variable in context. Importantly, sigils allow variables to be interpolated directly into strings. Perl has many built-in functions that provide tools often used in shell programming (although many of these tools are implemented by programs external to the shell) such as sorting, and calling on system facilities.

Perl takes lists from Lisp, associative arrays (hashes) from AWK, and regular expressions from Sed. These simplify and facilitate many parsing, text-handling, and data-management tasks.

Perl 5 added features that support complex data structures, first-class functions (that is, closures as values), and an object-oriented programming model. These include references, packages, class-based method dispatch, and lexically scoped variables, along with compiler directives (for example, the strict pragma). A major additional feature introduced with Perl 5 was the ability to package code as reusable modules. Larry Wall later stated that "The whole intent of Perl 5's module system was to encourage the growth of Perl culture rather than the Perl core" [37].

All versions of Perl do automatic data-typing and automatic memory-management. The interpreter knows the type and storage requirements of every data object in the program; it allocates and frees storage for them as necessary using reference counting (so it cannot deallocate circular data structures without manual intervention). Legal type-conversions, for example, conversions from number to string, are done automatically at run time and illegal type conversions are fatal errors.

Here is a Perl program which gets a web page from a URL and does a very simple analysis on the content of the page (Program 6).

Program 6 Get a web page in Perl

```
my $url="http://www.silanis.com/index.html";
use LWP::simple;
my $content = get $url;
die "Couldn’t get $url" unless defined $content;
#  analyze the page
If($content =~ m/eSignLive/i)
{
print "There is a tool called eSignLive"
}else{
Print "There is nothing about eSignLive"
}
```

There are three concepts in Perl provide a valuable reference for Erasmus: Packages, Modules and Objects to organize the programme.

In Perl, a package in fact is a namespace. Packages provide the fundamental building block on which the higher-level concepts of modules and classes are constructed. Packages are independent of files. We can have many packages in a single file, or a single package that spans several files. Code is always compiled in the current package which determines which symbol table is used for name lookups, so it protects different sections of code from inadvertently tampering with each other’s variables. At the beginning, the package main is the initial current package, but you can switch the current package to another one using the package declaration.

A module is a reusable package which is defined in a library file whose name is the same as the name of the package. A module may export some of its symbols into the symbol table of any other package using it. Or it may function as a class definition and make its operations available implicitly.

An object is a data structure with a collection of behaviors that the data structure is capable of. An object gets its behaviors by being an instance of a class. The class will also likely include class methods which are constructor methods or operation methods. Through classes, Perl provides inheritance for reusability. A class may be defined to inherit both class and instance methods from parent classes which is called base classes. This allows a new class to be created that is similar to an existing class, but with added behaviors [28].

3.1.1 Catalyst Framework (MVC Framework)

Catalyst is a framework for building web applications, based around the MVC design pattern, it provides us with lots of infrastructure, and a suggestion on how to lay out the codes, which is helpful to rapidly build new applications and reuse other people's code.

The Catalyst framework is a flexible and comprehensive environment for quickly building high-functionality web applications in Perl. Catalyst is an open source Perl-based Model-View-Controller framework that aims to ease the diffculties in web application developement. It reorganizes the web application and implements it in a natural, maintainable, and testable manner, which makes web development fun, fast, and rewarding.

For Catalyst web applications, the MVC implementation choices are most commonly:

Model: a database, using any of the drivers supported by DBI. A popular approach for object-oriented programming is to use one of the object-relational mapping modules like Class::DBI or DBIx::Class, which let you use databases without writing much (or in some cases, any!) SQL code. View: a module which formats data in HTML format, for example (Template::Toolkit, HTML::Template, HTML::Mason, ..). There are actually two parts to the view: a Perl "View" module which handles any manipulation of data, and a simple "template" which takes care of presenting the end result. Controller: the mastermind of the application, which maps request URIs to functions using the catalyst Dispatcher. Controller modules do the busy work of asking a model for data for a request, and then passing it to a view for presentation. With the variety of application requirements, there aren't many generic examples of controllers. To get a new Catalyst developer started, there are some useful "helper" controllers on CPAN, including Handel (a commerce framework) and Catalyst::Enzyme (for database create/update/... interaction) [14].

There are some features of Catalyst:

- Speed

  Catalyst is an enterprise level framework and is able to handle a significant load. In complile time, Catalyst applications has registered their actions in the dispatcher which makes the system can process runtime requests quickly, without needing elaborate checks. Furthermore, Regex dispatches are all precompiled to improve the performance of the system. At last, Catalyst builds only the structures it needs, so there are no delays to generate unused database relations.

- Simplicity

  Catalyst has a lot of prebuilt components for the developer. For any kind of business, we can find components to simplify our developement. For example, there are View classes available for Template Toolkit, HTML::Template, Mason, Petal, and PSP. Plugins are available for dozens of applications and functions, including

Data::FormValidator, authentication based on LDAP or Class::DBI, several caching modules, HTML::FillInForm, and XML-RPC.Catalyst supports component auto-discovery: If we put the components at the right place, Catalyst can automatically find them and load them.

- Flexibility

  Catalyst is very flexible for developers to choose different kinds of Controllers, Models and Views. Normally, you can have the combination of MYSQL Database as Model, Template::Toolkit as View and custom Perl code as Controller. You also can use Class::DBI for your database storage and LDAP for authentication; You can choose Template Toolkit for web display and PDF::Template for print output; etc.

Based on Moose, the Perl5 object system, Catalyst is easy to extend. Catalyst is able to run on many different web servers, including Apache's mod_Perl, FastCGI and a standalone server for development, so we can deploy on or alongside our existing architecture.

3.2 Ruby

Ruby is a dynamic, open source and scripting language. It was developed by a Japanese Yukihiro Matsumoto and was influenced primarily by Perl, Smalltalk, Eiffel, and Lisp. Ruby is a simple and productive language with an elegant syntax. From the example we know that Ruby is really a natural language which is easy to read and write.

3.2.1 Sinatra

Sinatra is a free and open source web application framework written in Ruby. Sinatra was designed and developed by Blake Mizerany and is small and flexible. It does not follow the typical model-view-controller pattern that is seen in other frameworks, such as Ruby on Rails. Instead, Sinatra focuses on “quickly creating Web-applications in Ruby with minimal effort”. So, Sinatra is a DSL (domain-specific language) for quickly creating web applications in Ruby with minimal effort.

Program 7 Hello World of Sinatra

```
class MyApp <Sinatra::Base
get '/' do
   'Hello World from MyApp'
end
end
```

In the example (Program 7), we demonstrate a router, When we go to the url: $http : //localhost : 4567/$, it will show the text: “Hello World from MyApp” in the browser.

We can have a more complicate router like(Program 8):

When a request comes in, the first route which matches the request is invoked, so, the handler attached to that route gets executed.

What is Sinatra?

“Sinatra is a minimal and elegant web framework. It doesn’t include hundreds of helpers, or an ORM library (like ActiveRecord), or any complex wrappers on views. It prides itself in smallness, and this is reflected in your applications. Sinatra applications are often single file, including the views. Complete applications range in tens, sometimes hundreds of lines of code, including the views” [7].

So what can we really use Sinatra for?

In fact, Sinatra doesn’t do anything that Rails or other frameworks can’t, however sometimes, Rails is just too big, we don’t need a big framwrok. A small framework is a

Program 8 Complicated router examples

```
get ’/’ do
’Get from MyApp’
end

post ’/’ do
’Post from MyApp’
end

put ’/’ do
’Put from MyApp’
end

delete ’/’ do
’Delete from MyApp’
end

get ’named\_params/:argument’ do
’argument->#{params[:argument]}’
end

get ’/star_dot_star/*.*’ do
   ’filename -> #{params[:splat][0]} <br>
   extension -> #{params[:splat][1]}’
  end

get %r{/regular_expression/([\w]+)} do
’Regular_expression -> #{params[:captures].first}’
end
```

good fit for such a small application. Another great use is services. web services should provide simple, clearly-defined roles. This makes Rails overkill for services and also makes Sinatra justs right for services.

3.2.2 Ruby on Rails

Rails is a very popular web framework for Ruby. Rails is modeled by MVC pattern, so it consists of Model, View and Controller components. For Model, we can very easily build a database application with ActiveRecord; For View, the idea is that a HTML files server as templates, and dynamic content is inserted at render time; Controllers provide the "glue" between models and views. In Rails, controllers are responsible for processing the incoming requests from the web browser, interrogating the models for data, and passing that data on to the views for presentation.

3.3 J2EE

J2EE introduced a lot of new concepts into web developments. To compare with our Erasmus, We describe only one feature of J2EE: EJB.

An Enterprise JavaBeans(EJB) component is a body of code having fields and methods to implement modules of business logic. An enterprise bean is a server-side component that encapsulates the business logic of an application.

There are 3 kinds of enterprise beans defined in J2EE.

- Session Bean : Performs a task for a client; implements a web service.
- Entity Bean: Represents a business entity object that exists in persistent storage.

- Message-Driven Bean: Acts as a listener for the Java Message Service API, processing messages asynchronously.

A session bean is a client inside the Application Server. If the client of the web application want to access an application which is deployed on the server, the client invokes the session beans methods.

A session bean is an interactive session and it is not shared; it has only one client. As an interactive session, a session bean is not persistent. (That is, its data is not saved to a database.) When the action terminates, the session bean is no longer associated with the client. It means the session bean does not belong to any client. When an action is finished, the session will be released and other client can reuse it.

There are two types of session beans: stateless and stateful. A stateless session bean does not maintain a conversational state for the client. The entire variable in the session can only be kept for the duration of the invocation. When the method is finished, the state is no longer retained. Except during method invocation, all instances of a stateless bean are equivalent, allowing the EJB container to assign an instance to any client. Because stateless session beans can support multiple clients, they can offer better scalability for applications that require large numbers of clients. In fact, majority of the beans are stateless beans. In a stateful session bean, it keeps the state for the duration of the client-bean session. If the client removes the bean or terminates, the session ends and the state disappears.

An entity bean represents a business object in a persistent storage mechanism, for example, database. Persistence means that the entity beans state exists beyond the lifetime of the application or the Application Server process.

A message-driven bean is an enterprise bean that allows J2EE applications to process messages asynchronously. The messages may be sent or received by all kinds of components: a application client, another enterprise bean, a web component or by a JMS application or system that does not use J2EE technology. The messages can be either JMS messages or other kinds of messages [41].

3.3.1 Resource Mangagement in Java

Memory Management Java has a built in garbage-collection system. Based on the reachability of an object (if there is reference in the program to this object) the JVM2 starts the finalization process. The two key states of objects are the reachability and the finalization state. Both of them have three possible values. Objects may be reachable, finalizer-reachable or unreachable in terms of reachability or unfinalized, finalizeable or finalized in terms of finalization [46]. The garbage collector of Parallel Garbage Collection is designed for throughput. It will take all CPU power to clear the old generation as soon as possible. The downside of this strategy, is that the bigger the heap, the longer the pause time will be. But a web application is a highly interactive application, and we need to make sure that response times are in an acceptable range. In this situation, we should choose CMS (Content Management System) collector which also known as the low-latency collector [15].

“A collection cycle for the CMS collector starts with a short pause, called the initial mark, that identifies the initial set of live objects directly reachable from the application code. Then, during the concurrent marking phase, the collector marks all live objects that are transitively reachable from this set. Because the application is running and updating reference fields while the marking phase is taking place, not all live objects are guaranteed to be marked at the end of the concurrent marking phase. To handle this, the application stops again for a second pause, called remark, which finalizes marking by revisiting any objects that were modified during the concurrent marking phase. Because the remark pause is more substantial than the initial mark, multiple threads are run in parallel to increase its efficiency" [30].

In summary, compared to the parallel collector, the CMS collector decreases pauses dramatically which is very important for web based application.

JNDI JNDI (Java Naming and Directory Interface) is an API which support to access Naming and Directory services in J2EE programming. Naming service associates names with objects to provide a way to access objects based on the names. JNDI not only can access the object but also the attributes of the object, which is called directory services. An association between a name and a object is called binding and a set of such bindings is called a context.

Retrieving an object by a JNDI name from a naming system or directory is called looking up the object. When we call lookup(), we specify the name of the child of the context we want find. *lookup*() returns a *java.lang.object* that represents the child.

Object object = InitialContext.lookup(name);

The object we retrieved from the underlying name system may or may not implement Context which is determined by the JNDI service provider. For example, if we use Sun Filesystem provider and we look up a child which is a file, it will return a *java.io.File* object which doesn't implement Context. But if you look up a directory, it will return FScontext which implements the Context [17].

JDBC JDBC includes a set of call-level API for SQL-based database access, which is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC API contains two sets of interfaces: the first is the JDBC API for application writers, and the second is the lower-level JDBC driver API for driver writers. Using pure Java JDBC technology-based drivers, Java applications and applets can access databases via the JDBC API .

As a core part of the Java 2 Platform, the JDBC API is available anywhere, which means that the applications can truly write database applications once and access data anywhere. The JDBC API is included in both the Java 2 Platform, Standard Edition (J2SE) and the Java 2 Platform, Enterprise Edition (J2EE).

The JDBC API provides metadata access which enables the development of sophisticated applications that need to understand the underlying facilities and capabilities of a specific database connection.

JDBC technology allows Internet-standard URLs to identify database connections. The object of DataSource makes code more portable and easier to maintain. Additionally, DataSource objects can provide connection pooling and distributed transactions, essential for enterprise database computing and this functionality is provided transparently to the programmer [4].

3.3.2 JMS: Java Message Service

JMS is a Java Message Oriented Middleware for sending messages between two or more clients. JMS is a part of the Java Platform, Enterprise Edition, and is defined by a specification. It allows application components based on the Java Enterprise Edition (JEE) to create, send, receive, and read messages. It also allows the communication between different components of a distributed application to be loosely coupled, reliable, and asynchronous [5].

A JMS application is composed of the following parts:[6]

- A JMS provider: A messaging system that implements the JMS specification.
- JMS clients: Java applications which can send and receive messages.
- Messages: Message objects that are sent and received between JMS clients.
- Administered objects: Pre-configured JMS objects created by an administrator for the use of JMS clients.

JMS supports two different message delivery models:

1. Point-to-Point (Queue destination): With this model, a message is delivered from a producer to one consumer. The messages are delivered to a queue, and then delivered to one of the consumers registered for the queue. While producers send messages to the queue, each message is guaranteed to be delivered, and consumed by one consumer. If there are no consumers registered to consume the messages, the queue holds them until a consumer registers to consume them.
2. Publish/Subscribe (Topic destination): With this model, a message is delivered from a producer to a number of consumers. Messages are delivered to the topic container, and then are sent to all active consumers who have subscribed to the topic. In addition, any number of the topic publisher can send messages to a topic destination, and each message can be delivered to any number of topic subscribers. comparing with Point

to Point model, if there are no consumers registered, the topic destination doesn't hold messages.

A JMS client can consume messages either synchronously or asynchronously.

- Synchronous: In this mode, a client receives a message with $receive()$ method of the MessageConsumer object. The application thread blocks until the method returns. If a message is not available, it blocks until a message becomes available or a timeout of the $receive()$ method.

- Asynchronous: In this mode, the client registers a MessageListener object with a message consumer. The client consumes a message when the session invokes the $onMessage()$ method which is a call-back function. In other words, the application's thread doesn't block.

JMS defines two delivery modes:[1]

- Persistent messages: Guaranteed to be successfully consumed once and only once. Messages are not lost.

- Non-persistent messages: Guaranteed to be delivered at most once. Message loss is not a concern.

Choosing which delivery mode is a performance trade-offs. The more reliable the delivery of messages, the more bandwidth and overhead required to achieve that reliability. Performance can be better by producing non-persistent messages, or can maximize the reliability by producing persistent messages.

With the ever-increasing requirements imposed by e-business, never before has the implementation of a message-processing layer within a distributed system been as complex as it is today. JMS defines a common enterprise messaging API which is very suitable to develop enterprise applications that asynchronously send and receive business data and events and provides developers a solution to support enterprise messaging products.

Chapter 4

Erasmus

4.1 Concept of Erasmus

Erasmus is being developed by Peter Grogono at Concordia University, Canada and Brian Shearing at The Software Factory UK. Erasmus is a new language designed especially for concurrent system because it is a kind of Process-Oriented language. Erasmus has a compiler which is responsible for compiling Erasmus scripts into C++ scripts. C++ is a OS independent language, so, Erasmus is also an OS independent language.

1. Scale-free programming

   Erasmus takes the view that software construction should be fractal : the same notation should be employed at all levels of scale. Erasmus facilitates scale-free software development by allowing cells and processes to be recursively nested. This is in contrast to languages with a hierarchy of abstractions such as method-class-package, each with slightly different syntax.

2. Type safety

Modern languages should be type-safe. This is a necessary condition for building reliable systems. Erasmus is a strongly typed language (all type errors are detected) and type checking is static: all checking is performed at the compilation time.

3. Encapsulation

   Erasmus provides encapsulation in various ways: program code is organized into isolated processes that communicate through message passing. A process cannot modify private variables of another process. Processes are organized into independent cells that communicate only by exchanging messages. A cell can have variables that are shared by processes within the cell.

4. Capabilities

   Cells are given capabilities or entities that are needed to carry out their tasks and, nothing more.

5. Large-scale Refactoring

   Refactoring is the process of changing the internal structure of a software system without changing its external behaviour. Refactoring is often desirable but sometimes infeasible especially in commercial setting where downtime can be extremely costly. Erasmus is designed to facilitate refactoring of large programs. Software components can be moved easily from one environment to another. For example, a “fat client” can be converted a “thin client” by moving components from the client to the server.

6. High-level abstractions

   Erasmus provides high-level constructs and abstractions that simplify programming for most programmers.

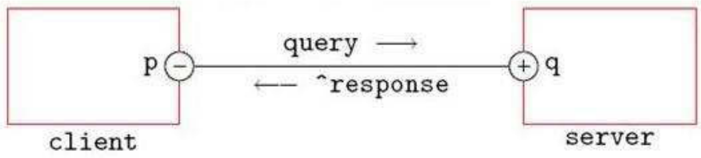

Figure 11: Basic idea of Erasmus

The main idea of Process-Oriented language is that the program consists of processes which affect each other only through communication interface. So, compared with Object-Oriented language, it is loosely coupled and only operations on private data which makes all processes are totally independent. Concurrency in Erasmus is based on communicating processes.

The simplest Erasmus project: Hello World (Program 9).

Program 9 Hello World in Erasmus

```
proc = { | sys.out := "Hello, world!" };
cell = ( proc() );
cell();
```

The program prints the string “Hello, world” to the standard output device. It has a process named *proc*, and a cell. When the cell is running, it instantiates the process *proc*, then the process starts to execute.

4.1.1 Basic Idea

The communication mode of Erasmus is based on client and server. A client sends queries to a server, and the server answers with responses (Figure 11). Note that ”‘Client”’ and ”‘Server”’ are used in Erasmus to indicate message directions. They should be confused with network clients and servers.

Communication between a client and a server is defined by a protocol. An Erasmus protocol defines the types and allowed sequences of messages. The process client uses a port with a protocol for communication. With the port and the protocol, we can send messages and receive the messages.

The client must send a query and receive a response. In Erasmus, sending and receiving are both expressed by assignment statements. A message to be sent is written on the left of ":=", and a message to be received is written on the right of ":=".

4.1.2 Protocol

A Protocol defines the format of the messages communicating between processes.

ProtocolExpression = ['ˆ'] Declaration

|[Multiplicity] ProtocolExpression

|{ ProtocolExpression };

|{ ProtocolExpression }|

|'(' ProtocolExpression ')'.

In the definition:

Multiplicity = '?' |'*' |'+'.

(no operator): the default, means exactly once

'?': means 'optional' (zero times or once)

'*': means 'many' (zero or more times)

'+': means 'at least once' (one or more times)

Not only Protocol defines the data format of communications of the processes, but also defines when the communication ends.

The following are examples of protocol expression *protocol 1= [num1: Integer; ^num2:Integer];*

A protocol transfers one integer and gets a integer response.

Protocol2=[num1: Integer; num2: integer; ^total: integer];

A protocol transfers two integers and server sends an integer back to the client.

Protocol3=[?(num1:Integer);stop; ^num2:integer];

We may transfer zero or one integer with this protocol followed with a stop sign.

Protocol4=[(num:Integer);stop; ^total:integer];*

This protocol is used in this scenario: The client sends more than one integer to server. After the client send a stop sign then the server gives the sum of the integers to client.

Protocol5=[+(num:Integer);stop; ^total:integer];

This protocol is similar to the protocol5, but in this protocol, the client has to send at least one integer to the server.

Protocol6=[num1:Integer |text1: Text; ^num2:integer | ^text2: Text];

The Client sends either a integer or a text to the server; The server responses a integer or a text;

4.1.3 Port

Ports, like variables, must be declared before they are used. A port declaration has syntax:

PortDeclaration = PortName ('+' |'-') Protocol .

The operator determines the direction of the port:

'+' : the port is a server port

'-' : the port is a client port

A port is associated with a protocol. If we have a protocol defined:

Protocol = [num1: Integer; ^num2: Integer];

Then, we can define a client port

p: -Protocol

Or a server port

q: +Protocol

Erasmus uses lvalue to send a request to the server. The communication is expressed as an assignment statement:

p.num1 := 1;

A rvalue is used to receive a reply from the server.

reply: Integer := p.num2;

Ports can also be declared within a protocol, it means that we can send a port through a protocol.

Protocol1=[*(num:Integer);stop; ^total:integer];

Protocol2=[port: Protocol1];

4.1.4 Select Statements in Erasmus

The select statement in Erasmus is a key method to manage concurrency. It is one of the most important characteristics of Erasmus. A select statement consists of several branches. Each branch has guard and a sequence of statements. The sequences contain at least one send or receive operation.

Select = (select |loopselect) Policy Guard Sequence end .

Policy = [fair |ordered |random] .

Guard = '|' [Rvalue] '|'.

The Rvalue of a guard must be a Boolean expression. The empty guard ' ' is equivalent to the guard 'true'. The sequence in a branch of a select statement cannot be empty and the first statement must be a communication statement. This is called the principal communication of the branch. A branch of a select statement is ready if its guard evaluates to true and the principal communication is Feasible. The principal communication may be a send or a receive statement. The Policy determines the order in which the ready branch of the select statement is tested.

The workflow of Select statement is [35]:

- If no branches are ready, set the program counter back to the beginning of the select statement and relinquish control.
- If one branch is ready, execute it.
- If more than one branch is ready, apply the appropriate rule from the Table below.

Table 1: Policy and Meaning

Policy	Meaning
ordered	Execute the first ready branch, using the ordering defined by the source text of the program.
fair	Choose a branch fairly and execute it. A branch b of a select statement with N branches is said to be chosen fairly if, when b becomes ready, it is executed after no more than N-1 executions of the select statement.
random	Choose a branch at random and execute it.

4.2 An Example: The Sleeping Barber Problem

To understand Erasmus deeply, we use Erasmus to solve a practical problem to illustrate how Erasmus works. The problem we will discuss first appears in an early report by Dijkstra (1965), who gives a program which is known as "The Sleeping Barber":

A barbershop has a single barber and a waiting-room with a number of chairs. If the

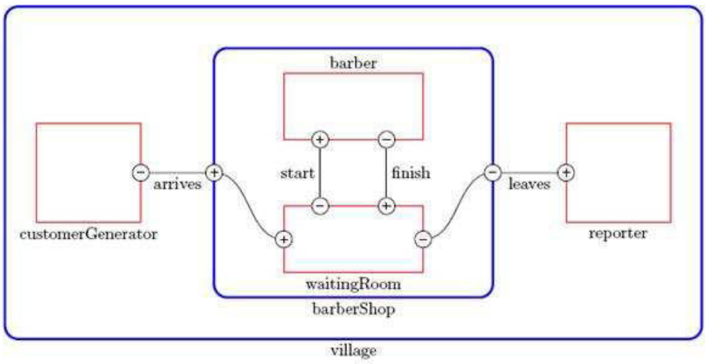

Figure 12: the sleeping barber

barber is not busy, he sleeps. If there is a customer, he wakes up and cuts the customer's hair. A new customer acts as follows:

- If the barber is asleep, the customer wakes him up and has his hair cut.
- If the barber is busy, and there are empty chairs in the waiting-room, the customer waits in a chair.
- If the barber is busy and there are no free chairs, the customer goes away.
- When the barber finishes with a customer, any waiting customer is served immediately. If there are no waiting customers, the barber goes to sleep.

Figure 12 shows the structure of the program described in this section. The waiting-room has a finite number of chairs:

NUM_CHAIRS: Integer = 3;

We represent a customer by a Text, consisting of the customer's name and an annotation describing what happens to him.

*prot = [*customer: Text]*

Process barber simulates the barber's actions. It shares, and controls the variable *sleeping*. Since sleeping is shared, it must be declared as an alias (Program 10).

Program 10 Process barber

```
barber = { start: +prot; finish: -prot; alias sleeping: Bool |
loop
sleeping := true;
customer: Text := start.customer;
sleeping := false;
finish.customer := customer + ' had hair cut'
end
}
```

The heart of the solution is the process waitingRoom, which simulates the waiting-room (Program 11). It has four ports: arrives for customers arriving; leaves for customers leaving; start for customers going to the barber; and finish for customers who have had their hair cut. It shares the variable sleeping with process barber. We use an indexed value to represent the chairs in the waiting-room. Initially, each chair is empty. The core of the process is a loop that performs an action when a customer arrives or when a customer is finished [19].

The barber's shop is represented by a cell that contains instances of barber and waitingRoom, the ports, and the shared variable (Program 12).

The process reporter issues a message as each customer leaves the shop (Program 13).

Program 13 Process reporter

```
reporter = { finish: +prot |
loop
sys.out := finish.customer + '\n'
end
}
```

Program 11 Process waitingRoom.

```
waitingRoom = { arrives: +prot; leaves: -prot;
alias sleeping: Bool
start: -prot; finish: +prot |

chairs: Integer indexes Text;
for c: Integer in 0 to NUM_CHAIRS do
chairs[c] := '';
end;

loopselect random
|sleeping|
start.customer := arrives.customer
|not sleeping|
customer: Text := arrives.customer;
any name: Text in range chairs such that name = '' do
name := customer
else
leaves.customer := customer + ' went away'
end
||
leaves.customer := finish.customer;
any name: Text in range chairs such that name <> '' do
start.customer := name;
name := ''
end
end
}
```

Program 12 Process barerShop

```
barberShop = ( arrives: +prot; leaves: -prot |
sleeping: Bool := false;
start, finish: prot;
barber(start, finish, sleeping);
waitingRoom(arrives, leaves, sleeping, start, finish) )
customerGenerator = { start: -prot |
for custnum: Integer in 0 to 20 do
start.customer := 'C' + text custnum
end
}
```

The village generates and reports on customers, and has a barber shop (Program 14).

Program 14 Cell village

```
village = (
arrives, leaves: prot;
customerGenerator(arrives); reporter(leaves);
barberShop(arrives, leaves) )
village()
```

Chapter 5

A Web Application in Erasmus

This chapter describes how to implement a web application in Erasmus. In the first section, we introduce a method of communication in distributed environment. The second section shows how to implement an HTTP listener to listen to the HTTP request and dynamically create a process to deal with the request and send back the response to clients. The third section describes how to manage database resource in Erasmus. At the end, we give an example to illustrate how to develop a web application in Erasmus.

5.1 Communication in Distributed Environment

Because Erasmus is a loosely coupled language, Erasmus should be suitable as a language for distributed environment. In distributed environment, the nodes are connected by network and communicate with a protocol which is agreed by nodes. Correspondingly, all the communication among processes in Erasmus are defined by a protocol. What we need to do is to give Erasmus ports an ability for remote communication.

If the Cell A and Cell B are not on the same physical machine, we say: Cell A and

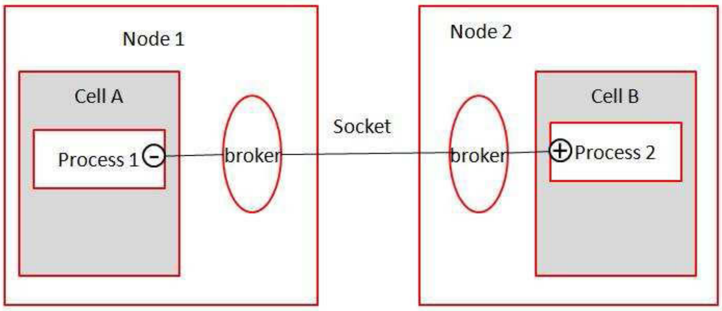

Figure 13: Communication between nodes

Cell B are on different nodes. In this kind of situation, we use two brokers to communicate between these two nodes. The broker is a proxy process created only to deal with the communication between nodes (Figure 13).

For every node which wants to communicate with a remote node, it has a config file (Table 2) which tells the broker where are the remote ports and the type of the ports (server or client). When the Erasmus application is starting, it loads the config file into memory as an array. While a process tries to send a request to another process, the system at first scans the array to determine if the port is local or remote. If the port is a remote port on another machine, the process then sends the request to the broker and waits the response from the broker. In fact, we cannot guarantee that the process on the remote machine is running or the network is working, so the process will throw a timeout exception if it cannot get response within a certain time.

In this example, when there is any communications using port p and q, the Erasmus processes send the data to the broker. Then, the brokers build a bridge between two nodes and finish the communications. Because brokers communicate with each other via socket and socket is supported by all platforms, it means the process can run on different platforms:

Table 2: Configuration of Client and Server

The config file on Node 1	The config file on Node 2
$< No > 1$	$< No > 1$
$< Port >$ 10001 $< /Port >$	$< Port >$ 10001 $< /Port >$
$< type > CLIENT < /type >$	$< type > SERVER < /type >$
$< remote >$ 10.0.2.2 $< /remote >$	$< remote >$ 10.0.2.1 $< /remote >$
$< process > process1 < /process >$	$< process > process2 < /process >$
$< portofprocess > p < /portofprocess >$	$< portofprocess > q < /portofprocess >$
$< protocol > Erasmus < /protocol >$	$< protocol > Erasmus < /protocol >$
$< /No >$	$< /No >$

UNIX, Linux, Windows and so on. Furthermore, even one of the nodes in communication is not an Erasmus node but a Java node, if it follows the Erasmus broker communication protocol, there is no problem to exchange information between them.

5.1.1 The Broker Workflow

The broker is a proxy which is a process running in the background. If one process has an Erasmus port which can communicate between different machines via a broker, in the $config$ file, we have to set a TCP/IP port for them to send and receive data between brokers, in other words, the ports show up in pairs: a server port always is along with a client port. In the example of config files, the brokers are using 10001 as the port to communicate with each other. Therefore, when broker on the client 10.0.2.2 has a request for the server 10.0.2.1, the client broker will first establish connection with server broker via port 10001, then the client broker sends the message to the server broker. The connection will be kept until the server sends a response back to the client.

Another important connection is between the broker and the Erasmus process on the same machine. In my system, the broker connects with Erasmus processes via a default port. Currently we are using 5550 as the default port to connect the broker and the process. The process sends query via this port to the broker to check if there is a request from other nodes in period. While the broker receives data from other nodes, it will forward the data to the destination process. To illustrate the workflow, I am using an example (Program 15) to explain the whole process.

Program 15 An example of communication

```
prot = [ num1: Integer;num2: Integer ];

client = { p: -prot |
sys.out:="sent number 1\n";
p.num1 := 1;
sys.out:="receive "+ text p.num2";

}

Cell = (p: prot; server(p); );
Cell();

---------------------------------------------

prot = [ num1: Integer;num2: Integer ];

server = { p: +prot |
 num1 : Integer= p.num1;
 p.num2 := num1+1;
end

Cell = (p: prot; server(p); );
Cell();
```

(client) p.num1 :=1

The process *client* whose ip address 10.0.2.1 uses LData to send an integer to the process *server* (10.0.2.2):

- The process *client* sends a string through port 5550 to broker.

  The format of the data is:

  Process name: Port name: Port type('-' is client; '+' is server): Field Name: Data.

  In this example, the string is: $client : p : - : num1 : 1$.

- The broker of process1 receives the string, and puts it into the sending buffer. Then it checks the Config Table to get the destination address, process and port. In this example: the destination is 10.0.2.2; process is process2; port is 10001. After that, the broker sets the correct destination process and port type in string. So, the string now is: $server : p : + : num1 : 1$.

- The broker of *client* tries to connect the 10.0.2.2:10001 which is the broker of *server*;

- The broker of *server* listens for this request, it receives the string from *client* and puts it into receive buffer.

(server) num : Integer= p.num1;

Process *server* uses RData to receive an integer from process *client*

- Process *server* sends a request to his broker via port 5550 to check if there is a string already in the receiving buffer.

  The checking request format is: $Processname : Portname : Porttype : FieldName$. In this example, the string is: $server : q : + : num1$.

- When the broker receives this request, it checks the receiving buffer. If there is no item to match this request, it sends back: No Data. If there is an item matching this request, it sends data back to *server* then erases the item from the receive buffer.

- When *server* gets the string, it extracts the data from the string and then converts to the correct data type (Integer, Float, String...). In this example, the variable *num* is set to 1.

(server) p.num2 := num+1;

The process *server* uses LData to send the result to *client*.

- *server* sends his broker a string: server: q: +: num2:2;
- The broker receives the data, and puts it into the sending buffer. Then, it checks the Config Table to get the destination address, process and port. Now, the string becomes: client: p:-: num2: 2;
- The broker sends result back to *client* because connection is not be terminated.
- The broker of the *client* receives the String and put it into receiving buffer, then disconnects the connection.

(client) sys.out:="reveive "+ text p.num2;

client uses RData to receive an integer from *server*

- *client* sends a request: $client : p : - : num2$ to his broker to check if there is an item already in the receiving buffer.
- When the broker receives this request, it checks the receiving buffer. If there is an item matching this request, it sends the string to *client* and then erases the item from

the receiving buffer. If there is no item matches the request, it sends `'No Data'` back.

- When the *client* gets `'No Data'`, it rests a while and sends the request to broker.
- When the *client* gets the Data, it extracts the data from the string and then converts to the correct data type (Integer, Float, String). In this example, it outputs "receive 2".

Through these 4 steps, the two nodes finish a communication. In fact, through different ports of the broker, we can build as many remote connections as possible with other nodes. In the view of an Erasmus developer, the location of the cell is transparent because the Erasmus system is responsible for all the underneath operations. No matter the cells are on the same machine, or the cells are on different machines, the design of Erasmus code is the same. A complicated distributed environment is never a burden for Erasmus developers. Another advantage of this structure is that the sending and receiving cache of the broker can synchronize the distributed cells. For example, node1 is running and send a request to node2 which is not started. The broker will wait node2 to start to finish this talk; Or node1 is much faster than node2, the broker can synchronize the speeds of two nodes by caching the requests into the receiving buffer of node2. This kind of scenario is very common in distributed environment.

- With this kind of structure, Erasmus programs are really expandable. We can easily move one process from one node to another node and then add this new node into the system. For example, with the evolution of the system, a node of the system becomes too busy to be a bottleneck. To improve the performance of the system, we can move some of the processes from this node to a new node to reduce the burden of this node.

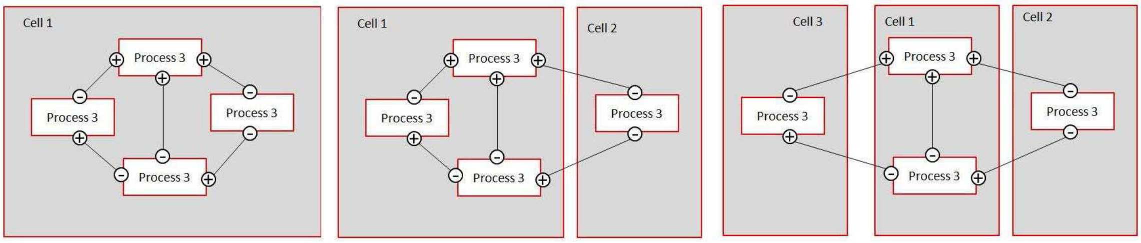

Figure 14: Transform of a Cell

This operation in Erasmus is very easy and almost at no cost (Figure 14).

From the Figure, Cell1 is divided into 3 cells: Cell1,Cell2 and Cell3. Then, we can move Cell2 and Cell3 to other nodes. So, the brokers will be responsible for the communication among the nodes.

In the current version of Erasmus, The kind of change described here require changeing a few lines of code and recompiling the project. In later version, it should be possible to perform such changes at run-time.

When we move a process from one node to another, it means that the communication between this process and other processes becomes remote communication which is far slower than local communication. So, when we transform a cell, we should choose a process or processes which don't exchange much data with other processes. According to this rule, if there are processes, they frequently communicate with each other and don't exchange much data with other processes, we should move these processes along to another node to avoid performance deterioration.

- After the introduction of broker, we can very easily fulfill Service-Oriented Architecture (SOA). A new service can be added in the system and provide service for other nodes via a service port. For example, we already have some nodes and want to add

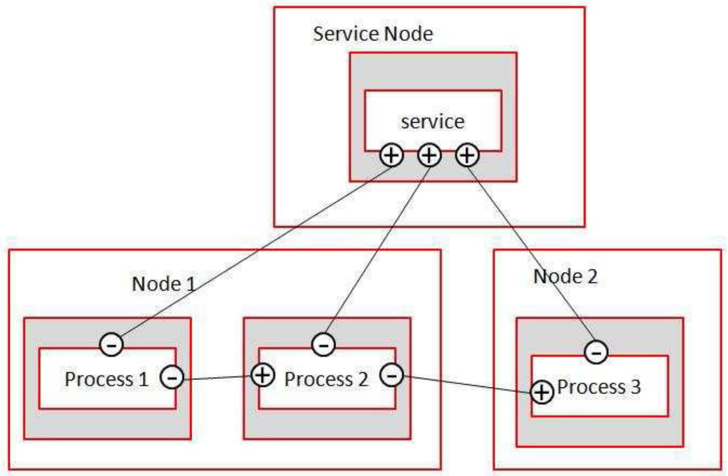

Figure 15: SOA architecture

a new node into the system which provides service for other processes. To do it, we build a new process on a new node which has some service ports. For every process, they can get service from the node by a client port which connect to the service port (Figure 15).

- Web applications always run in distributed environment. Some nodes provide database access service; some nodes provide file reading and writing service; some nodes hold the main processes of the web application; some nodes can access other resources, for example, tape, DVD or special network etc. Erasmus easily combines all nodes into a whole system in distributed environment (Figure 16).

5.2 Web Application

In Chapter 3, we already introduced the architecture of web applications. Erasmus is a process-oriented language, the basic component of Erasmus is *process*, so, we expect it to

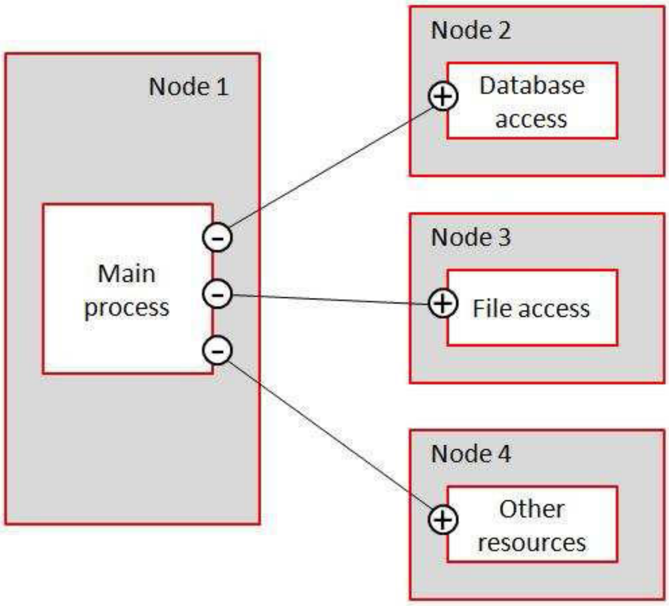

Figure 16: Web application architecture

be suitable to fulfill the process-based architecture.

In the config file of the last session, all the protocol types are defined as 'Erasmus' whose port is responsible for communication between Erasmus processes. To listen HTTP requests, the protocol can be defined as 'HTTP', which means that the broker can listen for HTTP requests and sends to the Erasmus web process to deal with the HTTP request and sends back response. In fact, not only HTTP, the protocol can also be SOAP REST or any protocol which is built on Socket.

The architecture of Erasmus web application is similar to the process-based architecture of web application. The only difference is that our Erasmus project listen to HTTP requests via a broker (Figure 17).

As a web application, it should have a HTTP listener which can listen to the requests, transfer the requests to Erasmus processes and after the transaction, it can send response

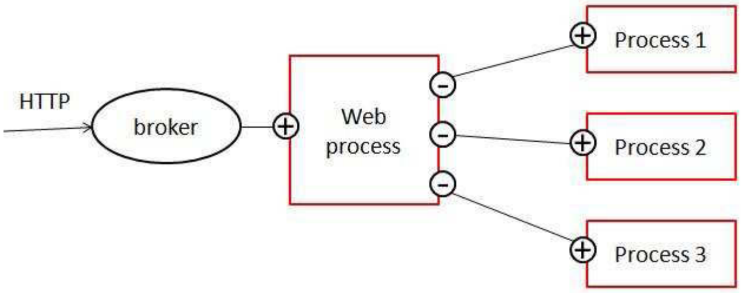

Figure 17: Architecture of Erasmus web application

to the clients. To set up a HTTP listener, we define this listener in the broker. So, in the web application framework, the broker is not only responsible for the communication in distributed environment, it is also used as a HTTP listener. Additionally, from theory, the broker can provide any service using any kind of protocol.

Furthermore, creating a new service instance is very expensive, so we always launch a certain number of service instances when the system starts. For Erasmus web framework, the Service Management process manages these instances and even can works as a load balancer. When the service instances are starting up, they report to Service Management process. The Service Management process has an array to remember the status of the service instances. When a service request is coming, the Service Management process checks the array to choose the least busy node, and then points this request to a service instance on that node, subsequently this service instance in the array is set to busy. After the request is finished, the service instance sends the result back to the client, at the same time, the Service Management process sets this service instance to free. If a service request comes and there is no service instance available at this time, the request will be pushed into a queue and will be popped while a service instance becomes free (Figure 18).

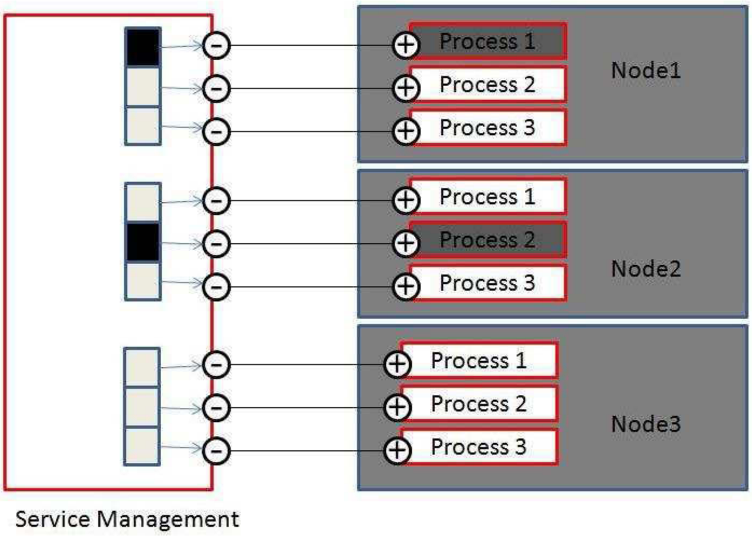

Figure 18: Architecture for static service instances

To be more complicated, for every node, we can have a Monitor process, this process reports the status of the node to "Resource Management", the information include CPU utilization percentage, memory usage, disk available, etc. When a service request is coming, the Resource Management process collect all the status information of the nodes, and choose a node to deal with the request. Then, the Resource Management process sends a command to the monitor on this node to start a new service instance. When the service is started, the Monitor points the service port to the Resource Management process. At the end, the Service Management process gets the port and establishes connection with this service instance. Even more, the Resource Management monitors the status of database node, if the CPU utilization of database reaches the threshold, the Resource Management process can pause dealing with the service requests to release the load of the database (Figure 19).

The architecture of Erasmus web framework is easily for us to be transformed for special purpose, the flexibility of the framework is inherited from Erasmus itself. The Process

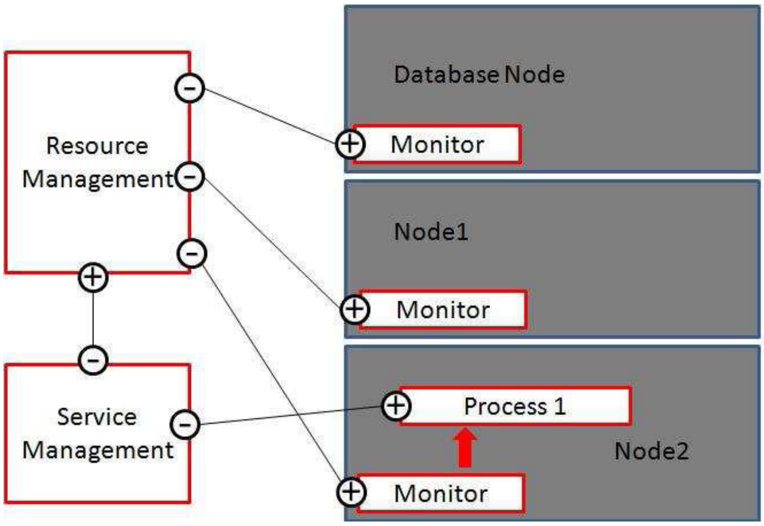

Figure 19: Web application framework with monitors

Oriented Language shows its advantages when we use it to implement service oriented applications.

5.2.1 HTTP Listener

HTTP Listener

To set a HTTP listener in the broker, we add a listener definition into the config file which defines the listening port as 80 and set the protocol type to HTTP. Comparing Erasmus protocol which is one-to-one, the HTTP protocol is many to one.

```
<No>1
        <Port>80 </Port>
        <type>Server </type>
        <remote>*</remote>
        <process>httpProcess1</process>
        <portofprocess>p</portofprocess>
```

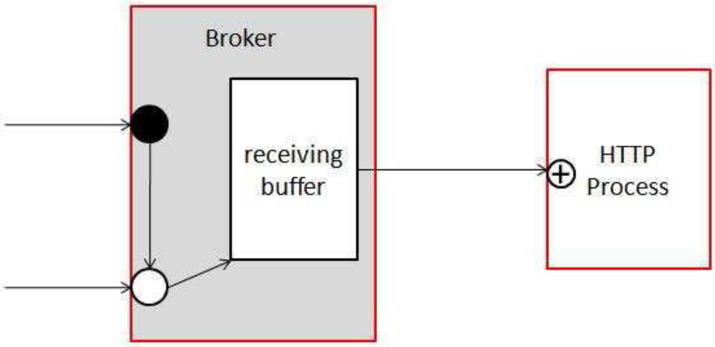

Figure 20: Listen a HTTP request

<protocol>HTTP</protocol>

</No>

The protocol definition of port p is very simple

HTTP = [*(request: Text; ˆresponse: Text)];

We can also define other kind of protocol, for example, the SOAP server, SOAP client, REST Server etc. So the Erasmus is protocol independent. When the broker is up, it will set a listener for HTTP port.

Listen a HTTP request

When the HTTP listener gets a request, it creates a new socket to communicate to the client. So the broker receives the request from the new socket and puts a string into the receiving buffer. The string includes the socket number and the receiving data. On the other hand, there is a main HTTP process which is responsible to deal with the HTTP request in our web application. The HTTP process always scans the receiving buffer by sending a query to the broker to check if there is a HTTP request. If the broker has new data received, it sends the string to the HTTP process(Figure 20).

Create a process to deal with the request

Now, the request goes into the HTTP process. The HTTP process interpret the request string and puts all the request fields into a structure: Request which includes socket number, heads, process name and parameters. After that, The HTTP process launches a new process to dispose this request (Figure 21).

The new created process has a service port whose protocol is

Request_Response = [request: Request; ^response: Response)];

For instance, there are two requests the broker received:

(get)hello?name=john

(post)withdraw (body)account=43567&amount=1000;

For the first request, the HTTP process understands that the process name is: hello and there is a parameter: name. Then the HTTP process launches a new process: 'hello' with a HTTP request structure in which there is a parameter: name.

For the second request, the HTTP process understands that the process name is: withdraw and there are two parameters: account and amount. Then the HTTP process launches a new process: withdraw with a HTTP structure in which there are two parameters: account and amount.

Send response to the client

Now, the request goes into the business process. After the calculation of the business process, the business process suicides after sending a response structure which includes the socket number, response head, and response body back to the HTTP process. Then the HTTP process translates the response structure into a string and send to the broker along

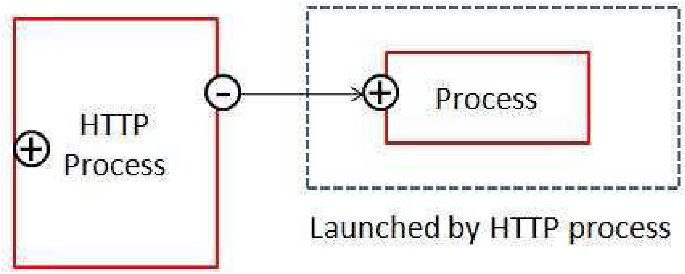

Figure 21: Create a process to deal with the request

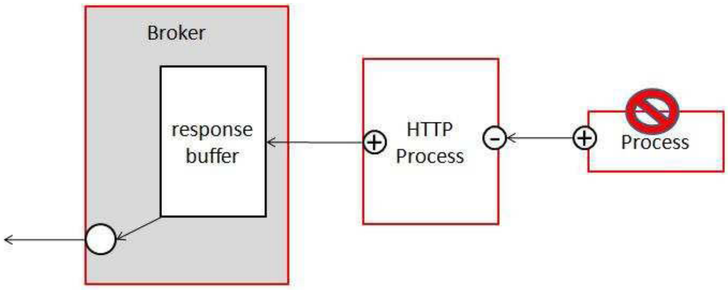

Figure 22: Send response to the client

with request socket number. At the end, the broker sends the response to the client because the broker knows the socket number. At this point, the process terminated (Figure 22).

Specially, in my project, when the response is a file, for example a jpg file, the HTTP process just sends the file name to the broker. The broker is responsible for reading the content of the file from the disk and transfers it to the client.

Strategy of concurrency

The advantage of this workflow is that the receiving buffer of the broker can cache the requests. When thousands of requests rush into the web server, all the requests will store into the receiving buffer. The HTTP process deals with the request according to its ability.

So, no matter how many requests the web server receives concurrently, the system will not be crashed. On the other hand, the broker can filter suspicious requests according to its safety strategy, and the broker can scan the receiving buffer to check timeout requests periodically, for the timeout requests, it sends back 'timeout' and then remove them from the buffer.

In my project, the HTTP process only interprets HTTP protocol. But in fact, this framework can extend to any protocols: POP3, SOAP or REST. For example, SOAP and REST are built on HTTP protocol, so they can have a pre-processor: HTTP process.

SOAP message is a kind of XML document and has its own schema, namespace and processing rules. A SOAP message implements the SOAP 1.1 XML schema, which requires that elements be fully qualified. A SOAP may have a body element, optionally a header element. SOAP doesn't limit the data type of the body, so SOAP message are extremely flexible. It can be an arbitrary XML element like a client information or an element that maps to the argument of a procedure call. The Header element can contain XML elements that describe security credential, transaction IDs, routing instructions, debugging information or any other information about the message in the body (Program 16) [31].

Program 16 A SOAP message of request

```
<?xml version="1.0" encoding="UTF-8"?>
<soap: Envelope
xmlns: soap="http://schema.xmlsoap.org/soap/envelope/"
xmlns: gs="http://www.ecerami.com/wsdl/GreetingService.wsdl" />
<soap:Body>
  <gs:SayHelloRequest>
   <firstName>Duo Peng</firstName>
  </gs:SayHelloRequest>
</soap:Body>
</soap:Envelope>
```

When the broker receives this message, it sends the SOAP message to SOAP process to open the envelope and then creates a dynamic process to deal with this “SayHelloRequest” request. Then, the dynamic process sends back a response with a string “Hi, Duo Peng”, The SOAP process envelopes the string into a SOAP message and sends back to the SOAP client (Program 17).

Program 17 A SOAP message of response

```
<?xml version="1.0" encoding="UTF-8"?>
<soap: Envelope
xmlns: soap="http://schema.xmlsoap.org/soap/envelope/"
xmlns: gs="http://www.ecerami.com/wsdl/GreetingService.wsdl" />
<soap:Body>
  <gs:SayHelloResponse>
   <greeting>Hi,Duo Peng</greeting>
  </gs:SayHelloResponse>
</soap:Body>
</soap:Envelope>
```

From the steps described, a web service is triggered and successfully completed. This Erasmus web application provide a web service called “Greeting” via SOAP. Because of the flexibility of the Erasmus, fulfilling a SOAP service is as easy as implementing a HTTP service. Not only SOAP, we can easily extend web services to other protocols: POP3, REST or other protocols.

5.3 Database Access

From the framework of our web application server, we know that every business process is stateless process which means the business process doesn’t remember any status of the last operation of the client. All the status of the client is stored in database and cookies.

Database access is a very import resource in web application. In Erasmus, we have two ways to do it. The first method is to directly access database in the business process, but there are three critical shortcomings:

- The resource of database access is not controllable.
- If a process needs to access, it has to setup connection to the database, run the SQL query and release the connection. Not only it makes the process to be complicated, but also it deteriorate the performace of the system
- When the database system changes, we have to change the code in all the business processes.

Based on these reasons, our project choose another database access method: indirect database access. All the resource of database access is controlled by a process called 'Database Access'. When a process wants to access database, it has to send a request to the process 'Database Access' to get a *port* which provide database access service.

When process A wants to access database, it firstly sends a request which contains the SQL language codes to the process 'Database Access'. The process 'Database Access' launches a process 'Database process' which has a service port. The process 'Database process' executes the SQL codes on database, get the result set from the database and wait for process A to retrieve the dataset(Figure 23).

When process A get the *port* from the 'Database Access', it knows that there is a process is waiting for it to retrieve the dataset. So, it begins to retrieves data from this *port* till gets a stop sign (Figure 24).

The whole process finishes after the process A gets the result set. From this structure,

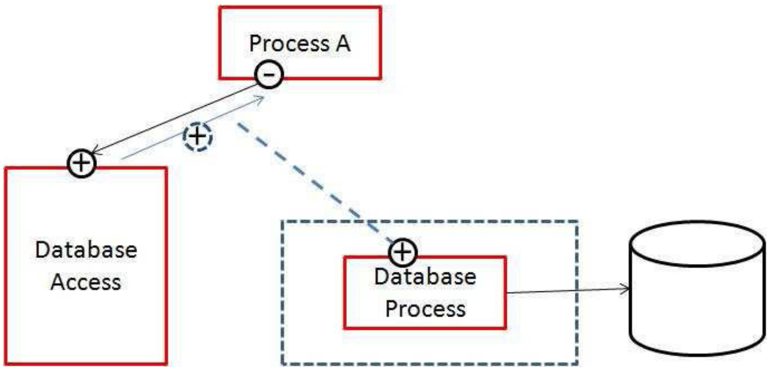

Figure 23: Get service port from ‘Database Access’

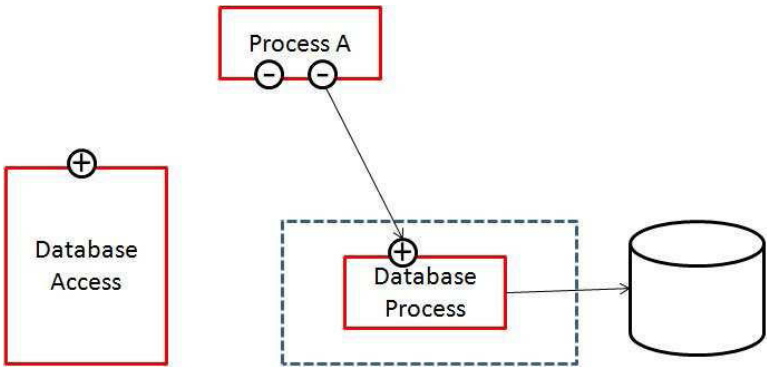

Figure 24: Get result set

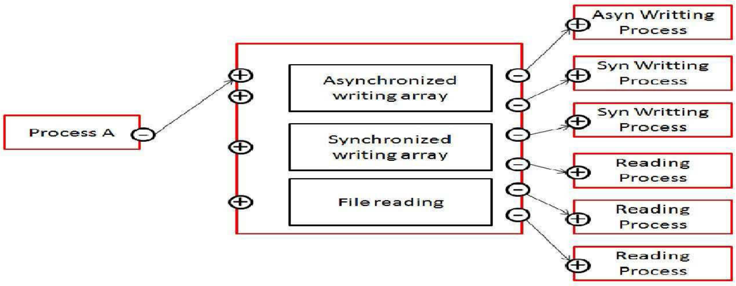

Figure 25: Write and read file

all the details about the database are hidden from the business processes and the database connection is reusable because building a database connection is really expensive.

Program 18 is a demo of this model (Program 18).

Except database, files are also kind of resource in web environment. The file is locked when one process is writing. Meanwhile, other processes which want to read or write this file have to wait to get control. As same as the strategy of database, in Erasmus, we can set a special process which provides service of reading and writing files and also manages the cache of the reading buffer to improve the performance of the system (Figure 25).

Furthermore, the Erasmus WriteFile Process provides two kinds of writing modes: synchronized writing and asynchronized writing. If a program wants to write to a file, it can choose synchronous writing: After it sends content to the FileWrite process, it waits untill the WriteFile Process sends back a response to tell that the file is successfully written or there is a writing failure. But, for some writing job, for example, writing logs, the process can choose asynchronize writing mode. The process only send the content to the WriteFile process, don't need to wait the writing process be finished. The WriteFile process sends back a unique ID to the client and the client can query the status of the job at any time. In the Figure 25, there are 2 service ports related to asynchronous Writing, one is for writing

Program 18 Database access

```
prot = [ num1: Integer;num2: Integer ];

sql=[*(^textField:Text)|^stopSign:Bool];
sqlask=[sqlquery:Text;^ch:sql];

DatabaseProcess ={s:+sql;request:Text|
success:Bool;
success:=ODBC_query(request);
stringValue:Text;
loop while (success)
success:=ODBC_nextValue(stringValue);
if(success) then
s.textField:=stringValue;
end;
end;
s.stopSign:=true;
success:=ODBC_closeQuery();
}

DatabaseAccess={ sqlAsk:+sqlask|
success:Bool :=ODBC_open("odbcsql","sa","password");
loopselect
  || request:Text:= sqlAsk.sqlquery;
     s:sql;
   DatabaseProcess(s,request);
   sqlAsk.ch:=s;
end;
}

ProcessA={ sqlAsk: -sqlask |
sqlQuery:Text := "select * from table"
sqlAsk.sqlquery :=sqlQuery;
ch:sql :=sqlAsk.ch;
loopselect
||sys.out:="\n value:"+ch.textField+"\n";
||ch.stopSign;
exit
end;
}

Cell = (p: sqlask; DatabaseAccess(p); ProcessA(p); );
Cell();
```

and another is for query the status of the task. Not only the client process can save a lot of waitting time, but also, the WriteFile process can set special strategy to adjust the priorities of the writing task. In Figure 25, we can tell that the resource assigned to "asynchronized writing, synchronized writing and Reading" is 1:2:3.

5.4 Erasmus Web Application

In this chapter, we introduced the main idea of our Erasmus web application and the framework of the system.

An Erasmus web application includes:

- A broker which is responsible for distributed communication.
- A HTTP listener to provide HTTP service.
- A HTTP process which can generate process to deal with the HTTP request and send back response.
- A Process 'Database Access' which provide database access service.

To illustrate the development of a web application in Erasmus, we give an example: Login which demonstrates the main steps to develop an Erasmus web application.

At first, we define two protocols:

HTTP= [request: Text; ˆresponse: Text];

Login= [username: Text; password: Text; ˆresult: Text];

Then we define the main process which works as a router. When it gets a HTTP request to verify the credential, it creates a login process to check the username and password (Program 19).

Program 19 HTTP process

```
HTTPProcess = {+p : http; -q: Login |
loopselect ordered:
          ||  request : text= http. Request;
              if( compare(request , POST, 'Login . process')
       //send to Login Process
        q.username=getParameter(request,'username');
                  q.password=getParameter(request,'password');
            end;
          || loginResult: text = q.result;
               if( result = 'success')
                      response=loadHtml('success.html');
               else
                     response=loadHtml('fail.html');
              end;
     end;
}
```

After that we define the process:Login which is responsible for verifying the username and password from database (Program 20).

At the end, we define the Cell (Program 21). The Cell includes three processes: DatabaseAccess, Login and Main.

With these Erasmus scripts, A Login web application is setup. If the clients send correct username and password, they will get a page shows the checking is success, or they will get a page tells them the password is not correct.

Program 20 Login process

```
Login ={ +q:Login; sqlAsk: -sqlask |
     loopselect
||username=q.username; password=q.password;
    sqlQuery:Text := "select count(*) from user where username="
     + username+ " and password=" + password;
sqlAsk.sqlquery :=sqlQuery;
ch:sql :=sqlAsk.ch;

   If(ch.textField = "0")
        q.result='fail'
   else
        q.result='success'
   end;
   ch.stopSign;
  end;
}
```

Program 21 Cell

```
Cell( p:Http; q:Login; sql: sqlask;
  DatabaseAccess(sql);
        Login(q);
        Main(p);
};
```

Chapter 6

MVC Framework in Erasmus

6.1 MVC Framework in Java

When we design a web application, we have to refer to the MVC (Model-View-Controller) framework. Model-View-Controller is a software architecture, a MVC application is a collection of model/view/controller triads, each is responsible for a different UI element. MVC structure is to reduce the complexity in design and to increase flexibility and maintainability of code (Figure 26).

The model manages the behaviour and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller). In event-driven systems, the model notifies observers (usually views) when the information changes so that they can react.

The view renders the model into a form suitable for interaction, typically a user interface element. Multiple views can exist for a single model for different purposes. A viewport typically has a one to one correspondence with a display surface and knows how to render to it.

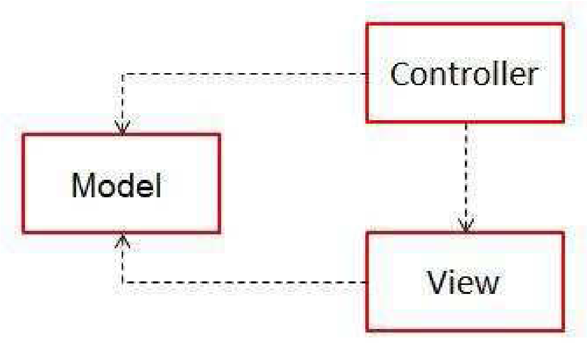

Figure 26: MVC framework

The controller receives user input and initiates a response by making calls on model objects. A controller accepts input from the user and instructs the model and viewport to perform actions.

- The user interacts with the user interface in some way (for example, presses a mouse button).
- The controller handles the input event.
- The controller notifies the model of the user action, possibly resulting in a change in the model's state.
- A view queries the model in order to generate an appropriate user interface. The view gets its own data from the model.
- The user interface waits for further user interactions, which restarts the cycle.

Model

The model is a collection of Java classes that form a software application intended to store, and optionally separate, data. A single front end class that can communicate with any user interface (for example: a console, a graphical user interface, or a web application).

View

The view is represented by a JavaServer Page, with data being transported to the page in the HttpServletRequest or HttpSession.

Controller

The Controller servlet communicates with the front end of the model and loads the HttpServletRequest or HttpSession with appropriate data, before forwarding the HttpServletRequest and Response to the JSP using a RequestDispatcher.

6.2 MVC in Erasmus

The purpose of using MVC in Erasmus is to improve the efficiency of development of Erasmus web applications. With MVC, developers can easily build a web application which implements MVC model, because they need not care about the communications among the processes of Controllers, Models and Viewers but can focus on business logic of the application.

As same as the MVC in other languages, MVC in Erasmus also includes the Passive model and the Active model.

6.2.1 Passive model

To implement the MVC Passive Model in Erasmus, we will have 3 kinds of processes: Controller, Model, and View. These 3 kinds of processes are correspond with the 3 triads of MVC framework: model/view/controller.

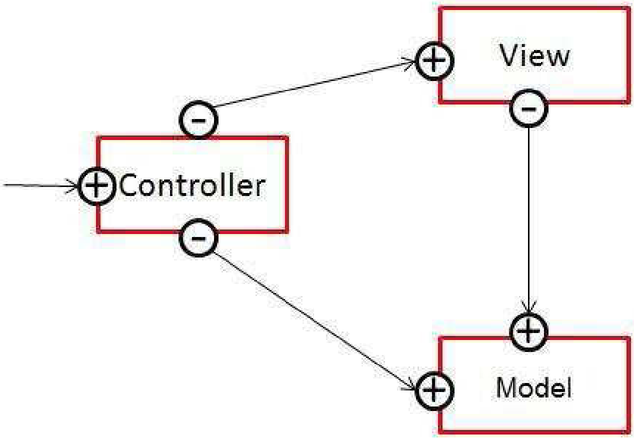

Figure 27: MVC Passive Model framework in Erasmus

The workflow in Erasmus is really similar to the MVC workflow (Figure 27).

- The user interacts with the user interface in some way (for example, presses a mouse button).
- The controller handles the input event. According to the request, Erasmus creates two processes: Model and View. The Model has a service port, so the view can get data from Model when the view renders the response page.
- The Controller sends a request to ask the Model to change the state.
- The View queries the model in order to generate an appropriate user interface. The view gets its own data from the Model.
- The user interface waits for further user interactions, which restarts the cycle.

According to this framework, a web application is composed of a set of processes which are Model or View or Controller. So, we need a special language to describe them. Using

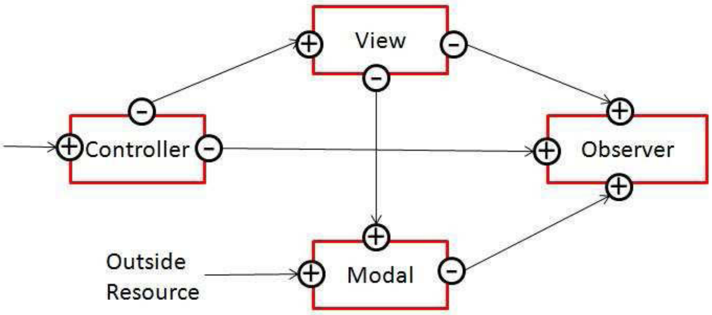

Figure 28: MVC Active Model framework in Erasmus

this script to describe a web application is far simpler than using Erasmus. Then we have a special compiler to compile the scripts into Erasmus. To explain the details, we give a simple example about this design in the section: An Example of MVC Application (6.3).

6.2.2 Active model

For Active Model, we add one more process which is called Observer to detect the change of the model process. In Active Model, the Controller is not involved in the change of the Model process which is modified by other resource (Figure 28).

In the Active Model, the Model is changed by outside resource. When there is a change of the Model, the Model will notify Observer. If at this time, Controller get a request to display the view, Controller will first send a request to Observer to see if there is a change of the Model. Now, Controller knows that the view needs to be refreshed. So, the View process refreshes the view and sends it back to Controller. By contraries, if the Observer reports to Controller there is no change of the Model, Controller will get the existed view from the View process.

We still use stock price as an example to illustrate the workflow. If the stock price

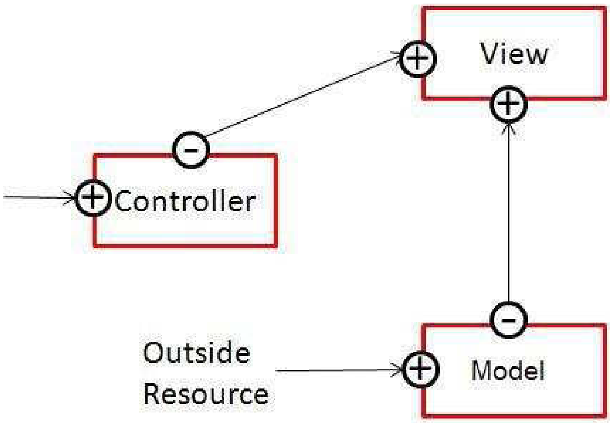

Figure 29: Simplified Active Model framework in Erasmus

changed, the Model will send Observer a message: there is a new price. When Controller get a request from internet to check the price of the stock, Controller first ask Observer if there is a change of the price. Next, Controller asks the View to refresh. At end, Controller send new price back to the Internet user. However, if a new request is coming from Internet when there is no change of the price, Observer will tell Controller There is no update of the price. Therefore, Controller will get old price from View and forward to clients.

From the example of stock price, the Active Model of MVC can provide the newest price of the stock for Internet users. But, in fact, we can simplify this Model, because in the securities market, there are thousands of stocks whose price is changing quickly, providing the new price in a second is enough for our internet user. For this scenario, we update our Active Model to the Simplified Active Model which reduces the load from the web server(Figure 29).

In Simplified Active Model of MVC, when there are changes in a period, the Model will notify the view to update for every period. When the Controller receives requests from

Internet, it always gets the view from the View process and provides it to internet users, no matter there is a change or not in Model.

The Active Model and the Simplified Active Model are using for different scenarios. The Active Model is suitable for the scenarios in which the Model is changing less frequently; on the other hand, the Simplified Active Model is more efficient for the scenarios in which the Model is changing too quickly for server to refresh the view for every update which will overburden the web server.

6.3 An Example of MVC Application

We still use Login as an example, a user login system with username and password. If the password is not correct, the user will get a page saying that the username or password is not correct; if the username and password are correct, it shows: Welcome to the system.

When user input username and password and click the button "Login", then the browser send a request to $http://system.com/Login$ with two parameters: username and password.

At first, the request comes to the controller which in fact is a router. When the Controller knows that this request is a Login, then it creates two processes: a Model and a View (Program 22).

We can find that we only define one protocol: Result and one port: q which is in between the Models: LoginModel and LoginView. In fact, we hide the protocol between the Controller and Model, and the protocol between the Controller and view because they are default protocols:

Request = [Request: Text] Response = [ˆResponse: Text]

So, to simplify the code, the framework will define these ports and protocols by default.

We don't need to define it in the code specially.

In Program 22, we tell the system, when user send a request to http://system.com/Login, the Controller should create two processes LoginModel and LoginView. Also we can have more choices, for example: Withdraw, Deposit, Check etc.

The MVC Framework compiler will compile this controller component into Erasmus code (Program 22):

Now, we use our scripts to define the Model process of Login (Program 23).

Program 23 runs a SQL query to search the database to check if there is user whose username and password are as same as the request. It calls the Erasmus Database service to get the query result. If the result is bigger than one, it means that the user is a legal user. The compiler for MVC will translate the scripts into Erasmus code (Program 23).

At the end, we define the View of Login. The View of Login in fact is a segment of HTML scripts, in the scripts, according the result from LoginModel, we render a different page to the client. So, there are two key characters of the LoginView. It uses the result from the LoginModel and it returns a HTML page to the client (Program 24).

In the scripts, according the result from LoginModel, we return to client a different HTML page. The compiler of MVC will translate this script into Erasmus code (Program 24).

Of course, to listen to the HTTP port, we need a config file to set up HTTP listener which receives requests form HTTP port and forwards them to Controller (Program 25).

To make it clear, we put all the MVC scripts together, we can get a conclusion that it dramatically reduce the complexity of the web development since it hides the communication in detail and expresses business logic clearly (Program 26).

Program 22 MVC scripts and Erasmus code of the Controller

```
----------------------------------------------------
MVC scripts
----------------------------------------------------

Controller= {
Login:
LoginModel;
LoginView;
}

----------------------------------------------------
Compiled Erasmus Scripts
----------------------------------------------------

HTTP = [*(request: Text; ^response: Text)];
Request= [Request: Text]
Response = [^ Response: Text]
Result = [ ^result: Text ];

Controller ={http:+HTTP; toModel: -Request; toView: -Response |
Loopselect
     || httpRequest: text := http.request;
     If(httpRequest.equals("Login"))
        q: Result;
      LoginModel(toModel,q);
      LoginView( toView,q);
        toModel.Request:=httpRequest;
     endif
     || response: text := Response.Response;
        http: response:=response;
end
}
```

Program 23 MVC scripts and Erasmus code of the LoginModel

```
----------------------------------------------------
MVC scripts
----------------------------------------------------

LoginModel = {
sql:Text := "Select count(*) from Users
where Username="+Request.get("Username")+
    " and password="+Request.get("password");
ErasmusDbf(sql,ch);
return ch.result;
}

----------------------------------------------------
Compiled Erasmus Scripts
----------------------------------------------------

Request= [Request: Text]
Result = [ ^result: Text ];

LoginModel = {toModel: +Request; q: +Result
select
|| Request: text := toModel.Request;
    sql:Text := "Select count(*) from Users where Username="+
     Request.get("Username")+"and password="+
     Request.get("password");
ErasmusDbf(sql,ch);
q.result=ch.result;
end;
}
```

Program 25 HTTP Listener of Erasmus

```
<No>1
    <Port> 80 </Port>
   <type> Server </type>
    <remote>*</remote>
    <process>Controller</process>
   <portofprocess>http</portofprocess>
    <protocol>HTTP</protocol>
</No>
```

Program 24 MVC Scripts and Erasmus code of the LoginView

```
----------------------------------------------------
MVC scripts
----------------------------------------------------

LoginView ={
return "
    <html xmlns="http://www.w3.org/1999/xhtml">
   <head>
<title>Login</title>
   </head>
   <body>
   <% if(LoginModel.result=="0" then %>
<p> UserName or Password is not correct !!
   <% else %>
<p> Welcome to System!!
   <%end>
   </body>
   </html>";
}

----------------------------------------------------
Compiled Erasmus Scripts
----------------------------------------------------

Response = [^ Response: Text]
SQLResult = [ ^result: Text ];

LoginView ={ toView: +Response; q: -Result||
select
       || result: text := q.result;
      If(result=="0")
               Response.Response="
    <html xmlns=\"http://www.w3.org/1999/xhtml\">
    <head>
<title>Login</title>
    </head>
    <body>
  <p> UserName or Password is not correct !!
 </body>
    </html>"
   Else
       Response.Response="
    <html xmlns=\"http://www.w3.org/1999/xhtml\">
    <head>
<title>Login</title>
    </head>
    <body>
   <p>Welcome to System!!
 </body>
    </html>"
  Endif;
End;
```

Program 26 MVC scripts

```
Controller= {
Login:
LoginModel;
LoginView;
}

LoginModel = {
sql:Text := "Select count(*) from Users"+
"‘where Username="+Request.get("Username")+
    " and password="+Request.get("password");
ErasmusDbf(sql,ch);
return ch.result;
}

LoginView ={
return "
    <html xmlns="http://www.w3.org/1999/xhtml">
   <head>
<title>Login</title>
    </head>
    <body>
   <% if(LoginMode l.result=="0" then %>
<p> UserName or Password is not correct !!
    <% else %>
<p> Welcome to System!!
    <%end>
    </body>
    </html>";
}
```

With this example, we introduced the primary idea of MVC pattern in Erasmus. Although the business logic of this example is rather simple, it proves MVC pattern is feasible in our framework. But, to use this framework for complicate web application of all kinds of business logics, we still need to extend the functions of our MVC compiler and has a long way to achieve it.

Chapter 7

Contribution and Future Work

7.1 Contribution

First, I built a background application called Broker running as a Windows service, which can support communications among nodes. My work is based on the project of Nurudeen Lameed who was a master student of professor Peter Grogono [26]. In his project, his broker can send and receive messages among nodes. I added a new feature to the broker: the broker can listen for HTTP request and send response. Furthermore I modified Erasmus compiler to make the Erasmus use this broker to commincate among nodes and listen for HTTP requests. My broker is compiled and is running on Windows environment.

Second, in my project, I built a web application framework which can support web service and database access. I implemented a simple web application which dynamically creates a web service process to verify the username and password by accessing database, and sends back an Html file and some image files to browsers. This application was compiled into a C++ scripts with my compiler which is built on Erasmus compiler and is running on Windows.

Third, I built a MVC compiler which can compile MVC file into Erasmus scripts. The compiled Erasmus scripts accord with MVC pattern. However, this compiler is not flexible and can only compile the files which I was using as an example to show that MVC pattern is feasible in Erasmus. The compiler can not deal with complex scenarios, for example the passive model of the MVC because I only designed the algorithms but didn't implement them.

7.2 Discussion

Our web framework provides a solution in development a web application in Erasmus, which indicates that Erasmus, as a process oriented language, is suitable for web application development. But, compared with other applications, web application is more complicated and need some special tools to help the developers. These tools are developed in object oriented languages and are built on the concept of object. Erasmus is a process oriented language and doesn't have the concept of object, should our Erasmus web framework introduce similar tools and how can we implement them in Erasmus.

7.2.1 ORM

Does Erasmus need ORM(Object Relational Mapping)?

With the evolution of the database, the ORM, for example, Hibernate, ActiveRecord, EJB entity bean, becomes more and more popular. Why the ORM can be quickly accepted by developers because there is a lot of advantages with ORM tools.

- Facilitates implementing the Domain Model pattern.

  In short, using this pattern means that your model entities based on real business concepts rather than based on your data base structure. ORM tools provide this functionality through mapping between the logical business model and the physical storage model.

- Huge reduction in code.

  ORM tools provide a host of services thereby allowing developers to focus on the business logic of the application rather than repetitive CRUD (Create Read Update Delete) logic.

- Changes to the object model are made in one place.

  Once we update our object definitions, the ORM will automatically use the updated structure for retrievals and updates. There are no SQL Update, Delete and Insert statements strewn throughout different layers of the application that need modification.

- Navigation.

  we can navigate object relationships transparently. Related objects are automatically loaded as needed.

- Data loads are completely configurable allowing to load the data appropriate for each scenario.

- Concurrency support.

  Support for multiple users updating the same data simultaneously.

- Cache management.

  Entities are cached in memory thereby reducing load on the database.

- Transaction management and isolation.

  All object changes occur scoped to a transaction. The entire transaction can either be committed or rolled back. Multiple transactions can be active in memory in the same time, and each transaction changes are isolated form on another.

ORM is based on the concept of Object. But our Erasmus, as an OPL, is a process oriented language. We don't have the concept of Object, so it is not possible to introduce ORM into Erasmus. In the future, maybe we should introduce a similar concept into Erasmus. For example, PRM (Process Relational Mapping) is a possible framework which is suitable for the process oriented language.

7.2.2 Message Service Framework

Messaging is a form of loosely coupled distributed communication where in this context the term 'communication' can be understood as an exchange of messages between software components. Message Service makes our application loosely coupled and more flexible. There are two kinds of Message Service Modal:

- Point-Point model

- Publish and Subscribe model

In Point-Point Model, the sender posts a message which is only for a special receiver; In Publish and Subscribe model, the sender posts messages with a special topic, and the receives register with topic to get messages.

In Erasmus, a protocol is a message pipe which defines the format of the message. Each message is sent and received via protocol between ports. The difference is that the message is exchanged directly between processes. However, in Message Service Model, the messages are communicated indirectly through a middleware. Introduction of Message Service Model in Erasmus should be able to increase the flexibility of the Erasmus system.

With Erasmus, we expect that it will be very easy to fulfill the Message Service Framework. In this model, some processes don't communicate each other, they send and receive messages through a process which is responsible for the distribution of the messages by Point by Point or publish/subscribe methods.

For Publish Model, because the relationship among ports is one-to-many, we need protect the channel before the communication is finished. Only after the channel becomes free, another process then can use this channel to send or retrieve messages. For Subscribe Port, subscribers register to the Message process with a topic as a queue. When there is a message available, Message process chooses a subscriber from queue and sends the message to the client.

Erasmus does not yet support this kind of one-to-many ports structure. The new version of Erasmus will support this kind of framework.

7.2.3 The Advantages and Disadvantages

Comparing with other web application language, Erasmus has its advantages: First, Erasmus is a process oriented language and web application is a process based application. Erasmus is a natural web application language. So, the web application in Erasmus is clearer and easier to understand and implement.

Second, our framework is very powerful in distributed environment. The distributed environment is transparent for the developers.

Third, Erasmus is a language designed for concurrent system. With Erasmus, the system should be able to fully take advantage of the multi CPU of the machine. The web application in Erasmus of course can inherent the advantages of this language. We can image how fast for a machine with 10 CPUs to dispose 10 requests at the same time, which will dramatically improve the performance of the system.

However, Erasmus is a language in developing and web application is a very complicated system. Our framework is only at design stage and we don't have much experience in developing real web applications. From the thesis, we have shown that Erasmus is suitable for web application development, but, when we really put it into practice, we will find a lot details we need consider and a lot of difficulties we need to overcome.

7.3 Conclusion

In this thesis, we introduced a process oriented language: Erasmus which is designed by Peter Grogono at Concordia University, Canada and Brian Shearing at The Software Factory UK. Because of the special ability to deal with the concurrency and the distributed environment, Erasmus is very fitting as a web application language. To fulfill the communication in distributed environment, we introduce the notion of broker to be responsible for the communications between nodes. Because of Broker, Erasmus has the potential to be a powerful language for distributed environment.

Then, we introduce a HTTP listener into Broker to service HTTP requests. In Chapter four, we describe how the system listens for the requests, creates new process to deal with the requests, and sends back the responses to the HTTP client.

In Chapter five, we specially illustrate how to access database in Erasmus which is very important for a web application language, since database, as a resource, it is a critical component when developing a web application. In Erasmus, we are using indirect method to access database, means that the database resource is controlled by a process called ‘Database Access’.

At the end, we introduce MVC model in Erasmus which reduces the complexity in design and increases flexibility and maintainability of code. Our MVC example shows that the MVC pattern is feasible and efficient in Erasmus web application framework.

In conclusion, Erasmus a potential good web development language, but Erasmus is still an experimental language, before it becomes a practical language to develop a real web application, there is still much work to do.

Bibliography

[1] Delivery mode. http://docs.oracle.com/javaee/1.4/api/javax/jms/DeliveryMode.html. [Online; accessed 23-Dec-2011].

[2] Eve online records 45,000 simultaneous players. http://www.tutorialspoint.com/web_developers_guide/web_server_types.htm. [Online; accessed 23-July-2010].

[3] Eve online records 45,000 simultaneous players. http://www.shacknews.com/article/56605/eve-online-records-45000-simultaneous. [Online; accessed 23-July-2010].

[4] Jdbc technology. http://www.oracle.com/technetwork/java/overview-141217.html. [Online; accessed 17-Dec-2011].

[5] Jms. http://en.wikipedia.org/wiki/Java_Message_Service. [Online; accessed 23-Dec-2011].

[6] Oracle jms. http://java.sun.com/developer/technicalArticles/Ecommerce/jms/. [Online; accessed 23-Dec-2011].

[7] Sinatra the book. http://sinatra-book.gittr.com/. [Online; accessed 23-July-2011].

[8] Web application. http://en.wikipedia.org/wiki/Web_application. [Online; accessed 19-July-2010].

[9] Joe Armstrong. Programming Erlang: Software for a Concurrent World, pages 5 – 15. July 18, 2007.

[10] M. Ben-Ari. Principles of Concurrent and Distributed Programming, pages 3 – 46. Israel Institute of Technology, 1990.

[11] Steve Burbeck. Applications programming in smalltalk-80(tm): How to use model-view-controller (mvc). http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html, 1987. [Online; accessed 23-Dec-2011].

[12] Ethan Cerami. Web Services Essentials, pages 119 – 121. O'Reilly, 1005 Gravenstein High way North CA 95472, February 2002.

[13] Edsger W. Dijkstra. Programming languages: Nato advanced study institute. Academic Press, pages 43 – 112, 1968.

[14] Kieren Diment. The Definitive Guide to Catalyst: Writing Extendable, Scalable and Maintainable Perl-Based Web Applications, pages 21 – 24. July 9, 2009.

[15] Alexandru Ersenie. J2ee-full garbage collections when running with cms garbage collection strateg, Jan,2011.

[16] Dino Esposito. Programming Microsoft ASP.NET MVC, pages 62 – 70. October 28, 2011.

[17] Jim Farley and William Crawford. Java Enterprise In a NutShell, pages 249 – 255. O'Reilly, 1005 Gravenstein High way North CA 95472, November 2005.

[18] Jean Dollimore G Coulouris and Tim Kindberg. Distributed Systems: Concepts and Design, pages 11 – 13. Addison Wesley, August 21, 2000.

[19] Peter Grogono and Brian Shearing. Sketches. Examples of Erasmus codes.

[20] Peter Grogono and Brian Shearing. A modular language for concurrent programming. Journal of Parallel and Distributed Computing (1995), Volume: 26, Issue: 1, 2006.

[21] Peter Grogono and Brian Shearing. Desi: Erasmus for 2010. url-http://users.encs.concordia.ca/ grogono/Erasmus/desi-report.pdf, November 2011. [Online; accessed 19-Jan-2012].

[22] Rachid Guerraoui and Luis Rodrigues. Introduction to Reliable Distributed programming, pages 3 – 5; 26 – 28. Springer Berlin Heidelberg New York, April 2006.

[23] Harumi Kuno Gustavo Alonso, Fabio casati and Vijay Machiraju. Web Service concepts, Architectures and Applications, pages 94 – 95. Springer, 1501 Page Mill Road, MS1142 CA,USA, 2004.

[24] C. A. R. Hoare. Communicating sequential processes. ICSOFT 2008 - Proceedings of the Third International Conference on Software and Data Technologies, pages 666 – 677, 2004.

[25] Balachander Krishnamurthy and Jennifer Rexford. Web Protocols and practice, pages 176 – 177. Addison-Wesley, May 2001.

[26] Nurudeen Lameed. Implementing concurrency in a process-based language. Master's thesis, Department of Computer Science, Concordia University, March 2008.

[27] Nurudeen Lameed and Peter Grogono. Separating program semantics from deployment. ICSOFT 2008 - Proceedings of the Third International Conference on Software and Data Technologies, PL/DPS/KE:54 – 67, 2008.

[28] Tom Christiansen Larry Wall and Randal Schwartz. Programming perl, pages 17 – 31. O'reilly, September 1996.

[29] Gordon McComb. Web Programming Languages Sourcebook, pages 1 – 21. John Wiley & Sons Canada, Ltd., 1997.

[30] Sun Microsystems. Memory Management in the Java HotSpot. Virtual Machine, pages 7 – 12. 2006.

[31] Richard Monson-Haefel. J2EE Web Services, pages 79 – 127. Addison-Wesley, 2004.

[32] SAPE MULLENDER. Distributed Systems, pages 12 – 45. ACM Press, New York, 1993.

[33] Jamie Munro. Programming MVC 3: Faster, Smarter Web Development, pages 34 – 56. O'Reilly Media, 11 Oct 2011.

[34] Mark Tomlinson Olaf Zimmermann and Stefan Peuser. Perspectives on Web Services, pages 55 – 75. Springer, 1501 Page Mill Road, MS1142 CA,USA, 2003.

[35] Nurudeen Lameed Peter Grogono and Brian Shearing. Modularity + concurrency = manageability. pages 1 – 3.

[36] Christian Queinnec. Continuations and web servers. Higher-Order and Symbolic Computation, Volume 17 Issue 4, December 2004.

[37] Tom Phoenix Randal Schwartz and brian d foy. Learning perl, pages 1 – 3. O'reilly, July 2005.

[38] rj45. Simple example of mvc (model view controller) design pattern for abstraction. `http://www.codeproject.com/KB/tips/ModelViewController.aspx`. [Online; accessed 22-July-2010].

[39] Frank Leymann Tony Storey Sanjiva Weerawarana, Francisco Curbera and Donald F. Ferguson. Web Services Platform Architecture, pages 109 – 121. Pearson Education, Inc., Upper Saddle River, NJ 07458, February 2006.

[40] Adrian Lienhard Stephane Ducasse and Lukas Renggli. Seaside. A multiple control flow web application framework. In Proceedings of ESUG Research Track 2004, pages 232 – 233, L. 2004.

[41] Shaun Terry. Enterprise JMS Programming, pages 10 – 58. Wiley, 2002.

[42] Stephen A. Thomas. HTTP Essentials: Protocols for Secure, Scaleable Web Sites, pages 55 – 56. Wiley, first edition, March 8, 2001.

[43] Kim Topley. J2ME In a Nutshell, pages 10 – 25. O'Reilly, 2002.

[44] Sing Li Vivek Chopra and Jeff Gendender. Professional Apache Tomcat 6, pages 51 – 69. Wrox, 2007.

[45] Bill Brogden William B. Brogden. SOAP Programming with Java, pages 36 – 78. Sybex Inc, 2002.

[46] Markus Winand. A practical comparison of the java and c++ resource management facilities. `http://fatalmind.com/papers/java_vs_cplusplus/resource.pdf`, 2003.